AGERE! at SPLASH 2012

2nd International Workshop on Programming based on Actors, Agents, and Decentralized Control

Workshop held at ACM SPLASH 2012
21-22 October 2012
Tucson, Arizona, USA

PRE-PROCEEDINGS

These pre-proceedings are available on the web site at:
http://agere2012.apice.unibo.it

AGERE! is an ACM SIGPLAN workshop,
sponsored by Typesafe, Inc.

Introduction

ago, agis, egi, actum, **agere**
latin verb meaning to act, to lead, to do,
common root for actors and agents

"The Free Lunch is Over" also for Abstractions

The fundamental turn of software into concurrency, interactivity, and distribution is not only a matter of performance, but also design and abstraction. *The free lunch is over* [14] calls for devising new programming paradigms – possibly as evolution of existing ones — that would allow for natural ways of thinking about, designing, developing, executing, debugging and profiling systems that exhibit different degrees of concurrency, autonomy, decentralization of control, and physical distribution. Almost any application today requires the programming of software components that actively –proactively and reactively –carry out multiple tasks, react to various kinds of events, and communicate with each other. Relevant research questions include: how to properly program these entities and systems of entities, what kinds of programming abstractions can help in systematically structuring complex reactive and proactive behaviors, and what kinds of programming abstractions can be effective in organizing applications as ensembles of relatively autonomous entities working together.

Actors, Agents and Abstractions for Decentralized Control

Given this premise, in SPLASH 2011 the AGERE! workshop [1] was proposed for the first time to investigate the definition of proper levels of abstraction, programming languages, and platforms to support and promote a decentralized mindset [11] in systems development. To this end, *agents* (and multi-agent systems) and *actors* were taken as a starting point, as two main broad families of concepts described in the literature. These abstractions and programming tools explicitly promote such a decentralized-control mindset from different facets, depending on the context in which they are discussed, e.g., concurrent programming or distributed artificial intelligence.

Actors [3] and object-oriented concurrent programming [15, 2] couple object-oriented programming with concurrency, providing a clean and powerful computation model which is nowadays increasingly adopted in mainstream languages, frameworks and libraries. Agents and agent-oriented programming [5, 6, 7, 10, 13, 12] provide a rich abstraction layer on top of actors and objects. This approach aims at easing programming of concurrent/distributed systems conceived as societies of autonomous and proactive task-oriented individuals interacting in a shared environment.

The wave of interest on concurrency and distribution in mainstream programming has been clearly witnessed also through the good number of contributions accepted to OOPSLA and OnWard! in SPLASH 2011 (and in other recent editions) that addresses those same issues. However, the main focus in those contributions (including invited talks and panels) so far has been mainly on issues related to performance, and *mechanisms for extending mainstream paradigms* to effectively exploit the power of e.g. multi-core and many-core architectures. While acknowledging the importance of those objectives, at the same time we argue for the importance of strengthening the research on new paradigms aiming *first* at improving the conceptual modeling and the level of abstraction used to design and program such complex software systems.

With that main objective in mind, AGERE! is organized in SPLASH 2012 to promote the investigation of the features that would make agent-oriented and actor-oriented programming languages effective and general-purpose in developing software systems as an evolution of OOP. Besides actors and agents, the workshop is meant, more generally, to serve as a venue for all programming approaches and paradigms investigating how to effectively specify and structure control when programming reactive systems [9, 8, 4] providing new abstractions for dealing, e.g., with management of asynchronous events and the efficient execution of concurrent activities.

Theory and Practice of Programming with Actors, Agents and Decentralized Control Abstractions

All stages of software development are considered interesting for the workshop, including requirements, modeling, prototyping, design, implementation, testing, and any other means of producing running software based on actors and agents as first-class abstractions. The scope of the conference includes aspects that concern both the theory and the practice of design and programming using such paradigms, so as to bring together researchers working on the models, languages and technologies, as well as the practitioners using such technologies to develop real-world systems and applications.

Finally, the overall perspective of the workshop is what distinguishes this event from related venues (e.g. about agents) organized in different contexts (e.g. AI) with the intent to hopefully impact mainstream programming paradigms and software development. Another purpose of the workshop is to serve as a forum for collecting, discussing, and confronting related research work that typically appears in different communities in the context of (distributed) artificial intelligence, distributed computing, computer programming, and software engineering.

Acknowledgment

The organizing committee would like to thank all program committee members, authors and participants. Thank you to ACM and SPLASH organizers, and to Typesafe Inc., for their support. We look forward to a productive workshop.

Committee

Program Committee

Gul Agha, University of Illinois at Urbana-Champaign, USA
Joe Armstrong, SICS / Ericsson, Sweden
Saddek Bensalem, Verimag, France
Rafael H. Bordini, FACINPUCRS, Brazil
Gilad Braha, Google, USA
Rem Collier, UCD, Dublin
Tom Van Cutsem, Vrije Universiteit, Brussel, Belgium
Amal El Fallah Seghrouchni, LIP6 Univ. P and M. Curie, Paris, France
Jurgen Dix, Technical University of Clausthal, Germany
Philipp Haller, Typesafe, Switzerland
Tom Holvoet, Dept. Computer Science K.U.Leuven, Belgium
Einar Broch Johnsen, University of Oslo, Norway
Hillel Kugler, Microsoft, USA
Assaf Marron, Weizmann Institute of Science, Israel
Mark Miller, Google, USA
Olaf Owe, University of Oslo, Norway
Jens Palsberg, UCLA, Los Angeles, USA
Ravi Pandya, Microsoft, USA
Arnd Poetzsch-Heffter, University of Kaiserslautern, Germany
Alessandro Ricci, University of Bologna, Italy
Birna van Riemsdijk, Delft University of Technology, The Netherlands
Giovanni Rimassa, Whitestein Technologies, Switzerland
Munindar Singh, North Carolina State University, USA
Marjan Sirjani, Reykjavik University, Iceland
Gera Weiss, Ben Gurion University, Israel
Guy Wiener, HP, Israel
Akinori Yonezawa, University of Tokyo, Japan

Organizing Committee & PC Chairs

Gul Agha, University of Illinois at Urbana-Champaign, USA
Rafael H. Bordini, FACIN–PUCRS, Brazil
Assaf Marron, Weizmann Institute of Science, Israel
Alessandro Ricci, University of Bologna, Italy

Bibliography

[1] *SPLASH '11 Workshops: Proceedings of the compilation of the co-located workshops on DSM'11, TMC'11, AGERE!'11, AOOPES'11, NEAT'11 & VMIL'11*, New York, NY, USA, 2011. ACM.

[2] G. Agha. Concurrent object-oriented programming. *Commun. ACM*, 33:125–141, September 1990.

[3] G. A. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott. A foundation for actor computation. *J. Funct. Program.*, 7(1):1–72, Jan. 1997.

[4] A. Basu, M. Bozga, and J. Sifakis. Modeling heterogeneous real-time components in bip. In *Proceedings of the Fourth IEEE International Conference on Software Engineering and Formal Methods*, SEFM '06, pages 3–12, Washington, DC, USA, 2006. IEEE Computer Society.

[5] R. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni, editors. *Multi-Agent Programming Languages, Platforms and Applications - Volume 1*. Springer, 2005.

[6] R. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni, editors. *Multi-Agent Programming Languages, Platforms and Applications - Volume 2*. Springer, 2009.

[7] R. H. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni. Special issue on multi-agent programming. *Autonomous Agents and Multi-Agent Systems*, 23 (2), 2011.

[8] D. Harel, A. Marron, and G. Weiss. Behavioral programming. *Commun. ACM*, 55(7):90–100, July 2012.

[9] D. Harel and A. Pnueli. *On the development of reactive systems*, pages 477–498. Springer-Verlag New York, Inc., New York, NY, USA, 1985.

[10] J. J. Odell. Objects and agents compared. *Journal of Object Technology*, 1(1):41–53, 2002.

[11] M. Resnick. *Turtles, Termites and Traffic Jams. Explorations in Massively Parallel Microworlds.* MIT Press, 1994.

[12] A. Ricci and A. Santi. Agent-oriented computing: Agents as a paradigm for computer programming and software development. In *Proc. of Future Computing '11*, Rome, Italy, 2011.

[13] Y. Shoham. Agent-oriented programming. *Artificial Intelligence*, 60(1):51–92, 1993.

[14] H. Sutter and J. Larus. Software and the concurrency revolution. *ACM Queue: Tomorrow's Computing Today*, 3(7):54–62, Sept. 2005.

[15] A. Yonezawa and M. Tokoro. *Object-oriented concurrent programming.* MIT Press series in computer systems. MIT Press, 1987.

Invited Talks

1. *On the integration of the actor model in mainstream technologies – The Scala perspective*
   Philipp Haller – Typesafe

2. *20 years of Agent-Oriented Programming in Distributed AI: History and Outlook*
   Birna van Riemsdijk – Delft University of Technology (The Netherlands)

3. *Agents, Concurrent Objects, and High Performance Computing*
   Akinori Yonezawa – University of Tokyo (Japan)

On the Integration of the Actor Model into Mainstream Technologies

A Scala Perspective

Philipp Haller
Typesafe, Inc.
philipp.haller@typesafe.com

Abstract

Integrating the actor model into mainstream software platforms is challenging because typical runtime environments, such as the Java Virtual Machine, have been designed for very different concurrency models. Moreover, to enable integration with existing infrastructures, execution modes and constructs foreign to the pure actor model have to be supported. This paper provides an overview of past and current efforts to address these challenges in the context of the Scala programming language.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming— Distributed and parallel programming; D.2.13 [*Software Engineering*]: Reusable Software—Reusable libraries

Keywords Concurrent programming, distributed programming, actors, threads

1. Introduction

Actors are a powerful abstraction for structuring highly concurrent software systems which scale up to many-core processors, as well as scale out to clusters and the cloud. The Scala community is well-known for its effort to bring the actor model to mainstream software engineering. The first actors implementation was released as part of the Scala standard library in 2006 [6, 15]. Since then, there has been a steady stream of both research results and industrial development, contributing to a renewed interest in actors in academia, as well as innovations powering state-of-the-art frameworks like the Akka event-driven middleware [16].

Integrating the actor model into mainstream software platforms is a formidable challenge. On the one hand, industrial-strength implementations have to make optimal use of underlying runtime environments which typically have not been designed to support actors. On the other hand, in order to integrate with existing infrastructures, it is necessary to support execution modes and constructs that are rather foreign to a pure notion of actors, such as blocking operations, and interoperability with native platform threads.

This paper provides an overview of the challenges of providing an industrial-strength actor implementation on the Java Virtual Machine (JVM), in the context of the Scala programming language [13]. It aims to serve as an experience report on addressing these challenges through a combination of research and engineering advances.

We're going to focus on the two main actor implementations for Scala: Scala Actors [7] and Akka [16]. The former has been part of Scala's standard library since Scala version 2.1.7. Beginning with Scala version 2.10.0, Scala Actors are deprecated in favor of Akka's actor implementation, a new member of the Scala distribution. One goal of this paper is to motivate this transition, and to examine which ideas of Scala Actors are adopted in Akka, and what has changed in the design and implementation.

1.1 Overview

The design and implementation of industrial-strength actor implementations on mainstream platforms, such as the JVM, is driven by numerous requirements. Some requirements guided the design and implementation of Scala's first actors framework; these "early requirements" were:

- *Library-based implementation (R1).* It is unclear which concurrency abstraction is going to "win". Real-world concurrency tasks might even benefit from a combination of several different abstractions. Rather than betting on a single candidate and providing built-in language support, Scala's approach has been to enable flexible concurrency *libraries*.
- *High-level domain-specific language (R2).* While actors are provided using library abstractions in Scala, it is important that their programming interface is "competitive" with languages with specialized concurrency support.
- *Event-driven implementation (R3).* Actors should be mapped to lightweight tasks triggered by messaging events. Spending an entire virtual machine thread per actor does not scale to large numbers of actors. At the same time, the benefits of thread-based programming should remain accessible in cases where a purely event-driven model would be too restrictive.

In retrospect, these requirements are still valid, however, other requirements turned out to be more important in the context of industrial software development. With the growing use of actors in production Scala applications, it became clear that satisfying *only* these early requirements was not sufficient to meet all demands. Other requirements had to be added, and existing ones turned out to be useful beyond their initial goals. These "later requirements" were:

- *High performance (R4).* The majority of industrial applications where actors provide most benefits are highly performance sensitive. Past experience with industrial Erlang applications [1, 12] suggests that *scalability* is more important than raw performance. On the other hand, it is known that a high-performance virtual machine such as the JVM can enable applications to scale out much more gracefully than other runtime environments. In addition, benchmarking offers a simple evaluation strategy if the compared benchmark programs are of similar complexity.

AGERE! October 21-22, 2012.

- *Flexible remote actors (R5).* Many medium-sized and large applications can benefit from remote actors, i.e., actors that communicate transparently over the network. In many cases, flexible deployment mechanisms, e.g., using external configuration, are very important.

The early requirement of a library-based implementation turned out to provide additional benefits: first, it enables existing tools, such as IDEs and debuggers, to be readily supported. Second, it is possible to provide APIs for several languages. For example, the Akka framework has both a Scala and a Java API.

In the following we are going to "tackle" these requirements, in two groups: the first group is concerned with the *programming interface* (see Section 2) which addresses requirements R1 and R2. The second group is concerned with the *actor runtime* (see Section 3) which addresses requirements R3 and R4. Remote actors (R5) are beyond the scope of this paper.

2. The Programming Interface

This section provides an overview of the programming interface of both Akka and Scala Actors, and how the interface is realized as a library in Scala.

An actor is a process that communicates with other actors by exchanging messages. The principal message send operation is asynchronous. Therefore, an actor buffers incoming messages in a message queue, its *mailbox*. The *behavior* of an actor determines how the messages in its mailbox are processed. Since defining an actor's behavior is a rather important activity when programming with actors, it is crucial that an actor programming system has good support for it. In Scala, the behavior of an actor can be defined by creating a new class type that extends a predefined `Actor` trait.[1] Figure 1 shows an example using Scala Actors (top) and Akka Actors (bottom), respectively.

In Scala Actors, the body of the `act` method (inherited from `Actor`) defines an actor's behavior. In the above example, it repeatedly calls the `receive` operation to try to receive a message. The receive operation has the following form:

```
receive {
  case msgpat₁ => action₁
  ...
  case msgpatₙ => actionₙ
}
```

The first message which matches any of the patterns $msgpat_i$ is removed from the mailbox, and the corresponding $action_i$ is executed. If no pattern matches, the actor suspends.

The example in Figure 1 (top) uses `receive` to wait for two kinds of messages. The `Order(item)` message handles an order for `item`. An object which represents the order is created and an acknowledgment containing a reference to the order object is sent back to the sender. The `Cancel(o)` message cancels order `o` if it is still pending. In this case, an acknowledgment is sent back to the sender. Otherwise a `NoAck` message is sent, signaling the cancellation of a non-pending order.

The API of Akka's actors is similar to that of Scala Actors. The principal way of defining a message handler for incoming messages is the implementation of the `receive` method, which is inherited from the `Actor` trait. The body of the `receive` method has the same form as in the case of `receive` in Scala Actors.

For simplicity, we're going to refer to the latter as "sreceive" and to the former as "areceive" (Akka's `receive`) in the following. The main difference between "sreceive" and "areceive" is that the

[1] A trait in Scala is an abstract class that can be mixin-composed with other traits.

```
class OrderManager extends Actor {
  def act() {
    while (true) {
      receive {
        case Order(item) =>
          val o = handleOrder(sender, item)
          sender ! Ack(o)
        case Cancel(o) =>
          if (o.pending) {
            cancelOrder(o)
            sender ! Ack(o)
          } else sender ! NoAck
      }
    }
  }
}

class OrderManager extends Actor {
  def receive = {
    case Order(item) =>
      // same as above
    case Cancel(o) =>
      // same as above
  }
}
```

Figure 1. Example: orders and cancellations.

former operation is *blocking*, i.e., the current actor is suspended until a matching message can be removed from its mailbox. On the other hand, "areceive" is used to define a global message handler, which, by default, is used for processing all messages that the actor receives over the course of its life time. Moreover, the message handler defined by "areceive" only gets activated when a message can be removed from the mailbox. Another important difference is that "areceive" will never leave a message in the mailbox if there is no matching pattern which is different compared to "sreceive". Whenever the actor is ready to process the next message, it is removed from the mailbox; if there is no pattern that matches the removed message, an event is published to the system, signaling an unhandled message.

The example in Figure 1 (bottom) defines a global message handler which handles the same two kinds of messages as the example at the top.

2.1 Bridging the Gap

The semantics of "sreceive" and "areceive" are quite different. "sreceive" has the same semantics as "receive" in Erlang [2]. On the other hand, "areceive" can be implemented more efficiently on the JVM. Each construct enables a different programming style for messaging protocols. To support both styles, Akka 2.0 introduces a `Stash` trait which an `Actor` subclass can optionally mix in. Together with methods to change the global message handler of an actor (called `become` and `unbecome` in Akka), the stash enables the familiar Erlang style also using Akka.

2.2 Creating Actors

In Scala Actors, creating a new instance of a subclass of `Actor` (such as `OrderManager` in Figure 1) creates an actor with the behavior defined by that class. All interaction with the actor (message sends etc.) is done using references to that instance.

In Akka Actors, an actor is created using one of several factory methods of an instance of type `ActorRefFactory`, say `factory`:

```
val actor = factory.actorOf(Props[OrderManager])
```

In many cases, the `factory` object is the “actor system”, the container which provides shared facilities (e.g., task scheduling) to all actors created in that container. (The `factory` can also be a “context” object which is used to create supervision hierarchies for fault handling.) The expression `Props[OrderManager]` in Scala is equivalent to `Props.apply[OrderManager]`, an invocation of the `apply` factory method of the `Props` singleton object. Singleton objects have exactly one instance at runtime, and their methods are similar to static methods in Java. The `Props.apply` method returns an instance of the `Props` class type, which contains all information necessary for creating new actors.

The main difference between creating an actor in Scala Actors and in Akka is that the above `actorOf` method in Akka returns an instance of type `ActorRef` instead of an instance of the specific `Actor` subclass. One of the main reasons is *encapsulation*.

2.3 Encapsulation

The actor runtime guarantees thread safety of actor interactions only if actors communicate only by passing messages. However, in Scala Actors it is possible for an actor to directly call a (public) method on a different actor instance. This breaks encapsulation and can lead to race conditions if the state of the target actor is accessed concurrently [10].

To prevent such encapsulation breaches, in Akka actors have a very limited interface, `ActorRef`, which basically only provides methods to send or forward messages to its actor. Akka has built-in checks to ensure that no direct reference to an instance of an `Actor` subclass is accessible after an actor is created. This mechanism works surprisingly well in practice, although it can be circumvented. [9]

An alternative approach to ensuring encapsulation of actors is a typing discipline such as uniqueness types [3]. The capability-based *separate uniqueness* type system [8] has been implemented as a prototype for Scala [5]. However, more research needs to be done to make such type systems practical.

2.4 Implementation

Looking at the examples shown above, it might seem that Scala is a language specialized for actor concurrency. In fact, this is not true. Scala only assumes the basic thread model of the underlying host. All higher-level operations shown in the examples are defined as classes and methods of the Scala library. In the rest of this section, we look “under the covers” to find out how selected constructs are defined and implemented.

The send operation `!` is used to send a message to an actor. The syntax `a ! msg` is simply an abbreviation for the method call `a.!(msg)`, just like `x + y` in Scala is an abbreviation for `x.+(y)`. Consequently, `!` can be defined as a regular method:

```
  def !(msg: Any): Unit = ...
```

The `receive` constructs are more interesting. In Scala, the pattern matching expression inside braces is treated as a first-class object that is passed as an argument to “sreceive”, and returned from “areceive”, respectively. The argument’s type is an instance of `PartialFunction`, which is a subtrait of `Function1`, the type of unary functions. The two traits are defined as follows.

```
trait Function1[-A, +B] {
  def apply(x: A): B
}
trait PartialFunction[-A, +B]
  extends Function1[A, B] {
  def isDefinedAt(x: A): Boolean
}
```

Functions are objects which have an `apply` method. Partial functions are objects which have in addition a method `isDefinedAt` which tests whether a function is defined for a given argument. Both traits are parameterized; the first type parameter `A` indicates the function’s argument type and the second type parameter `B` indicates its result type[2].

A pattern matching expression
`{ case` p_1 `=>` e_1`; ...; case` p_n `=>` e_n `}` is then a partial function whose methods are defined as follows.

- The `isDefinedAt` method returns `true` if one of the patterns p_i matches the argument, `false` otherwise.
- The `apply` method returns the value e_i for the first pattern p_i that matches its argument. If none of the patterns match, a `MatchError` exception is thrown.

The two methods are used in the implementation of “sreceive” as follows. First, messages in the mailbox are scanned in the order they appear. If the argument `f` of “sreceive” is defined for a message, that message is removed from the mailbox and `f` is applied to it. On the other hand, if `f.isDefinedAt(m)` is `false` for every message `m` in the mailbox, the receiving actor is suspended.

The Akka runtime uses partial functions differently: first, the behavior of an actor is defined by implementing the `receive` method; this method *returns* a partial function, say, `f`. The messages in the actor’s mailbox are processed in FIFO order. The Akka runtime guarantees that at most one message (per receiving actor) is processed at a time. Each message, say `msg` is removed from the mailbox regardless of `f`. If `f` is defined for `msg`, `f` is applied to it. On the other hand, if `f.isDefinedAt(msg)` is `false`, `msg` is published as an “unhandled message” event to the system (wrapped in an object which additionally contains references to the sender and receiver).

3. The Actor Runtime

As motivated in the introduction, the most important features of the actor runtime are (a) a lightweight execution environment, and (b) high performance. In the following we will outline how these features are realized in Akka and which ideas of Scala Actors stood the test of time.

3.1 Event-Based Actors

Scala Actors [6] introduced a new approach to decouple actors and threads by providing an event-based operation for receiving messages, called “react”. In this approach, an actor waiting for a message that it can process is not modeled by blocking a thread; instead, it is modeled by a closure which is set up to be scheduled for execution when a suitable message is received. At that point a *task* is created which executes this continuation closure, and submitted to a thread pool for (asynchronous) execution.

In this approach, the continuation closure is actually an instance of type `PartialFunction[Any, Unit]` (see Section 2.4). Akka has adopted this idea: the continuation of an actor waiting for a message is an instance of the same type. The main difference is that when using “react”, this continuation closure is provided *per message* to be received; in contrast, in Akka the continuation closure is defined once, to be used for many (or all) messages. Additionally, Akka provides methods to change the global continuation (`become`/`unbecome`). The main advantage of Akka’s approach is that it can be implemented much more efficiently on the JVM. In

[2] Parameters can carry `+` or `-` variance annotations which specify the relationship between instantiation and subtyping. The `-A`, `+B` annotations indicate that functions are contravariant in their argument and covariant in their result. In other words `Function1[X1, Y1]` is a subtype of `Function1[X2, Y2]` if `X2` is a subtype of `X1` and `Y1` is a subtype of `Y2`.

the absence of first-class continuations, implementing "react" requires the use of *control-flow exceptions* to unwind the call stack, so that each message is handled on a call stack which is basically empty. Throwing and catching a control exception for each message is additional overhead compared to Akka's execution strategy.

3.2 Lightweight Execution Environment

A key realization of Scala Actors is the fact that for actor programs a workstealing thread pool with local task queues [11] scales much better than a thread pool with a global task queue. The main idea is as follows: when creating a task which executes the message handler to process a message, that task is submitted to the *local task queue* of the current worker thread. This avoids an important bottleneck of thread pools with a global task submission queue which can quickly become heavily contended.

Like Scala Actors, Akka uses Lea's fork/join pool (an evolution of [11], released as part of JDK 7 [14]). In addition, and unlike Scala Actors, Akka uses non-blocking algorithms for inserting messages into actor mailboxes, and scheduling tasks for execution, which results in a substantial performance boost.

3.3 Integrating Threads and Actors

Integrating (JVM) threads and event-based actors is useful to enable powerful message-processing operations also for regular threads. This facilitates interoperability with existing libraries and frameworks and offers additional convenience, since it enables actors to be more easily used from Scala's interactive REPL (read-eval-print-loop). Besides Scala Actors, there are other approaches attempting an integration of threads and event-based actors [4].

In Scala Actors, calling `receive` on a regular thread, which is not currently executing an actor, establishes an actor identity and mailbox in thread-local storage. This actor identity can be passed to other actors in messages, so as to add the thread actor to their set of acquaintances.

Akka version 2.1 introduces an `Inbox` abstraction which let's one create a first-class actor mailbox as follows:

```
implicit val i = ActorDSL.inbox()
someActor ! someMsg // replies will go to 'i'

val reply = i.receive()
val transformedReply = i.select(5 seconds) {
  case x: Int => 2 * x
}
```

The message send in the second line above *implicitly* transmits an `ActorRef` obtained from the `Inbox i` as the sender of `someMsg`. As a result, responses of the receiving actor (via `sender ! someResponse`) are enqueued in `i`. Methods such as `receive` and `select` enable blocking access to one message at a time. The downside of a first-class mailbox is, of course, that it does not come with a guarantee that there is only a single thread receiving from the same mailbox, since it could be shared among multiple threads. On the other hand, the advantage is that it allows an efficient implementation, and it is relatively straight-forward to avoid subtle memory leaks.

3.4 Summary

- Scala's partial functions are well-suited to represent an actor's continuation.
- The overhead of unwinding the call stack through exceptions can be avoided by using a single, global message message handler. Loss in flexibility can be recovered through constructs to replace the global message handler.
- A workstealing thread pool with local task queues is an ideal execution environment for event-based actors.
- Threads and actors can be integrated in a robust way using first-class actor mailboxes. On the other hand, it does not guarantee unique receivers.

4. Conclusion

In this paper we have outlined the requirements and forces of mainstream software engineering which have influenced past and present actor implementations for Scala. Based on these requirements, principles behind the design and implementation of actors in Scala are explained, covering (a) the programming interface, and (b) the actor runtime. It is our hope that the learned lessons will be helpful in the design of other actor implementations for platforms sharing at least some features with Scala and/or the Java Virtual Machine.

References

[1] J. Armstrong. Erlang — a survey of the language and its industrial applications. In *Proc. INAP*, pages 16–18, Oct. 1996.

[2] J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in Erlang, Second Edition*. Prentice-Hall, 1996.

[3] D. Clarke, T. Wrigstad, J. Östlund, and E. B. Johnsen. Minimal ownership for active objects. In *Proceedings of the 6th Asian Symposium on Programming Languages and Systems (APLAS'08)*, pages 139–154. Springer, Dec. 2008.

[4] T. V. Cutsem, S. Mostinckx, and W. D. Meuter. Linguistic symbiosis between event loop actors and threads. *Computer Languages, Systems & Structures*, 35(1):80–98, 2009.

[5] P. Haller. *Isolated Actors for Race-Free Concurrent Programming*. PhD thesis, EPFL, Switzerland, Nov. 2010.

[6] P. Haller and M. Odersky. Event-based programming without inversion of control. In D. E. Lightfoot and C. A. Szyperski, editors, *JMLC*, volume 4228 of *Lecture Notes in Computer Science*, pages 4–22. Springer, 2006. ISBN 3-540-40927-0.

[7] P. Haller and M. Odersky. Scala actors: Unifying thread-based and event-based programming. *Theor. Comput. Sci*, 410(2-3):202–220, 2009.

[8] P. Haller and M. Odersky. Capabilities for uniqueness and borrowing. In *Proceedings of the 24th European Conference on Object-Oriented Programming (ECOOP'10)*, pages 354–378. Springer, June 2010. ISBN 978-3-642-14106-5.

[9] P. Haller and F. Sommers. *Actors in Scala*. Artima Press, 2012.

[10] R. K. Karmani, A. Shali, and G. Agha. Actor frameworks for the JVM platform: a comparative analysis. In *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java (PPPJ'09)*, pages 11–20. ACM, Aug. 2009. ISBN 978-1-60558-598-7.

[11] D. Lea. A Java fork/join framework. In *Java Grande*, pages 36–43, 2000.

[12] J. H. Nyström, P. W. Trinder, and D. J. King. Evaluating distributed functional languages for telecommunications software. In *Proc. Workshop on Erlang*, pages 1–7. ACM, Aug. 2003. ISBN 1-58113-772-9.

[13] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala, Second Edition*. Artima Press, 2010.

[14] Oracle, Inc. Java SE Development Kit 7. `http://openjdk.java.net/`.

[15] The Scala Programming Language. Home page. `http://www.scala-lang.org/`.

[16] Typesafe, Inc. Akka framework. `http://www.akka.io`.

Accepted Papers

1. *Domains: Safe sharing among actors*
   Joeri De Koster, Tom Van Cutsem and Theo D'Hondt – Vrije Universiteit (Belgium)

2. *Soter: An Automatic Safety Verifier for Erlang*
   Emanuele D'Osualdo, Jonathan Kochems and Luke Ong – Oxford University (United Kingdom)

3. *Leveraging Actors for Privacy Compliance*
   Jeffery Von Ronne – The University of Texas at San Antonio (United States)

4. *Adding Distribution and Fault Tolerance to Jason*
   Álvaro Fernández Díaz, Clara Benac Earle and Lars-Ake Fredlund – Universidad Politcnica de Madrid (Spain)

5. *Programming Abstractions for Integrating Autonomous and Reactive Behavior: An Agent-Oriented Approach*
   Alessandro Ricci, Andrea Santi – University of Bologna (Italy)

6. *Empirical Software Engineering for Agent Programming*
   Birna Van Riemsdijk – TU Delft (The Netherlands)

7. *Messages with Implicit Destinations as Mobile Agents*
   Ahmad Ahmad-Kassem, Stphane Grumbach and Stphane Ubda – INRIA INSA Lyon, INRIA (France)

8. *A Decentralized Approach for Programming Interactive Applications with JavaScript and Blockly*
   Assaf Marron, Gera Weiss and Guy Wiener – Weizmann Institue of Science, Ben Gurion University, HP Labs (Israel)

9. *Optimized Distributed Implementation of Multiparty Interactions with Observation*
   Saddek Bensalem, Marius Bozga, Jean Quilbeuf and Joseph Sifakis – VERIMAG, VERIMAG/CNRS (France)

10. *Distributed Priority Synthesis using Knowledge*
    Chih-Hong Cheng, Rongjie Yan, Harald Ruess and Saddek Bensalem – Fortiss (Germany), State Key Laboratory of Computer Science Institute of Software (China), VERIMAG (France)

11. *Parallel Gesture Recognition with Soft Real-Time Guarantees*
    Thierry Renaux, Lode Hoste, Stefan Marr and Wolfgang De Meuter – Vrije Universiteit (Belgium)

12. *A Relational Trace Logic for Simple Hierarchical Actor-Based Component Systems*
    Ilham W. Kurnia and Arnd Poetzsch-Heffter – University of Kaiserslautern (Germany)

13. *Timed-Rebeca Schedulability and Deadlock-Freedom Analysis Using Floating-Time Transition System*
    Ehsan Khamespanah, Zeynab Sabahi Kaviani, Ramtin Khosravi, Marjan Sirjani and Mohammad-Javad Izadi – Tehran University (Iran), Reykjavik University (Iceland)

14. *Actor Idioms*
    Dale Schumacher – United States

Domains: Safe sharing among actors

Joeri De Koster Tom Van Cutsem Theo D'Hondt
Vrije Universiteit Brussel,
Pleinlaan 2,
B-1050 Brussels, Belgium
jdekoste@vub.ac.be tvcutsem@vub.ac.be tjdhondt@vub.ac.be

Abstract

The actor model has already proven itself as an interesting concurrency model that avoids issues such as deadlocks and race conditions by construction, and thus facilitates concurrent programming. While it has mainly been used in a distributed context it is certainly equally useful for modeling interactive components in a concurrent setting. In component based software, the actor model lends itself to naturally dividing the components over different actors and using message passing concurrency for implementing the interactivity between these components. The tradeoff is that the actor model sacrifices expressiveness and efficiency especially with respect to parallel access to shared state.

This paper gives an overview of the disadvantages of the actor model in the case of shared state and then formulates an extension of the actor model to solve these issues. Our solution proposes *domains* and *synchronization views* to solve the issues without compromising on the semantic properties of the actor model. Thus, the resulting concurrency model maintains deadlock-freedom and avoids low-level race conditions.

1. Introduction

Traditionally, concurrency models fall into two broad categories: message-passing versus shared-state concurrency control. Both models have their relative advantages and disadvantages. In this paper, we explore an extension to a message-passing concurrency model that allows safe, expressive and efficient sharing of mutable state among otherwise isolated concurrent components.

A well-known message-passing concurrency model is the actor model [3]. In this model, applications are decomposed into concurrently running actors. Actors are isolated (i.e., have no direct access to each other's state), but may interact via (asynchronous) message passing. While originally designed to model open, distributed systems, and thus often used as a distributed programming model, they remain equally useful as a more high-level alternative to shared-memory multithreading. Both component-based and service-oriented architectures can be modeled naturally using actors. It is important to point out that in this paper, we restrict ourselves to the use of actors as a concurrency model, not as a distribution model.

AGERE! @ SPLASH '12 21-22 October 2012 - Tucson, Arizona, USA.
Copyright © 2012 ACM [to be supplied]. . . $10.00

In practice, the actor model is either made available via dedicated programming languages (actor languages), or via libraries in existing languages. Actor languages are mostly *pure*, in the sense that they often strictly enforce the isolation of actors: the state of an actor is fully encapsulated, cannot leak, and asynchronous access to it is enforced. Examples of pure actor languages include Erlang [5], E [18], AmbientTalk [23], Salsa [24] and Kilim [20]. The major benefit of pure actor languages is that the developer gets very strong safety guarantees: low-level race conditions are avoided. On the other hand, these languages make it difficult to express shared mutable state. Often, one needs to express shared state in terms of a shared actor encapsulating that state, which has several disadvantages, as will be discussed in Section 2.4.

On the other end of the spectrum, we find actor libraries, which are very often added to an existing language whose concurrency model is based on shared-memory multithreading. For Java alone, there exist the ActorFoundry [6], AsyncObjects [2], ... Scala, which inherits shared-memory multithreading from Java, features multiple actor frameworks, such as Scala Actors [12] and Akka [1]. What these libraries have in common is that they cannot typically enforce actor isolation, i.e. they do not guarantee that actors don't share mutable state. On the other hand, it's easy for a developer to use the underlying shared-memory concurrency model as an "escape hatch" when direct sharing of state is the most natural or most efficient solution. However, once the developer chooses to go this route, all of the benefits of the high-level actor model are lost, and the developer typically has to resort to manual locking to prevent data races.

The goal of this work is to enable safe, expressive and efficient state sharing among actors:

safe : the isolation properties of actors are often helpful to bring structure to, and help reason about, large-scale software. Consider for instance a plug-in or component architecture. By running plug-ins in their own isolated actors, we can guarantee that they do not violate invariants of the "core" application. Thus, as in pure actor languages, we want an actor system that maintains strong language-enforced guarantees, such as the fact that low-level data races and deadlocks are prevented by design.

expressive : many phenomena in the real world can be naturally modelled using message-passing concurrency (e.g. telephone calls, e-mail, digital circuits, discrete-event simulations, etc.). Sometimes, however, a phenomenon can be modelled more directly in terms of shared state. Consider for instance the scoreboard in a game of football, which can be read in parallel by thousands of spectators. As in impure actor libraries, we want an actor system in which one can directly express access to shared mutable state, without having to encode shared state via a shared actor. Furthermore, by enabling direct *synchronous* access to shared state, we gain stronger synchronization constraints and prevent the inversion of control that is characteristic of interacting with actors (as interaction is typically asynchronous).

efficient : today, multicore hardware is becoming the prevalent computing platform, both on the client and the server [21]. While multiple isolated actors can be perfectly executed in parallel by different hardware threads, shared access to a single actor can still form a serious sequential bottleneck. In particular, in pure actor languages that support mutable state, all requests sent to an actor are typically serialized, even if some requests could be processed in parallel (e.g. requests to simply read or query some of the actor's state). Pure actors lack multiple-reader, single-writer access, which is required to enable truly parallel reads of shared state.

In this paper, we propose *domains*, an extension to the actor model that enables safe, expressive and efficient sharing among actors. Since we want to provide strong language-level guarantees, we present domains as part of a small actor language called *Shacl*[1]. In terms of sharing state, our approach strikes a middle ground between what is achievable in a pure actor language versus what can be achieved using impure actor libraries. An interpreter for the whole SHACL language can be found on our website[2].

In the next section 2 we present a number of problems that occur when representing shared state within the actor model. In section 3 we present our domain and view abstractions. In section 4 we list a number of important additional features of SHACL. And to conclude the paper we have a related work section and finally a conclusion.

2. The problem: Accessing non-local shared state

In this section we introduce SHACL, a small implementation of a pure event-loop actor language for which we introduce new features to allow synchronous access to shared state as we go along. There are two ways to represent shared state in the event-loop actor model: either by replicating the shared state over the different actors or by encapsulating the shared state as an additional independent actor. In this section we discuss the disadvantages of both approaches using a motivating example.

2.1 Shacl: An event-loop actor language

The sequential subset of SHACL implements a prototype-based object model similar to Self [22]. This object model also has an inheritance model, supports late binding and static super references. However, these are not relevant in the context of this paper and thus will not be discussed. The concurrency model of SHACL is based on the event-loop model of E [18] and AmbientTalk [23] where actors are represented by *vats*. Each vat/actor has a single thread of execution, an object heap, and an event queue. Each object in the object heap of an actor is *owned* by that actor."Owning" an object gives that actor exclusive access rights to that object. Any reference to an object owned by the same actor is called a *local reference*. A reference to an object owned by another actor is called a *remote reference*. The type of reference determines the access capabilities of the actor on the referenced object. While objects pointed to by local references can be synchronously accessed, any remote object can only be accessed through asynchronous message passing. Thus, sending a message to another actor in this model is just a matter of sending a message to a remote object in that actor's object heap. Any incoming message is then added to the event queue of the actor that owns the remote object. The thread of execution of that actor combined with the event queue form the event-loop of that actor. This event-loop processes arriving messages one by one. The processing of a single message is called a *turn*. Each of those turns is processed in a single atomic step. As in E and AmbientTalk, SHACL supports non-blocking futures to implement two-way messages without using callbacks (See section 4.2). This model was extended with the notion of *domains* to allow shared state between different actors in a controlled way.

2.2 Motivating example

As a motivating example we looked into plugin architectures. Isolation and encapsulation are important properties for such architectures to model the different plugins. Hence, the event-loop actor model is a good fit for such application as it already enforces these properties on the level of the language. The actor model lends itself to naturally model the different plugins each as an actor and using message passing concurrency to model the communication between these different components of the application. However, in such applications, it is common for different plugins to require access to a shared resource. Whether it be globally available for all plugins or just shared between a subset of the plugins.

Figure 1 shows a model of an application where two different plugins use a binary search tree (BST) as a shared data-structure. The binary search tree can be queried for a certain key using `query` and users can insert key-value pairs using `insert`. In our application Plugin 1 will periodically insert new key-value pairs in the tree and Plugin 2 will first query the tree and depending on the result insert a new key-value pair in the tree.

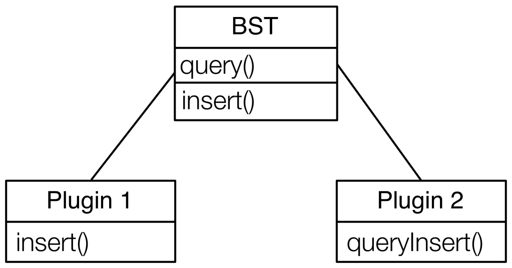


Figure 1. Two plugins using the same shared resource

In the next two sections we will use this example to motivate the issues of using shared state within the actor model.

2.3 Replication

One option for representing shared state in the actor model would be to replicate this state inside different actors that require access to it. For our specific example this would mean that the BST would be replicated in both plugin 1 and 2. This approach has a number of issues:

Consistency Keeping replicated state consistent requires a consistency protocol that usually does not scale well with the number of participants. In our specific example this approach can be used but if we consider applications with hundreds of components this no longer feasible. Lowering the availability or consistency of shared state can be a solution to this problem [9]. Unfortunately, whether lowering either availability or consistency is possible is entirely application dependent. In general, keeping replicated state consistent is a hard problem that usually leads to inefficient code.

Memory usage increases linearly with the amount of shared state and the number of actors. Depending on the granularity with which actors are created, this might incur a memory overhead that is too high.

[1] Pronounce as "shackle", short for **sh**ared **ac**tor **l**anguage

[2] http://soft.vub.ac.be/~jdekoste/shacl

Copying cost Sometimes short lived objects need to be shared between different actors and the cost of copying them is greater than the cost of the operation that needs to be performed on them.

For our approach, we could try to hide these consistency protocols and try to lower the memory usage and copying costs. Unfortunately this solution does not scale very well with the number of actors, making it very unfeasible to use.

2.4 Shared state as an additional independent actor

Using a separate actor to encapsulate shared state is the more natural solution as it does not require any consistency protocols and also scales well with the number of other actors accessing that shared state. There are however three different classes of problems when using this approach:

Continuation-passing style enforced. Using a distinct actor to represent conceptually shared state implies that this resource can not be accessed directly from any other actor since all communication happens asynchronously within the actor model. Thus, the programmer needs to explicitly handle a request-response situation, which usually forces the programmer to employ an explicit continuation-passing-style.

No synchronization conditions. The traditional actor model does not allow specifying extra synchronization conditions on different operations since the order in which events from different senders are handled is nondeterministic.

No parallel reads. State that is conceptually shared can never be read truly in parallel because all accesses to this state are sequentialized by the event queue of the encapsulating actor.

Figure 2 shows an implementation of the BST that is encapsulated by a separate actor. Similarly to E [18] and AmbientTalk [23] in SHACL the `actor{<expression>}` syntax evaluates to a new actor with its own separate object heap and event-loop. That object heap is then initialized with a single new object initialized by `<expression>`. The actor syntax immediately evaluates to a remote reference to that newly created object. Any asynchronous message that is sent to that reference will then be scheduled as an event in the event-loop of the newly created actor. For this example we intentionally left out the implementation details of the BST. What is important here is its interface and how it can be accessed. In this example, querying or updating the BST requires the use of asynchronous communication. Because of that, querying the BST for a value requires the use of callbacks to implement the response message. In our example this is done by passing a callback object, namely the `client` (lines 19–25), as a second argument of the query method. This callback object then has to implement a queried method that is called with the result of the query method.

In this example the `insert` method of plugin 1 just delegates any insert calls to the BST. On the other hand, plugin 2 provides a `queryInsert` method that will query the BST for a certain key and will then decrement the value of that key if it is positive.

In this section we will discuss the three issues raised above in more detail using our motivating example.

Asynchronous communication leads to continuation passing style

The style of programming where a computation is divided into different execution steps is called continuation passing style (CPS), also known as programming without a call stack [15]. This problem of using CPS to access a remote resource is typical and can be found in various other actor languages like Salsa [24], Kilim [20], etc. The problem with this style of programming is that it leads to "inversion of control".

```
1  let bst = actor {
2    insert(key, value) {
3      ...
4    }
5    query(key, client) {
6      result := ...
7      client<-queried(result);
8    }
9  }
10
11 let plugin1 = actor {
12   insert(bst, key, value) {
13     bst<-insert(key, value)
14   }
15 }
16
17 let plugin2 = actor {
18   queryInsert(bst, key) {
19     bst<-query(key, object {
20       queried(value) {
21         if(value > 0) {
22           bst<-insert(key, value - 1)
23         }
24       }
25     });
26   }
27 }
```

Figure 2. A shared bst encapsulated in a separate actor

Figure 2 shows that the restriction of only being able to communicate asynchronously with a remote shared resource forces the programmer to structure his code in a very unintuitive way (CPS, lines 19–25). If we want to query and afterwards insert a new value in the BST to update it, we either have to extend the implementation of our BST with an `update` method or we combine the `query` and `insert` method in some way. Let us assume that changing the interface of the BST is not possible[3] and we need to employ the latter solution. Inter-actor communication always happens asynchronously in the event-loop model and therefore does not yield a return value. If we want to access items in our BST we will need a way to send back the result of the `query` method. The common approach to achieve this is to add an extra argument to each message that represents a callback. This client implements the continuation of our program given the return value of the message. In our example this is done via the `queried` method.

On line 19 we asynchronously send a `query` message to the BST passing a key as a parameter as well as a reference to an object that implements the continuation of our program given the return value of the `query` method (lines 20–24). Once the `bst` actor is processing the event it will eventually send back the result of the query to the client object via an asynchronous message (line 7). Because the client object was created by the `plugin 2` actor that message will then be scheduled as an event in the event-queue of the `plugin 2` actor. Note that while the `bst` actor is busy processing the query request, the `plugin 2` actor is available for handling other incoming events. Once the `plugin 2` actor is ready to process the "queried" event it can then decide whether or not to send an insert message to the bst actor depending on the return value of the query (lines 21–23).

The lack of synchronous communication with remote resources forces us to write our code in a CPS. Ideally we would want the

[3] This can be true for various reasons. Either legacy reasons or it might be that the query and insert messages need to be combined in a non-trivial way that also involves other remote objects.

`query` and `insert` method to be evaluated in the context of one event, which is not possible in either the event-loop actor model.

Extra synchronization conditions on groups of messages are not possible

In some cases it is possible that a certain interleaving of the evaluation of different messages leads to event-level race conditions. For example, in Figure 2 we introduce a race condition when both plugin 1 and plugin 2 try to insert a new value in the BST. Even if we would somehow avoid having to use CPS, any unwanted interleaving of the `query` an `insert` methods might lead plugin 2 to update the BST using old information. For example, if the bst actor first receives a query event from plugin 1, then the insert event from plugin 2 and only then the insert event from plugin 1, then plugin 1 updated the value of the bst depending on old information which is a race condition.

The reason race conditions like these occur when programming in an actor language is because different messages, sent by the same actor, cannot always be processed atomically. Programmers cannot specify extra synchronization conditions on groups of messages. A programmer is limited by the smallest unit of non-interleaved operations provided by the interface of the objects he or she is using and there are no mechanisms provided to eliminate unwanted interleaving without changing the implementation of the object (i.e. there are no means for client-side synchronization). There are ways to circumvent this, such as batch messages [25], but they do not solve the problem in the case where there are data dependencies between the different messages (e.g. in our example we need the value of the `query` method to be able to pass it to the `insert` method).

One way to solve this issue in our specific case would be to introduce a “coordination actor” that synchronizes access to the BST. Figure 3 illustrates how we could implement this.

The coordinator implements an asynchronous lock that can be acquired when the lock is available and released otherwise. Using a coordinator like this to guard critical sections has a number of disadvantages:

- Because all operations are asynchronous all of the actors will stay responsive to any message. However, this approach just reintroduces all the issues of traditional locking techniques. For example, similarly to deadlocks, progress can still be lost if different client objects are waiting to acquire a lock on a coordinator locked by the other client.
- Because the coordinator actor is a shared resource as well, asynchronous locking mechanism introduces another level of CPS code (lines 25 and 39).
- Introducing locks like this has the additional overhead of having to use the message passing system to both acquire and release a lock which makes it unsuitable for fine-grained locking.

No parallel reads

The main inefficiency of the actor model with respect to parallel programming is the fact that data cannot be read truly in parallel. This is assuming that we represent shared state as a separate actor. If we want to read (part of) an actor’s state in parallel we have to go through the message passing system and the event-loop of the actor, which will handle each received event sequentially.

In Figure 2, our shared `bst` resource needs to be encapsulated by an actor. This means that all `query` messages will be needlessly sequentialized (in the absence of a `queryInsert` method)

Not only does this make accessing a large data structure from within different components of an application inefficient, it also makes it difficult to implement typical data-parallel algorithms efficiently within the actor model.

```
1  let bst = actor {
2    insert(key, value, client) {
3      result := ...
4      client<-inserted(result);
5    }
6    query(key, client) {
7      result := ...
8      client<-queried(result);
9    }
10 }
11
12 let coordinator = actor {
13   acquire(client) {
14     ...
15     client<-aqcuired();
16     ...
17   }
18   release() {
19     ...
20   }
21 }
22
23 let plugin1 = actor {
24   insert(coordinator, bst, key, value) {
25     coordinator<-acquire(object {
26       aqcuired() {
27         bst<-insert(key, value, object {
28           inserted(ignore) {
29             coordinator<-release();
30           }
31         });
32       }
33     });
34   }
35 }
36
37 let plugin2 = actor {
38   queryInsert(coordinator, bst, key) {
39     coordinator<-acquire(object {
40       aqcuired() {
41         bst<-query(key, object {
42           queried(value) {
43             if(value > 0) {
44               bst<-insert(key, value - 1, object {
45                 inserted(ignore) {
46                   coordinator<-release();
47                 }
48               });
49             } else {
50               coordinator<-release();
51             }
52           }
53         });
54       }
55     });
56   }
57 }
```

Figure 3. Synchronizing access to the BST

2.5 Our approach

Ideally we would want a third option in which we represent shared state as objects that do not belong to any particular actor but rather to a separate entity on which multiple actors can have synchronous access in a controlled way. This way we avoid all the issues with replicating state and also avoid all the issues that come with asynchronously communicating with that shared state.

3. The solution: Domains and views

Our approach allows the programmer to bundle any number of objects in the shared state as a domain. A domain does not belong to a specific actor but is rather a separate entity on which actors can have synchronous access. This synchronous access is important as most of the problems we identified are caused by the use of asynchronous communication to access the shared state. In our approach this synchronous access is represented by a "view". Views are a synchronization mechanism that allows one or more actors to have synchronous access to a shared domain for the duration of one event-loop event. There are two kinds of views, a shared and an exclusive view which mimic multiple reader, single writer access as a synchronization strategy.

As we discussed in section 2.4 an actor is a combination of an object heap and an event loop. The `actor{<expression>}` syntax creates a new event-loop and an object heap initialized with a single object initialized with `<expression>`. Evaluating the actor expression will result in a remote reference to that object. Similarly, a domain is just a container for a number of objects. An actor can never have a direct reference to a domain as a whole. Rather it can have references to objects inside that domain. From now on we will refer to these kinds of references as *domain references*. The `domain{<expression>}` syntax will create a new domain and initialize that domain's object heap with one object initialized by `<expression>`. Evaluating the domain expression will result in a domain reference to that object.

SHACL has a number of primitives to **asynchronously** request access rights to a particular domain using a domain reference. Once the corresponding domain becomes available for shared or exclusive access, an event is queued in the event-loop of the requesting actor. During that event, that actor has a window to synchronously access any object encapsulated by that domain using a domain reference.

Figure 4 gives a quick illustration of the usage of domains and views. Note that the `bst` actor of figure 2 has been replaced by a `domain`. As we saw in the previous section, the `domain` syntax on line 1 will create a new domain with a single object that implements two methods, `insert` and `query`. The return value of the domain syntax is always a domain reference to that object. This means that in our example the `bst` variable will contain a domain reference. Any object created by an expression nested inside the `domain` syntax cannot have access to variables that outside of the scope of that `domain`. The domain reference contained in the variable can be arbitrarily passed around between actors but can only be dereferenced when obtaining a view.

In Figure 4 sending a `queryInsert` message to plugin 2 will first asynchronously request an exclusive view on the bst domain reference (line 21). Once the corresponding domain becomes available for exclusive access, an event is scheduled in the event queue of the plugin 2 actor which will evaluate the block of code provided to the `whenExclusive` primitive (lines 22–25). Note that this block of code is executed by the actor that created it (`plugin2`) and it has access to all lexically available variables such as `bst` and `key`. The plugin 2 actor can synchronously access the BST within that block of code. It can synchronously query it for a certain key and then synchronously update that key-value pair depending on the result of that operation. The same holds if we want to read the same value multiple times, read and/or update different values, etc. Additionally, during the event on which we acquired the view the actor code no longer has to be written in CPS to read and/or write values from and to our shared resource, we can synchronize different messages to the same resource and in the case of a shared view we can even parallelize reads to that resource.

```
1  let bst = domain {
2    insert(key, value) {
3      ...
4    }
5    query(key) {
6      ...
7    }
8  }
9
10 let plugin1 = actor {
11   insert(bst, key, value) {
12     whenExclusive(bst) {
13       bst.insert(key, value);
14     }
15   }
16 }
17
18
19 let plugin2 = actor {
20   queryInsert(bst, key) {
21     whenExclusive(bst) {
22       value := bst.query(key);
23       if(value > 0) {
24         bst.insert(key, value - 1)
25       }
26     }
27   }
28 }
```

Figure 4. Illustration of domains and views

3.1 View primitives

In this section we will only consider view primitives that acquire a view on a single domain at a time. For SHACL this set of primitives was extended to also allow acquiring shared and/or exclusive views on a set of domains (See section 4). SHACL supports the following primitives for requesting views on a domain reference:

$$\text{whenShared}(e)\{e'\}$$
$$\text{whenExclusive}(e)\{e'\}$$

Here, e is a valid SHACL expression that evaluates to a domain reference and e' is any valid SHACL expression. Note that these primitives are asynchronous operations, they will schedule a view-request and immediately return. After the request is scheduled, the event-loop of the actor can resume processing other events in its event-queue. Once the domain becomes available two things happen. First the domain is locked for exclusive or shared access. Then an event that is responsible for evaluating the expression e' is put in the event-queue of the corresponding actor. Once that event is processed the domain is freed again, allowing other actors to access it.

Figure 5 illustrates how views are created. Both actor `A` and actor `B` have a reference to the shared object. If they want to access this shared object, first they need to request a view on that object. Attempting to access a domain reference outside of a view results in an error. Once the request is handled by the domain, any reference to an object inside that domain becomes synchronously available for the duration of one event. When e' is evaluated, the actor loses its access rights to that domain. A shared view allows the actor to synchronously invoke read-only methods of all the objects within the corresponding domain. Any attempt to write a field of a domain object during a shared view will result in an error. An exclusive view allows the actor to synchronously invoke any

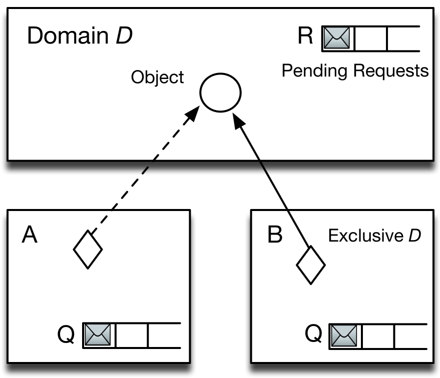


Figure 5. Actor A and B share a reference to an object inside domain D on which Actor B has an exclusive view.

method on objects inside the corresponding domain, regardless of whether they change the state of the object(s) inside that domain.

3.2 Semantic properties

In this section we will evaluate and discuss our approach with regard to the original actor model. The following topics will be discussed: deadlock freedom, race condition freedom and macro-step semantics.

3.2.1 Deadlock freedom

The absence of deadlocks and low-level race conditions are the two properties of the actor model that differentiate it from lower-level models and provide the required guarantees to build large concurrent applications in a sustainable manner. To maintain deadlock-freedom, the following two restrictions are enforced for views:

View requests are non-blocking. All primitives that request views are non-blocking. As explained in Section 3.1, the request for a view is scheduled as an asynchronous event which is processed only once the domain becomes available. This implies that all operations in our language terminate, meaning that all events can be processed in a finite operational time (if programmed correctly). And thus, the event for which a lock was acquired will eventually terminate and release the lock again, which is important to ensure that our language remains deadlock free.

A view on a domain only exists for the duration of one event. Events in our model can be considered atomic operations with a finite operational time. This means that any domain that is currently unavailable due to a view will become available at some point in the future.

With those restrictions in place SHACL is guaranteed to be deadlock free. With view requests being non-blocking, and the absence of any other blocking operation in the model, it is guaranteed that *wait-for* cycles can not be constructed with the basic primitives provided.

As discussed in Section 3.1, views are only held for the duration of a single event, and requesting a "nested" view while holding another view is an asynchronous operation. All this supports the notion of an event being executed as an atomic operation with a finite number of operational steps. Barring any sequential infinite loops included by the programmer.

3.2.2 Race condition freedom

To maintain race condition freedom, the following three restrictions are enforced for views:

Only allow view requests on domain references
All our primitives require that the reference on which a view is requested is a domain reference. In contrast with other remote objects, domain objects are not allowed to have direct references to objects contained in another domain. This way, accidentally shared state is avoided, which ensures that no low-level race conditions can occur in our model. Without this restriction, multiple actors could access the same free variable in the lexical scope of a domain object, opening the door for classical data races.

Domain references cannot be accessed outside of a view
Care has to be taken of views which had been previously lexically captured but haven been only available in an asynchronous operation. Since the lexical scope would hold a reference to a domain object that is not protected by a view anymore, race conditions could be introduced. To avoid such data races, any attempt to access a domain object outside of a view throws an error.

Object creation inside a domain.
Any object creation expression lexically nested inside a domain generates objects owned by that domain. Views are acquired on a domain and dereferencing an object owned by that domain can never expose that domain's content. As such, any reference to an object that is owned by a domain is always a domain reference.

These three rules ensures that, in any scenario, any domain reference or lexically nested state is no longer accessible while processing later events without requesting a new view. They also ensure that any concurrent updates of shared state are impossible and thus ensures that we avoid race conditions by construction.

3.2.3 Macro-step semantics

The actor model provides one important property for formal reasoning about different program properties. This property is the macro-step semantics [4].

In an actor model, the granularity of reasoning is a message/event. For the properties of a program, each event is processed in a single atomic step. This leads to a convenient reduction of the overall state-space that has to be regarded in the process of formal reasoning. Furthermore, this property is directly beneficial to application programmers as well in their development process. Programmers can design the semantics of message sends as coarse-grained as appropriate, reducing the potential problematic interactions.

After introducing domains and views, the question is whether the macro-step semantics still holds. Arguable, this is still the case, since the macro-step semantics only requires the atomicity of the evaluation of a message, but does not imply any locality of changes. Thus, changing the state of an object for which a view was obtained does not violate atomicity, since we only allow exclusive views for state modifications. Shared views for reading state are also not violating the semantics since state is not actually changed.

Based on this reasoning, an actor-model with the concept of views presented in Section 3.1 still maintains the macro-step semantics, and thus keeps the main properties of the actor model that are beneficial for formal reasoning intact.

Furthermore, the semantics of all writes in our model, being restricted either to local writes inside an actor, or writes protected by an exclusive view, result in a memory model which enforces sequential consistency [11]. Thus, the original semantics remains preserved and allows the application of relevant reasoning techniques.

3.3 Expressiveness

Since the actor model relies solely on asynchronous event processing to avoid deadlocks, the expressiveness of such a language is typically impaired.

With the mechanisms proposed here, it is however possible to grant synchronous access to domain objects protected by views. Listing 4 introduced the corresponding example and demonstrates how to access a shared resource synchronously, which could not be expressed before. For this specific case, the alternative solution would be to change the interface of the remote object, to be able to request and update its state in a single step. However, that approach is neither always possible, e. g., for third-party code, nor desirable. Also, this solution would not be appropriate for synchronizing updates to different domain objects as ensuring synchronized access in the traditional actor model would require to bundle these objects into one actor. Thus, with views the expressiveness is extended considerably over the standard actor model. Programmers can model their shared state without taking into account how this shared state will be accessed from the client side and the client side can synchronize and compose access to different objects in an arbitrary way.

3.4 Conclusion

In this section we have shown that with the use of views we can avoid the problems discussed in section 2. Firstly, by **not replicating** the shared state we avoid the need to keep replicas consistent. Secondly, from within a view we do **not** need to employ **CPS** to access shared state. If we have synchronous access to the domain object we can directly access it's fields without using the message passing system. Thirdly, we can safely build more coarse-grained synchronization boundaries by combining messages to objects within the same domain in an arbitrary way during the event in which we acquired the view. Lastly, if we only use shared views on a resource we can **read** from that resource **in parallel**.

4. SHACL further features

Section 3 discussed only the core features of SHACL. In this section we will discuss a number of other important features of SHACL.

4.1 Views on multiple domains

Currently, SHACL only supports shared and exclusive views, which mimic single writer, multiple reader locking. This means that it is impossible to do parallel updates of objects a single domain. A workaround for this problem would be to subdivide the shared data structure into several domains. We could for example put each node of the binary search tree of our example in a separate domain. This however also means that any parallel updates to that data structure need to be synchronized by the program. SHACL has a primitive that allows the programmer to synchronize access to multiple domains:

$$\text{when}(e, e')\{e''\}$$

The `when` primitive takes any 2 SHACL expressions e and e' that evaluate to two arrays of domain references. The first array has to contain all the domain references for which the programmer wants to have shared access and similarly the second array has to contain all the domain references for which the programmer wants to have exclusive access. e'' is the expression that will be scheduled as an event in the event-loop of the executing actor once all the necessary resources become available.

In SHACL there is a global ordering in which all domains are locked for shared and/or exclusive access. This is to prevent deadlocks when views are requested on multiple domains.

4.2 Futures

In section 1 we already mentioned that SHACL supports future-type messages. Futures introduce a synchronization mechanism for actors to synchronize on the reception of a message without using callbacks. Traditional asynchronous messages have no return value. A developer needs to work around this lack of return values by means of an explicit customer object as seen in all the examples throughout the paper. Future type messages allow the programmer to hide this explicit callback parameter.

In contrast to regular asynchronous messages, a future-type message does have a return value. It returns a future-value that represents the "eventual" return value of the message that was sent. The developer can then register an observer with that future-value using a special `whenBecomes` primitive. When the original message is processed by the receiving actor, the future is "resolved" with the return value of that message and any registered observer is notified. A notified observer triggers an event that is scheduled in the event-loop of the actor that executed the `whenBecomes` primitive.

The following example illustrates the usage of futures:

```
let cell = object {
  c := 0;
  get() {
    c;
  }
  set(n) {
    this.c := n;
  }
}

let a = actor {
  increase(counter) {
        future := counter<-get();
    whenBecomes(future -> c) {
      counter<-set(c + 1);
    }
  }
}

a<-increase(cell);
```

Figure 6. Illustration of futures

In Figure 6 `get` is sent as a future-type message to the remote reference counter and immediately returns a future-value. An event is registered with that future that is responsible for updating the counter by sending it a regular asynchronous `set` message. Notice that using futures does not solve the issues discussed in section 2. We still need to employ CPS if we want to access several values of our remote object, event-level data races can still occur and reads are not parallelized.

The reason that futures are interesting for our model is because they work well together with domains and views. In fact, a view request in SHACL returns a future-value on which can be synchronized. If part of our computation depends on the atomic update of a shared resource but does not necessarily require synchronous access to that resource, these futures can be used to schedule code that can be executed after the view was released.

There is also a mechanism in SHACL to group futures into a single future (namely the primitive `group`). This mechanism can be used in conjunction with future-type messages to branch work to other actors and then synchronize on all of them. Or it can be used in conjunction with domains and views to schedule a number of atomic updates and then synchronize on the completion of all of them.

5. Related work

The engineering benefits of semantically coarse-grained synchronization mechanisms in general [10] and the restrictions of the actor model [16] have been recognized by others. In particular the notion of domains and *view*-like constructs has been proposed before.

Demsky and Lam [10] propose views as a coarse-grained locking mechanism for concurrent Java objects. Their approach is based on static view definitions from which at compile time the correct locking strategy is derived. Furthermore, their compiler detects a number of problems during compilation which can aid the developer to refine the static view definitions. For instance they detect when a developer violates the view semantics by acquiring a read view but writing to a field. The main distinction between our and their approach comes from the different underlying concurrency models. Since Demsky and Lam start from a shared-memory model, they have to tackle many problems that do not exist in the actor model. This results in a more complex solution with weaker overall guarantees than what our approach provides. First of all, accessing shared state without the use of views is not prohibited by the compiler thereby compromising any general assumptions about thread safety. Secondly, the programmer is required to manually list all the incompatibilities between the different views. While the compiler does check for inconsistencies when acquiring views, it does not automatically check if different views are incompatible. Forgetting to list an incompatibility between different views again compromises thread safety. Thirdly, acquiring a view is a blocking statement and nested views are allowed, possibly leading to deadlocks. They do recognize this problem and partially solve this by allowing simultaneously acquiring different views to avoid this issue. But avoiding the acquiring of nested views is not enforced by the compiler. Finally, their approach does not support a dynamic notion of a view which could be used to safely access shared state depending on runtime information.

Hoffman et al. [14] show the need for programs to isolate state between different subcomponents of an application. They propose protection domains and ribbons as an extension to Java. Similarly to our approach, protection domains dynamically limit access to shared state from different executing threads. Access rights are defined with ribbons where different threads are grouped into. While their approach is very similar to ours, they started from a model with less restrictions (threads) and built on top of that while we started from the actor model which already has the necessary isolation of processes by default. While access modifiers on protection domains do limit the number of critical operations in which race conditions need to be considered. If two threads have write access to the same data structure, access to that data structure still needs to be synchronized.

Axum [17] is an actor based language that also introduced the concept of domains for state sharing. Similarly to our approach single writer, multiple reader access is provided to domains. Access patterns in Axum have to be statically written down, which does give some static guarantees about the program but ultimately suffers from the same problems as the views abstractions from Demsky and Lam. Although the Axum project was concluded it also showed that there is an interest in a high level concurrency model that allows structuring interactive and independent components of an application.

ProActive [7] is middleware for Java that provides an actor abstraction on top of threads. It provides the notion of *Coordination actors* to avoid race conditions similar to views. However, the overall reasoning about thread safety is hampered since its use is not enforced. Furthermore, coordination actors are proxy objects that sequentialize access to a shared resource, and thus, are not able to support parallel reads, one of the main issues tackled with our approach. In addition, it is neither possible to add synchronization constraints on batches of messages, nor is deadlock-freedom guaranteed, since accessing a shared resource through a proxy is a blocking operation.

In Deterministic Parallel Java [8] the programmer has to use effect annotations to determine what parts (*regions*) of the heap a certain method accesses. They ensure race condition free programs by only allowing nested calls to write disjoint sub-regions of that region. This means that this approach is best suited for algorithms that employ a divide and conquer strategy. In our approach we want a solution that is applicable to a wider range of problems including algorithms that randomly access data from different regions.

Parallel Actor Monitors [19] (PAM) is a related approach to enable parallelism inside a single actor by evaluating different messages in the message queue of an actor in parallel. The difference with our approach is that the actor that owns the shared datastructure is still the only one that has synchronous access on that resource. In our approach we apply an inversion of control where the user of the shared resource has exclusive access instead of the owner. This inversion of control allows an actor in SHACL to synchronize access to multiple resources which is not possible in the case of PAM.

6. Conclusion

The Actor Model is a good model for concurrent programming, it provides a number of safety guarantees for issues that are often problematic in other models (Deadlock freedom, data-race freedom, macro-step semantics). Unfortunately the restrictions on this model often limit the expressiveness of the model in comparison with less strict implementations, limiting its adoptability as a mainstream programming model. The issue of accessing shared state is one shared between all actor languages. Others solve this issue by allowing the programmer to break actor boundaries as an escape hatch (e.g. Scala). In this case, the programmer has to rely on traditional locking mechanisms to synchronize access to that state, reintroducing all problems that come with locks. Others combine several concurrency models to solve this issue. For example, Clojure [13] both implements actor based concurrency primitives as well as a Software Transactional Memory. In our approach we tried to tailor our solution specifically for the Actor Model ensuring maximum interoperability between the different primitives. The advantages of our model over the traditional event-loop model are threefold. Firstly we avoid the continuation passing style of programming when accessing shared state. Secondlyndly we allow the programmer to introduce extra synchronization constraints on groups of messages and lastly we are able to model true parallel reads.

7. Acknowledgements

Joeri De Koster is supported by a doctoral scholarship granted by the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen), Belgium.

Tom Van Cutsem is a Postdoctoral Fellow of the Research Foundation, Flanders (FWO)

References

[1] Akka. `http://akka.io/`.

[2] Asyncobjects framework. `http://asyncobjects.sourceforge.net/`.

[3] G. Agha. Actors: a model of concurrent computation in distributed systems. *AITR-844*, 1985.

[4] G. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7(1):1–72, 1997.

[5] J. Armstrong, R. Virding, C. Wikstrom, and M. Williams. Concurrent programming in erlang. 1996.

[6] M. Astley. The actor foundry: A java-based actor programming environment. *University of Illinois at Urbana-Champaign: Open Systems Laboratory*, 1998.

[7] L. Baduel, F. Baude, D. Caromel, A. Contes, F. Huet, M. Morel, and R. Quilici. *Grid Computing: Software Environments and Tools*, chapter Programming, Deploying, Composing, for the Grid. Springer-Verlag, January 2006.

[8] R. Bocchino Jr, V. Adve, D. Dig, S. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A type and effect system for deterministic parallel Java. *ACM SIGPLAN Notices*, 44(10):97–116, 2009.

[9] E. Brewer. Towards robust distributed systems. In *Proceedings of the Annual ACM Symposium on Principles of Distributed Computing*, volume 19, pages 7–10, 2000.

[10] B. Demsky and P. Lam. Views: Object-inspired concurrency control. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 395–404. ACM, 2010.

[11] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. *ACM SIGARCH Computer Architecture News*, 18:15–26, May 1990. ISSN 0163-5964.

[12] P. Haller and M. Odersky. Scala actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410(2-3): 202–220, 2009.

[13] S. Halloway. *Programming Clojure*. Pragmatic Bookshelf, 2009.

[14] K. Hoffman, H. Metzger, and P. Eugster. Ribbons: a partially shared memory programming model. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, pages 289–306. ACM, 2011.

[15] G. Hohpe. Programming Without a Call Stack–Event-driven Architectures. *Objekt Spektrum*, 2006.

[16] R. Karmani, A. Shali, and G. Agha. Actor frameworks for the jvm platform: A comparative analysis. In *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, pages 11–20. ACM, 2009.

[17] Microsoft Corporation. Axum programming language. `http://tinyurl.com/r5e558`.

[18] M. S. Miller, E. D. Tribble, J. Shapiro, and H. P. Laboratories. Concurrency among strangers: Programming in e as plan coordination. In *In Trustworthy Global Computing, International Symposium, TGC 2005*, pages 195–229. Springer, 2005.

[19] C. Scholliers, É. Tanter, and W. De Meuter. Parallel actor monitors. Technical report, 2010. vub-tr-soft-10-05.

[20] S. Srinivasan and A. Mycroft. Kilim: Isolation-typed actors for java. *ECOOP 2008–Object-Oriented Programming*, pages 104–128, 2008.

[21] H. Sutter. Welcome to the jungle. http://herbsutter.com/welcome-to-the-jungle/, 2011.

[22] D. Ungar and R. Smith. *Self: The power of simplicity*, volume 22. ACM, 1987.

[23] T. Van Cutsem, S. Mostinckx, E. Gonzalez Boix, J. Dedecker, and W. De Meuter. Ambienttalk: Object-oriented event-driven programming in mobile ad hoc networks. In *Chilean Society of Computer Science, 2007. SCCC'07. XXVI International Conference of the*, pages 3–12. Ieee, 2007.

[24] C. Varela and G. Agha. Programming dynamically reconfigurable open systems with salsa. *ACM SIGPLAN Notices*, 36(12):20–34, 2001.

[25] A. Yonezawa, J. Briot, and E. Shibayama. Object-oriented concurrent programming ABCL/1. *ACM SIGPLAN Notices*, 21(11):268, 1986.

Soter: An Automatic Safety Verifier for Erlang

Emanuele D'Osualdo
University of Oxford
emanuele.dosualdo@cs.ox.ac.uk

Jonathan Kochems
University of Oxford
jonathan.kochems@cs.ox.ac.uk

C.-H. Luke Ong
University of Oxford
luke.ong@cs.ox.ac.uk

Abstract

This paper presents Soter, a fully-automatic program analyser and verifier for Erlang modules. The fragment of Erlang accepted by Soter includes the higher-order functional constructs and all the key features of *actor concurrency*, namely, dynamic and possibly unbounded spawning of processes and asynchronous message passing. Soter uses a combination of static analysis and infinite-state model checking to verify safety properties specified by the user. Given an Erlang module and a set of properties, Soter first extracts an abstract (approximate but sound) model in the form of an *actor communicating system* (ACS), and then checks if the properties are satisfied using a Petri net coverability checker, BFC. To our knowledge, Soter is the first fully-automatic, infinite-state model checker for a large fragment of Erlang. We find that in practice our abstraction technique is accurate enough to verify an interesting range of safety properties such as mutual-exclusion and boundedness of mailboxes. Though the ACS coverability problem is EXPSPACE-complete, Soter can analyse these problems surprisingly efficiently.

1. Introduction

This paper presents Soter, a tool that automatically verifies safety properties of concurrent Erlang programs, based on the framework of [4]. Erlang is an open-sourced language with support for higher-order functions, concurrency, communication, distribution, fault tolerance, on-the-fly code reloading and multiple platforms [2]. The sequential part of Erlang is a higher order, dynamically typed, call-by-value functional language with pattern-matching algebraic data types. Following the *actor model* [7], a concurrent Erlang computation consists of a dynamic network of processes that communicate by asynchronous message passing. Each process has a unique process identifier (pid), and is equipped with an unbounded mailbox. Message send is non-blocking. Retrieval of messages from the mailbox is not FIFO but First-In-First-Firable-Out (FIFFO) via pattern-matching. A process may block while waiting for a message that matches a certain pattern to arrive in its mailbox. Thanks to a highly efficient runtime system, Erlang is a natural fit for programming multicore CPUs, networked servers, distributed databases, GUIs, and monitoring, control and testing tools. For an introduction to Erlang, see Armstrong's *CACM* article [1].

Safety Verification by Static Analysis and Model Checking

The challenge of verifying Erlang programs is that one must reason about the asynchronous communication of an unbounded set of messages, across an unbounded set of Turing-powerful, higher-order processes. The inherent complexity of the verification task can be seen from several diverse sources of infinity in the state space.

(∞ 1) Function definitions are not necessarily tail-recursive, so a call-stack is needed.
(∞ 2) Higher-order functions are first-class values; closures can be passed as parameters or returned.
(∞ 3) Data domains, and hence the message space, are unbounded: functions may return, and variables may be bound to, terms of an arbitrary size.
(∞ 4) An unbounded number of processes can be spawned dynamically.
(∞ 5) Mailboxes have unbounded capacity.

This motivates our model checking approach: we automatically extract an abstract model that simulates the semantics of the program by construction, then we use decision procedures on the abstract model to prove safety properties.

Our abstract model, called *Actor Communicating System*, is highly expressive: it can model dynamic spawning and unbounded mailboxes. An ACS is defined by a finite set of rules but it is infinite-state i.e. its dynamic semantics includes traces that go through infinitely many different configurations. It follows that one cannot establish reachability by exploring all the possible runs. However ACS are equivalent to Petri nets for which model-checking algorithms do exist. Our tool uses a Petri net coverability checker called BFC [8]. ACS models are described in Section 2.

Overview of Soter

Soter is an experimental, prototype Haskell implementation of the framework of [4]. It accepts a (concurrent) subset of the Erlang language: supported features include algebraic data-types with pattern-matching, higher-order, spawning of new processes, asynchronous communication. See Section 4 for the Erlang constructs that are not currently supported by Soter.

As presented in Figure 1, Soter's workflow has three phases.

In phase 1, the input Erlang module with correctness annotations is compiled using the standard Erlang compiler `erlc` to a module of *Core Erlang* — the official intermediate representation of Erlang. The code is then normalised in preparation for the

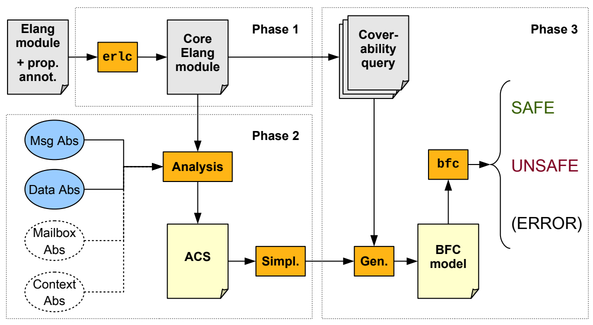
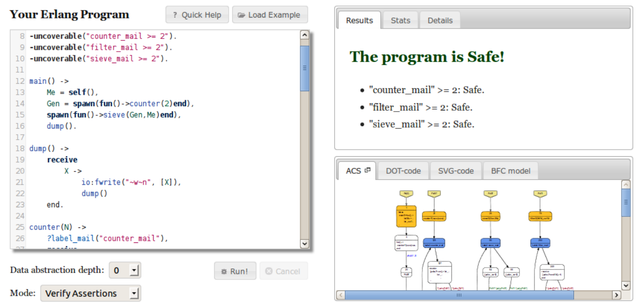


Figure 1. Workflow of Soter



```
-uncoverable("counter_mail >= 2").
-uncoverable("filter_mail >= 2").
-uncoverable("sieve_mail >= 2").

main() ->
    Me = self(),
    Gen = spawn(fun()->counter(2)end),
    spawn(fun()->sieve(Gen,Me)end),
    dump().

dump() ->
    receive
        X ->
            io:fwrite("~w~n", [X]),
            dump()
    end.

counter(N) ->
    ?label_mail("counter_mail"),
```

The program is Safe!

- "counter_mail" >= 2: Safe.
- "filter_mail" >= 2: Safe.
- "sieve_mail" >= 2: Safe.

Data abstraction depth: 0

Mode: Verify Assertions

Figure 2. Screenshot of Soter's web interface

next phase. Correctness properties, expressible in various forms, are specified by annotating the program source. The user can insert assertions, or label program points and mailboxes and then state constraints they must satisfy.

The main purpose of phase 2 is to soundly abstract sources of infinity (∞ 1), (∞ 2) and (∞ 3). This is done as follows. A control-flow based analysis is performed on the program, yielding a control flow graph on which we bootstrap the generation of the ACS rules. The analysis is parametric in D and M, the depth of the data and message abstraction respectively. We abstract data by truncating terms at the specified depth. By default D is set to zero, so calls to the same function with different arguments are merged in the abstract model; the runtime of the analysis is exponential in D. M is by default set to $D + P$ where P is the maximum depth of the receive patterns of the program; using large values for M does not incur the same slowdown as adjusting D. In future releases, we plan to introduce parameters to tune the precision of the abstraction so that users can control the context sensitivity of the analysis.

In phase 3, Soter generates a Petri net in the format of BFC [8], which is a fast coverability checker for Petri nets with transfer arcs, developed by Alexander Kaiser. For each property Soter needs to prove, BFC is called internally with the BFC model and a query representing the safety property as input.

Soter can be run in three modes: "analysis only" which produces the ACS, skipping phase 3; "verify assertions" which extracts the properties from user annotations; "verify absence-of-errors" which generates BFC queries asserting the absence of runtime exceptions. Currently not all the exceptions that the Erlang runtime can throw are represented; the supported ones include sending a message to a non-pid value, applying a function with the wrong arity, and spawning a non-functional value. A notable omission is pattern-matching failures which will be supported by the next release of Soter.

Soter is a practical implementation of a highly complex procedure. Phases 1 and 2 are polytime in the size of the input program [4]. Despite the EXPSPACE-completeness of the Petri net coverability problem, phase 3 is surprisingly efficient; see the outcome of the experiments in Table 1.

In addition to a command-line interface, we have built a web interface for Soter at http://mjolnir.cs.ox.ac.uk/soter/. The user interface allows easy input of Erlang programs. A library of annotated example programs is available to be tried and modified. Soter presents the generated abstract model as a labelled graph for easy visualisation, and reports in detail on the performance and results of the verification. A screenshot of the web interface is shown in Figure 2.

Related Work There are a few popular bug-finding tools for Erlang, notably Dyalizer [3, 9] which implements a variety of static analyses. McErlang [5] is a finite-state on-the-fly model-checker

```
main() -> Me = self(),
          Gen = spawn(fun()->counter(2)end),
          spawn(fun()->sieve(Gen,Me)end),
          dump().

dump() -> receive X -> io:write(X), dump() end.

counter(N) ->
    ?label_mail("counter_mail"),
    receive {poke, From} ->
        From!{ans, N}, counter(N+1)
    end.

sieve(In, Out) ->
    ?label_mail("sieve_mail"),
    In!{poke, self()},
    receive {ans,X} ->
        Out!X,
        F = spawn(fun()->
            filter(divisible_by(X), In)
          end),
        sieve(F,Out)
    end.

filter(Test, In) ->
    ?label_mail("filter_mail"),
    receive {poke, From} ->
        filter(Test, In, From)
    end.

filter(Test, In, Out) ->
    In!{poke, self()},
    receive {ans,Y} ->
        case Test(Y) of
            false -> Out!{ans,Y}, filter(Test, In);
            true  -> filter(Test, In, Out)
        end
    end.

-ifdef(SOTER).
    divisible_by(X) ->
        fun(Y) -> ?any_bool() end.
-else.
    divisible_by(X) ->
        fun(Y) -> case Y rem X of
                      0 -> true;
                      _ -> false
                  end
        end.
-endif.
```

Figure 3. Eratosthenes' Sieve, actor style

for Büchi properties and EtomCRL2 [6] translates Erlang programs to μCRL which allows verification. Soter instead operates on the semantics of Erlang directly and model-checks an infinite-state transition system.

2. Actor Communicating Systems

The abstract model we extract from the input Erlang program is an *Actor Communicating System* (ACS), which models the interaction of an unbounded set of communicating processes. An ACS has a finite set Q of *control states*, a finite set P of *pid classes*, a finite set M of message kinds and a finite set of rules. An ACS rule has the shape $\iota\colon q \xrightarrow{\ell} q'$ which means that a process of pid class ι can transition from state q to state q' with (possible) *communication side effect* ℓ, of which there are four kinds:

(i) the process makes an internal transition,
(ii) it extracts and reads a message m from its mailbox,
(iii) it sends a message m to a process of pid class ι' and
(iv) it spawns a process of pid class ι'.

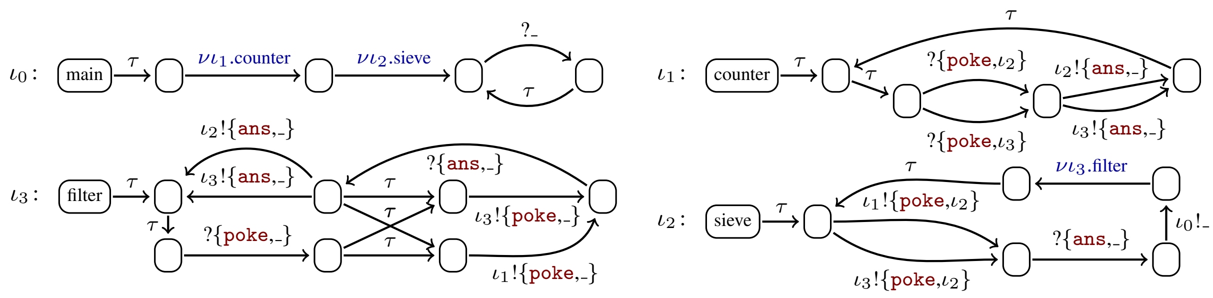


Figure 4. The ACS graph generated by Soter from the sieve example. The ι_0 component represents the starting process which sets up the counter agent (ι_1) and the sieve agent (ι_2) and then becomes the dump agent. During its execution, the sieve agent spawns new filter agents, all represented by the ι_3 component.

To help user visualise the model, we present an ACS as a collection of label graphs (called components), each showing a pid-class, where the vertices are the control states and the labelled edges are the rules. An example of such a graphical presentation is shown in Figure 4.

The semantics of ACS is as follows. Each control state is a counter holding tokens; when a $\iota\colon q \xrightarrow{\tau} q'$ rule is executed a token is extracted from q and transferred to q'; if q contains no tokens the rule is not enabled. Spawn rules insert a new token in the control state of the process to be created while making the transition. Message passing is dealt with analogously: for each component there is a counter for each message; these counters keep track of the number of messages that have been sent to that component so far, thus merging all the mailboxes of the processes of the component. When a message is sent, a token is inserted in the relevant counter. A receive rule can fire only when the counter for the message to be extracted contains at least one token; when fired, a token gets consumed. Note that the order of arrival of messages is not recorded.

An ACS can be interpreted naturally as a *vector addition system* (VAS), or equivalently Petri net. Recall that a VAS of dimension n is given by a set of n-long vectors of integers regarded as transition rules. A VAS defines a state transition graph whose states are just n-long vectors of non-negative integers. There is a transition from state $\mathbf{v}$ to state $\mathbf{v}'$ just if $\mathbf{v}' = \mathbf{v}+\mathbf{r}$ for some transition rule $\mathbf{r}$. In this paper, we are concerned with the EXPSPACE-complete decision problem *Coverability* [10]: given a VAS, a start vector s and a target non-negative vector t of the same dimension, is it possible to reach some v that covers t (i.e. $v \geq t$)? Note that *LTL Model Checking* is also EXPSPACE-complete for VAS; *Reachability* is decidable but its complexity is open.

A wide range of properties can be encoded as coverability queries on the ACS. Examples include reachability of error states, mutual exclusion, bounds on the number of enqueued messages in a mailbox. Some of these correctness properties can be exploited by optimising compilers. Bounds on mailboxes of a class of processes, for example, allow the compiler to allocate a fixed number of cells for that mailbox, resulting in programs that can be efficiently garbage-collected.

Liveness properties such as deadlock freedom cannot currently be checked by Soter because there are no efficient implementations of LTL model checking for Petri nets. Should such implementations become available, Soter can quickly take advantage of them.

3. Demo: A Concurrent Eratosthene's Sieve

We illustrate the workings of Soter by an example. Figure 3 shows an implementation of Eratosthenes' sieve inspired by a NewSqueak program by Rob Pike.[1] The actor defined by `counter` provides the sequence of natural numbers as responses to `poke` messages, starting from 2; the `dump` actor prints everything it receives. The `sieve` actor's goal is to send all prime numbers in sequence as messages to the `dump` actor; to do so it pokes its current `In` actor waiting for a prime number. After forwarding the received prime number, it creates (`spawn`) a new `filter` process, which becomes its new `In` actor. The filter actor, when poked, queries its `In` actor until a number satisfying `Test` is received and then it forwards it; the test (an higher-order parameter) is initialized by `sieve` to be a divisibility check that tests if the received number is divisible by the last prime produced. The overall effect is a growing chain of `filter` actors each filtering multiples of the primes produced so far; at one end of the chain there is the counter, at the other the sieve that forwards the results to `dump`.

Since Soter does not have native support for arithmetic operations, line 41 defines a stub to be used by Soter that returns true or false non-deterministically, thus soundly approximating the real definition based on division given in line 44.

The communication here is synchronous in spirit: whenever a message is sent, the sender actor blocks waiting for a reply. To check this is the case, we can verify the property that every mailbox contains in fact at most one message at any time. To be able to express this constraint we label the mailboxes we are interested in with the `?label_mail()` macro: the instructions in lines 9, 15 and 26 mark the mailbox of any process that may execute them with the corresponding label.

Then we can insert the following lines at the beginning of the module

```
-uncoverable("counter_mail >= 2").
-uncoverable("filter_mail >= 2").
-uncoverable("sieve_mail >= 2").
```

which state the property we want to prove. The `-uncoverable` directive is ignored by the Erlang compiler but it is interpreted by Soter as a property to be proved: all the states satisfying the constraint are considered to be "bad states". These inequalities state that if the total number of messages in the labelled mailboxes exceed the given bound, we are in a bad state.

Soter allows user-defined labels for program locations as well with the macro `?label()`; the inequalities in this case state that the total number of processes executing the labelled instruction at the same time must be less that the given bound.

When executed on the code in Figure 3, Soter will compute the ACS in Figure 4; its semantics is a sound approximation of the actual semantics of the program. A VAS description of it,

[1] see "Concurrency/message passing Newsqueak", http://video.google.com/videoplay?docid=810232012617965344

Example	LOC	PRP	SAFE?	ABSTR		ACS SIZE		TIME			
				D	M	Places	Ratio	Analysis	Simpl	BFC	Total
reslockbeh	507	1	yes	0	2	40	4%	1.94	0.41	0.85	3.21
reslock	356	1	yes	0	2	40	10%	0.56	0.08	0.82	1.48
sieve	230	3	yes	0	2	47	19%	0.26	0.03	2.46	2.76
concdb	321	1	yes	0	2	67	12%	1.10	0.16	5.19	6.46
state_factory	295	2	yes	0	1	22	4%	0.59	0.13	0.02	0.75
pipe	173	1	yes	0	0	18	8%	0.15	0.03	0.00	0.18
ring	211	1	yes	0	2	36	9%	0.55	0.07	0.25	0.88
parikh	101	1	yes	0	2	42	41%	0.05	0.01	0.07	0.13
unsafe_send	49	1	no	0	1	10	38%	0.02	0.00	0.00	0.02
safe_send	82	1	no*	0	1	33	36%	0.05	0.01	0.00	0.06
safe_send	82	4	yes	1	2	82	34%	0.23	0.03	0.06	0.32
firewall	236	1	no*	0	2	35	10%	0.36	0.05	0.02	0.44
firewall	236	1	yes	1	3	74	10%	2.38	0.30	0.00	2.69
finite_leader	555	1	no*	0	2	56	20%	0.35	0.03	0.01	0.40
finite_leader	555	1	yes	1	3	97	23%	0.75	0.07	0.86	1.70
stutter	115	1	no*	0	0	15	19%	0.04	0.00	0.00	0.05
howait	187	1	no*	0	2	29	14%	0.19	0.02	0.00	0.22

Table 1. Soter Benchmarks. The number of lines of code refers to the compiled Core Erlang. The PRP column indicates the number of properties which need to be proved. The columns D and M indicate the data and message abstraction depth respectively. In the "Safe?" column, "no*" means that the program satisfies the properties but the verification was inconclusive; "no" means that the program is not safe and Soter finds a genuine counterexample. "Places" is the number of places of the underlying Petri net after the simplification; "Ratio" is the ratio of the number of places of the generated Petri net before and after the simplification. All times are in seconds.

incorporating the property, is then generated and fed to BFC to check for the uncoverability of bad states; in this instance BFC is successful in proving the program safe.

4. Experiments, Limitations and Extensions

Evaluation In Table 1 we summarise our experimental results. Soter is a fully automatic tool. All our example programs are higher-order and use dynamic (and unbounded) process creation and non-trivial synchronisation. The properties checked fall into three groups: mutual exclusion, unreachability of error states, and bounds on mailboxes. The annotated example programs in Table 1 can all be viewed and verified using Soter at the web interface `http://mjolnir.cs.ox.ac.uk/soter/`. As indicated by the experimental outcome, the abstractions employed by Soter are sufficiently precise to prove safety for a wide variety of examples. We observe that the ACS simplification is especially effective in reducing the problem size. BFC implements an algorithm that is EXPSPACE-hard in the ACS size. The experiments show there are other factors such as transition structure that strongly influence the runtime complexity of BFC, although it is not yet clear what these parameters are. In conclusion, despite the worst-case exponential complexity of the underlying algorithm, Soter is surprisingly efficient. We believe that the experimental outcome justifies further development of the tool.

Limitations Features of Erlang currently unsupported by Soter can be organised into three groups: (i) constructs such exceptions, arithmetic primitives, built-in data types and the module system are not difficult to integrate into the current framework; (ii) features such as time-outs in receives, registered processes, input-output and type guards could be supported by providing specific abstractions; (iii) the monitor / link primitives and the multi-node semantics. These features need to be supported explicitly by the abstract model for them to be usefully approximated, and may require a major extension of the theory. How to extend our framework to explicitly and precisely model the last two groups of features is an interesting research problem. Despite these limitations, it is usually possible to adapt existing programs so that they are accepted by Soter: often it is sufficient to provide "dummy" implementations of the unsupported functions as it has been done in line 41 of the example code in Figure 3.

In addition certain problem instances remain out of Soter's scope: if the proof of safety for a program requires accurate modelling of the stack or precise sequencing information on the arrival order of messages, then our abstractions are not suitable; further our analysis assumes a closed program — the ability to analyse and model-check open programs would enable a compositional approach which we expect would enhance Soter's scalability.

Extensions and Future Directions We plan the following extensions: (i) handle arbitrary Core Erlang programs (ii) formalise and implement specific abstractions for time-outs and I/O (iii) develop fine-tuned, flexible and refineable abstractions for data, mailboxes and context-sensitivity, which would facilitate the construction of a CEGAR loop (iv) exploit the decidability of LTL-properties for ACS to enable Soter to prove liveness and other path properties.

References

[1] J. Armstrong. Erlang. *CACM*, 53(9):68, 2010.

[2] Francesca Cesarini and Simon Thompson. *Erlang Programming - A Concurrent Approach to Software Development*. O'Reilly, 2009.

[3] M. Christakis and K. Sagonas. Detection of asynchronous message passing errors using static analysis. *PADL*, pages 5–18, 2011.

[4] E. D'Osualdo, J. Kochems, and C.-H. L. Ong. Automatic verification of Erlang-style concurrency. Technical report, 2011. `http://mjolnir.cs.ox.ac.uk/soter/soterpaper.pdf`.

[5] L. Fredlund and H. Svensson. McErlang: a model checker for a distributed functional programming language. In *ICFP*, pages 125–136, 2007.

[6] Qiang Guo and John Derrick. Verification of timed erlang/otp components using the process algebra mucrl. *Erlang Workshop*, pages 55–64. 2007.

[7] C. Hewitt, P. Bishop, and R. Steiger. A universal modular actor formalism for artificial intelligence. In *IJCAI*, pages 235–245, 1973.

[8] A. Kaiser, D. Kroening, and T. Wahl. Efficient coverability analysis by proof minimization. In *CONCUR*, 2012. `http://www.cprover.org/bfc/`.

[9] T. Lindahl and K. Sagonas. Practical type inference based on success typings. In *PPDP*, pages 167–178, 2006.

[10] C. Rackoff. The covering and boundedness problems for vector addition systems. *TCS*, 6:223–231, 1978.

Leveraging Actors for Privacy Compliance

Jeffery von Ronne
The University of Texas at San Antonio
vonronne@cs.utsa.edu

Abstract

Many organizations store and process personal information about the individuals with whom they interact. Because incorrect handling of this information can be harmful to those individuals, this information is often regulated by privacy policies. Although non-compliance can be costly, determining whether an organization's systems and processes actually follow these policies is challenging. It is our position, however, that such information systems could be formally verified if it is specified, designed, and implemented according to a methodology that prioritizes verifiability of privacy properties. This paper describes one such approach that leverages an actor-based architectural style, formal specifications of personal information that is allowed and required to be communicated, and a domain-specific actor-based language. Specifications at the system-, component- Actor-level are written using a first-order temporal logic. We propose that the software implementation be mechanically-checked against individual actor specifications using abstract interpretation. Whereas, consistency between the different specification levels and would be checked using model checking. By restricting our attention to programs using a specific actor-based style and implementation technology, we can make progress towards the very challenging problem of rigorously verifying program implementations against complex privacy regulations.

Keywords formal specifications, program verification, programming languages

1. Introduction

Many organizations store and process personal information about the individuals with whom they interact. Health care providers and insurers maintain medical records of patients' health problems and treatments. Credit card companies have records of their customers' purchasing habits. Search engines have records of every search an individual makes. Modern information technology makes collecting and storing such information cheaper and consequently more pervasive than ever before. Inappropriate disclosure of such information, however, can be harmful to the individual. For instance, 98,000 University of Hawaii students, alumni, and faculty and staff members were affected [14] when their social security numbers and other private information were published online by a retired professor.

Consequently, various laws, industry norms, contractual obligations, and organizational policies have arisen to protect the privacy of individuals and to regulate the conditions under which personal information can be stored, used, and disseminated. Examples include the Health Insurance Portability and Accountability Act (HIPAA) [10], the Family Education Rights and Privacy Act (FERPA) [6], Fair Credit Reporting Act, the UK Internet Advertising Bureau Good Practice Principles for Online Behavioral Advertising, and the ACM Privacy Policy. We will refer to these collectively as privacy policies. Failure to adhere to such policies is not only harmful to the individuals whose information is being disclosed, but it can also be damaging to the organization that violates the individuals' policies (e.g., [2]).

At the same time, organizations are processing the data with ever-more complex and interconnected information systems. For example, electronic medical record systems (which can be as large as sixty million lines of code [8]) are currently being adopted throughout the health care industry. Since it is difficult to know whether such large and complex systems are compliant with applicable privacy policies, there is an urgent need to be able to statically verify the software systems against privacy policies. Although challenging, there has been progress towards formalizing the communications allowed and required by privacy policies using temporal logics [3, 4, 7, 11]. Determining whether an arbitrary software systems complies with even a formalized privacy policy is an even more daunting task. We can, however, get leverage on the problem of verifying software systems against privacy policies by constraining the class of software systems that we wish verify. Specifically, in this paper, we will discuss how one might build distributed information systems using an Actor-based architectural style such that they can be mechanically checked.

2. Policy Specifications in Temporal Logic

Building on Barth *et al.*'s work formalizing Contextual Integrity [3], it is possible to formalize most aspects of privacy policies by abstracting all activity as communications between entities. These communications may either be a *speech act* that is meant as an action performed by the sender or a message that is meant to convey one or more personal attributes (e.g., protected health information (phi) under HIPAA) of a subject from the sender to the receiver.

In this way, an organization's privacy-relevant activities can be represented formally as a sequence or trace of messages that are sent among various entities, including the organization's information system's components and the system's users. Furthermore, a privacy policy can be considered formally to define a set of traces of messages that are compliant with the policy. These sets of traces can be specified using first-order linear temporal logic formulas that are satisfied only by traces in which the organization is privacy compliant.

2.1 Background: Linear Temporal Logic

Temporal Logic [12] characterizes the behavior of reactive systems in terms of traces, a sequence of states and/or events. Our privacy policy language is a many-sorted, first-order linear temporal logic (FOTL) [5]. Due to space constraints, we summarize FOTL only briefly.

FOTL formulas can contain unary and binary temporal operators where the operand(s) are FOTL subformula(s). The temporal operators we use are standard and are discussed below. *Future Operators.* Henceforth: $\Box\phi$ says that ϕ holds in all future states. Eventually: $\Diamond\phi$ says that ϕ holds in some future state. *Past Operators.* Historically: $\boxminus\phi$ says that ϕ held in all previous states. Once: $\diamondminus\phi$ says that ϕ held in some previous state. Since: $\phi_1 \, \mathcal{S} \, \phi_2$ says that ϕ_2 held at some point in the past, and since then ϕ_1 has held in every state. FOTL formulas also include *non-temporal formulas*, which are constructed from atomic formulas, possibly with variables, logical connectives, and quantifiers over variables.

A *logical environment* η maps each variable to a value in the carrier according to the variable's sort. That a formula ϕ is satisfied by a trace σ at an index i under η is denoted by $\sigma, i, \eta \models \phi$, which can be defined inductively on the structure of ϕ.One says that σ satisfies ϕ, written $\sigma \models \phi$, if and only if for any η, we have $\sigma, 0, \eta \models \phi$.

2.2 Policy Formulas in First Order Linear Temporal Logic

The policy formulas can be decomposed into norms by restricting temporal logic formulas to the form found in Figure 1 in a manner similar to that done in Barth *et al.*'s work on Contextual Integrity.

The sorts are P, T, M, and R (denoting agents, attributes, messages, and roles) and their associated carriers are given by $\mathcal{P}$, $\mathcal{T}$, $\mathcal{M}$, and $\mathcal{R}$. The variables p_1, p_2, and q are of sort P, $\hat{r}, \hat{r}_1$, and $\hat{r}_2$ are constants of sort R, t is a variable of sort T, and m is a variable of sort M. We use meta-variables $\vec{y_1}$, $\vec{y_2}$, $\vec{y_3}$, and $\vec{y_4}$ to denote vectors of zero or more additional arguments, drawn from p_1, p_2, and q, and used to support parameterized roles. "Procedure" and "diagnosis" are examples of attributes. The meta-variable $\hat{t}$ stands for an attribute set-valued constant.

A communication action is denoted by $\mathrm{send}(p, q, m)$, in which p is the sender, q is the receiver, and m is the message being sent. Each message contains a set of agent, attribute pairs, $content(m) \subseteq \mathcal{P} \times \mathcal{T}$. The predicate $\mathrm{contains}(m, q, t)$ holds if message m contains attribute t of subject q. A *knowledge state* κ is a subset of $\mathcal{P} \times \mathcal{P} \times \mathcal{T}$. If $(p, q, t) \in \kappa$, this means p knows the value of attribute t of agent q. For example, Alice knows Bob's height. A transition between knowledge states occurs when a message is transmitted, and the attributes contained in the message become known to the recipient. Transmitting a message m is permissible when all the negative norms and at least one positive norm are satisfied.

We allow roles to be parameterized, as illustrated in Figure 2. For this we use the inrole predicates of differing arities. We have $\mathrm{inrole}(p, \vec{y}, r)$ if $(p, \vec{y}, r) \in \mathrm{roleAssignment} \subseteq \mathcal{P} \times \mathcal{P}^* \times \mathcal{R}$. For example, if $(\mathit{Alice}, \mathit{Bob}, \mathit{psychiatrist}) \in \mathrm{roleAssignment}$, then Alice is Bob's psychiatrist and if $\mathrm{inrole}(\mathit{Bob}, \mathit{Fred}, \mathit{Carol}, \mathit{child})$ holds, then Bob is the child of Fred and Carol. We make the simplifying assumption that principals are statically assigned to roles.

Figure 2 shows some example norms that could be used to instantiate a policy of the required syntactic form. The first is a positive norm that allows information to be released to another healthcare provider. The second is a negative norm, that requires that a covered entity p_1 only release a patient q's medical records to an individual (not a healthcare provider) p_2 if q has previous givien permission to release their information to p_2.

3. Distributed Information Systems and Actors

As described above, the formal privacy policies describe the actions humans and organizations (e.g., "covered entities") may perform. These *principals* may, however, make use of automated information systems to store information for them and perform tasks on their behalf. If an information system is used to (automatically) transmit personal information on behalf of a *principal*, it is necessary that the actions that the information system takes or fails to but is supposed to take do not put the *principal* out of compliance with applicable privacy policies. Since a typical organization (such as a medical clinic) will consist of multiple human and automated agents acting on behalf, this requires the system designers to carefully delineate, within the context of the organization, what the responsibilities of the human agents and the information system is.

This can be accomplished by developing a set of temporal logic formulas for each agent of an organization that describe under what circumstances that agent can or must communicate information to other agents, including those that are both internal and external to the organization. The structure of policy formulas for organizations can be adopted for this purpose, by distinguishing between legal *principals* and physical *agents* and by adding a new type of predicate to constrain whether some agent $\mathit{acts\text{-}on\text{-}behalf\text{-}of}$ some principal. Formal methods, such as model checking (after applying a small-model theorem to eliminate the first-order quantification), can then be used to show that these agent responsibilities entail the privacy policy formula when the agent actions are attributed to the appropriate principals.

The temporal logic formula giving the responsibilities of the information system is a partial specification of that system's allowed behavior. If information systems were simple and monolithic, this would be sufficient and the actual program code could be verified directly against this partial specification of the information system. In practice, however, the information systems we care about, such as electronic medical record system, are not monolithic, and include many different software agents interacting. Some of these might be acting on behalf of the organization, some on behalf of particular individuals (such as doctors or nurses) within the organization and some might act on behalf of external entities (such as insurance companies). Furthermore, the number and identity of these software agents and the principals they represent may vary dynamically. They may also be distributed across multiple computer hardware-systems (e.g., backend servers, user workstations) and locations (e.g., a hospital and an affiliated out-patient client).

The Actor Model [1, 9] provides clarity to this situation. In this model, a collection of concurrently operating actors communicate through asynchronous message passing. Each of these actors has a mailbox through which it receives messages. Based on its state/behavior, an actor reacts to the messages it receives one at a time—but not necessarily in the order they arrive—by performing some action. The action an actor takes in response to the receipt of a message can involve sending a finite number of messages to other actors it knows about, creating a finite number of new actors, and/or changing its state/behavior so that it will take different actions in response to future messages.

By considering the information system to be composed of actors (with external I/O being just messages sent to/from outside entities), one can dynamically associate, each actor instance with the principal it acts on behalf of and create privacy specifications for actor classes in temporal logic. These privacy specifications should describe what kind of information can be sent by the actor and under what circumstances. It may also include assumptions about the circumstances under which it can receive messages that are required to be fulfilled by the rest of the system. The advantage of this approach is that there is a natural correspondence between

$$\Box \forall p_1, p_2, q : P.\forall m : M.\forall t : T.\text{send}(p_1, p_2, m) \land \text{contains}(m, q, t) \rightarrow \bigvee_{\varphi^+ \in \text{norms}^+} \varphi^+ \land \bigwedge_{\varphi^- \in \text{norms}^-} \varphi^- \quad (1)$$

$$\text{positive norm} : \text{inrole}(p_1, \vec{y_1}, \hat{r}_1) \land \text{inrole}(p_2, \vec{y_2}, \hat{r}_2) \land \text{inrole}(q, \vec{y_3}, \hat{r}) \land (t \in \hat{t}) \land \phi \quad (2)$$

$$\text{negative norm} : \text{inrole}(p_1, \vec{y_1}, \hat{r}_1) \land \text{inrole}(p_2, \vec{y_2}, \hat{r}_2) \land \text{inrole}(q, \vec{y_3}, \hat{r}) \land (t \in \hat{t}) \land \phi_1 \rightarrow \phi_2 \quad (3)$$

Figure 1. General form of Knowledge-Transmission Policy

$$\text{inrole}(p_1, \textit{covered-entity}) \land \text{inrole}(p_2, q, \textit{provider}) \land (t \in \textit{phi}) \quad (4)$$

$$\text{inrole}(p_1, \textit{covered-entity}) \land \text{inrole}(p_2, \textit{individual}) \land \text{inrole}(q, \textit{patient}) \land (t \in \phi) \rightarrow \Diamond\text{send}(q, p_1, \langle p_2, \textit{release-phi}\rangle) \quad (5)$$

Figure 2. Example Privacy Policy Norms

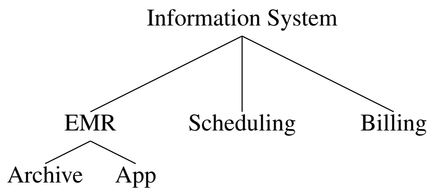


Figure 4. Decomposition of the Information System

messages in the implementation and the communication regulated by the policy.

In order to deal with the complexity of such software systems, however, it is probably necessary to support aggregating Actors into components to which specifications can be applied. Agha *et al.* [1] and Talcott [13] give semantics for actor components in terms of the evolution of *configurations of actors*, which is a group of actors with an interface for interaction with their environment. Although these configurations can evolve over time, an initial configuration can be defined statically and a temporal logic specification can be associated with the static definition that applies to the dynamically changing configuration. These configurations can then be nested statically and formal methods can be applied to show that the inner components' specifications entail the outer components' specification.

4. Defining Components and Actors

This development approach and verification of the software system against formal privacy policies can be facilitating by implementing the software system in an Actor-based domain specific language that contains abstractions for Actors, messages, components, principals, and protected information about particular principals.

An example of how the initial configuration of the components of a medical clinic information system might be defined in a language (called the History Aware Programming Language (HAPL)) that we are developing for this purpose is shown in Figure 3. (Note, for space-reasons, we do not include the HAPL actor class definitions, which use imperative code and per-actor state to define the behavior of actors.) The components of this example are nested as illustrated in Figure 4.

The system as a whole is an instance of the System component. This component is parameterized with the dependently-typed parameter `clinic` and a corresponding type-parameter α, which is constrained to be a covered entity. (A variable of type `P[`α`]` holds a runtime representation of the principal α.) System has one externally visible actor that is specified using the channel keyword. Furthermore, System contains three sub-component instances records which is an instance of EMR, scheduling which is an instance of Scheduling, and billing which is an instance of Billing; the component classes Billing, Scheduling, and EMR are also nested inside of System. The EMR component definition, in turn consists of two actor instances. The first is an `archive` that is responsible for storing in EMR data and is an instance of EMRArchive. It is parameterized with the same `clinic` that as the System of which it is a part. The second is an instance of EMRApp; it is parameterized by the archive and provides an interface for interacting with the archive.

The definitions of the Billing and Scheduling components and the EMRArchive and EMRApp actors are omitted for brevity. HAPL's actor definitions, however, are statically typed with types parameters for principals, and types for protected information. The syntax resembles conventional class-based object oriented languages such as C++, Java, C♯, and Scala. This language is designed with static semantics that use an abstract interpretation to verify the actor against its specification.

5. Example

Consider a medical clinic for which the information system in Figure 3 is being built. This organization might have the following privacy policy norms:

1. If a patient requests a copy of their medical records, the clinic will provide them to the patient.
2. Patient records will not be disclosed, except to:
   (a) clinic employees
   (b) the patient, or
   (c) another third party if the patient has given prior authorization.

The statement that "patient records will not be disclosed except to . . . " is a safety requirement. It can be applied independently to the human agents and to each of the actors and components comprising the information system. As long as both the information system and the human agents only disclose information when permitted, the organization as a whole will only disclose information when permitted.

The requirement that obligates an organization to respond to a patient's requests for copies of her medical records, however, is more complicated. It raises the question of from whom the patient is allowed to make such requests. Must the clinic's janitor be prepared to cope with such requests? Must the information system? A similar provision in HIPAA allows a medical provider to require such requests to be made in writing, as long as it informs the patient of how to submit such requests.

```
component System [$\alpha$ | inrole($\alpha$, covered-entity)](clinic:P[$\alpha$]) {
  channel emr = records.app

  records = new component EMR[]()
  scheduling = new component Scheduling[]()
  billing = new component Billing[]()


  component Billing[]() { ... }

  component Scheduling[]() { ... }

  component EMR[]() {
    archive = new actor EMRArchive[$\alpha$](clinic)
    app = new actor EMRApp[$\alpha$](clinic, archive)
  }
}
```

Figure 3. Component Definition in HAPL

One could imagine, that an organization might be allowed to implement this requirement by telling their patients that they can directly access their own records the electronic medical records app (that is part of the clinic's information system). In this case, the obligation becomes a liveness property required of the information system. Formalizing this, however, requires that the communication channels (e.g., the electronic medical records app) be identified (as was done in Figure 3) and incorporated into the information system's formal partial specification.

6. Summary and Outlook

Ensuring the privacy compliance of information systems is an increasingly urgent problem. There is a need to be able to formally verify information system's compliance against privacy policies described in formal logics. This problem is very challenging, but by restricting our attention to programs using a specific actor-based style and implementation technology, we can make progress towards rigorously verifying program implementations against complex privacy regulations. We have begun developing a programming language, the History Aware Programming Language (HAPL), and verification tools that will take us this next step.

Acknowledgments

This paper is dedicated to the memory of William Hale Winsborough who collaborated in the development of many of the ideas discussed in this paper. The author is also indebted to Jianwei Niu, Omar Chowdhuri, and Shamim Ashik, who have contributed to the project of which this paper is a part and contributed to the policy language description and the figures found in this paper.

References

[1] G. A. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7(01): 1–72, 1997. doi: 10.1017/S095679689700261X.

[2] amednews.com. Clinics fined $4.3 million for hipaa violations, 2011. Available at *http://www.ama-assn.org/amednews/2011/03/07/bisb0307.htm*.

[3] A. Barth, A. Datta, J. C. Mitchell, and H. Nissenbaum. Privacy and contextual integrity: Framework and applications. In *IEEE Symposium on Security and Privacy*, pages 184–198, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2574-1. doi: http://dx.doi.org/10.1109/SP.2006.32.

[4] H. DeYoung, D. Garg, L. Jia, D. Kaynar, and A. Datta. Experiences in the logical specification of the hipaa and glba privacy laws. In *Proceedings of the 9th annual ACM workshop on Privacy in the electronic society*, WPES '10, pages 73–82, New York, NY, USA, 2010. ACM.

[5] E. A. Emerson. Handbook of theoretical computer science. chapter Temporal and modal logic, pages 995–1072. MIT Press, Cambridge, MA, USA, 1990. ISBN 0-444-88074-7. URL `http://portal.acm.org/citation.cfm?id=114891.114907`.

[6] FERPA. Family educational rights and privacy act (ferpa), 1974. (20 U.S.C. §1232g; 34 CFR Part 99).

[7] D. Garg, L. Jia, and A. Datta. Policy auditing over incomplete logs: theory, implementation and applications. In *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, pages 151–162, New York, NY, USA, 2011. ACM.

[8] M. Hagland. Thinking about Clinical IT as a Strategic Investment (Healthcare Informatics), 2012. Available at http://www.healthcare-informatics.com/article/thinking-about-clinical-it-strategic-investment.

[9] C. Hewitt. Viewing control structures as patterns of passing messages. *Artificial Intelligence*, 8(3):323 – 364, 1977. ISSN 0004-3702. doi: DOI:10.1016/0004-3702(77)90033-9.

[10] HIPAA. Health insurance portability and accountability act (hipaa), 1996. (42 U.S.C. §300gg, 29 U.S.C §1181 *et seq.*, and 42 U.S.C §1320d *et seq.*; 45 CFR Parts 144, 146, 160 162, and 164).

[11] M. J. May, C. A. Gunter, and I. Lee. Privacy apis: Access control techniques to analyze and verify legal privacy policies. In *CSFW*, pages 85–97, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2615-2. doi: http://dx.doi.org/10.1109/CSFW.2006.24.

[12] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science*, volume 526, pages 46–67, 1977.

[13] C. L. Talcott. Composable semantic models for actor theories. *Higher-Order and Symbolic Computation*, 11(3):281–343, 1998.

[14] The Chronicle. U. of Hawaii Settles Lawsuit Over Data Breaches Affecting 98,000 People, 2012. Available at http://chronicle.com/blogs/ticker/u-of-hawaii-settles-lawsuit-over-data-breaches-affecting-98000-people/40001.

Adding Distribution and Fault Tolerance to Jason [*]

Álvaro Fernández Díaz Clara Benac Earle Lars-Åke Fredlund

Babel group, DLSIIS, Facultad de Informática, Universidad Politécnica de Madrid

{avalor,cbenac,fred}@babel.ls.fi.upm.es

Abstract

In this paper we describe an extension of the multiagent system programming language Jason with constructs for distribution and fault tolerance. The standard Java-based Jason implementation already does provide a distribution mechanism, which is implemented using the JADE library, but to use it effectively some Java programming is often required. Moreover, there is no support for fault tolerance. In contrast, this paper develops constructs for distribution and fault tolerance wholly integrated in Jason, permitting the Jason programmer to implement complex distributed systems entirely in Jason itself. The fault tolerance techniques implemented allow the agents to detect, and hence react accordingly, when other agents have stopped working for some reason (e.g., due to a software or a hardware failure) or cannot be reached due to a communication link failure. The introduction of distribution and fault tolerance in Jason represents a step forward towards the coherent integration of successful distributed software techniques into the agent based software paradigm. The proposed extension to Jason has been implemented in eJason, an Erlang-based implementation of Jason. In fact, in this work we essentially import the distribution and fault tolerance mechanisms from the Erlang programming language into Jason, a task which requires the adaptation of the basic primitives due to the difference between a process based functional programming language (Erlang) and a language for programming BDI (Belief-Desire-Intention) agent based systems (Jason).

1. Introduction

The increasing interest in multiagent systems (MAS) is resulting in the development of new programming languages and tools capable of supporting complex MAS development. One such languages is Jason [6]. Some of the more difficult challenges faced by the multiagent systems community, i.e., how to develop scalable and fault tolerant systems, are the same fundamental challenges that any concurrent and distributed system faces. Consequently, the agent-oriented programming languages provide mechanisms to address these issues, typically borrowing from more mainstream frameworks for developing distributed systems. For instance, Jason allows the development of distributed multiagent systems by interfacing with JADE [4, 5]. However, Jason does not provide specific mechanisms to implement fault-tolerant systems.

MAS and the actor model [2] have many characteristics in common. The key difference is that agents normally impose extra requirements upon the actors, typically a rational and motivational component such as the Belief-Desire-Intention architecture[10, 11].

Some programming languages based on the actor model are very effective in addressing the aforesaid challenges of distributed system. Erlang [3, 7], in particular, provides excellent support for concurrency, distribution and fault tolerance. However, Erlang lacks some of the concepts, like the Belief-Desire-Intention architecture, which are relevant to the development of MAS.

This article forms part of a research programme to evaluate whether the BDI architecture provides useful programming paradigms which can improve the design of (non AI-based) complex distributed systems too. However, to be able to do such an evaluation we found that it was first necessary to improve the support for programming distributed systems available in Jason implementations (which is the topic of this article).

In recent work [8], we presented eJason, an open source implementation of a significant subset of Jason in Erlang, with very encouraging results in terms of efficiency and scalability. Moreover, some characteristics common to Jason and Erlang (e.g. both having their syntactical roots in Prolog) made the implementation quite straightforward. However, the first eJason prototype did not permit the programming of distributed or fault-tolerant multiagent systems.

In this paper, we propose a distribution model and a fault tolerance mechanism for Jason closely inspired by Erlang. This extension of Jason has been implemented in eJason, thus making it possible to develop complex distributed systems fully in Jason itself. Our implementation of eJason and the sample multiagent systems described in this and previous documents can be downloaded at:

$$git://github.com/avalor/eJason.git$$

The rest of the paper is organized as follows: Section 2 provides background material introducing Jason, Erlang and eJason. Sections 3 and 4 describe the proposed distribution model and fault tolerance mechanisms for Jason programs, respectively. Some details on the implementation in eJason of these extensions can be found in Section 5. An example that illustrates in detail the use of the proposed extension is included in Section 6. Finally, Section 7 presents the conclusions and future lines of work.

2. Background

In this section we briefly introduce Jason, Erlang and eJason. Some previous knowledge of both Jason and Erlang is assumed.

* This work has been partially supported by the following projects: DESAFIOS10 (TIN2009-14599-C03-03), PROMETIDOS(P2009/TIC-1465) and EU FP7 Prowess (GA Nr. 317820).

2.1 Jason

Jason is an agent-oriented programming language which is an extension of AgentSpeak [9]. The standard implementation of Jason is an interpreter written in Java.

2.1.1 The Jason Programming Language

The Jason programming language is based on the Belief-Desire-Intention (BDI) architecture [10, 11] which is highly influential on the development of multiagent systems. The first-class constructs of the language are: beliefs, goals (desires) and plans (intentions). This approach allows the implementation of the rational part of agents by the definition of their "know-how", i.e., *how* each agent should act in order to achieve its goals, based on its subjective *know*ledge.

The Jason language also follows an environment-oriented philosophy, i.e., an agent exists in an environment which it can perceive and with which it can interact using so called external actions. In addition, Jason allows the execution of internal actions. These actions allow the interaction with other agents (communication) or to carry out some useful tasks such as, e.g., string concatenation and printing on the standard output, among others.

2.1.2 The Java Implementation of Jason

A complete description of the Java implementation of Jason can be found in [6]. This implementation of Jason allows the programming of distributed multiagent systems by interfacing with the well-known third-party software JADE [4, 5], which is compliant to FIPA recommendations [1]. JADE implements a distribution model where the agents are grouped in agent containers which are, in turn, grouped again to compose agent platforms.

The distribution of a Jason system using JADE is not transparent from the programmer's perspective as he/she must declare the architecture of the system (centralised and JADE being the ones provided by default), and, most likely, execute some actions that rely on the third-party software used to distribute the system.

The Java interpreter of Jason provides mechanisms to detect and react to the failure of a plan of an agent. However, there are no mechanisms to detect and react to the failure of an entire agent. That is, there are no constructs which permit to detect if some agent has stopped working (has died) or has become isolated. One can, of course, *program* a "monitor agent" to continuously interact with the agent that should stay alive according to some pre-established communication protocol, in order to detect if the monitored agent fails. However, having to program such monitors by hand is an error prone and a tedious task.

2.2 Erlang

Erlang [3, 7] is a functional concurrent programming language created by Ericsson in the 1980s which follows the actor model. The chief strength of the language is that it provides excellent support for concurrency, distribution, and fault tolerance on top of a dynamically typed and strictly evaluated functional programming language. It enables programmers to write robust and clean code for modern multiprocessor and distributed systems.

An Erlang system (see Fig. 1) is a collection of Erlang nodes. An Erlang node (or Erlang Run-time System) is a collection of processes (actors), with a unique node name. These processes run independently from each other and do not share memory. They interact via communication. Communication is asynchronous and point-to-point, with one process sending a message to a second process identified by its process identifier (pid). Messages sent to a process are put in its message queue, also referred to as a mailbox.

As an alternative to addressing a process using its pid, there is a facility for associating a symbolic name with a pid. The name, which must be an atom, is automatically unregistered when the associated process terminates. Message passing between processes in different nodes is transparent when pids are used, i.e., there is no syntactical difference between sending a message to a process in the same node, or to a remote node. However, the node must be specified when sending messages using registered names, as the pid registry is local to a node. For instance, in Fig. 1 let the process Proc C be registered with the symbolic name `procC`. Then, if the process Proc B wants to send a message to the process `procC`, the message will be addressed to `procC@Node2`.

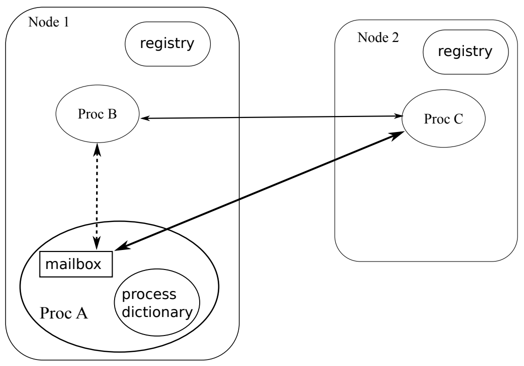


Figure 1. An Erlang multi-node system

A unique feature of Erlang that greatly facilitates building fault-tolerant systems is that one process can monitor another process in order to detect and recover from abnormal process termination. If a process P_1 monitors another process P_2, and P_2 terminates with a fault, process P_1 is automatically informed by the Erlang runtime of the failure of P_2. It is possible to monitor processes at remote nodes. This functionality is provided by an Erlang function named `erlang:monitor`.

2.3 eJason

eJason is our Erlang implementation of the Jason programming language which exploits the support for efficient distribution and concurrency provided by the Erlang runtime system to make a MAS more robust and performant. It is interesting to note the similarities between Jason and Erlang; both are inspired by Prolog, and both support asynchronous communication among computational independent entities (agents/processes), which makes the implementation of Jason in Erlang rather straightforward.

The first prototype of eJason was described in [8]. This prototype supported a significant subset of Jason. This subset included the main functionality: the reasoning cycle, the inference engine used by test goals and plan contexts, and the knowledge base. We continue developing eJason by increasing the Jason subset supported (which now includes plan annotation and an improved inference engine that generates matching values for the variables in the queries upon request, instead of unnecessarily computing all matching values), and by improving the design and implementation of eJason.

Probably the most relevant fact of the implementation of eJason is the one-to-one correspondence between an agent and an Erlang process (a lightweight entity), enabling the eJason implementation to execute multiagent systems composed of up to hundreds of thousands of concurrently executing agents with few performance problems. This compares very favorably with the Java based standard Jason implementation which has problems in executing systems with no more than thousands of concurrent agents (even if executed using a pool of threads) on comparable hardware (see [8] for benchmarks).

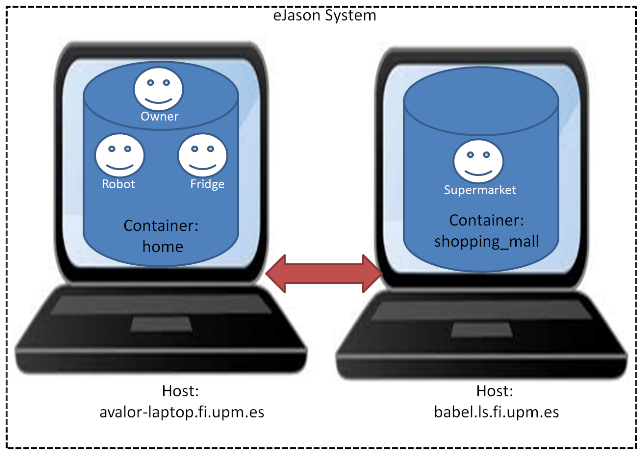


Figure 2. Sample eJason Distributed System

3. Distribution

In this section we describe the proposed agent distribution model extension to Jason, which has been implemented in eJason. It is inspired by the distribution model of Erlang, as, in our opinion, it is a sound and efficient one. The distribution model has been designed with three goals in mind: distribution transparency (i.e., ensuring that distributed agents can communicate), efficiency, and minimizing the impact on the syntax of Jason programming language.

3.1 Distribution Schema

Below we introduce the terminology used to describe the distribution model.

- A **multiagent system** is composed by one or more agent containers.
- Each **(agent) container**[1] is comprised of a set of agents, and is located on a computing host. It is given a *name* that is unique in the whole system. Concretely the name has the shape `NickName@HostName`, where `NickName` is an arbitrary short name given to the container when it is started and `HostName` is the full name of the host in which the container runs. For instance, `home@avalor-laptop.fi.upm.es` corresponds to a container that runs in a host whose name is `avalor-laptop.fi.upm.es`.
- Each **agent** is present in a single container. At the moment of agent creation, it is given a symbolic name that is unique within the container. A single agent is uniquely identified in the system by the combination of its own symbolic name and the name of its container. Using the unique name of an agent, agents can communicate with each other irrespectively of the containers they reside in.

As an example, consider the distributed Jason multiagent system in Figure 2. This system is composed by four agents distributed over two different hosts. The agents with the names `owner`, `robot` and `fridge` reside in a container named `home@avalor-laptop.fi.upm.es`. The agent `supermarket` runs in the container `shopping_mall@babel.ls.fi.upm.es`, located on a different host.

[1] we use the name container to emphasize the similarities with a JADE container

3.2 Distribution API

Recall that Jason agents communicate with each other using internal actions. Thus, to support distribution, we add support for communicating in existing internal actions such as, e.g., `.send`, as well as adding a few new internal actions. In practice, most of the new internal actions are just an extension of existing ones to allow the inclusion of the name of the container where the agent receiving the effect of the internal action (e.g. receiving a message) runs. This name is added as an annotation. A complete example showing how the new API is used in practice is included in Section 6.

3.2.1 Agent Communication

The communication between agents in Jason requires the use of the internal action

a) `.send(AgentName,Performative,Message [Reply,Timeout]),`

where the parameters in brackets are optional and `AgentName` is the symbolic name of the agent receiving the message. The parameters `Reply` and `Timeout` can only be used when the performative is of type `ask`. We omit the exact description of their meaning from this article as they are present with the same semantics in "non-distributed Jason"; see [6] for details.

In the Java implementation of Jason, the symbolic name must uniquely identify one agent within the system, while, as mentioned above, the distributed extension guarantees only that agent names are unique in a container. Thus the above internal action can be used only to communicate between agents located in the same container (intra-container communications).

Therefore, to permit communication between agents located in different containers (inter-container communication) the distributed extension of Jason provides a new internal action

b) `.send(Address,Performative,Message, [Reply,Timeout])`

where the parameter `Address` is an annotated atom with the structure `AgentName[container(Container)]` and `Container` is the name of the container in which the agent with symbolic name `AgentName` runs. Most internal actions similarly accept such an address structure to specify an agent; in the following we will not list these variant internal actions.

The guarantees[2] provided by the aforementioned internal actions differ:

- If the execution of **a)** succeeds, the reception of the message by the receiver is guaranteed. However, it does not ensure that the receiving agent considers the message socially acceptable (i.e. considered suitable for processing, cf. not socially acceptable messages are automatically discarded and do not generate any event, see [6]) nor that it has a plan which can be triggered by the reception of the message. The execution will fail if there is no agent in the same container whose symbolic name is `AgentName`.
- The execution of **b)** always succeeds. This internal action, thus, provides no guarantees regarding the successful delivery of the message.

3.2.2 Agent Creation and Destruction

Agents can *create* agents in a named container using the internal action

[2] We encourage the reader to read the content at http://www.erlang.org/faq/academic.html#id54296 to get a feel of how useless and inefficient the imposition of strong guarantees on message delivery can be for distributed systems.

- `.create_agent(AgentName, Source,[InitialBeliefs])`

where the parameter in brackets is optional. If `Container` is not provided, using an `Address` structure, the new agent is created in the same container as the agent executing the internal action. As in [6], the parameter `Source` indicates the implementation of the agent (its plans, initial goals and initial beliefs). Finally, `InitialBeliefs` is a list of beliefs that should be added to the set of initial beliefs of the new agent upon creation. This internal action will succeed if (1) the named container exists in the system, and (2) there is no other agent with symbolic name `AgentName` already in `Container`, and (3) `Source` correctly identifies an agent implementation.

An agent can also explicitly *kill* other agents. The following internal action allows this:

- `.kill_agent(AgentName)`.

The parameters and the meaning of their absence are analogous to those above. The execution of this internal action does not fail, even if the agent to terminate does not exist.

3.2.3 Container Name Discovery

An agent can *discover* the name of its own container by executing the internal action

- `.my_container(Var)`

If the variable `Var` is unbound, the execution of the internal action does not fail and, as a result, `Var` will be bound to the name of the container of the agent. If `Var` is bound to any value different from the name of the container of the agent executing it, the internal action will fail, otherwise it succeeds.

An agent can *discover* the name of the agent and container (the complete address) from which it has received a message. Every belief and goal generated from a communication is labeled with the annotation

- `source(AgentName[container(Container)])`

Notice that this extension guarantees backwards compatibility with Jason legacy code. The arity of the annotation `source`, see [6], is maintained and the `Address` structure can be used to identify the sender agent, e.g. in a `.send` internal action that represents the reply to the message received.

4. Fault Tolerance

The fault detection extension that we describe in this section enables an agent to express its desire to be notified when another agent terminates or becomes unreachable. As we describe later, this notification is carried out by the runtime system by adding a new belief to the belief base of the agent being notified.

4.1 Agent failures

The reasons why an agent may stop working are numerous. For instance, the host in which the agent runs has been shut down, the agent itself has been stopped or has crashed due to some error in the source code, or the host in which the agent runs may have become isolated from the rest of agents in the system.

4.2 Monitoring agents

Agents are not informed about failures in each other by default. Instead, a (monitoring) agent interested in the state of a (monitored) agent must explicitly request to be notified when the status of the monitored agent worsens. This request is implemented as the internal action:

- `.monitor(AgentName)`

where the parameter `AgentName` is the symbolic name of the monitored agent in the desired container. This action is executed by the monitoring agent. The execution of this internal action never fails.

After the execution of the `.monitor` internal action, the monitoring agent will be informed about failures in the monitored agent at most once, i.e. after one error in the monitored agent is detected. Nevertheless, the `.monitor` internal action can be called again after the notification. If the monitored agent fails, the monitoring agent will receive the following new belief:

- `+agent_down(AgentName[container(Container)])`
  `[reason(RType)]`

along with the corresponding belief addition event. The possible values of `RType` are `unknown_agent`, `dead_agent` and `unreachable_agent`. Their meaning is as follows:

- **unknown_agent**. There is no agent whose symbolic name is `AgentName` in the container `Container` as specified in the invocation of the internal action `.monitor`.
- **dead_agent**. The monitored agent with the symbolic name `AgentName` has stopped working.
- **unreachable_agent**. The containers of the monitored and monitoring agent are not connected, e.g. caused by a network problem, or by a problem with the container host, or by a failure in the container itself. The reconnection of the containers, which renders the monitored agent reachable again if it is still alive, may happen but it is not notified to the monitoring agent.

Clearly these errors are not mutually exclusive. For instance, an unreachable agent (**unreachable_agent**) may be dead as well (**dead_agent**).

In Section 6 we illustrate how this fault detection mechanism is used to detect and help recover from different agent failures.

5. Implementation

In this section, we provide some details about the implementation in eJason of the proposed extensions to Jason. We do not intend to give a low-level description of how every single element has been implemented. Instead, we describe the correspondence between the elements introduced and their Erlang counterparts (recall that the extensions are inspired by the distribution and fault tolerance mechanism of Erlang). A very basic knowledge about Erlang constructs and their semantics suffices to understand the contents of this section.

5.1 Distribution

The new concept introduced by our distribution model is the agent container. It is implemented using an Erlang node. Therefore, each container must be given a symbolic name unique in its host, as it is not possible to have two Erlang nodes with the same name running in the same host (Erlang nodes can be given short- or full-names. For convenience, eJason only considers the latter possibility). For each agent container in a system, a new Erlang node is started. Besides, in eJason each agent is represented by a different Erlang process. Therefore, given an agent with symbolic name `AgentName` running in a container with symbolic name `NickName` in host `Host`, there exists an Erlang process running in the Erlang node `NickName@Host`. This process is locally registered as `AgentName`.

The guarantees for the message exchange achieved by executing the internal action `.send`, described in Section 3, derive from the Erlang semantics of their implementation:

- When internal action `.send(AgentName,Performative, Message,[Reply,Timeout])` is executed, a message is sent to the Erlang process locally registered as `AgentName`. This operation fails if there is no process registered as `AgentName`.

- When the internal action executed is `.send(Address, Performative,Message,[Reply,Timeout])`, where the parameter `Address` is the annotated atom described in Section 3.2.1, a message is sent to the Erlang process that runs in the Erlang node with name `Container` and that is registered as `AgentName` in that same node. This operation cannot fail even if the Erlang node does not exist (which does not ensure the correct delivery of the message).

5.2 Faul Tolerance

The internal action `.monitor` is implemented by the Erlang function: `erlang:monitor`. Consider the case where an agent `A` monitors another agent `B`. When `A` executes the `.monitor` internal action, the Erlang process corresponding to `A` invokes the function `erlang:monitor` giving among its parameters the identifier of the Erlang process corresponding to `B` (either its registered name alone or along with the name of its Erlang node).

If there is a failure on the agent `B`, the Erlang process of agent `A` receives a so-called message 'DOWN' message, which provides, among others, information about the failure. Depending on that information, which is represented by an Erlang construct (atom or tuple), one of the different failures considered is generated:

- If the information received is the atom `noproc`, the Erlang process corresponding to the agent `B` cannot be found while the Erlang node it should be running in can. Therefore, a failure of type `unknown_agent` is detected.
- If the information received is the atom `noconnection`, the Erlang node corresponding to the container of agent `B` cannot be found. A failure of type `unreachable_agent` is detected.
- The reception of any other kind of information (e.g. the atom `killed` meaning that the Erlang process of agent `B` has been killed or a tuple providing information about the reason why that process has crashed) means that the monitored agent has died. Therefore, a failure of type `dead_agent` is detected.

6. Example

In this section we illustrate the Jason extensions using a sample multiagent system that is distributed and which makes use of the fault tolerance mechanisms described in Sections 3 and 4. The example is inspired by a similar one provided with the Jason distribution and also described in [6]. The example runs unchanged under eJason.

6.1 The system

The system is composed by the following agents:

- **Agent owner:** this agent monitors a robot (another agent) and is continuously avid for beer. It asks the robot for cans of beer. If it gets a beer, it drinks it whole sip by sip, then asks for another beer. If the robot agent reports that there are no more beers in the fridge, the owner takes a short nap and, immediately after waking up, starts asking for a beer again. Upon reception of a message from the robot informing that it is not allowed to give any more beer to the owner, it disconnects the robot. The owner monitors the robot, so if it dies (because the owner disconnects it or otherwise), the former is immediately aware of it and starts the robot again. Finally, if this agent drinks more beer than its physical limit can bear, it collapses (the agent owner dies).
- **Agent robot:** this agent fulfills the owner's requests for beer. Upon reception of a request from the owner, it goes to the fridge and checks whether it contains some beer or is empty. In the first case, it grabs one beer, goes to the location of the owner and gives him the beer. The second case is again split in two, depending on whether the supermarket (represented by another agent) is open (the supermarket agent is not dead and is reachable) or closed (the supermarket agent is dead or is unreachable). If the fridge is empty and the supermarket is open, the robot orders some beer, which will be delivered directly to the fridge, and tells the owner that the fridge is empty. Otherwise, if the fridge is empty and the supermarket is closed, the robot just tells the owner that the fridge is empty, without trying to make any order. In order to know whether the supermarket is open or not, the robot monitors the supermarket agent. Depending on the type of failure detected in the supermarket agent, the robot emits a different speech (prints a different message on the standard output). Finally, the robot also monitors the owner and, if it dies, emits a short speech and waits, idly, for future requests.
- **Agent fridge:** this agent receives requests for beer from the robot. If it is not empty, it gives a beer to the robot, hence decrementing its current stock by one unit. If it is empty, it does not hand out any beer to the robot and just reports back this fact. Finally, anytime it receives a delivery from the supermarket, it updates its contents.
- **Agent supermarket:** the behaviour of this agent is the simplest in the system. It waits for a delivery order from any agent (the robot in this case) and fulfills it.

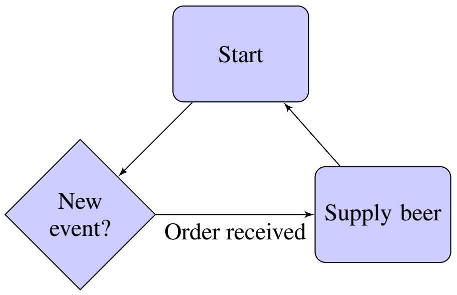


Figure 3. Flowchart for agent "supermarket"

The eJason code for each of these agents is included in Appendix A, Figures 7, 8, 9, and 10.

For the sake of clarity, we also include a series of flowcharts in Figures 3, 4, 5, and 6 that provide the diagrammatic representation of the behaviour of each of the agents, respectively.

Note that the environment entity is not implemented in eJason yet, hence not allowing the execution of external actions or the gathering of information through perception. Therefore, some actions of the system, like the displacement of the robot or checking the contents of the fridge, have been assumed to be always successful and do not require any interaction with the environment.

6.2 Code Excerpts

In this section we provide some brief code excerpts showing agent plans, to illustrate the new features introduced in Jason (and implemented in eJason).

6.2.1 Communication

Consider the following plan from the agent robot, where both intra- and inter-container communication take place:

```
+no_more(beer) :                              //Trigger
    (not  closed(supermarket)) &              //Context
```

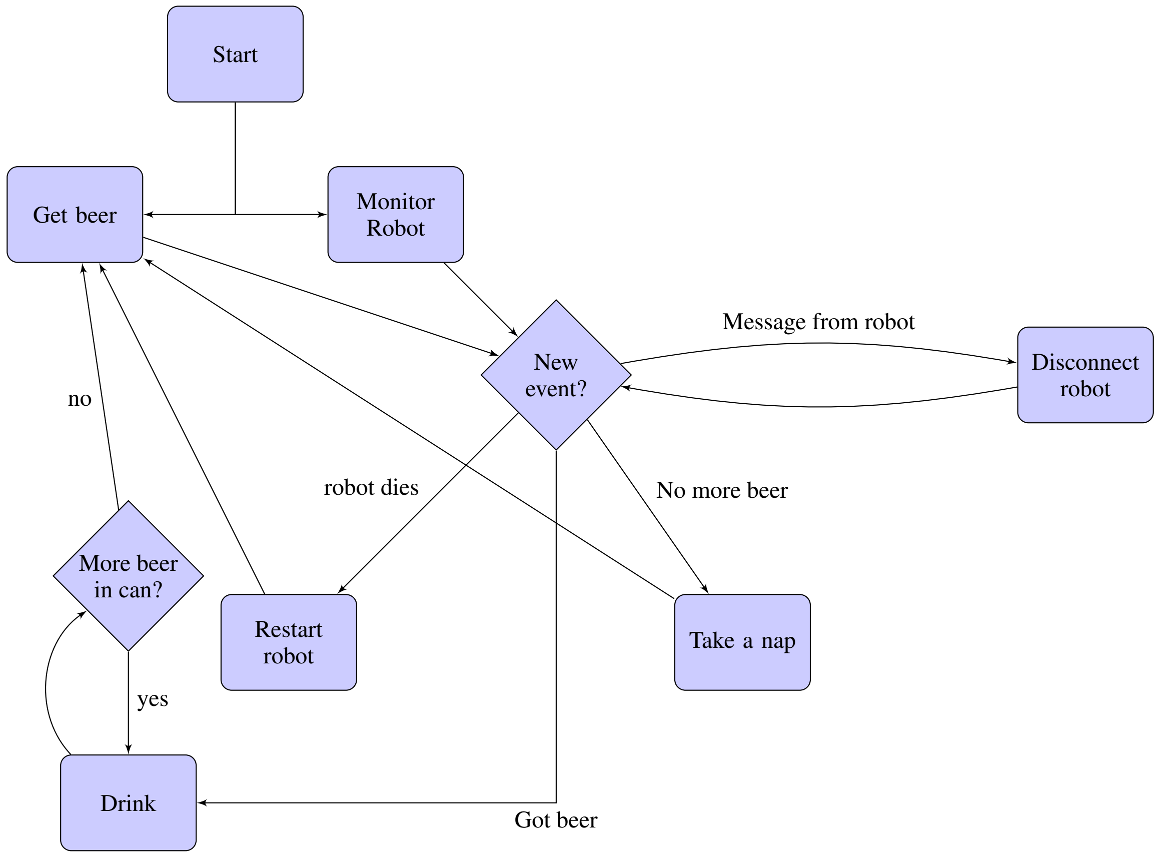


Figure 4. Flowchart for agent "owner"

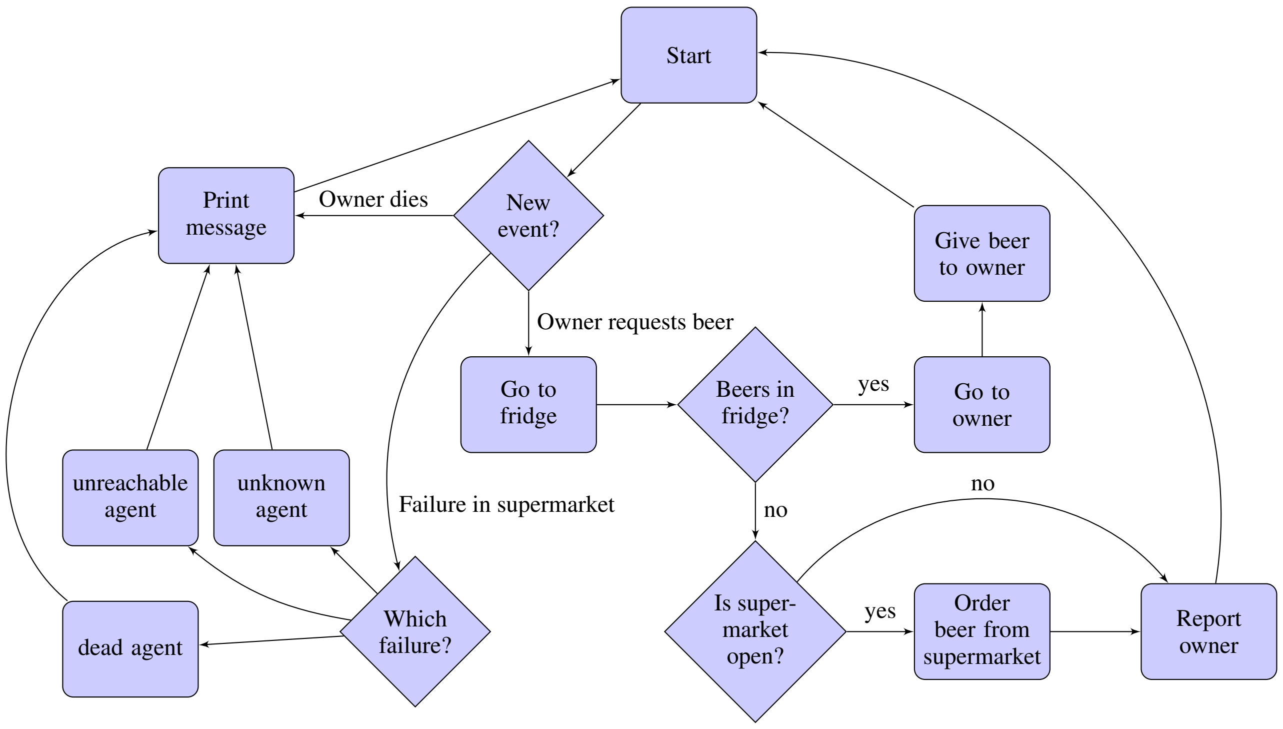


Figure 5. Flowchart for agent "robot"

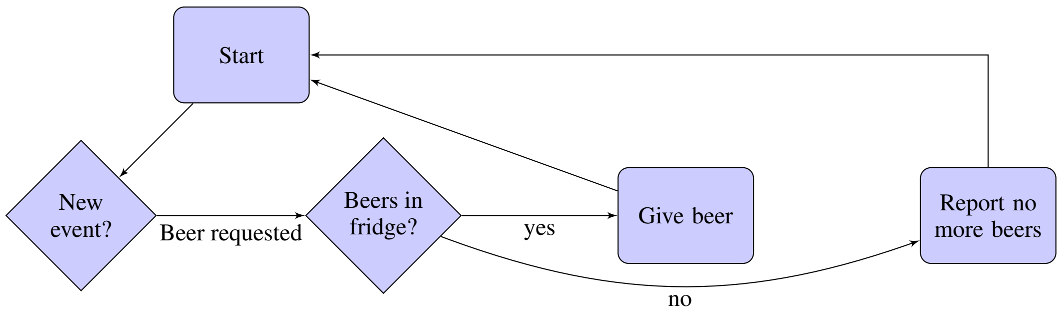


Figure 6. Flowchart for agent "fridge"

```
   address(supermarket,SupContainer) <- //_____
  -no_more(beer);                      //Body
  .print("Fridge is empty.");          //
  .send(supermarket                    //
        [container(SupContainer)],     //
        achieve, order(beer,5));       //
  .send(owner,tell,no_more(beer));     //
  !at(robot,fridge).                   //_____
```

This plan is triggered when the robot notices that the fridge is empty. The context requires the supermarket not to be closed (i.e. the agent supermarket must be alive and reachable) and the address of that agent to be known. When the plan is triggered and the context is met, the robot executes, in appearance order, the actions in the body of the plan. The first action deletes the belief that states that the fridge is empty while the second makes the robot print a message on the standard output. The third action is a `.send` action of type `b)` according to Section 3.2.1, where `AgentName` is `supermarket`, `Container` has the value of the variable `SupContainer` (which in this case is `shopping_mall@babel.ls.fi.upm.es`), `Performative` is `achieve` and `Message` is `order(beer)`. By executing this action, the robot is ordering beer from the supermarket agent, which runs in a different container. The fourth action is a `.send` action of type `a)` and does not include any reference to the container in which the agent owner runs, as it is the same one for robot agent. Finally, the last action represents an achievement goal that requires the robot to go to the location of the owner.

6.2.2 Monitoring an agent

The code for the agent robot also shows how an agent can monitor another agent. Consider the following excerpt:

```
+!monitor(Agent):
   address(Agent,Container) <-
  -closed(supermarket);
  .monitor(Agent[container(Container)]).

+!monitor(owner): true <-
  .monitor(owner).
```

The first plan can only be executed if the full address of the monitored agent is known.

The second plan will only be executed if the monitored agent is the owner agent. Notice that, this time, the container of the monitoring and monitored is the same, hence being omitted from the parameters of the `.monitor` action.

6.2.3 Detecting different failures

Finally, consider these four plans, again from the source of the robot agent:

```
+agent_down(supermarket)
     [reason(unknown_agent)]: true <-
   +closed(supermarket);
   .print("I cannot find the supermarket
          in the shopping mall").


+agent_down(supermarket)
     [reason(unreachable_agent)]: true <-
    +closed(supermarket);
    .print("I cannot find the shopping mall").


+agent_down(supermarket)
     [reason(dead_agent)]: true <-
    +closed(supermarket);
    .print("The supermarket has just closed").


+agent_down(owner): true <-
   .print("Oh, oh. My master has passed out.").
```

The first plan is triggered when a failure of type `unknown_agent` is detected on the supermarket agent. This can only happen if the container `shopping_mall@babel.ls.fi.upm.es` is reachable but does not contain any supermarket agent. A possible reaction from the robot agent could have been starting that agent, but we did not consider reasonable the option of allowing the robot to "open" the supermarket.

The second plan is only triggered when a failure of type `unreachable_agent` is detected on the supermarket agent. This failure is generated when the connection to the container named `shopping_mall@babel.ls.fi.upm.es` is lost, hence leaving the agent supermarket unreachable.

The third plan is triggered when a failure of type `dead_agent` is detected again on the supermarket agent, i.e. the Erlang process corresponding to that agent is dead. Again, a possible reaction from the robot agent could have been starting that agent.

The fourth plan is triggered by the detection of any kind of failure on the owner agent, as the annotation of the event is ignored. These four plans represent all the possible agent failure detections that are possible in the example.

Notice that, in all four plans, the `[container(Container)]` annotation corresponding to each `AgentName` has been ignored in their triggers, as this information is not used neither in their contexts nor in their bodies.

6.3 Experiments

In this section we report on some experiments that we performed on the multiagent system described above. In all of them, the distribution of the agents was organized in the following way, which is also depicted in Figure 2:

- The agents owner, robot and fridge run in the same container whose name is *home@avalor-laptop.fi.upm.es*.
- The agent supermarket runs in a different host and container. The name of this container is *shopping_mall@babel.ls.fi.upm.es*. Checking Figure 10, one may notice that this address is hard-coded as an initial belief of the robot agent. It is done this way for convenience, as no service discovery is available for the agents yet.

The experiments carried out were the following:

6.3.1 Experiment 1: distribution

The goal of this experiment is checking whether the implementation of the proposed distribution works correctly. To do it, we start all the agents of the system but do not connect the two hosts until some time after the start of the experiment. If the distribution works properly, after emptying the fridge, the owner must not get more beer until the robot can order more from the supermarket, which is not possible while the hosts remain disconnected. Next, we list the steps of the experiment (enumerated using letters) together with a description of the observable behaviour of the agents that is relevant to the experiment (presented between angle brackets).

a) Disconnect the hosts (isolating one of them suffices).

b) Start all the agents.

<The owner drinks all four beers in the fridge. The robot informs that the fridge is empty, but does not attempt to order new beers because it is aware of the fact that the supermarket is unreachable. The owner sleeps and wakes up periodically asking the robot for more beer. The system does not evolve.>

c) Connect the hosts via internet or intranet

<The robot orders more beer from the supermarket. The fridge gets refilled. The owner continues drinking.>

c) Terminate the experiment (kill all the agents).

The experiment shows that the distribution model implemented works as expected in this case. Besides, note that the two varieties of the internal action `.send`, described in Section 3.2.1, are successfully used for both intra- and inter-container communication.

6.3.2 Experiment 2: intra-container fault tolerance

This experiment seeks to test the performance of the fault tolerance mechanisms implemented when they involve agents running in the same container. Briefly, it follows an execution flow in which the agents owner and robot stop working (die) several times. Anytime the robot is not alive, the owner must notice it and start it again (even if the owner killed it). If the owner agent dies, the robot must be aware of it and print a message. The steps and output of the experiment are:

a) Connect the hosts and start all the agents but the robot.

< The owner immediately realizes that the robot agent is not alive and starts it. The owner starts drinking beer. The robot refills the fridge whenever it is empty. The sequence continues until the robot tells the owner not to drink any more beer. Upon reception of the message from the robot, the owner kills it. The owner becomes immediately aware that the robot is dead and restarts it. The owner continues drinking beer, killing and restarting the robot whenever it refuses to bring more beer, until it surpasses its physical limit and passes out (the agent dies). The robot notices the death of the owner immediately after it happens and prints a message.>

b) Terminate the experiment (kill all the agents).

Note that the failures detected in this experiment are all of type `dead_agent`. This experiment shows that the agents are immediately aware of the death of the agents they monitor, at least in the same container, and react properly (the owner restarting the robot and the robot printing a message).

6.3.3 Experiment 3: inter-container fault tolerance

This third experiment tests the performance of the fault tolerance mechanisms when the system is distributed. In its execution flow the three types of agent failures described before are generated. The correct detection of these failures can be checked through the reactions of the agent robot. If the agent supermarket is not alive but its container is, the robot must say: “I cannot find the supermarket in the shopping mall”. If the agent supermarket dies, the robot says “The supermarket has just closed”.Finally, if the agent supermarket becomes unreachable, the robot must say: “I cannot find the shopping mall”. The steps followed and the relevant observable behaviour of the agents are:

a) Connect the hosts and start all the agents but the supermarket.

<The robot is immediately aware that there is no agent with symbolic name supermarket in the container shopping_mall and prints the message: “I cannot find the supermarket in the shopping mall”. The owner drinks beer until the fridge is empty, then gets asleep and wakes up periodically.>

b) Start the agent supermarket

<The fridge gets filled again and the owner continues drinking beer.>

c) Disconnect the hosts

< The fridge becomes empty. Then, the robot tries to order beer from the supermarket for a while. After about 60 seconds after the disconnection of the hosts (due to Erlang implementation issues, the disconnection of two nodes running in different hosts is detected, by default, from 45 to 70 seconds after the network disconnection actually happened), the robot realizes that the supermarket is unreachable and prints the message: “I cannot find the shopping mall”.>

d) Reconnect the hosts.

<The fridge gets filled again and the owner continues drinking beer.>

e) Kill the agent supermarket.

<The robot agent is immediately aware of the death of the agent supermarket and prints the message: “The supermarket has just closed”>

f) Start the agent supermarket.

<The fridge gets filled again and the owner continues drinking beer.>

g) Destroy the container `shopping_mall`

<The robot agent is immediately aware of the death of the agent supermarket and prints the message: “The supermarket has just closed”>

h) Terminate the experiment (kill all the remaining agents).

The experiment shows that all the different failures are correctly detected by the implementation of our proposed fault tolerance system.

7. Conclusion and Future Work

In this paper we have described an extension to the Jason multiagent systems programming language, which provides a new distribution model and constructs for fault detection and fault recovery. Moreover, our implementation of Jason in Erlang – eJason – contains a prototype implementation of the extension.

The addition of a proper, Jason based, distribution model to Jason systems, removes the need of interfacing to third-party software such as e.g. JADE. The addition of fault-detection and fault tolerance mechanisms to Jason addresses one of the key issues in the development of robust distributed systems. Concretely, these mechanisms allow the detection of failures in a multiagent system caused by malfunctioning hardware (computers, network links) or software (agents).

The distribution model and fault-tolerant mechanisms are inspired by Erlang, an actor based language which is increasingly used in industry to develop robust distributed systems, in part precisely because of the elegant approach to fault detection and fault tolerance. Somewhat surprising, Erlang and Jason share many common features, and this has made the design and implementation of the distribution model and the fault-tolerant mechanisms a rather straightforward task.

The lines of future work are many. First we need to provide higher-level agents (components) that ease the task of programming fault tolerant systems. In Erlang this is accomplished, for example, by providing a general component (the supervisor) to supervise and, if need be, restart failing processes. In Jason this will correspond to a monitoring agent. We expect to extend the usual notion of supervision (responding to termination of agents) with a notion of "semantic termination", i.e., detecting when an agent is still alive but no longer contributing useful results. In this same line, we plan to elaborate use cases showing how higher-level fault-tolerance properties could be implemented in eJason. Nevertheless, some issues like the preservation of the state of an agent across failures or the identification/development of useful high-level constructs must be dealt with first.

Second, we should develop at least a "semi-formal" semantics for this extension of Jason, describing exactly the behaviour of the internal actions (sending, monitoring, etc).

Third, we need to evaluate this extension on further examples, to ensure that the new internal actions have sufficient expressive power, and are integrated well enough in Jason, to permit us to design distributed multiagent systems cleanly and succinctly, and in the general spirit of BDI reasoning systems.

Considerably more speculative, we are not certain that the model of distribution offered by this extension is sufficient to model real-world multiagent system with respect to agent mobility. It might be necessary, for instance, to permit an agent to migrate between different process containers. This will complicate the implementation, but should not be impossible

With respect to eJason, there is a need to complete the implementation with regards to environment (perception of external events, etc) handling, with all its related functionality. Moreover, we plan to permit the interoperability between the agents running in eJason and agents belonging to other (e.g., JADE based) agent platforms. Therefore, we will study how to best interface eJason with a FIPA based agent platform. For instance, the concept of Directory Facilitator appears similar to the global registry service provided by Erlang.

References

[1] *Foundation for Intelligent Physical Agents, Agent Communication Language. http://www.fipa.org/specs/fipa00061/SC00061G.html.*

[2] G. Agha and C. Hewitt. Actors: A conceptual foundation for concurrent object-oriented programming. In *Research Directions in Object-Oriented Programming*, pages 49–74. 1987.

[3] J. Armstrong. Programming Erlang: Software for a concurrent world). The Pragmatic Bookshelf, 2007.

[4] F. Bellifemine, F. Bergenti, G. Caire, and A. Poggi. JADE - a Java agent development framework. In R. H. Bordini, M. Dastani, J. Dix, and A. E. Fallah-Seghrouchni, editors, *Multi-Agent Programming*, volume 15 of *Multiagent Systems, Artificial Societies, and Simulated Organizations*, pages 125–147. Springer, 2005. ISBN 0-387-24568-5.

[5] F. L. Bellifemine, G. Caire, and D. Greenwood. *Developing Multi-Agent Systems with JADE (Wiley Series in Agent Technology)*. Wiley, Apr. 2007. ISBN 0470057475. URL `http://www.worldcat.org/isbn/0470057475`.

[6] R. H. Bordini, M. Wooldridge, and J. F. Hübner. *Programming Multi-Agent Systems in AgentSpeak using Jason (Wiley Series in Agent Technology)*. John Wiley & Sons, 2007. ISBN 0470029005.

[7] F. Cesarini and S. Thompson. *Erlang Programming*. O'Reilly Media, 2009. ISBN 978-0-596-51818-9—ISBN 10:0-596-51818-8.

[8] Á. Fernández-Díaz, C. Benac-Earle, and L.-A. Fredlund. ejason: an implementation of jason in erlang. Proceedings of the 10th International Workshop on Programming Multi-Agent Systems (ProMAS 2012), pages 7–22, 2012.

[9] A. S. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. In W. Van de Velde and J. W. Perram, editors, *Agents Breaking Away*, volume 1038 of *LNCS*, pages 42–55. Springer, 1996. ISBN 978-3-540-60852-3. doi: 10.1007/BFb0031845. URL `http://www.springerlink.com/content/5x727q807435264u/`. 7th European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW'96), Eindhoven, The Netherlands, 22-25 Jan. 1996, Proceedings.

[10] A. S. Rao and M. P. Georgeff. BDI agents: From theory to practice. In *In Proceedings of the first Interntional Conference on Multi-Agent Systems (ICMAS-95*, pages 312–319, 1995.

[11] M. Wooldridge. Reasoning about rational agents. MIT Press, 2000.

A. Appendix A

```
contents(beer,4).

+!give(Item)[source(Who[container(Where)])]:
    contents(Item,Stock) &
    Stock > 0 <-
   .print("Giving beer to ",Who,
          " in ", Where);
   NewStock =  Stock + -1;
   -+contents(Item,NewStock);
   .print("Beers left:  ",
          NewStock);
   .send(Who[container(Where)],tell,
         holding(Item)).

+!give(Item)[source(Who)] :
    contents(Item,Stock) &
    Stock < 1 <-
   send(Who,tell,
        no_more(Item)).

+delivered(Item,Qtd, OrderId):
    contents(Item,Stock)<-
   -delivered(Item,Qtd,OrderId);
   .print("Received ", Qtd,
          " units of ", Item);
   NewStock = Qtd + Stock;
   -+contents(Item,NewStock).
```

Figure 7. Code for agent "fridge"

```
last_order_id(1).

+!order(Product,Qtd)[source(Ag[container(Container)])]:
   last_order_id(N) &
   OrderId = N +1<-
  -+last_order_id(OrderId);
  .print("Sending ", Qtd, " units of ",
         Product, "to ", Container);
  .send(fridge[container(Container)],tell,
        delivered(Product,Qtd,OrderId)).
```

Figure 8. Code for agent "supermarket"

```
physical_limit(21).
beers_drunk(1).
inactive(robot).

!monitor(robot).

+!get(beer):
     physical_limit(Limit) &
     beers_drunk(Drunk) &
     Drunk <= Limit &
     not inactive(robot)<-
   .send(robot, achieve,
         has(owner,beer));
   .print("Getting a beer").

+!get(beer):
      physical_limit(Limit) &
      beers_drunk(Drunk) &
      Drunk >Limit &
      not inactive(robot)<-
     .print("I feel strangg..");
     .kill_agent(owner).

+has(owner,beer) : true <-
   ?beers_drunk(Beers);
   .print("I got my beer number ",
           Beers, ". Yeepe!");
   +remaining_sips(3);
   !!drink(beer).


+!drink(beer) :
     remaining_sips(Sips) &
     Sips > 0 <-
    NewSips = Sips + -1;
    .print("Sip");
    -+remaining_sips(NewSips);
    ?remaining_sips(X);
    !!drink(beer).

+!drink(beer) :
     remaining_sips(Sips) &
     Sips < 1 <-
    ?beers_drunk(Beers);
    NumBeers = Beers +1;
    -+beers_drunk(NumBeers);
    -has(owner,beer);
    .print("Finished beer!");
    !!get(beer).
```

```
+msg(M)[source(Ag)] : true <-
    .print(Ag," says: ",M);
    -msg(M);
    .print("Unacceptable!");
    .print("Let's restart my ",
           "mechanic friend.");
    .kill_agent(Ag).

+closed(supermarket): true <-
    -closed(supermarket);
    !sleep.

+no_more(beer): true <-
    -no_more(beer);
    !sleep.

+!sleep: true <-
    .print("No beer means nap ",
           "time Zzz.");
    -closed(supermarket);
    .wait(2000);
    !!get(beer).

+agent_down(robot): true <-
    +inactive(robot);
    -agent_down(robot);
    .create_agent(robot,robot);
    .print("robot has stopped ",
           "working. Start anew!");
    !monitor(robot).

+!monitor(robot):true <-
    -inactive(robot);
    .monitor(robot);
    !!get(beer).
```

Figure 9. Code for agent “owner”

```
consumed(beer,0).
at(robot,owner).
limit(beer,10).
address(supermarket,
   'shopping_mall@babel.ls.fi.upm.es').

too_much(Beverage) :-
     limit(Beverage,Limit) &
     consumed(Beverage,Consumed) &
     Consumed > Limit.

!monitor(supermarket).

+!has(owner,beer):
     not too_much(beer) <-
    !at(robot,fridge);
    .send(fridge,achieve,give(beer)).


+!has(owner,beer):
     too_much(beer)<-
    ?limit(beer,Y);
    .print("The Department of Health ",
          "does not allow me to give",
          " you more than ",Y,
          " beers a day! I am very ",
          "sorry about that!");
    ?consumed(beer,X);
    .print("Consumed ",X,
          " beers when the limit is ",
          Y, ".");
    .send(owner,tell,
         msg("I am very sorry!")).


+!has(owner, beer):
     closed(supermarket) <-
    .send(owner,tell,
         closed(supermarket));
    !monitor(supermarket).


+holding(beer) : true <-
    -holding(beer);
    !at(robot,owner);
    ?consumed(beer,X);
    Y = X +1;
    -+consumed(beer,Y);
    .send(owner, tell, has(owner,beer)).


+no_more(beer) :(not
     closed(supermarket)) &
    address(supermarket,SupContainer) <-
   -no_more(beer);
   .print("Fridge is empty.");
   .send(supermarket[container(SupContainer)],
        achieve, order(beer,5));
   .send(owner,tell,no_more(beer));
   !at(robot,fridge).
```

```
+no_more(beer):
    closed(supermarket)<-
   -no_more(beer);
   .print("Fridge is empty.
          And supermarket is closed.");
   .send(owner,tell,
          closed(supermarket));
   !monitor(supermarket);
   !at(robot,fridge).

+!at(robot,P):
    at(robot,P) <-
   true.

+!at(robot,P):
    not at(robot,P)<-
   -+at(robot,P).


+!monitor(Agent):
    address(Agent,Container) <-
   -closed(supermarket);
   .monitor(Agent[container(Container)]).

+!monitor(owner): true <-
   .monitor(owner).

+agent_down(supermarket)
     [reason(unknown_agent)]: true <-
   +closed(supermarket);
   .print("I cannot find the supermarket
           in the shopping mall").


+agent_down(supermarket)
     [reason(unreachable_agent)]: true <-
    +closed(supermarket);
    .print("I cannot find the shopping mall").


+agent_down(supermarket)
     [reason(dead_agent)]: true <-
    +closed(supermarket);
    .print("The supermarket has just closed").


+agent_down(owner): true <-
   .print("Oh, oh. My master has passed out.").
```

Figure 10. Code for agent "robot"

Programming Abstractions for Integrating Autonomous and Reactive Behaviors: An Agent-Oriented Approach

Alessandro Ricci
University of Bologna, Italy
a.ricci@unibo.it

Andrea Santi
University of Bologna, Italy
a.santi@unibo.it

Abstract

The integration of autonomous and reactive behavior is a relevant problem in the context of concurrent programming, related to the integration of thread-based and event-driven programming. From a programming paradigm perspective, the problem can not be easily solved by approaches based on object-oriented concurrent programming or by the actor model, being them natively based on the reactivity principle only. In this paper we tackle the problem with agent-oriented programming, using an agent-oriented programming language called simpAL.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming; D.3.2 [*Programming Languages*]: Language Constructs and Features

General Terms Languages, Design

Keywords event-driven programming, agent-oriented programming, actors

1. Introduction

In modern concurrent programming, many interesting problems and applications call for developing software components capable of integrating a process-oriented autonomous behavior with an event-driven, reactive one. A simple example is a web crawler that has to search pro-actively information over the Internet and at the same time must be able to react to asynchronous inputs generated by the user through a GUI, either to interrupt the crawler or to dynamically refine its research. Another example is given by cooperative distributed algorithms like Ricart-Agrawala's or Token-Ring's [8], for distributed mutual exclusion and critical section. In these algorithms, the behavior of each distributed node typically includes both an autonomous part, periodically entering in critical section (CS), and a reactive part, which needs to receive and send messages aside to the first one to ensure the correct coordination among the nodes. The two parts then need to cooperate, since the behavior of the reactive part can depend on the state of the pro-active one (i.e., being in CS or not). The same situation can be found in producers-consumers architectures in which producers and consumers, while doing their job, need to be reactive to some kind of asynchronous events—e.g., producers should stop producing as soon as some condition related to consumers occurs. This capability is often required also in real-time concurrent programs doing some control task – as exemplified by the cruise control system example [17]. The software controller has to continuously act on the environment to keep or maintain some condition (e.g., the value of the speed) and at the same time it must react to user inputs (e.g. pressing the brake) and possibly change operation modality.

In concurrent programming literature, the problem of integrating autonomous and reactive behaviors is strongly related to the one contrasting thread-based and event-based programming [27, 35], and to those works that look for unifying the approaches [18]. In this paper we take a programming paradigm viewpoint in looking at the problem, taking as the background context object-oriented concurrent programming [3, 13] and actor-based approaches [2]. Being based on a pure *reactivity principle* [13, 23], actors – as objects as well – do not provide native means to effectively integrate also pro-activity, so actor-based solutions to this problem – as will be clarified in Section 2 – suffer of a weak abstraction and modularity. So the question at the core of this paper is: can we identify proper programming/computation abstractions that integrate these two aspects at the conceptual and foundation level, going beyond the reactivity principle?

In this paper we develop an answer to this question based on agent-oriented programming, using an agent programming language called simpAL. Agents as defined in simpAL can be conceptually conceived as an extension (or specialization, depending on the viewpoint) of actors introducing high-level programming concepts (main examples are tasks and plans) that aim to ease the design and development of concurrent and distributed programs. A core feature of the agent model is the control architecture, that allows for natively integrating autonomy and reactivity.

The remainder of the paper is organized as follows: in Section 2 we describe the problem in more detail, using some toy examples and state-of-the-art actor technologies; then, in Section 3 we introduce our agent-oriented model and simpAL, describing in particular the features of the agent control loop compared to the actor event loop; in Section 4 we describe programming examples in simpAL of increasing complexity, integrating an autonomous and reactive behavior. In Section 5 we consider a real-world example, discussing its design and programming in simpAL. Finally, in Section 6 we discuss related work and in Section 7 we conclude the paper providing some remarks on the performance.

2. Background and Problem Statement

The dualism between reactivity and autonomy in the context of concurrent object-oriented programing is discussed in [13]. Both actors and objects are based on reactivity and the *reactivity principle* [3, 13, 23]. They are reactive in the sense that they react to an event, i.e. the receipt of a message. The only way to activate an object or an actor is by sending a message. In [13] this is opposed to the idea of a *process*, or a pure autonomous behavior, that starts processing as soon as it is created.

The integration of object with process (the concept of active object) raises the issue of whether reactivity will be preserved or shadowed by the autonomous behavior of the process. Two broad families are identified[13]: *(i) reactive active objects* – these approaches adhere to the reactivity principle (such as actors); *(ii) autonomous active objects* – in these approaches, the active entity may compute before being sent a message. Although the models are opposite they can easily simulate each other [13]. On the one side, a reactive active object can have a method whose body in an endless loop, turning it into an autonomous active object after receiving a corresponding message. On the other side, an autonomous active object whose activity is to keep accepting messages actually models a reactive active object.

However, this is not useful when dealing with problems that call for exploiting both of them in an *integrated* way. Abstracting from the details, all these problems have some kind of process doing pro-actively actions to accomplish some (possibly long-running) task that requires also to react to some asynchronous events from their environment.

2.1 An Abstract Example

In the remainder of the section we analyze the problem by considering an abstract example that captures some core issues. We will use actors as reference model and related state of the art programming technology.

```
public class TestActor0 extends Actor {
  private int c = 0;

  @message
  public void doTaskT() {
    ta();
    tb();
    tc();
  }

  @message
  public void react() {
    call(stdout, "println","react! "+c);
  }

  private void ta(){ c = c + 1; }
  private void tb(){ c = c + 1; }
  private void tc(){ c = c + 1; }
}
```

Figure 1. A first solution using ActorFoundry.

```
public class TestActor1 extends Actor {
  private int c = 0;

  @message
  public void doTaskT() {
    send(self(), "doingTa");
  }

  @message
  public void doingTa() {
    send(self(), "doingTb");
    ta();
  }

  @message
  public void doingTb() {
    send(self(), "doingTc");
    tb();
  }

  @message
  public void doingTc() { tc(); }

  @message
  public void react() throws RemoteCodeException {
    call(stdout, "println","react! "+c);
  }
  ...
}
```

Figure 2. A solution in ActorFoundry that does not shadow reactivity.

Let's consider a task T which is supposed to be long-term, articulated in a sequence of three sub-tasks: Ta, Tb, Tc—for sake of simplicity we suppose initially that these sub-tasks are fully computational, without interactions. This constitutes the autonomous/pro-active part of the job. Then, the task requires to promptly react to a message react that could be sent in any moment while doing T, and upon receiving the message the actor must do a sub-task Td. In this first example, we assume that such a sub-task Td is not going to alter the execution of the sub-tasks, but simply prints the `react!` message in standard output.

Figure 1 shows a first solution in ActorFoundry [22], a Java-based framework which we will consider in the following as reference technology for implementing pure actor solutions. Note that we could use any framework/language

```
test_actor() ->
    self() ! doTaskT, loop(0).

loop(C) ->
    receive
      doTaskT ->
        C1 = ta(C),
        self() ! doingTb,
        loop(C1);
      doingTb ->
        C1 = tb(C),
        self() ! doingTc,
        loop(C1);
      doingTc ->
        C1 = tc(C),
        loop(C1);
      react ->
        io:format("react! ~w~n", [C]),
        loop(C)
    end.

ta(C) -> C+1.
tb(C) -> C+1.
tc(C) -> C+1.
```

Figure 3. A solution in Erlang.

strictly implementing the actor model. The only peculiarity that we exploited of ActorFoundry is the `call` primitive, which realizes a request-reply message exchange pattern. The problem with this solution is that given the macro-step semantics adopted by the actor model, the actor can react to the the `react` message only after fully executing the body of the method `doTask` executed when receiving the corresponding message. So the message printed on standard output is always `react! 3`. In this case reactivity is shadowed.

In order to be reactive while doing the tasks, the autonomous behavior of the actor must necessarily be broken in sub-behaviors so as to allow the actor event loop to consider the receipt of the react message. An example is shown in Figure 2. The problem in this case is the fragmentation of the code in handlers, which does not necessarily corresponds to a good modularization from the point of view of organization of the autonomous behavior. One is forced to artificially break the behavior so that the event loop can take the control and check the availability of messages possibly sent by other actors. Besides, self-sending messages is needed to proceed the computation, in the case that no messages are available, not to get stucked. This is clearly a programming trick, decreasing the level of abstraction used to describe the strategy identified at the design level.

No substantial improvements can be obtained if we consider actor approaches based on explicit acceptance of messages (following the classification discussed in [13]), i.e. providing a `receive` primitive to explicitly retrieve messages from the mailbox (and not encapsulating then the event-loop). Figure 3 shows the solution in Erlang [7], which is a main representative case—whose model inspired other more recent technologies, such as Scala Actors [18]. Here the fragmentation occurs by splitting the autonomous behavior into the arms of the receive primitive.

No improvements can be obtained neither if we consider the programming abstractions that have been proposed in literature upon the basic actor model – e.g., local synchronization constraints, synchronizers [5]. This because all such extensions are finally targeted to ease the management of messages, so improving the programming of the reactive part. The same applies for extensions introducing mechanisms to overcome handler/callback fragmentation—by means, for instance, of join continuations [4] or promises [26]. These mechanisms are effective to improve the organization of the callbacks handling asynchronous events, avoiding obscure nesting.

A radically different approach to overcome the problem could be using a network of cooperating actors instead of one, in which each actor has either a purely proactive or a purely reactive job. For instance: an actor for each sub-task to be executed and one "input" actor responsible to receive the react message. In that case, the input actor could immediately react and print the message – without caring about the other actors, which go on doing their job. However the problem persists as soon as we consider a slightly more complex case, in which by receiving the message one need to have some immediate effect on the ongoing sub-tasks, such as stopping them or adapting them according to some new information. This would require anyway sending a message to the actor who is in charge of autonomously doing the sub-task, which should be able to react.

In this paper we develop a solution based on a programming abstraction layer which would allow to express in a modular way an autonomous behavior – targeted to the fulfillment of some task – which is capable to react to asynchronous events that can occur while doing the task. The programming model should allow to keep the same level of modularization identified at design time. More generally, we aim at identifying a foundational computation model that would capture the essence of being pro-active, besides reactivity.

3. An Agent-Oriented Approach: simpAL

Agent-Oriented Programming has been originally introduced in (Distributed) Artificial Intelligence (AI) contexts [32], with the purpose of finding appropriate computation models and architectures to design *intelligent* software entities exhibiting some level of autonomy in achieving complex goals [21]. Most of the available agent programming languages and frameworks are based on models/architectures explicitly inspired to the BDI (Belief-Desire-Intention) cognitive model of human practical reasoning [12].

Differently from this AI perspective, in our research work in general we are exploring the definition and use of agent-orientation as a *programming paradigm* for concurrent and distributed systems, providing features that extend (or specialize, depending from the view point) Object-Oriented Programming (OOP) and actor-based approaches [31]. To this purpose we designed simpAL [30], a programming language that provides agent-oriented first-class abstractions to deal with aspects related to concurrency, I/O, asynchronous event management, distribution—and then using OOP for all those parts that concern pure computation and data structure modeling. In this paper we discuss in detail those aspects of the simpAL agent model that allow for natively integrating autonomous and reactive behavior.

3.1 From Reactivity to Pro-Activity Principle

Agents in simpAL can be generally defined as software components which are designed to accomplish autonomously *tasks*, both by exploiting the computational environment where they are situated and by communicating with other agents. Therefore, differently from objects and actors, agents are not based on the reactivity principle, since they do something not necessarily only when receiving a message, but because they have one or multiple tasks to accomplish. In that sense, we can say that they are based on a *pro-activity* principle, autonomously choosing and executing *actions* in order to fulfill some tasks. Autonomously means that they fully encapsulate the decision about what to do and the control of their execution.

Actions are defined by the environment, which is modularized in terms of first-class abstractions called *artifacts*, conceptually representing tools and resources shared and used by agents to do their individual and cooperative tasks. Artifacts are used to model non-autonomous computational entities of the program that can be dynamically created and disposed by agents, and that provide some kind of functionality to agents. Examples of artifacts are a blackboard, a bounded-buffer, a shared counter. Artifacts provide operations (e.g., `put` and `get` for a bounded buffer) which correspond to the actions that agents can do. So the repertoire of actions that an agent can do is dynamic and depends on the set of artifacts available in the environment.

Along with pro-activity, agents typically need to be also reactive, that is: in order to fulfill a task, they have to act upon the environment but also to react to asynchronous events that concern, for instance, some change in the environment observable state. This is represented by artifact *observable properties*, i.e. state variables that can be modified by the execution of operations. Like actors, agents are reactive also to messages sent by other agents. It is worth noting that pro-activity could be defined in terms of being reactive but not only to events – e.g. the assignment of a task to do, communicated by another agent – but to also a *state*, i.e. having a task which has not been fulfilled yet.

3.2 From Actor Event Loop to Agent Control Loop

In order to integrate pro-activity and re-activity, the *control architecture* adopted in agents extends the actor one, based on event-loop. From a behavioral point of view, actors can be conceived as processes continuously executing the following loop, sometimes called *event loop* [26][1]:

Algorithm 1 Actor Event Loop

```
1: while $true$ do
2:     $msg \leftarrow$ PICKMSGFROMMSGQUEUE()
3:     $method \leftarrow$ SELECTHANDLER($msg$)
4:     EXECUTE($method$)
5: end while
```

All the extensions introduced in literature, such as making it explicit the receive primitive or providing a way to order the messages to receive, can be finally translated into this basic loop [6]. Two aspects are important in particular for this paper: *(i)* if no messages are available in the queue, the loop is blocked (this is consistent with the reactivity principle); *(ii)* the method selected for handling a message must be executed until completion, atomically, before fetching the next message—this is also called *macro-step semantics* [6].

The loop characterizing agent execution (agent *control loop* or *reasoning cycle*) in simpAL – inspired by the BDI model/architecture [29] – can be considered an extension of the actor event loop and is composed by three conceptual stages – *sense*, *plan*, *act* – that are cyclically executed:

Algorithm 2 simpAL Agent Control Loop

```
1: while $true$ do
2:                                                        ▷ SENSE stage
3:     if external events queue not empty then
4:         $ev \leftarrow$ PICKEXTEVENT()
5:         UPDATEAGENTSTATE($ev$)
6:     end if
7:                                                        ▷ PLAN stage
8:     if new tasks todo then
9:         for each new task to-do $task$ do
10:            $plan \leftarrow$ SELECTPLAN($task, belBase, planLib$)
11:            CREATENEWINTENTION($plan, task$)
12:        end for
13:    end if
14:    $actList \leftarrow []$
15:    for each ongoing intention $i$ do
16:        if TASKFULFILLED($i$) then
17:            DROPINTENTION($i$)
18:        end if
19:        $actList \leftarrow actList +$ SELECTACTIONS($i, belBase$)
20:    end for
21:                                                        ▷ ACT stage
22:    for each action $act$ in $actList$ do
23:        EXECUTE($act$)
24:    end for
25: end while
```

[1] Event loops are widely recognized to be part of the actor model by research community working on actors but Carl Hewitt: *"[...] In any case, 'event loop' is confusing terminology because there can be "holes in the cheese." [Hewitt and Atkinson 1979, ActorScript], which means code is just nested expressions (i.e. no loop)..."* `http://lambda-the-ultimate.org/node/4453`

In the sense stage, the state of the agent is updated with what perceived from the outside, fetching one event – if available – from the external event queue. Such a state includes agent *beliefs* – i.e., the informational part of the state, composed by private state variables, possibly keeping track of the observable state of the artifacts the agent is using/observing – ongoing tasks and tasks to do. The event can concern either some change in the observable state of artifacts, or a new message sent by another agent, or the notification of the completion with success or failure of an action executed by the agent on the environment.

In the plan stage, first, if there are new tasks to do, then for each one a *plan* is selected from the agent *plan library* to handle the task and a new *intention* is instantiated—i.e., a new activity committed to the fulfillment of the task, keeping track of the plan in execution. Plans as programming abstraction will be described in next section: they are module of *procedural knowledge* [29], composed by a set of action rules that specify what to do and when to do it. For each ongoing intention, all the actions which can be executed – according to the plans and the current beliefs of the agent – are collected. Intentions that achieved their goal are dropped. Finally, in the act stage, all the collected actions are executed. Internal actions – i.e., actions accessing/modifying the internal state of the agent – are executed atomically in this stage, in one cycle. External actions instead – i.e., actions that correspond to the execution of an operation provided by some artifact of the environment – are just started. The completion of such events may be perceived asynchronously in the sense stage in future cycles.

Two main differences compared to the actor event loop are important here. First, an agent can execute a cycle even if *there are no external events to process*. This happens when the agent has one or more intentions about tasks to be executed – that have been previously assigned – and following the related plans some actions must be selected pro-actively. For instance: a plan stating that some action a must be continuously selected and executed, like a simple non terminating process. This is consistent with the idea that an agent follows the *pro-activity principle*. So, conceptually, an agent doing some task(s) is *never blocked*—always cycling until the task(s) have been achieved or failed. At the same time, an agent is reactive and event-driven: an event perceived in the sense stage can result in updating some agent beliefs and this could trigger the selection of some actions in the plan stage.

Second, intentions – i.e., plans in execution – are not meant to be fully executed and completed in one cycle: typically their execution requires multiple cycles, each one selecting zero or one or multiple actions to be executed (depending on the plan). By doing an analogy between methods in the actor case and plans, this means that in the agent case the *macro-step semantics* is relaxed, or it has finer granularity, which is at the level of the actions composing a plan.

```
roledef = "role" id "{" { beliefdef } { taskdef } "}" ;
beliefdef = id ":" type ;
taskdef = "task" id "{" [ inputparam ] [ outputparam ]
               [ understands ] [ talksabout ] "}" ;
inputparam = "input-params" "{" { id ":" type } "}" ;
outputparam = "output-params" "{" { id ":" type } "}" ;
understands = "understands" "{" { id ":" type } "}" ;
talksabout = "talks-about" "{" { id ":" type } "}" ;
```

Figure 4. Syntax of role and task definition.

```
role RoleR {
  task Booting { }
  task TaskT {
    understands {
      react: boolean;
}}}
```

Figure 5. The definition of the role `RoleR` used in the example.

These features together allow to tackle the integration of the autonomous and reactive behavior directly at the foundation level—raising performance issues, that will be discussed in Section 7. Now the point is: how to program the plans. We need to introduce a proper model which would be effective at the programming level in specifying such procedural knowledge, eventually integrating an imperative and declarative style in describing *when* to do *what*.

3.3 Tasks and Plans in simpAL Programming

In simpAL tasks and plans are the main first-class constructs to specify and organize agent behavior. The notion of task is used to define declaratively a job that has to be done, while the notion of plan is used to define the recipes to accomplish some task, in some condition. So tasks represents *what* to do, plans *how* to do it.

Tasks are typed data abstraction and the definition of a task type can be done in the context of *roles*. Roles are used to define the notion of type for agents, grouping together the definition of set of task types. So an agent of type R – i.e., which is declared to play a role R – at runtime can be dynamically assigned by other agents of any instance of task T whose type is declared in the role R.

Figure 4 sketches the EBNF syntax of role/task definition in simpAL. A task is described in terms of: input parameters – information that must be specified when the task is assigned to some assignee agent; output parameters – information that must be specified when the task has been completed by the assignee agent; messages that can be understood by the task assignee (`understands` block); messages that can be sent by the task assignee (`talks-about` block). The complete and detailed description of the task model, along with all the other aspects of simpAL, is outside the scope of this paper: the interested reader can find these information elsewhere [30]. Here we focus only on those aspects that are relevant for the integration of autonomous and reactive behavior. Figure 5 shows the definition of the role

```
agent-script TestScript implements RoleR {
  c: int = 0;

  plan-for TaskT  {
    do-task new-task Ta();
    do-task new-task Tb();
    do-task new-task Tc()
  }

  plan-for Ta { c = c + 1
}
  plan-for Tb { c = c + 1
}
  plan-for Tc { c = c + 1
}

  task Ta {}
  task Tb {}
  task Tc {}
}
```

Figure 6. A simple simpAL script.

```
1  agent-script TestScript implements RoleR {
2    c: int = 0;
3
4    plan-for TaskT completed-when: is-done tc {
5      do-task new-task Ta();
6      do-task new-task Tb();
7      do-task new-task Tc() #tc
8
9      when told this-task.react => using: console@main {
10       println(msg: "react! "+c)
11   }}
12   ...
13 }
```

Figure 7. A simple simpAL script with a plan integrating both autonomous and reactive behavior.

RoleR, including the task type TaskT mentioned in the previous section. An agent who is playing a role RoleR can be dynamically requested by other agents to do instances of TaskT tasks; in doing a TaskT task, the agent can receive messages telling him to react[2].

Plans – which specify how to concretely fulfill tasks – are defined in *scripts*, which represent modules of agent behavior collecting sets of plans useful to play some role R. A script contains both the definition of a set of plans useful to accomplish the tasks of the role declared to be implemented by the script, and a set of beliefs that can be accessed by all the plans declared into that script. By loading a script, an agent adds to its belief-base the beliefs declared in the script and the plans of the script to its plan library[3].

The plan model adopted for plan definition is a key aspect for this paper, discussed in next section.

[2] i.e., telling him that the value of the react information, that in this case can be true or false.

[3] The description of the management of the conflicts that can arise is outside the scope of this paper.

4. Integrating Autonomous and Reactive Behavior in Plans

A plan in simpAL can describe any arbitrary combination of sequences of actions with *reactions*, i.e. actions that must be taken if/when/every time some condition holds.

4.1 Sequential Behaviors and Reactions

Sequential behaviors can be expressed by sequences of actions separated by a ;. Figure 6 shows the simpAL version of the first example, purely autonomous, without reactivity. The definition of a plan includes the specification of the type of task for which the plan can be used (TaskT), a context condition specifying when the plan can be used (not used in the example) and a plan *body*, which defines the plan behavior. In the example, the plan for fulling the task TaskT breaks the task in three private sub-tasks Ta, Tb, Tc, to be fulfilled in sequence. The body of the plan is a simple sequence of do-task actions, which is one of the available actions in simpAL to manage tasks. do-task action in particular self-assigns a task and completes when the task has been completed. The parameter of the do-task is an instance of a type of task (Ta, Tb, Tc), defined in this case the in the script being them private. The script includes also the simple plans to manage them, where a global belief c is incremented.

Then, reactivity can be added by including action rules that specify reactions to some specific event/condition—called *when* part. The syntax of action rule in EBNF with the *when* part specified is:

```
("when"|"every-time")[Ev][":"Cond]"=>" Act [#lbl] |
 "always" "=>" Act [#lbl]
```

Ev is an event template, specifying what kinds of event can trigger the rule; Cond is a boolean expression, specifying the condition over the belief base that must hold for a triggered rule to be applicable; finally Act is the action to be executed if the rule is triggered and is applicable, and lbl is a label to identify the invoked action. So informally, an action rule states that when the general condition expressed by the event and the condition holds given the current state of the agent, then the specified action can be chosen to be executed. The keyword when can be used to specify that the rule can be selected and triggered only once; every-time otherwise, to mean the general case that the rule can be triggered every time that the specified event/condition holds. The action to be executed can be an internal action {...} instantiating a new action rule block. The keyword always is syntactic sugar for everytime true, i.e. the action can selected at each cycle.

A first example is given in Figure 7, which extends the previous example adding a first level of reactivity. The rule in lines 9–11 says that when the agent receives a message about reacting (told this-task.react is the event), it eventually interrupts the sequential course of action of the current intention and executes a new action rule block. Each intention has a stack of action blocks, which is initialized

```
agent-script MyClient implements Client {
  inputGUI: UserView
  ...
  plan-for UseServices  using: inputGUI
                        completed-when: stopped {
    always =>
           completed-when:is-done reqC || stopped
           using:serviceA,serviceB {
      doReqA() #reqA
      doReqB() #reqB
      when is-done reqA && is-done reqB
                              => using:serviceC {
        doReqC() #reqC
}}}}
```

Figure 8. Another example of plan behavior with actions and reactions.

with the plan body when the intention is created: each time a new action block is instantiated, it is pushed on top of the stack. In the reaction block the agent prints the `react` message on standard output using a `console` artifact available in the `main` workspace. The attribute `using:` specifies which artifacts will be used (and observed) inside an action rule block. At runtime, when entering a block where an artifact is used, automatically the observable properties of the artifact are continuously perceived and their value is stored in corresponding beliefs in the belief base—updated in the sense stage of the agent execution cycle. The block is completed after the agent perceives the completion of this action and then popped from the stack of the intention. The plan is completed when the last action labeled with `tc` completes. The attribute `completed-when:` is used to declaratively specify when an action rule block can be considered completed.

In this example, the action rule block is given by a simple sequence of actions and a reaction. Actually, also the sequence of actions is – under the hood – realized by a set of action rules, in which the *when* part is set up in order to obtain such a specific control flow. So an action rule block is always uniformly represented by a set of action rules, expressing patterns of behavior possibly mixing sequences and reactions. Then, some syntactic sugar is provided to specify sequences (as rules represented only by actions, separated by a `;`), to specify rules that should be triggered only once (`when` keyword), and other frequent patterns. As a further simple example, Figure 8 shows a plan in which the agent repeatedly does concurrently (in the same cycle) two requests on two services (modeled as two artifacts: `serviceA`, `serviceB`) and then does a further request on a third service as soon as the two previous requests have been completed. This is repeated until a button stop has been pressed on a GUI (modeled by the artifact `inputGUI`).

It is worth clarifying how such a plan model works considering the agent control cycle. In the plan stage, when an agent perceives that a new task has to be done, it selects from its plan library an applicable plan given the type of the task and the context condition, and then it instantiates a new intention, representing the plan in execution. The intention is kept until the task is accomplished or failed. Each intention has a stack of action rule blocks, which drives the selection of actions cycle by cycle. At the beginning, the stack contains only the body of the plan. Action selection consists in collecting *all* the rules of the block on the top of the stack that are triggered in that cycle – i.e., with the *when* part satisfied. For each action rule in the block, an action rule is selected if *(i)* the event template specified in the rule matches the event (if any) at the head of the internal event queue, and *(ii)* the boolean expression defined in the condition part – which may predicate about the current value of any beliefs of the belief base – is true. At each cycle, only one event (if available) from the internal event queue is fetched. Actions selected in the plan stage are executed in the act stage.

4.2 Managing Tasks as First-class Entities

The capability of managing tasks in execution is an important feature for programming more structured and complex behaviors, in particular for those cases in which the reactive part has to influence the execution of the autonomous one—as mentioned in Section 2. In the following we consider two examples, of increasing complexity.

First, we consider an extension of the previous example with the further requirement that, as soon as the agent receives the `react` message, besides printing a message on the console, it has also to stop what is currently doing and has to complete its job by executing a further `Td` task. A solution in simpAL is shown in Figure 9. The reaction in this case drops the sub-task (if any) in execution and the ongoing interrupted plan, and then does a new sub-task `Td` to complete the job. `drop-all-tasks` and `forget-old-plans` are an example of the repertoire of internal actions available in the language to manage tasks in execution and intentions. In particular, the `drop-all-tasks` action drops all the ongoing tasks (and related intentions) but the current one, and `forget-old-plans` acts on the intention rule stack of the current intention, by removing all the action rule blocks but the one at top level.

Generalizing the example, an agent in simpAL can carry on concurrently multiple tasks – which may be sub-tasks instantiated by the agent itself in a plan of a parent task. So multiple intentions can be in execution at a certain time; at each execution cycle, action selection concerns all the ongoing intentions—so it is like to say that the agent carry on all its ongoing tasks in parallel. A family of internal built-in actions and predicates are provided to respectively control and inspect tasks in execution – for instance: to drop or suspend tasks, to check if a specific task has been completed or has failed. This support allows for keeping a certain level of abstraction and modularity in writing plans.

In the second example we consider the case in which the reaction is meant to produce a different future behavior depending on what the agent was doing. In particular: if the `react` message is received when doing the `Ta` sub-task, then the sub-task has to be interrupted and a task `Td` must be executed to complete `TaskT`. Instead, if the `react` message

```
agent-script TestScript implements RoleR {
  c: int = 0;

  plan-for TaskT completed-when: is-done tc {
     do-task new-task Ta();
     do-task new-task Tb();
     do-task new-task Tc() #tc

     when told this-task.react => using:console@main {
       drop-all-tasks ;
       forget-old-plans ;
       println(msg: "react! "+c) ;
       do-task new-task Td()
     }
  }

  plan-for Td using: console@main {
    c = c * 10 ;
    println(msg: "done td: "+c)
  }

  task Td{}
  ...
}
```

Figure 9. An example of reaction affecting the autonomous behavior of the agent.

```
1   agent-script TestScript implements RoleR {
2     c: int = 0;
3
4     plan-for TaskT completed-when: is-done tc {
5       ta: Ta = new-task Ta();
6       do-task ta;
7       tb: Tb = new-task Tb();
8       do-task tb;
9       tc: Tc = new-task Tc();
10      do-task tc
11
12      when told this-task.react :
13                is-ongoing ta || is-ongoing tb =>
14          using: console@main
15          completed-when: is-done te || is-done newTc {
16        te: Te; newTc: Tc;
17        forget-old-plans ;
18        atomically: {
19          println(msg: "react! "+c) ;
20          if (is-ongoing ta) {
21            drop-task ta ;
22            newTc = new-task Tc() ;
23            assign-task newTc ;
24          } else-if (is-ongoing tb) {
25            te = new-task Te( prev: tb) ;
26            assign-task te
27    }}}}
28
29    plan-for Te {
30      taskToWait: Tb = this-task.prev;
31      completed-when: is-done te  {
32        when is-done taskToWait => using: console@main {
33          c = c * 100 ;
34          println(msg: "done te: "+c)
35        } #te
36      }
37    }
38
39    task Te {
40      input-params {
41        prev: Tb;
42    }}
43    ...
44  }
```

Figure 10. A more complex example of reactive behavior, that inspects ongoing tasks.

is received when doing `Tb` sub-task, then the sub-task must be carried on until the end and after that, instead of doing a `Tc` task, a new task `Te` must be executed. All the other cases are not relevant for the agent.

The solution in simpAL is shown in Figure 10 and it is a good example of what kind of flexibility is possible by having tasks as first-class abstractions. Differently from previous cases, beliefs are used (`ta`, `tb`, `tc`) to keep track of the tasks as soon as they are instantiated and assigned. Built-in predicates are available to check the state of tasks – in the example, `is-ongoing` returns true if the specified task is defined and it has been assigned but it is not completed, while `is-done` is true if the specified task has been completed. So the rule reacting to the event (lines 12-27) is triggered only if either the task `ta` or `tb` are ongoing. Then, by inspecting the state of the tasks, the future course of action for the plan is decided. In particular, if the task `ta` is ongoing (line 20-23), then it is immediately dropped and a new task `tc` is assigned[4]; otherwise, if the task `tb` is ongoing (line 24-26), a new sub-task `Te` is instantiated, without dropping `tb` that can proceed until completion. The plan handling `Te` waits for the completion of the ongoing task `Tb` – which has been passed as input parameter of the sub-task – before proceeding and doing its job, which accounts to update `c` and print a message on the console.

When dealing with agents that need to carry on concurrently multiple related tasks (that are e.g. sub-tasks of a main tasks), an important feature is the possibility to specify that, when doing some specific *critical* part of a plan, the agent should keep the focus on that, without interleaving with other plans in execution (that may interfere). This can be specified at the action rule block level, by means of the `atomically:` attribute. In the example, the attribute is used in (lines 18-27), when reacting and deciding what to do, depending on the state of the specific sub-task in execution.

4.3 Event Programming Without Inversion of Control

A well-known problem in literature which is strongly related to the one discussed in the paper is the capability of doing event-driven programming without the problems that typically are found when using call-backs and inversion of control [18, 26]. In simpAL there is no inversion of control, since events are managed by the agent control loop.

As an example, Figure 11 shows a possible implementation in simpAL of the *observer* pattern [15]. simpAL natively provides a support for publish/subscribe and event-based kind of interactions. Observed objects can be directly modeled as artifacts with some specific observable properties. Observers can be modeled as agents, simply declaring to use those artifacts. The example shows the script of an agent observing a counter, reacting each time the observable property `count` changes, because of the execution of the `inc`

[4] differently from `do-task`, `assign-task` succeeds when the task is assigned, not necessarily accomplished.

```
role Observer {
  task Observing {
    input-params {
     sharedCounter: CounterUI;
}}}

agent-script ObserverScript implements Observer {
  plan-for Observing using: console@main {
    println(msg: "start observing...");
    using: this-task.sharedCounter {
      every-time changed count => {
        println(msg: "new count perceived! ")
}}}}
```

```
usage-interface CounterUI {
  obs-prop count: int;

  operation inc();
}


artifact Counter implements CounterUI {

  init() { count = 0; }

  operation inc () { count = count + 1; }
}
```

Figure 11. *(Left)* Role and script for observer agents (on the left) and usage interface and implementation of an observed counter artifact.

operation by some (other) user agents, and the source code of the artifact implementing such CounterUI usage interface. Since this kind of interaction is part of the simpAL programming model, no specific code for managing observer registration / notification is necessary.

Every time the agent perceives a change of the observable property count, it prints a message on the console. The control flow executing this action is (conceptually) the agent control loop one. No race conditions and concurrency problems can occur even with multiple observers and users, thanks to the computation model of artifacts (in which the execution of operations is mutually exclusive) and of agents.

5. A Real-World Example: a Reactive File Searcher

In this section we consider a real-world programming example to show the effectiveness of our approach besides toy models. The example concerns the realization of a software component to search and then print in standard output the list of all the files of a certain directory whose size is greater than a threshold provided in input. As a further requirement, the file size threshold can be changed dynamically during the search process[5].

The analysis of the problem leads quite naturally to identify two macro-behaviors: one autonomous/pro-active – searching and then printing the list of files that meet the desired size – and one reactive — managing threshold updates. The integration of these behaviors must be properly designed in order to take into the account the effects that a dynamic update of the threshold must have on an ongoing search process—e.g., files that have been discarded can become relevant and vice versa.

A solution that naturally follows from this analysis is given by the following simple abstract algorithm: recursively, starting from the directory provided in input and for each sub-directory found, check each file and keep track of all those that have a size which is greater than the reference threshold. If a new threshold is communicated while searching and it is greater than the previous one, then suspend the search process, update the threshold with the new one, filter the current list of found files – i.e. keep only the files that are greater than the new threshold – and finally resume the search in the file system. Otherwise, if the new threshold is lower than the previous one, end the current search process and restart from scratch. Files that have been discarded because too small can now become relevant since the threshold has been lowered. When the search process is completed, print in standard output the result—i.e., the list of found files.

```
agent-script Searcher implements ReactiveSearcher {
 foundFiles: java.util.List
 currThr: int

 plan-for SearchFiles completed-when: done printRes{
   currThr = this-task.thr;
   searchTask = new-task
     SearchFilesInDir(dir: this-task.dir);
   assign-task searchTask;

   when done searchTask => {
     /* print results */
   }#printRes

   every-time told this-task.newThr
    : !(is-ongoing printRes) => atomically: {
     if (this-task.newThr > currThr){
       currThr = this-task.newThr;
       foundFiles = filter(foundFiles, currThr)
     } else if (this-task.newThr < currThr) {
       currThr = this-task.newThr;
       drop-task searchTask;
       foundFiles.clear();
       searchTask <- new-task
         SearchFilesInDir(dir: this-task.dir);
       assign-task searchTask
 }}}

 plan-for SearchFilesInDir{
   for-each elem in this-task.dir {
     if (isDir(elem)) {
       do-task new-task SearchFilesInDir(dir: elem)
     } else-if (size(elem)>currentThr) atomically:{
         foundFiles.add(foundFiles)
     }
}}}
```

Figure 12. Implementation of the reactive file searcher in simpAL.

[5] For sake of simplicity threshold changes that can occur when printing in standard output the results, so when the search process is already finished, are discarded—i.e., we consider the threshold updates arrived too late.

The implementation of this strategy in simpAL is quite straightforward (Figure 12), and the source code preserves the organization and structure defined at the design level. For sake of space, the implementation reported here abstracts from technical details that are not important for the scope of this paper—interested readers can find the full sources in the example folder of the standard simpAL distribution.

There is a main task `SearchFile`, with `dir` and `threshold` input parameters and `newThr` representing the message(s) that can be told to notify new thresholds. The plan for the task (lines 5-27) has an autonomous part and a reactive one. The autonomous behavior accounts for instantiating the sub-task `searchTask` of type `SearchFilesInDir` (line 7-9) first, providing as input parameter the starting directory which corresponds to the `dir` parameter of the `SearchFiles` task, and then printing the results when the search sub-task is completed. The plan for handling the `SearchFilesInDir` sub-task (line 29-36) collects recursively, instantiating a new `SearchFilesInDir` sub-task for each sub-directory found (lines 31-32), the set of files that meet the desired size.

The reactive behavior is given by an action rule (lines 15-27) which handles the updates of the threshold on the basis of the strategy described above. The threshold value can be updated by the agent that has assigned the `SearchFiles` task to the `ReactiveSearcher` by sending to it a `newThr` message. In the case of a threshold increase (line 17-19) the action rule simply filters the list of files found so far. In the case of a threshold decrease instead (line 20-27), the task `searchTask` is first dropped (line 22) and then re-instantiated (lines 24-26) in order to restart the search from scratch since there is no guarantee that we have not already discarded files that have become relevant given the new threshold. The action rule block is executed atomically so that all the SearchFiles sub-tasks and related intentions are suspended until the action rule is executed up to completion, giving hence the opportunity to realize the required integration between autonomous and reactive behaviors.

6. Related Work

This paper is strongly related to existing research work in literature discussing the problem of integrating threads with event-driven programming, in particular in the context of object-oriented concurrent programming [3, 13] and actors [2].

In [18] authors describe the approach used to implement the Scala Actors library unifying thread-based and event-based actors. An actor can suspend with a full stack frame (using the *receive* primitive) or it can suspend with just a continuation closure (using the *react* one). The first form of suspension corresponds to thread-based programming, the second form to event-based programming. In simpAL the agent programmer never suspends or acts upon threads directly: concurrency at the agent level is totally logical, threads are managed by the simpAL runtime at a lower level, applying a classic pool-based strategy to maximize parallelism. Unification in our approach occurs at another level of abstraction, in terms of integration of autonomous and reactive behavior.

Kilim [33] is one of the first actor frameworks for Java using a Continuation Passing Style (CPS) technique to integrate ultra-lightweight threads and events; in particular, a weaver transforms methods identied by an `@pausable` annotation into CPS to provide cooperatively-scheduled lightweight threads with automatic stack management [1] and trampolined call stack [16].

Besides the actor context, the dualism between multi-threaded and event-driven models is a well-known topic discussed in literature in particular in the context of Operating Systems [24, 27, 35], as well as asynchronous I/O management. This paper is related in particular to those works that aim at integrating the models, so as to finally simplify programming and improve modularity, avoiding problems such as *stack-ripping* [1], in which the logical control flow between operations is broken across a series of callbacks. Recent approaches include: AC [20], extending native languages such as C/C++ with constructs for asynchronous I/O; the asynchronous programming model of F# [34]; GHC Haskell, combining call-back based and thread-based communication abstractions [25]; TaskJava [14], proposing tasks as a new programming model for organizing event-driven programs in Java.

The paper is strongly related also to research works that in general propose new abstraction levels to ease the development of reactive systems and programs that integrate a concurrent and reactive behavior. A recent one is Behavioral Programming (BP) [19]. BP is based on languages for capturing formal requirements of reactive systems (i.e., systems that constantly interact with their environment) in a way that allows their execution. In particular, system behaviors identified in the requirement analysis stage can be coded in executable software modules using behavioral programming idioms and infrastructure—choosing different kinds of languages (Erlang and Java are two examples).

Finally, the design of simpAL has been strongly influenced by existing Agent Programming theory and languages in the context of Multi-Agent Systems [11]. The control cycle adopted in simpAL derives from the BDI agent reasoning cycle [12]. Besides, some main concepts and features of the language have been inspired by BDI agent languages, in particular AgentSpeak(L) [28] and its extension Jason [10], and from our previous works about JaCa [31] and JaCaMo frameworks [9]. Generally speaking, these languages and frameworks have been conceived specifically to tackle Distributed AI problems—so developing in particular those features that are important for that purpose—for instance, first-order logic for knowledge representation; simpAL instead has been design with a different objective, i.e.

Number of Iterations	Erlang	ActorFoundry	Jason	simpAL
10000 Iterations	1.52 s	8,92 s	12,56 s	1,58 s
10000 Iterations (no print)	0.04 s	0.06 s	6,30 s	0.89 s
25000 Iterations	8.82 s	56,94 s	69.57 s	4,52 s
25000 Iterations (no print)	0.06 s	0.14 s	50.69 s	2,29 s
50000 Iterations	35,34 s	218,64 s	281,16 s	9,15 s
50000 Iterations (no print)	0.11 s	0.31 s	252,95 s	4,55 s

Table 1. Execution time in seconds of the test program in Erlan, ActorFoundry, Jason and simpAL. The time reported in each cell refers to the average of twenty different runs.

to explore agent-oriented programming as a mainstream programming paradigm, devising agent-oriented abstractions, concepts and mechanisms with the purpose of improving programming and software development.

7. Concluding Remarks

The importance of abstraction in programming is well-known and in this paper we discussed the application of agent-oriented programming abstractions for tackling the problem of integrating autonomous and reactive behavior in concurrent and reactive programming.

Often abstraction comes with a price in terms of performance, which could be either acceptable or not depending on the application domain considered. In this perspective, in the abstraction layer introduced by simpAL there are two critical points related to agent execution. The first is the uncoupling between physical concurrency and logical concurrency, so that there is not one raw OS thread for each agent, but all the agents (and artifacts) on a node[6] are executed by a pool of threads—whose size depends on the number of processors available on that node. The second one is related to the agent control architecture and the plan model adopted to drive the selection of actions at each agent cycle. Conceptually, the sense-plan-act cycle is executed continuously, even in the case of, e.g., a simple sequence of actions that must be executed without reacting to any external events. This brings a severe penalty on performances when considering – in particular – the execution of pure computational blocks compared to e.g. actor technologies implementing a macro-step semantics. When we consider – instead – behaviors that mix computations and reactions/interactions, then the agent control architecture could bring some benefits even from the point of view of the performances.

To start investigating the performance issue, we made some tests comparing the performance of simpAL with: *(i)* Erlang and ActorFoundry as reference actor-based languages/frameworks; and *(ii)* Jason as one of the main agent-programming languages in the state-of-the-art. The test program we used is a slightly extended version of the first example described in Section 2 where the execution of the task T and the sending of the react message have been repeated for 10K, 25K and 50K times respectively. Actually, two slightly different test programs are considered: the first one in which the test agent reacts to the message by incrementing a reaction count and printing a message on the console – so involving some I/O – and the second one without printing, so a purely computational reaction. Each program has been executed for twenty times. The test has been executed with Erlang version 5.8.3, ActorFoundry 1.0, Jason 3.7 and simpAL version 0.7, on a PC with a Intel Core 2 Duo P8400 2.26GHZ (dual core) and 3GiB RAM[7]. The average of the execution time experienced in the tests is reported in Table 1.

The results are quite promising, in particular by considering that simpAL technology (which is Java based) is still in its infancy and no specific optimizations have been devised yet. simpAL performs better than Jason of about an order of magnitude – this result was quite expected, since Jason (whose runtime is based on Java too) has been designed for the (D)AI context and it strongly relies on logic programming. Compared to Erlang and ActorFoundry, quite surprisingly performances are comparable or even in favor of simpAL when considering reactions with I/O; instead, both largely outperform simpAL in tests with a pure CPU-bound reaction, as expected.

Optimizations – that will be part of out future work – will be explored in two main directions. The first one is about optimizing the sense-plan-act cycle and in particular action selection, avoiding as much as possible to do unnecessary cycles and minimize the time to select actions in specific cases, e.g. when blocks are composed by simple sequences of actions – with no reactions. The second one is about the management of beliefs; currently all the beliefs (that are like variables or fields in other programming languages) are managed through maps, accessed by a string-based key; this naive strategy results in a severe overhead indeed, compared to e.g. classic techniques used in compiled languages where access to variables is index-based.

Finally, two relevant issues related to asynchronous events have not been discussed: the management of (asynchronous) errors – e.g., the failure of an action requested in a plan, of task, of internal actions such as exceptions gener-

[6] simpAL programs can be distributed.

[7] The source code of the test programs is included in the appendix, Section A

ated by the execution of a method on a inner object – and the management of events related to time (e.g., timeouts or periodic temporal behaviors). In simpAL these aspects are modeled uniformly as external events fetched in the sense stage, so that it is possible to write action rules reacting to them. Current support however is quite naive and we plan to provide a more extensive treatment in future work.

References

[1] A. Adya, J. Howell, M. Theimer, W. J. Bolosky, and J. R. Douceur. Cooperative task management without manual stack management. In *Proc. of ATEC '02*, pages 289–302, Berkeley, CA, USA, 2002. USENIX Association.

[2] G. Agha. *Actors: a model of concurrent computation in distributed systems*. MIT Press, Cambridge, MA, USA, 1986.

[3] G. Agha. Concurrent object-oriented programming. *Commun. ACM*, 33(9):125–141, Sept. 1990.

[4] G. Agha and C. Hewitt. Concurrent programming using actors. In A. Yonezawa and M. Tokoro, editors, *Object-oriented concurrent programming*, pages 37–53. MIT Press, Cambridge, MA, USA, 1987.

[5] G. Agha, S. Frølund, W. Kim, R. Panwar, A. Patterson, and D. Sturman. Abstraction and modularity mechanisms for concurrent computing. *IEEE Parallel Distrib. Technol.*, 1(2): 3–14, May 1993.

[6] G. A. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott. A foundation for actor computation. *J. Funct. Program.*, 7(1): 1–72, Jan. 1997.

[7] J. Armstrong. Erlang. *Communications of the ACM*, 53(9): 68–75, 2010.

[8] M. Ben-Ari. *Principle of Concurrent and Distributed Programming*. Wiley, 2005.

[9] O. Boissier, R. H. Bordini, J. F. Hübner, A. Ricci, and A. Santi. Multi-agent oriented programming with jacamo. *Science of Computer Programming*, 2012. doi: 10.1016/j.scico.2011.10.004. In press.

[10] R. Bordini, J. Hübner, and M. Wooldridge. *Programming Multi-Agent Systems in AgentSpeak Using Jason*. John Wiley & Sons, Ltd, 2007.

[11] R. H. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni. *Special Issue: Multi-Agent Programming*, volume 23 (2). Springer Verlag, 2011.

[12] M. E. Bratman. *Intention, Plans, and Practical Reason*. Cambridge University Press, Mar. 1999.

[13] J.-P. Briot, R. Guerraoui, and K.-P. Lohr. Concurrency and distribution in object-oriented programming. *ACM Comput. Surv.*, 30(3):291–329, 1998.

[14] J. Fischer, R. Majumdar, and T. Millstein. Tasks: language support for event-driven programming. In *Proc. of PEPM '07*, pages 134–143, New York, NY, USA, 2007. ACM.

[15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Pattens*. Addison Wesley, 1995.

[16] S. E. Ganz, D. P. Friedman, and M. Wand. Trampolined style. *SIGPLAN Not.*, 34(9):18–27, Sept. 1999.

[17] N. Gehani. *ADA: Concurrent Programming*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1984.

[18] P. Haller and M. Odersky. Actors that unify threads and events. In *Proceedings of the 9th international conference on Coordination models and languages*, COORDINATION'07, pages 171–190, Berlin, Heidelberg, 2007. Springer-Verlag.

[19] D. Harel, A. Marron, and G. Weiss. Behavioral programming. *Commun. ACM*, 55(7):90–100, July 2012.

[20] T. Harris, M. Abadi, R. Isaacs, and R. McIlroy. AC: composable asynchronous io for native languages. *SIGPLAN Not.*, 46 (10):903–920, Oct. 2011.

[21] N. R. Jennings. An agent-based approach for building complex software systems. *Commun. ACM*, 44(4):35–41, 2001.

[22] R. K. Karmani, A. Shali, and G. Agha. Actor frameworks for the jvm platform: a comparative analysis. In *Proc. of PPPJ'09*, pages 11–20, New York, NY, USA, 2009. ACM.

[23] A. C. Kay. *The reactive engine*. PhD thesis, The University of Utah, 1969. AAI7003806.

[24] H. C. Lauer and R. M. Needham. On the duality of operating system structures. *SIGOPS Oper. Syst. Rev.*, 13(2):3–19, Apr. 1979.

[25] P. Li and S. Zdancewic. Combining events and threads for scalable network services implementation and evaluation of monadic, application-level concurrency primitives. *SIGPLAN Not.*, 42(6):189–199, June 2007.

[26] M. Miller, E. Tribble, and J. Shapiro. Concurrency among strangers: programming in E as plan coordination. In *Trustworthy Global Computing*, volume 3705 of *LNCS*, pages 195–229. Springer Berlin / Heidelberg, 2005.

[27] J. Ousterhout. Why Threads Are A Bad Idea (for most purposes), 1996. Presented at USENIX Technical Conference.

[28] A. S. Rao. AgentSpeak(L): Bdi agents speak out in a logical computable language. In *Proc. of MAAMAW'96*, pages 42–55. Springer-Verlag New York, Inc., 1996.

[29] A. S. Rao and M. P. Georgeff. BDI Agents: From Theory to Practice. In *Proc. of ICMAS'95*, 1995.

[30] A. Ricci and A. Santi. Designing a general-purpose programming language based on agent-oriented abstractions: the simpAL project. In *Proc. of AGERE!'11*, SPLASH '11 Workshops, pages 159–170, New York, NY, USA, 2011. ACM.

[31] A. Ricci and A. Santi. Agent-oriented computing: Agents as a paradigm for computer programming and software development. In *Proc. of Future Computing '11*, Rome, Italy, 2011. IARIA.

[32] Y. Shoham. Agent-oriented programming. *Artificial Intelligence*, 60(1):51–92, 1993.

[33] S. Srinivasan and A. Mycroft. Kilim: Isolation-typed actors for java. In *Proc. of ECOOP '08*, pages 104–128, Berlin, Heidelberg, 2008. Springer-Verlag.

[34] D. Syme, T. Petricek, and D. Lomov. The F# asynchronous programming model. In *Proc. of PADL'11*, pages 175–189, Berlin, Heidelberg, 2011. Springer-Verlag.

[35] R. von Behren, J. Condit, and E. Brewer. Why events are a bad idea (for high-concurrency servers). In *Proc. of HOTOS'03*, pages 4–4, Berkeley, CA, USA, 2003. USENIX Association.

A. Sources of the Test Programs

```
test_actor(N) ->
  statistics(wall_clock),
  self() ! doTaskT,
  loop(N,0,0).

loop(0,C,R) ->
  trigger ! testDone;

loop(N,C,R) ->
  receive
    doTaskT ->
      C1 = ta(C),
      self() ! doingTb,
      loop(N,C1,R);
    doingTb ->
      C1 = tb(C),
      self() ! doingTc,
      loop(N,C1,R);
    doingTc ->
      C1 = tc(C),
      self() ! doTaskT,
      loop(N-1,C1,R);
    react ->
      io:format("react! ~w~n", [R]),
      loop(N,C,R+1)
  end.

ta(C) -> C+1.
tb(C) -> C+1.
tc(C) -> C+1.

trigger(Who,0) ->
  receive
    testDone ->
      {_, Time} = statistics(wall_clock),
      io:format("Time elapsed ~p ms~n",[Time])
  end;

trigger(Who,N) ->
  Who ! react,
  trigger(Who,N-1).

start() ->
  PID = spawn(test2loop, test_actor, [50000]),
  register(trigger, spawn(test2loop,
    trigger, [PID, 50000])).
```

Figure 13. Implementation of the test program in Erlang. The test_actor function has been used to implement the actor that has in charge the execution of the task T, while the trigger function has been used to implement the actor that sends the react messages.

```
public class TestActor extends Actor {
  private int c = 0;
  private int nTimes = 0;
  private long startTime;
  private int maxTimes;
  private int nReactions=0;
  private ActorName triggerActor;

  @message
  public void start(ActorName triggerActor,
   Integer maxTimes) throws RemoteCodeException {
    startTime = System.currentTimeMillis();
    this.triggerActor = triggerActor;
    this.maxTimes = maxTimes;
    send(self(), "doTaskT");
  }
  @message
  public void doTaskT() throws RemoteCodeException {
    send(self(), "doingTa");
  }
  @message
  public void doingTa() throws RemoteCodeException {
    send(self(), "doingTb");
    ta();
  }
  @message
  public void doingTb() throws RemoteCodeException {
    send(self(), "doingTc");
    tb();
  }
  @message
  public void doingTc() throws RemoteCodeException {
    tc();
    nTimes++;
    if (nTimes < maxTimes){
      send(self(),"doingTa");
    } else {
      send(triggerActor, "testDone", startTime);
    }
  }
  @message
  public void react() throws RemoteCodeException {
    nReactions = nReactions + 1;
    call(stdout, "println","react! "+nReactions);
  }
  private void ta(){c = c + 1;}
  private void tb(){c = c + 1;}
  private void tc(){c = c + 1;}
}
```

```
public class TriggerActor extends Actor {

  @message
  public void start(ActorName testActor,
   Integer maxTimes) throws RemoteCodeException {
    for (int i=0; i < maxTimes; i++){
      send(testActor,"react");
    }
  }

  @message
  public void testDone(Long t0)
   throws RemoteCodeException {
    call(stdout, "println","Time elapsed " +
     (System.currentTimeMillis() - t0) + " ms");
  }
}
```

Figure 14. Implementation of the test in ActorFoundry. *(Top)* Implementation of the testActor that has in charge the execution of the task T. *(Bottom)* Implementation of the triggerActor that has in charge the sending of react messages to the testActor.

```
c(0).
reactions(0).

!start(50000).

+!start(N)
 <- StartTime = system.time;
    +startTime(StartTime);
    !loop(N).

+!loop(0) : startTime(StartTime)
 <- .send(trigger, tell, done(StartTime)).

+!loop(N)
 <- !tA;
    !tB;
    !tC;
    !loop(N-1).

+!tA:c(Val)<--+c(Val+1).
+!tB:c(Val)<--+c(Val+1).
+!tC:c(Val)<--+c(Val+1).

+react(N):reactions(Val)
 <- -+reactions(Val+1);
    .println("react").
```

```
!loop(0).

+!loop(50000).

+!loop(N)
 <- .send(test, tell, react(N));
    !loop(N+1).

+done(StartTime)
 <- CurrTime = system.time;
    .println("Time elapsed ",
      CurrTime-StartTime, " ms").
```

Figure 15. Implementation of the test in Jason. *(Top)* The test agent that has in charge the execution of the task T. *(Bottom)* Implementation of the trigger agent that sends the react messages to the test agent.

```
role RoleRLoop {
  task TaskT {
    input-params{
      maxTimes: int;
    }
    understands {
      react: boolean;
}}}
```

```
agent-script TestAgentScriptLoop
          implements RoleRLoop {

  c: int = 0

  plan-for TaskT
   completed-when: is-done tRep
   using: console@main{
    nReactions: int = 0

    tRep: Trep =
      new-task Trep(maxTimes:this-task.maxTimes);
    assign-task tRep

    every-time told this-task.react =>
      atomically: {
        nReactions = nReactions + 1;
        println(msg: "react! " + nReactions)
    }
  }

  plan-for Trep {
    nTimes: int = 0

    while (nTimes < this-task.maxTimes) {
      do-task new-task Ta();
      do-task new-task Tb();
      do-task new-task Tc();
      nTimes = nTimes + 1
    }
  }

  plan-for Ta {c = c + 1}
  plan-for Tb {c = c + 1}
  plan-for Tc {c = c + 1}

  task Ta {}
  task Tb {}
  task Tc {}
  task Trep {input-params{maxTimes: int;}}
}
```

Figure 16. Implementation of the test in simpAL. Role and script of the agent in charge the execution of the task T.

```
role TriggerAgent {
  task SendReact {
    input-params{
      maxTimes:int;
    }
  }
}
```

```
agent-script TriggerAgentScriptLoop
          implements TriggerAgent {

  nTimes: int = 0

  plan-for SendReact
    completed-when: is-done printTime
    using: clock@main, console@main {

    startTime: long
    testAgent: RoleRLoop
    taskT: RoleRLoop.TaskT

    getTimeNow(currentTime: startTime);
    taskT = new-task
      RoleRLoop.TaskT(maxTimes: this-task.maxTimes);
    new-agent TestAgentScriptLoop()
            init-task: taskT ref: testAgent;
    while (nTimes < this-task.maxTimes) {
      tell taskT.react = true;
      nTimes = nTimes + 1
    }

    when is-done taskT => {
      endTime: long
      getTimeNow(currentTime: endTime);
      println(msg: "Time elapsed " +
        (endTime-startTime)+" ms") on console@main
    } #printTime
  }
}
```

Figure 17. Role and script of the trigger agent that has in charge the sending of react messages to the agent playing the role RoleRLoop.

Empirical Software Engineering for Agent Programming

M. Birna van Riemsdijk
Delft University of Technology
m.b.vanriemsdijk@tudelft.nl

Abstract

Empirical software engineering is a branch of software engineering in which empirical methods are used to evaluate and develop tools, languages and techniques. In this position paper we argue for the use of empirical methods to advance the area of agent programming. Through that we will complement the solid theoretical foundations of the field with a thorough understanding of how our languages and platforms are used in practice, what the main problems and effective solutions are, and how to improve our technology based on empirical findings. Ultimately, this will pave the way for establishing multi-agent systems as a mature and recognized software engineering paradigm with clearly identified advantages and application domains.

Categories and Subject Descriptors I.2.5 [*Artificial Intelligence*]: Programming Languages and Software; I.2.11 [*Artificial Intelligence*]: Distributed Artificial Intelligence—Intelligent agents, languages and structures; D.2 [*Software Engineering*]

General Terms Design, Languages

Keywords Agent programming languages, empirical software engineering, software quality, metrics

1. Introduction

Empirical software engineering is a branch of software engineering in which empirical methods are used to evaluate and develop tools, languages and techniques. The journal on Empirical Software Engineering (see [1]) started in 1996. As stated in [12]: 'The acceptance of empirical studies in software engineering and their contributions to increasing knowledge is continuously growing. The analytical research paradigm is not sufficient for investigating complex real life issues, involving humans and their interactions with technology.' That is, empirical research needs to complement theoretical studies in order to advance understanding with respect to the use of technologies.

We argue that empirical software engineering is important not only for mainstream software engineering, but also for agent-oriented programming [3, 4] and software engineering. Through empirical methods, different kinds of questions can be answered than through analytical approaches. In this position paper we propose several such questions and thereby sketch what such a line of research might look like. We focus on agent programming rather than agent-oriented software engineering in general. However, similar questions and issues as proposed below for agent programming may also be translated to agent-oriented software engineering. The term 'agent programming' should be understood to refer to programming autonomous agents and multi-agent systems.

2. Research Questions

By using empirical methods, data can be gathered for several purposes. For example, to get a better understanding of how software developers use agent programming technology (problems and possible solution patterns), to perform a within technology comparison, i.e., demonstrating that one variant of (using) agent programming technology is better than another, to improve the technology based on these findings, and to perform across technology comparison, i.e., demonstrating that agent programming technology is better than some other technology. Corresponding research questions that may be studied using empirical methods can concern any of the set of instruments that facilitate the development of high-quality agent programs, namely programming language, programming guidelines & teaching methods, and development environment. For example:

- how do programmers use agent-oriented languages?
  - which constructs do they use?
  - what is expressed using which constructs?
  - what kind of patterns do they use?
  - which problems do they experience while programming?
  - which aspects of the languages do they find difficult or easy to understand?
  - what kind of processes are used during development, e.g., which parts of a program are developed first?
- which ways of using agent-oriented languages improve agent software quality?
  - how should constructs be used?
  - which patterns and anti-patterns can be distinguished?
- how does the use of an agent programming language compare with the use of mainstream, general purpose languages like Java, and other paradigms for development of decentralized, concurrent applications?
  - do similar programming patterns emerge?
  - how does the speed of software development compare?
  - how do the resulting programs compare with respect to software quality measures like efficiency, maintainability and readability?
- how does the domain for which applications are developed, influence software development?

    - which domain characteristics call for an agent-oriented approach to software development?
    - which patterns are/can/should be used in which kinds of domains?
- which features should an Integrated Development Environment for agent programming have?
    - how do programmers use existing IDEs?
    - which difficulties do they encounter while programming?
    - to what extent do requirements for an IDE for agent-oriented software development differ from those for mainstream software development?
    - which approaches for debugging are needed in the context of agent programming?
    - to what extent does debugging in agent programming differ from debugging in mainstream software development?

Of course, several of these questions have already been addressed to some extent in various papers. For example, in [2] it is shown that the use of BDI technology incorporated within an enterprise-level architecture can improve overall developer productivity by an average 350%. They argue that agent technology is particularly suitable for applications that are "hard" to build, in which requirements change quickly and which are event and exception driven. Testing multi-agent systems has also been studied in several papers, e.g., [10, 11, 15], although only [11] reports some empirical results. In [6], an empirical study is performed in the area of game development, where the POSH reactive planner with a graphical editor is compared with Java for programming high-level behavior of a virtual agent in the Unreal Tournament 2004 environment. In [8], metrics for quantifying coupling and cohesion are proposed that can be applied to agents as well as object-oriented software. In [17] we have studied how programmers use the GOAL agent programming language, making several observations concerning, e.g., the use of programming constructs and patterns.

3. Software Quality

A recurring theme in the above research questions is *software quality*. We aim at developing techniques that facilitate building "better" software. The question is then what exactly we mean by better software, i.e., how do we define software quality? A starting point for this is the ISO/EIC 9126 standard which provides a software quality model. It defines several software quality characteristics and subcharacteristics: functionality (e.g., interoperability, functionality compliance), reliability (e.g., fault tolerance, recoverability), usability (e.g., understandability, learnability), efficiency (e.g., time behavior), maintainability (e.g., analyzability, testability), and portability (e.g., adaptability, co-existence).

These characteristics provide an indication of what kind of characteristics to address when aiming for better software, but they do not specify how to *measure* to what extent a certain piece of software exhibits a characteristic. To address this, the standard specifies that for each characteristic a set of attributes has to be defined that can be verified or measured in the software product. This can be done, for example by defining a set of *quality metrics* which evaluate the degree of presence of quality attributes in the software. These can be internal metrics (static), external metrics (defined for running software), or 'quality in use' metrics (defined for using the software in real conditions). These attributes and metrics vary between technologies and software products.

Research will have to identify what software quality means in the context of programming multi-agent systems (MAS). Questions that need to be addressed are:

- Which ISO software quality characteristics are suitable for MAS?
- Which ISO software quality characteristics are particularly important (problematic or strength) in MAS?
- Which are MAS-specific software quality characteristics?
- Which MAS-specific attributes and metrics can be defined for measuring the characteristics?

It will be interesting to make precise how to measure quality characteristics in MAS. Given the wide range of languages and platforms for programming MAS, it should be of particular concern to analyze to what extent language-independent measures can be defined (as in [8]), or whether certain quality metrics need to be defined specific to a particular technology. For example, in [13] several existing software engineering metrics are used to evaluate a methodology for creating affective applications. In [14] metrics are used for evaluating the quality of message sequence charts, in the context of evaluating a methodology for developing cross-organizational business models. It will have to be investigated how to compare agent-specific metrics for quantifying a certain quality characteristic with metrics for that characteristic in mainstream technologies. Finally, it will be interesting to identify quality characteristics that are specific to MAS. Examples of characteristics that may be considered are *explainability* (to what extent is the intelligent system able to explain its decisions; this can be important for acceptance of the technology and for debugging (see, e.g., [7, 16])), and *believability* (to what extent does an intelligent (virtual) character or group of characters display believable behavior; this can be important for example for creating natural interaction with a human user of the technology).

4. Methodological Aspects

In this paper we argue for recognition of a line of research on empirical software engineering for agent programming, and for a more systematic approach to addressing questions like those posed above. This also calls for discussion and research concerning appropriate *methodologies* for conducting empirical research in agent programming. It needs to be investigated whether methods from mainstream empirical software engineering can be applied in our context. For example, [12] proposes guidelines for conducting and reporting case study research in software engineering. In [17] we propose an approach for empirically studying how programmers use an agent programming language, in which we identify several analysis dimensions, such as a functional analysis which identifies what the available language constructs are used for, and which general principles are applied when using them; and a structural analysis which identifies structural code patterns, and which determines quantitative metrics on the code. Also we propose a stepwise research approach for conducting case study research in agent programming, which is based on [5].

We believe that both quantitative as well as qualitative research should be performed. Quantitative research is used for testing predetermined hypotheses and producing generalizable results using statistics, focusing on answering mechanistic 'what?' questions; for example, what is the effect of using a certain debugging tool on the number of errors in the resulting software. Qualitative research is used for illumination and understanding of complex psychosocial issues, and can be used for answering humanistic 'why?' and 'how?' questions [9]; for example, how do programmers use agent-oriented programming languages. We believe that in particular in the earlier stages of studying the use of agent programming language empirically, it is very important to also perform qualitative research. This will provide a better understanding of how they are used, and through this the techniques can be improved. Once sufficient improvement has been realized through this process, within and across technology comparisons can be performed in a quantitative manner.

5. Conclusion

In this position paper we have argued for the use of empirical methods to advance the area of agent programming. Through this we will complement the solid theoretical foundations of the field of agent programming with a thorough understanding of how our languages and platforms are used practice, what the main problems and effective solutions are, and how to improve our technology based on empirical findings. Ultimately, this will pave the way for establishing multi-agent systems as a mature and recognized software engineering paradigm with clearly identified advantages and application domains.

Acknowledgments

I would like to thank Koen V. Hindriks for the joint research and discussions on the theme of this position paper.

References

[1] V. R. Basili and L. C. Briand, editors. *Empirical Software Engineering: An International Journal*. Springer, 2012. `http://www.springer.com/computer/swe/journal/10664`.

[2] S. S. Benfield, J. Hendrickson, and D. Galanti. Making a strong business case for multiagent technology. In *Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems (AAMAS'06)*, pages 10–15. ACM, 2006.

[3] R. H. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni. *Multi-Agent Programming: Languages, Platforms and Applications*. Springer, Berlin, 2005.

[4] R. H. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni. *Multi-Agent Programming: Languages, Tools and Applications*. Springer, Berlin, 2009.

[5] K. Eisenhardt. Building theories from case study research. *The Academy of Management Review*, 14(4):532–550, 1989.

[6] J. Gemrot, Z. Hlávka, and C. Brom. Does high-level behavior specification tool make production of virtual agent behaviors better? In *Proceedings of the International Workshop on Cognitive Agents for Virtual Environments (CAVE'12)*, 2012.

[7] K. V. Hindriks. Debugging is explaining. In *Principles of Practice in Multi-Agent Systems (PRIMA'12)*, volume 7455 of *LNAI*, pages 31–45. Springer, 2012.

[8] H. R. Jordan and R. Collier. Evaluating agent-oriented programs: Towards multi-paradigm metrics. In *Proceedings of the Eigth International Workshop on Programming Multiagent Systems (ProMAS'10)*, volume 6599 of *LNCS*, pages 63–78. Springer, 2012.

[9] M. N. Marshall. Sampling for qualitative research. *Family Practice*, (3):522–525, 1996.

[10] S. Miles, M. Winikoff, S. Cranefield, C. D. Nguyen, A. Perini, P. Tonella, M. Harman, and M. Luck. Why testing autonomous agents is hard and what can be done about it. URL `http://www.pa.icar.cnr.it/cossentino/AOSETF10/docs/miles.pdf`. AOSE Technical Forum 2010 Working Paper.

[11] D. Poutakidis, M. Winikoff, L. Padgham, and Z. Zhang. Debugging and testing of multi-agent systems using design artefacts. In R. H. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni, editors, *Multi-Agent Programming: Languages, Tools and Applications*, pages 215–258. Springer, Berlin, 2009.

[12] P. Runeson and M. Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, 14(2):131–164, 2009.

[13] D. J. Sollenberger and M. P. Singh. Kokomo: an empirically evaluated methodology for affective applications. In *Proceedings of the tenth international joint conference on autonomous agents and multiagent systems (AAMAS'11)*, pages 293–300. IFAAMAS, 2011.

[14] P. R. Telang and M. P. Singh. Comma: a commitment-based business modeling methodology and its empirical evaluation. In *Proceedings of the eleventh international joint conference on autonomous agents and multiagent systems (AAMAS'12)*, pages 1073–1080. IFAAMAS, 2012.

[15] J. Thangarajah, G. Jayatilleke, and L. Padgham. Scenarios for system requirements traceability and testing. In *10th International Conference on Autonomous Agents and Multiagent Systems (AAMAS'11)*, pages 285–292. IFAAMAS, 2011.

[16] I. van de Kieft, C. M. Jonker, and M. B. van Riemsdijk. Explaining negotiation: Obtaining a shared mental model of preferences. In *24th International Conference on Industrial Engineering and Other Applications of Applied Intelligent Systems (IEA/AIE'11)*, volume 6704 of *LNCS*, pages 120–129. Springer, 2011.

[17] M. B. van Riemsdijk, K. V. Hindriks, and C. M. Jonker. An empirical study of cognitive agent programs. *Multiagent and Grid Systems (MAGS)*, 8(2):187–222, 2012.

Messages with Implicit Destinations as Mobile Agents

Ahmad Ahmad-Kassem
Université de Lyon, INRIA
ahmad.ahmad_kassem@inria.fr

Stéphane Grumbach
INRIA
stephane.grumbach@inria.fr

Stéphane Ubéda
INRIA
stephane.ubeda@inria.fr

Abstract

Applications running over decentralized systems, distribute their computation on nodes/agents, which exchange data and services through messages. In many cases, the provenance of the data or service is not relevant, and applications can be optimized by choosing the most efficient solution to obtain them. We introduce a framework which allows messages with intensional destination, which can be seen as restricted mobile agents, specifying the desired service but not the exact node that carries it, leaving to the system the task of evaluating the extensional destination, that is an explicit address for that service. The intensional destinations are defined using queries that are evaluated by other agents while routing. We introduce the Questlog language, which allows to reformulate queries, and express complex strategies to pull distributed data. In addition, intensional addresses offer persistency to dynamic systems with nodes/agents leaving the system. We demonstrate the approach with examples taken from sensor networks, and show some experimental results on the QuestMonitor platform.

Keywords routing by content, intensional destination, mobile agents, declarative networking

1. Introduction

Most of the applications of our everyday life (communication, search, social, etc.), as well as those of our environment (workplace, domotics, transportation, energy, etc.) rely on complex network infrastructure. They require complex distributed algorithms that are difficult to program, require skilled programmers, and offer limited warrantee on their behavior. The dynamics of some networks, with nodes joining or leaving the networks, not to mention the various types of failures increases further the complexity and raises considerable challenges. One of the fundamental barriers to their development is the lack of programming abstraction [28].

Such applications are decentralized and need to adapt dynamically to their environment in a reactive manner. They necessitate a high-level programming paradigm that defines a new level of abstraction and offers features such as interaction, reactivity, autonomy, modularity, and asynchronous communication.

In this paper we propose a framework which allows to program distributed applications in a message-oriented manner, allowing messages as a sort of mobile agents with implicit destinations, that are solved in the network while they are traveling. The destinations of messages are abstracted and defined by queries. The main contribution of the paper is (i) the design of a data centric language, Questlog, which allows to program agents exchanging messages which admit a complex semantics associated to the queries defining their destinations, and (ii) its implementation over a simulation platform. We thus distinguish between intensional destinations, defined by queries, and extensional destinations, defined by node addresses.

The idea of programming messages as active agents has been proposed long ago in [36], where network programs are encapsulated in active messages traveling in the network. It provides a simple way to describe and understand distributed programs. Mobile code such as scripts, applets, and mobile agents is widely used [13, 20]. Our approach though is more restricted. We propose messages that have implicit destinations, that will be solved in the contact of other agents while the message is traveling. Only the query defining the destination is mobile, while the code of the agent that helps solving it, is static.

Recently, the notion of agent-oriented abstractions is proposed in [32], where a new programming paradigm providing a set of abstractions is introduced to simplify the programming of modern applications. However, the mobility of agents as well as the abstraction of the destinations of messages are not supported.

Messages with implicit destination facilitate the programming of a large class of applications. Publish/subscribe systems constitute a good example of such systems, with publishers who do not have to specify specific receivers, leaving the system matching them. Publish/subscribe systems constitute an appealing paradigm for developing pervasive systems which enable the decoupling of interacting components, separating communication from computation. However, they generally require the use of mediators to match interest with published events. Different forms of publish/subscribe systems have been proposed.

Topic-based systems [16] rely on the notion of topics, a static scheme with limited expressiveness. *Content-based* systems [16] allow filtering on the content of an event, and only those events that match the filter are delivered to the subscribers. This approach might result in high numbers of topics and potentially redundant events that increase the overhead. *Type-based* systems [16] combine topic-based and content-based system. The idea is to replace the topic classification form by a scheme that filters events according to their type. *Location-based* systems [17] support location-aware communication between participants based on positioning mechanisms, and *context-based* [19] systems capture event context in a modular way.

The major difficulty with publish/subscribe systems rely in the events matching mechanism, the efficient routing of notifications to subscribers, while avoiding useless transmission of notifications that result in an extra level of complexity [29]. Different content-based routing approaches have been proposed to route efficiently notifications by messages based in their content. In [12], a routing scheme is proposed based on a combination of a traditional broad-

AGERE'12 October 2012, Tucson, Arizona, USA.
Copyright © 2012 ACM [to be supplied]. . . $10.00

cast protocol and a content-based routing protocol. However, it suffers from a high communication complexity to build spanning trees to send notifications. In [27], authors propose a new method to provide end-to-end reliability based on the publish/subscribe system but with the cost of increased overhead.

To facilitate programming, we propose to use intensional destinations defined by queries, and let active agents in the network solve them on the fly. To do so, active agents have at their disposal programs (local agents) that give a meaning to destinations. More precisely, a *destination* is a pair specified *extensionally*, by the address of the node, and *intensionally*, by a Questlog query.

Declarative query languages have already been used in the context of networks. Several systems for sensor networks, such as TinyDB [26] or Cougar [14] offer the possibility to write queries in SQL. These systems provide solutions to perform energy-efficient data dissemination and query processing. A distributed query execution plan is computed in a centralized manner with a full knowledge of the network topology and the capacity of the constraint nodes, which optimizes the placement of subqueries in the network [33].

Another application of the declarative approach has been pursued at the network layer. The use of recursive query languages has been initially proposed to express communication network algorithms such as routing protocols [24] and declarative overlays [23]. This approach, known as **declarative networking** is extremely promising. It has been further pursued in [25], where execution techniques for Datalog are proposed. Distributed query languages thus provide new means to express complex network problems such as node discovery [4], route finding, path maintenance with quality of service [9], topology discovery, including physical topology [8], secure networking [1], or adaptive MANET routing [22].

Declarative networking relies on the rule-based languages [5, 6, 31, 35] developed in the field of databases in the 1980's. Questlog follows the trend opened by declarative networking [23, 25]. Declarative languages allow to specify at a high level "what" to do, rather than "how" to do it. They facilitate not only code reuse among systems, but also the extension, and hybridization. It was shown that such languages augmented with communication primitives, allow to express distributed applications and communication protocols with code about two orders of magnitude shorter than imperative programs, and with reasonable execution models. They are more declarative, so facilitate programming, they parallelize well, so facilitate the execution, they manipulate explicitly data structures, so facilitate verification of their properties. Simple Netlog protocols for instance have already been verified [15] using the Coq proof assistant.

Different languages have been proposed such as Overlog [23], NDlog [25], Netlog [18], and Webdamlog [2] for high-level programming abstraction. To our knowledge, however, they all follow the *forward chaining* mechanism. They are very successful in expressing various applications and protocols in proactive mode, but less than in reactive mode.

In contrast to Overlog [23], NDlog [25], Netlog [18], and Webdamlog [2], Questlog has been designed to *pull* data from a network by firing a query. The query is associated with a rule program composed of a set of rules in the form *head* :- *body* that are evaluated in parallel. The program is installed on the nodes of a network and the evaluation of rules combines *backward* and *forward chaining*. When a node receives a query, it identifies the rules whose head matches the query. If there are such rules, the node applies each of them, that is it generates their body instantiated with the variable substitution imposed by the initial query.

The Questlog language includes complex primitives such as aggregation, non deterministic choice, etc., to facilitate the programming of complex application. Questlog programs are compiled into sets of queries in an SQL dialect, which are loaded on the nodes of the network.

We have developed a system which runs the Questlog programs, and extends the Netquest virtual machine, initially proposed in [18] to evaluate Netlog programs. The new functionalities include (i) a *Questlog Engine* to evaluate queries and programs, and (ii) a *Router* to evaluate intensional destination query that offers resilience of the system under node failure or departure. We demonstrate the approach with examples taken from sensor networks, and show some experimental results on the QuestMonitor [10] platform that allows to interact with a network and visualize the behavior of declarative protocols. The system has been tested on simple networking protocols as well as wireless sensor networks, WSN, applications.

The paper is organized as follows. In the next section, we present motivating examples to explain the use of intensional destinations, and introduce the rules. Questlog, with the language's primitives is presented through examples in Section 3, while its procedural semantics is defined in Section 4. Section 5 is devoted to the implementation on top of the Netquest system, while some experimental results are presented in Section 6.

2. Motivation

We are interested in applications running over networks, with data fragmented over participating nodes, which in general have no knowledge on the location of data. They communicate by exchanging messages with a *payload*, the content of the message, and a *destination*, the final destination. In classical networking approaches, the flow of messages from source nodes to destinations is driven by their addresses (e.g. IP, MAC, etc.) assigned explicitly by the source nodes. In an increasing number of applications however, it is desirable if not necessary to be able to delay the evaluation of the final destination of a message. Examples of such applications include:

- *Distributed hash tables*: A hash function is used to map data items to nodes. Given a value (e.g. Id, address, data, etc.), the hash function produces a key, in general over the domain of identifiers of nodes. The destination can then be for instance the closest node. In Chord [34] or VRR [11] for instance, the nodes are organized in a ring structure, and messages are routed on the ring to increasing or decreasing Ids, till the closest node is reached.
- *Wireless sensor networks*: Such networks consist of large numbers of sensor nodes with limited numbers of sinks, which collect information from sensor nodes. For instance, a sink can collect the positions of nodes which have a temperature greater than some threshold. The sink can thus send messages to subsets of nodes satisfying some property.
- *Publish-subscribe systems*: Users publish services without specifying specific destinations to them, while subscribers express their interest to services, and receive corresponding messages, without knowledge of the publishers. Such systems are handled by appropriate middleware taking care of the messages.
- *Social networks*: Users are organized in network structures with their friends (symmetric links of Facebook) or followers (asymmetric links of Twitter) for instance, with whom they exchange information. Some messages can be addressed to sets of users that are out of the knowledge of users, or difficult to enumerate, such as the friends of their friends. In some social matching networks, it is possible to send notifications of interest to users to be received only by users who have sent in a symmetric manner similar notification of interest to the sender. In this example, the destinations cannot be cleared by the users themselves.

In all these examples, it would make things easier, if it was possible to specify the destination implicitly by a query, defining in an *intensional manner*, the destination of (message) mobile agent, which can be cleared or evaluated while traveling in the network, in an *extensional* manner, as the explicit address of the destination nodes. In Publish-Subscribe systems, this is done by appropriate message oriented middleware. Our objective is to let mobile agent solve intensional destinations.

Let us consider the following more complex example from wireless sensor networks. Consider an application where some sink node monitors the positions of nodes which have, together with their neighbors to avoid individual measurement errors, a temperature higher than some threshold. How to program such queries? How to get neighbors' temperature values dynamically?

We propose a declarative language, *Questlog*, which allows to specify such problems in a rather declarative, data centric manner. For simplicity, we consider a relational model of data, with relations of some fixed schema. *Questlog* is a rule-based language with *rules* of the form:

$$head : -body$$

well-adapted to complex queries as well as to reactive protocols. *Questlog queries* are of a very simple form:

$$?R(x_1, \cdots, x_\ell)$$

where R is a *relation symbol* of arity ℓ, and $x_1, \cdots, x_\ell$ are variables or constants. They are associated to rule programs which define their semantics.

Let us illustrate the language on the previous WSN example. The query can be expressed very simply by a predicate of the form:

$$?WarnPos(v, x, y)$$

where v is a node Id and (x, y) its positions. The meaning of the query is defined by a program (agent), which is used to evaluate it. Let us consider the following program:

$$\uparrow WarnPos(v, x, y) : - \ Pos(v, x, y), Tmp(v, t), t > T. \quad (1)$$

We assume that each node, say v, stores its location (x, y) as $Pos(v, x, y)$, and its temperature t as $Tmp(v, t)$. When the agent on a node, say α, receives a query $?WarnPos(v, x, y)$, it checks if it matches the head of a rule. In this case, it matches Rule (1). Its body, $Pos(v, x, y), Tmp(v, t), t > T$, is instantiated with local data, and the tuples (v, x, y) satisfying the query are produced as answers to the query and sent ($\uparrow$ in front of the rule) to the source of the query.

Let us consider now the more complex example, of nodes v with location (x, y), which have, together with their neighbors, a temperature greater than T. We assume that each node v also stores links to its neighbors, say w, as $Link(v, w)$. The following program defines the new query.

$$\begin{aligned}\uparrow WarnPos(v, x, y) : - \ & Pos(v, x, y), Tmp(v, t), t > T, \\ & \forall w \ Link(v, w), ?HighTmp(@w). \quad (2)\\ \uparrow HighTmp(v) : - \ & Tmp(v, t), t > T. \quad (3)\end{aligned}$$

The program is interpreted as follows. The query now matches Rule (2). This rule is interpreted as follows. Its body contains facts $Pos(v, x, y)$; $Tmp(v, t)$; as well as an expression:

$$\forall w \ Link(v, w), ?HighTmp(@w).$$

The facts are instantiated locally as above. The new query $?HighTmp(@w)$ is generated for each neighbor w of v (universal quantifier), and sent to each neighbor w (symbol "@" in front of the variable). Suppose that there are nodes β and γ such that $Link(\alpha, \beta)$ and $Link(\alpha, \gamma)$ hold on α. Then α generates two new queries, $?HighTmp(@\beta)$ and $?HighTmp(@\gamma)$, which have to be sent to node β and γ respectively.

Suppose that neighbor β receives the query $?HighTmp(\beta)$. It matches the *head* of rule (3). This matching leads to $Tmp(\beta, t), t > T$, the body of Rule (3). If the rule is satisfied, then the *head*, $HighTmp(\beta)$, of the rule is generated and sent to α, due to the affectation operator ($\uparrow$), where α is the *origin* of the query. The evaluation of the query $?HighTmp(\gamma)$ is done in a similar fashion on node γ. The results of the initial query $WarnPos(\alpha, x, y)$ will be computed by Rule (2) once *all* the answers to the queries $?HighTmp(@w)$ have been obtained. This is the meaning of the $\forall$ symbol in front of variable w in the body of Rule (2). Then the result is sent to the initial source of the query $?WarnPos(v, x, y)$.

With Questlog, complex applications and protocols can be expressed easily. Consider for instance the query $?Route(\alpha, d, y, n)$, searching for a next hop y, and a length n, for a route from node α to destination d. The following two rules, Rule (4) and (5), define an *on-demand routing protocol*, which allows to evaluate the initial query $?Route(\alpha, d, y, n)$.

$$\begin{aligned}\updownarrow Route(x, w, w, 1) : - \ & Link(x, w). \quad (4)\\ \updownarrow Route(x, w, z, n+1) : - \ & Link(x, z), \\ & ?Route(@z, w, u, n). \quad (5)\end{aligned}$$

When node, say α, fires a query $?Route(\alpha, d, y, n)$, the agent on α checks if it matches the head of a rule. The matching results in the body $Link(\alpha, d)$. Two scenarios are then possible. Either, with Rule (4), $Link(\alpha, d)$ holds on node α, and the query can be answered by $Route(\alpha, d, d, 1)$ (d is a neighbor of node α), or Rule (5) generates a body $Link(\alpha, z), ?Route(@z, d, u, n)$, containing a fact $Link(\alpha, z)$, and a new query $?Route(@z, d, u, n)$. Suppose there is a node β such that $Link(\alpha, \beta)$ holds on α. Then Rule (5) generates a new query, $?Route(@\beta, d, u, n)$, which has to be sent to node β.

Suppose now that node β receives the previous query, and that $Link(\beta, d)$ holds on β. The query is evaluated on node β, in a similar fashion. The agent on β can now run Rule (4), and answer the query with $Route(\beta, d, d, 1)$. Two actions are then performed. First, the result is stored in the local store. This is due to the affectation operator ($\downarrow$) in front of Rule (4). Second, the result has to be sent to α, due to the affectation operator ($\uparrow$), where α is the *origin* of the query. When the agent on α receives the answer from the agent on β, it uses again Rule (5), but now in push mode to derive the answer to the query, $Route(\alpha, d, \beta, 2)$, and stores it. As a side effect, intermediate nodes that aggregate answers of subqueries save routes to the destination.

Messages are formed by a payload and a destination. The payload can consist either of data or queries. Similarly, the destination consists of an explicit address, and an implicit address, defined by a query. When the destination of a message is only implicitly known as a query Q, two strategies are possible. Either, Q is included in the destination part of the message, which is then handled only by node satisfying it, or it is included in the payload, and handled by all nodes. We will see in the sequel that it results in different evaluation strategies.

More generally, when the destination as well as the payload are represented by queries, we distinguish in messages between two queries:

- *content-query*: query in the payload,
- *dest-query*: query in the destination.

The *dest-query* might be very simple to solve. Only if a node satisfies the *dest-query*, is it authorized to read and compute the *content-query*. Interestingly, this distinction also allows to optimize the distributed computation of queries.

3. The Questlog language

The language Questlog is used to program the behavior of nodes. We are interested in networks, where the nodes have initially only the knowledge of their neighbors. The $Link$ relation is thus distributed over the network such that each node has only a fragment of it. This can be done with an agent that communicates periodically with other agents on nodes in its transmission range to update the $Link$ relation upon node joining or leaving. The *Questlog* programs are agents and are installed on each node, where they run concurrently. The computation is distributed and the nodes exchange information.

Agents interact with each other on the same node. They can query and update the data on the nodes. They interact also with agents on other nodes in the network by producing messages to send on the network. Questlog has been designed to pull data from a network. As it has been shown in Section 2, agents are used in association with a predicate, (e.g. $WarnPos$ in Rule (2) for instance) defining a query, which is solved by running the associated agent.

Before describing the language, let us explain the behavior of queries and agents. The evaluation of the rules combines *backward* and *forward chaining*. Intuitively, when an agent on a node receives a query, it identifies the rules whose head matches the query. If there are such rules, the node applies each of them, that is it generates their body instantiated with the variable substitution imposed by the initial query.

There are two possibilities for the body. The body might be *simple*, with no subquery included, it is then evaluated locally on the node, the answer to the query is deduced by applying the rule in a forward manner, and then sent to the requesting node. If the body is *complex*, with subqueries, then the part of the body without subqueries is evaluated locally. The partial results obtained, lead to partial instantiation of the subqueries, which are then sent to the appropriate nodes. Some bookkeeping is performed to keep track of the initial queries and the corresponding subqueries. When the answers are received, the initial query can be computed, and its answer sent to the requesting node.

The Questlog *queries* are of a very simple form: $?R(x_1, \cdots, x_\ell)$, where R is a relation symbol of arity ℓ, and $x_1, \cdots, x_\ell$ are variables or constants. They are associated to rule *programs* which define their semantics. *Questlog programs* are agents that consist of sets of rules that are executed in parallel.

We introduce Questlog and the primitives of the language through examples. Let us start with routing which is a fundamental functionality for network applications. *On-demand routing* protocols, such as AODV [30], are reactive protocols that flood the network with a route request to find a route from a source to some destination. When the route is found, each node along the route saves locally the next hop to the destination.

We have seen in Section 2, Rules (4) and (5), which express a simple route request. On-demand routing requires some more care though. Indeed, the rules are evaluated in parallel, and the previous two rules could lead at the same time to an answer to the query as well as to useless subqueries propagated to other nodes. For instance, suppose that the $Link$ relation has two facts corresponding to $Link(\alpha, d)$ and $Link(\alpha, \beta)$, where d is the requested destination. Then, Rule (4) leads to a fact $Route(\alpha, d, d, 1)$ as an answer to the query saved locally on α and sent to the source of the query, while Rule (5) leads to a useless subquery $?Route(@\beta, d, u, n)$ sent to neighbor β.

To prevent propagating subqueries when an answer of a query is found locally, we use *negation*. Accordingly, the following routing program, Rule (6) and (7), will be used to evaluate an on-demand routing query. Rule (7) makes use of the literal "$\neg Link(x, w)$" which can be interpreted as follows: there is no link from node x to destination w.

$$\updownarrow Route(x, w, w, 1) :- Link(x, w). \tag{6}$$

$$\updownarrow Route(x, w, z, n+1) :- \neg Link(x, w), Link(x, z), ?Route(@z, w, u, n). \tag{7}$$

When node α fires the query $?Route(\alpha, d, y, n)$, the agent on α checks if it matches the head of a rule. The matching rules, Rule (6) and (7), are loaded, and executed in parallel to evaluate the query. The two rules are instantiated by the instances of the variables in the query. Rule (6) leads to the body $Link(\alpha, d)$. Suppose node d is a neighbor of node α, then Rule (6) is satisfied.

The results of the rules can be either stored locally on the node, or send to other nodes. The arrow in front of the rules specifies it, with $\downarrow$ for local storage, and $\uparrow$ for results sent to the origin of the query, and $\updownarrow$ for both. The deduced answer, $Route(\alpha, d, d, 1)$, is stored in the local data store ($\downarrow$ in front of Rule (6)), and has to be sent ($\uparrow$) to the *origin* of the query. However, Rule (7) generates a body that is not satisfied since the fact $Link(\alpha, d)$ holds on node α.

Intermediate nodes that aggregate answers of subqueries save ($\downarrow$) routes to the destination. It would be interesting to use local knowledge of nodes to reduce the delay and the complexity in both communication and computation. An additional rule is required. The following program with Rules (8), (9), and (10) defines the semantics of the on-demand routing protocol.

$$\uparrow Route(x, w, \diamond y, n) :- Route(x, w, y, n). \tag{8}$$

$$\updownarrow Route(x, w, w, 1) :- Link(x, w), \neg Route(x, w, \_, 1). \tag{9}$$

$$\updownarrow Route(x, w, z, n+1) :- \neg Link(x, w), \neg Route(x, w, \_, \_), Link(x, z), ?Route(@z, w, u, n). \tag{10}$$

Suppose intermediate node γ has a fact, $Route(\gamma, d, \theta, 2)$, saved in the routing table. Rule (8), when receiving the query $?Route(\gamma, d, y, n)$, leads to the body $Route(\gamma, d, y, n)$. The rule is satisfied, then deduced result $Route(\gamma, d, \theta, 2)$ is sent ($\uparrow$) to the source of the query. In case of plurality, one route can be chosen non-deterministicaly using the choice operator, $\diamond$ in front of y. Alternatively, the shortest route can be chosen using aggregation, (e.g. $Route(x, w, y, min(n))$). The evaluation of Rule (9) leads to the body $Link(\gamma, d), \neg Route(\gamma, d, \_, 1)$, where underscore means "any value". The fact "$\neg Route(\gamma, d, \_, 1)$" is read as follow: there is no route from γ to d with next hop any value and number of hop is 1. The use of the negation prevents Rules (9) and similarly for Rule (10) to be satisfied when a route is found locally. This concludes of the on-demand routing protocol.

Let us now consider an example of *aggregation query* over sensor networks. Suppose that a tree rooted on a node α has been constructed in the network. Each node, say x, has the relation $Tree(x, y)$ where y is a child of x, and stores a temperature value t in a relation $Tmp(x, t)$. Suppose node α fires the query, $?ResultAvg(\alpha, v)$, asking for the average, v, of the temperature values of deployed sensors in the network. The following program defines its semantics.

$$\downarrow ResultAvg(x, v) :- v := t/n, ?Avg(@x, n, t). \tag{11}$$

$$\uparrow Avg(x, 1, t) :- \neg Tree(x, \_), Tmp(x, t). \tag{12}$$

$$\uparrow Avg(x, \Sigma n + 1, \Sigma v + t) :- \forall y\, Tree(x, y), Tmp(x, t), ?Avg(@y, n, v). \tag{13}$$

where $Avg(x, n, t)$ stores the number n of nodes in the tree rooted at x with the sum t of their temperatures. When node α initially fires the query $?ResultAvg(\alpha, v)$, the agent on α checks if it matches the head of a rule. The matching leads by Rule (11) to the body $v := t/n, ?Avg(@\alpha, n, t)$ which gives raise to a new query $?Avg(@\alpha, n, t)$.

The matching of the new query leads either to the body $\neg Tree(x, \_), Tmp(x, t)$ of Rule (12) if α is a leaf (i.e. satisfies $\neg Tree(\alpha, \_)$), or otherwise to $\forall y\ Tree(\alpha, y)$,$Tmp(\alpha, t)$, $?Avg(@y, n, v)$ by Rule (13). In this later case, a series of queries $?Avg(@y, n, v)$ are generated, which are sent to all the children y of α in the tree. The computation will recursively walk down the tree until reaching the leaf nodes. Suppose nodes γ and λ are two leaf nodes, and node β is their parent. When receiving the query $?Avg(@\gamma, n, v)$ on node γ, Rule (12) is satisfied, and deduced result $Avg(\gamma, 1, t)$ is sent to the source β of the query. Node λ evaluates similarly the query $?Avg(@\lambda, c, v)$.

The results of the query on parent node β will be computed by Rule (13) once *all* the answers to the queries $?Avg(@y, n, v)$ have been obtained, according to the $\forall$ symbol in front of variable y in the body of Rule (13). After the computation, the deduced result $Avg(\beta, \Sigma n + 1, \Sigma v + t)$ is sent ($\uparrow$) to the source of the query. The operator $\sum$ is the function *sum* and it is used to sum the number of children as well as their temperature. Node β increases by 1 the number of nodes, and adds its temperature to the sum of temperatures before sending the result to the source node. Rule (13) will perform a converge-cast of the intermediate results. When agent on node α receives the answer for the query $?Avg(\alpha, n, t)$, by Rule (11), it deduces the average temperature. It uses the assignment literal ":=" together with arithmetic operations (e.g. division "/"). The result is saved locally in the relation $ResultAvg$.

Due to fragile conditions, the measured temperature value of individual sensor nodes might be wrong. To improve the stability of such systems, it is possible to update temperature stored in the Tmp relation on each sensor node with new values such as the average temperature of their neighbors. The query $?Tmp(w, u)$ is fired from some node, say α, with all destinations.

$$\downarrow Tmp(x, avg(t)) :- !Tmp(x, t_1), \forall y\ Link(x, y), ?GetNghTmp(@y, t) \quad (14)$$

$$\uparrow GetNghTmp(x, t) :- Tmp(x, t). \quad (15)$$

On each node, say β, the query $?Tmp(\beta, u)$, matches the head of Rule (14) thus leading to the body $!Tmp(\beta, t_1), \forall y\ Link(\beta, y)$, $?GetNghTmp(@y, u)$. It gives raise to queries of the form $?GetNghTmp(@y, u)$ sent to all neighbors y. Each neighbor upon receiving the query, Rule (15), forwards ($\uparrow$) its own temperature value to the query expeditor β. When all answers (according to $\forall$) are received, Rule (14) continues the evaluation in the push mode, results in the head with a new value t stored ($\downarrow$) on β where t is the average temperature which is defined using aggregation.

The *consumption operator*, !, is used to delete the facts that are used in the body of the rules from local data store. The fact $!Tmp(\beta, t_1)$ is deleted upon evaluating the rule in the push mode.

Consider now an application where the sink node floods a query to sensor nodes to collect on-demand the sensed data about an object of interest x. Each sensor node upon receiving the query forwards its sensed data (e.g. the temperature value) to the sink node. This query can be mapped to a rule-based program which models its semantic. Suppose that the sink node floods the query $?GetData(w, x)$ to all nodes with destination all.

$$\uparrow GetData(x, t) :- Tmp(x, t). \quad (16)$$

Each node, say ν, stores its temperature in the relation Tmp. Upon receiving the query, the agent uses Rule (16) to evaluate it. Deduced result $GetData(\nu, t)$ is sent ($\uparrow$) to the sink.

In most approaches, all deployed sensor nodes are homogeneous and mono-service, and run one application at a time (e.g. measuring the temperature). It is worth noting that Questlog can express applications and protocols running on heterogeneous devices with mono- or multi-services.

In the next example, we explain the use of destination queries. Assume the sink node sends a message that contains (i) a *content-query* in the payload, and (ii) a *dest-query* in the destination. We have seen in the previous example with Rule (16) that data collection might involve all nodes in a network. However, due to sensor power constraints, it might be preferable [21] that data collection be performed from a subset of nodes only.

Assume that the sink node calls sensor nodes that have energy level greater than a threshold as cluster heads, to collect data (e.g. temperature) from their neighbors, aggregate the data, and then send aggregated value with the address of the cluster head to the sink. The sink node sends a message with *content-query* $?Collect(x, s)$ and *dest-query* $?Powerful(x)$ in the network. Suppose that the energy level is saved in the relation $Energy$. The following program defines its semantics:

$$Powerful(x) :- Energy(e), e > n. \quad (17)$$

$$\uparrow Collect(x, avg(s)) :- \forall y\ Link(x, y), ?GetData(@y, s). \quad (18)$$

$$\uparrow GetData(y, h) :- Tmp(x, h). \quad (19)$$

Each sensor node, say ν, upon receiving the message evaluates the *dest-query* $?Powerful(\nu)$ using Rule (17) after matching the head of the rule. If the body of the rule $Energy(e), e > n$ is satisfied, then the sensor node ν belongs to the destination, and is now allowed to evaluate the *content-query* $?Collect(\nu, s)$. Otherwise, the message is sent further.

The *content-query* matches the head of Rule (18), which leads to the evaluation of its body $\forall y\ Link(x, y)$, $?GetData(@y, d)$ that gives raise to queries $?GetData(@y, s)$ sent to all neighbors y of ν. Each neighbor upon receiving the query $?GetData(y, s)$, uses Rule (19) after matching and returns its temperature value to the source ν of the query. When all answers ($\forall$) are received, node ν continue the evaluation of Rule (18) in the push mode, leading to a fact $Collect(\nu, s)$ where s is the average temperature sent ($\uparrow$) to the sink.

A simplified grammar of the Questlog language is shown below. Optional items are enclosed in square brackets, and items repeating zero or more times are enclosed in curly brackets.

$$
\begin{aligned}
query &::= ?\ ident\ "("\ @term\ \{\ ","\ term\ \}\ ")" \\
term &::= (var \mid cst) \\
rule &::= (\ \uparrow \mid \downarrow \mid \updownarrow\ )\ \ head\ \ ":-"\ body \\
head &::= ident\ "("\ term\ \{\ ","\ term\ \}\ ")" \\
body &::= \{\ literal\ \{\ ","\ literal\ \}\ \}\ \ [","\ query]\ "." \\
literal &::= ([\ \neg \mid !\ \mid \forall\ \overline{var}\ ]\ atom\ ) \mid condition \\
atom &::= ident\ "("body\_term\ \{\ ","\ body\_term\ \}\ ")" \\
body\_term &::= exp \mid "\_" \\
exp &::= exp * exp \mid exp + exp \mid cst \\
condition &::= exp\ condition\_op\ exp \\
condition\_op &::= "=" \mid "\neq" \mid ">" \mid "\geq"
\end{aligned}
$$

4. Procedural Semantics

We make little assumptions on the networks. We consider nodes which communicate by exchanging messages as restricted mobile agents. The communication is asynchronous with no shared memory.

Each node is equipped with an embedded machine (Figure 1) which evaluates the Questlog programs. It is composed of three main components: (i) a *router* to handle the communication with the network; (ii) an *engine* to evaluate the queries; and (iii) a local *data store* to manage the information (Data and Programs) local to the node. The Questlog programs are installed on each node, and used to evaluate Questlog queries fired by the applications or received from other nodes through mobile agents.

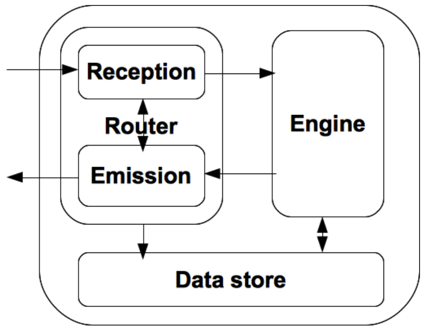


Figure 1. Global architecture of the virtual machine

The evaluation may lead to data (as answers) or subqueries sent to other nodes in the network. Pending queries need to be stored, some bookkeeping is thus performed in the local data store with timeouts. When an answer of a pending query is received, the corresponding query is retrieved and the evaluation is resumed.

4.1 Messages and Routing

We have seen in Section 2 that a message is composed of a payload and a destination. To define precisely the procedural semantics, additional informations in a message are also required. In particular, the source node address, the payload query Id, and the TTL (time-to-live). The TTL is the number of hops that a message is permitted to travel before being discarded by the router. A message has thus the following format:

$$msg = \ < src, qId, payload, dest, ttl >$$

The $payload$ is the content of the message which may contain either a query or data. It has the following format:

$$payload = \ < query \ | \ answer >$$

The $dest$ is the destination of the message. It is composed of both *extensional* and *intensional* destination. The extensional destination is defined by a node address, while the intensional destination is defined by a query. It has the following format:

$$dest = \ < extDest : intDest >$$

The Router is composed of two main modules: (i) Reception module that receives messages from the network, and (ii) Emission module to send messages to other nodes in the network.

When receiving a message, a node first checks the destination. Two cases have to be considered corresponding to extensional and intensional destinations. If the extensional destination is equal to the node address, then the node stores the received message in a local data structure ($BookKeeping$) with a unique Id and a timeout, and transfers the payload to the engine. Otherwise, the node address is not the destination, and the message is transferred to the emission module. For instance, when node β receives the message:

$$msg_1 = \ < \alpha, 4, payload, < \beta : - >, 10 >$$

with address specified extensionally by β which is equal to the node address. Then the message is stored locally ($BookKeeping$), and the $payload$ is transferred to the engine. However, if β receives a message with $dest$ = $< \gamma : - >$, then the message is transferred to the emission module since γ does not match the node address.

Consider now the second case. If the extensional destination is empty, then the router evaluates the intensional destination.

$$msg_2 = \ < \alpha, 4, payload, < - : query >, 10 >$$

The evaluation of the intensional destination $query$ passes through the engine, and the result is a set (σ_{Ans}) of node addresses. When receiving the set of answers, the router checks if the node address is in the set. If so, the router stores locally ($BookKeeping$) the message msg_2, and transfers the payload to the engine to be evaluated. Otherwise, the message is discarded.

At the same time, the initial message msg_2 is transferred to the emission module to be sent to other nodes. It is noteworthy to mention that an alternative strategy could have been used. For instance, instead of transferring the message msg_2 to the emission module, the router could take into consideration the set of answers (e.g. α, β, etc.), encapsulates messages based on msg_2 but with new destinations specified *extensionally* and *intensionally* (e.g. $msg_3.dest =< \alpha : query >, msg_4.dest =< \beta : query >, etc.$), and transfers them to the emission module. The important features of this strategy is: (i) toggling from broadcast mode into unicast mode, and (ii) benefiting from local knowledge of a node. The choice of the strategy can be made by an agent.

The Emission module is used to send messages to other nodes in the network. The router fetches the next hop to the extensional destination from the routing table and sends the message if the next hop is found. Otherwise, the message is discarded. Here again, other strategies can be made and applied as for instance: (i) send the message to neighbors, or (ii) fetch a route to the destination d which in our approach requires to fire the query $?Route(s, d, nh, n)$ while s is the node address, nh is the required next hop, and n is the number of hops, as we have seen in Section 2.

4.2 Computation

A message may contain queries (content-query, dest-query) or data. To evaluate a query, as seen in Figure 2, the mobile agent collaborates with local node agents such as Program agent, Timer agent, Strategy agent, etc. to achieve the task and produce a new mobile agent.

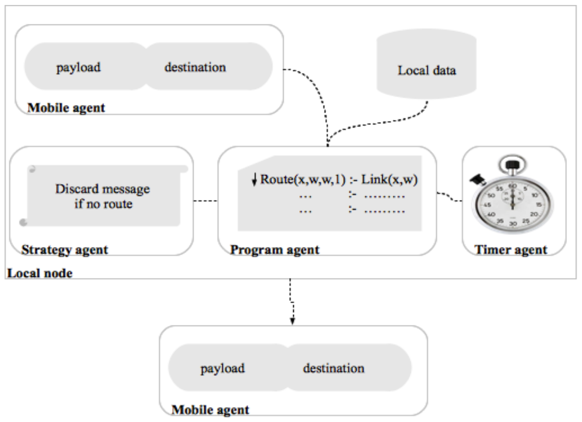
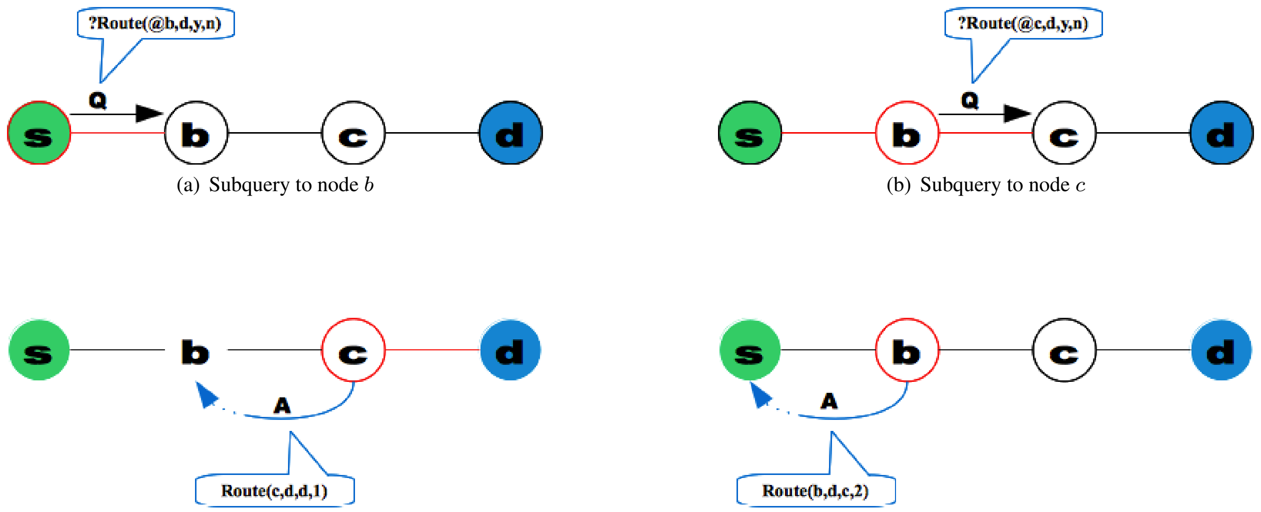


Figure 2. Emission of messages seen as mobile agents

A Timer agent manages program time events and timeout. The timer is defined as a high level specification as follows:

$$Timer(TimerName, Period, Occurrence, ProgramName)$$



(a) Subquery to node b (b) Subquery to node c (c) Intermediate answer to node b (d) Intermediate answer to node s

Figure 3. Propagation of subqueries and converge-cast of intermediate answers

where the name of the timer, the period to wait before sending an event, the occurrence for repetitive events, and the name of the program are specified.

The engine is in charge of evaluating the received queries/answers. The engine is constructed around two main modules to evaluate them (i) the query module, and (ii) the data module. The query module initiates the evaluation of queries, which may result either in a direct answer to be sent to the query origin, or to subqueries to be sent to other nodes in the network. The data module is used to carry further the computation, and evaluate answers and subsequently pending queries, which may result in an answer saved locally or sent to other nodes.

For each message, its content/dest-query is analyzed and transferred to the corresponding module. When receiving a query, the query module first loads the appropriate rules from the local data store. More precisely, the received query is matched with the *head* of each rule, and only matching rules are loaded. The rules are then evaluated in parallel. The first step towards their evaluation is the substitution of variables by constants. Rules are instantiated by: (i) potentially the constant values of the received query, and (ii) the local data of the node (where the evaluation is taking place).

Rules can be of two kinds: (i) simple rules, or (ii) complex rules. Simple rules have no subquery in their body, and are evaluated locally on the node. Potentially, local data might satisfy the query, resulting in an answer to be sent to the node source of the query. However, complex rules have subqueries in their body, and their evaluation leads to subqueries propagated to their appropriate destinations.

After the evaluation, two kinds of outputs, either (i) a query, or (ii) an answer can be produced.

- If the result is a query, then the destination to where the query should be sent is extracted from the query. The destination is the instance of the variable prepended by @ in the subquery (e.g. $?Route(@\beta, d, y, n)$). After that, the result is encapsulated in a message which is stored in the local data store ($BookKeeping$), and then transferred to the router.
- If the result is an answer, as a fact (e.g. $Route(\beta, d, \gamma, 2)$), then according to the affectation operator of the corresponding rule, the result (i) is stored locally ($\downarrow$), or (ii) sent ($\uparrow$) to the source node, or both stored and sent ($\updownarrow$). The result will be sent in a message, and that requires to collect some information. In particular, the address of the source node of the query is the destination of the message to which the result will be sent. The qId of the message should be the same as the Id of the initial query. The corresponding entry that holds these data is retrieved from the local data store ($BookKeeping$). After that, the message is encapsulated and transferred to the router.

Let us consider for instance the on-demand routing protocol, and suppose node source s fires the query $?Route(s, d, y, n)$ asking for a route to destination d. Figure 3 shows an example of a trivial network where node s fires the query. On node s, the engine matches the query with the head of Queslog rules saved in the local data store, and loads only the corresponding rules with matching heads. Loaded rules correspond to Rules (8), (9), and (10) shown in Section 3. The engine evaluates the rules in parallel. Only Rule (10) is satisfied, since there is neither a direct link to the destination d, nor a route, thus leading to a subquery $?Route(@b, d, y, n)$ as shown in Figure 3(a). Similarly, node b loads relevant rules and evaluates them leading to subqueries $?Route(@s, d, y, n)$, and $?Route(@c, d, y, n)$ since node s and c are neighbors as shown in Figure 3(b). However, the subquery $?Route(@s, d, y, n)$ can be avoided either by the engine upon evaluation of the initial query (do not send subquery to the source of the initial query) or discarded by the router of node s.

The engine on node c loads and evaluates the relevant rules in a similar fashion. However, the evaluation leads to the head of Rule (9), $Route(c, d, d, 1)$, as an answer of the query since the destination d is a neighbor as shown in Figure 3(c). The answer is saved in the local data store of node c and sent to the source node of the query. The engine determines the source node by retrieving the appropriate entry in the local data store based on a unique local Id. When receiving the answer $Route(c, d, d, 1)$, the engine on node b uses the data module to continue the evaluation as we will see in the following.

The data module is used to continue the evaluation of pending queries stored locally on a node. When receiving an answer, the data module first loads the appropriate rules from the local data store. More precisely, the engine knows the message qId, communicated by the router, with the payload. The engine matches the received qId with each entry in the $BookKeeping$ data structure, and retrieves the corresponding Questlog rules if the head is matched. Then, the engine evaluates the rules in parallel but now in the push mode. Deduced results are again sent to their appropriate destination exactly as we have seen previously.

When receiving the fact $Route(c, d, d, 1)$, the engine on node b in Figure 3(d), matches the qId with the query Id on the $BookKeeping$ data structure and loads the corresponding rule, Rule (10), which is evaluated in push mode. The evaluation leads to a new fact $Route(b, d, c, 2)$ saved locally on b and sent to the source node s.

5. Implementation

In this section, we present the system which supports the Questlog queries together with their corresponding programs. The network is constituted of nodes that have a unique *identifier*, *Id*, taken from $1, 2, \cdots, n$, where n is the number of nodes. Their communication are based on asynchronous message exchange, and have no shared memory. We make no particular assumption on the nodes/devices which all have the same architecture and the same behavior.

The Questlog programs are transformed into a sort of bytecode that can be smoothly handled. We compile the Questlog programs into an SQL dialect that is executed by the engine. An SQL query is built for each Questlog operator (query "?", store "↓", push "↑" and deletion "!") in a rule. Consider the following rule witch contains a subquery in the body:

$$\uparrow Route(x, y, z) :- \neg Link(x, y), Link(x, z), ?Route(@z, y, s). \quad (20)$$

The compiler transforms Rule (20) into two SQL queries, as shown in the following, corresponding to: (i) the results of the operator "?" (body SQL query) to be used in the pull mode for subquery, and (ii) the operator "↑" (head SQL query) to be used in the push mode when receiving an answer in the body of a rule.

```
SELECT Link.z, Route.y, Route.z
 FROM Route, Link AS Lk1
 WHERE Lk1.x=self,
  AND NOT EXISTS (
   SELECT Link.x, Link.y
    FROM Link AS LK
    WHERE LK.x=LK1.x
    AND LK.y=Route.y);

 SELECT Link.x, Route.y, Link.z
  FROM Link AS Lk1, Route
  WHERE Lk1.z=Route.z
  AND Lk1.x=self,
  AND NOT EXISTS (
   SELECT Link.x, Link.y
    FROM Link AS LK
    WHERE LK.x=LK1.x
    AND LK.y=Route.y);
```

The first attribute in the predicate of the head of a rule represents the node address, and it is used as a location specifier. The negation of $Link$ is translated with the SQL subquery into the section $not\ exists$. It is worth noting that the operators "↓" and "!" in a Questlog rule are transformed into an insert and delete SQL query respectively.

Each node is equipped with an embedded machine as we have seen in Section 4. We implemented an extended version of the *Netquest machine* (Figure 4) presented in [18]. Two important functionalities have been introduced (i) a *Router module* to evaluate intensional destinations and to communicate with the network, and (ii) a *Questlog Engine* to execute the Questlog queries and programs.

The Netquest Virtual Machine executes the bytecode, generated by the compiler, and manipulates data and messages. It is working as a daemon in the device, and applications can use it to communicate with other devices on the network. The virtual machine is portable and can be installed in small devices with embedded DMS. A previous implementation was done in iMote sensors [7].

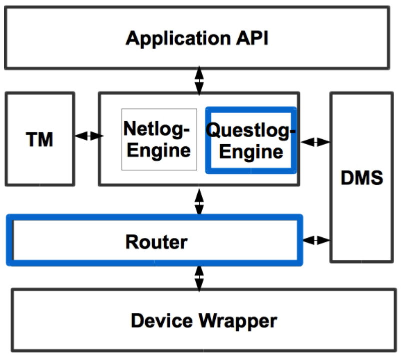


Figure 4. Architecture of the Netquest virtual machine

The Netquest Virtual Machine was initially proposed to evaluate Netlog [18] programs. It is composed of six components; (i) the device wrapper receives and sends data over the network, (ii) the DMS evaluates the bytecode and manipulates data, (ii) the router receives and sends Netquest messages through the device wrapper and chooses the next hop to route a message, (iv) the Netlog engine evalutes Netlog programs by loading the rules and evaluating them through the DMS, (v) the timer manager creates and manipulates timers and manages the time event of the system, and (vi) the application API is in charge of the interaction with local applications.

We next describe the new modules which were added to the Netquest machine, the *Questlog Engine* and the *Router*, together with their functionalities.

The Questlog engine executes received queries, either from local application or from mobile agents, based on the Questlog programs stored in the local data store. In the proposed model, the message is a mobile agent that may contain a Questlog query. It interacts with local agents that have Questlog programs at their disposition in order to execute the query. They all collaborate to achieve a task. More precisely, the query received by mobile agent is matched with the head of rules of an agent program, that in its turn may use the timer agent, the routing agent, the neighborhood agent, etc. to finally solve the query.

The engine maintains a data structure, *BookKeeping*, to store queries and answers together with information such as the origin of the query. The Questlog engine is composed of four modules:

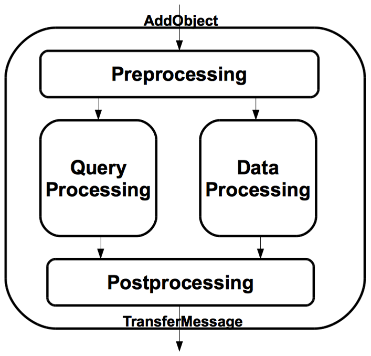


Figure 5. Architecture of the Quesltog engine

1. Preprocessing: This module analyses the incoming messages, in particular the *payload* of the messages. If the content is a query, then the module *query processing* is called to treat the query, otherwise the content is an answer and so the module *data processing* is called.
2. Query processing: This module computes the Questlog queries. For each query, the corresponding rules are retrieved from the local data store. More precisely, a matching operation is performed between the received query and the head of Questlog rules, and then the corresponding SQL queries are retrieved. After that, the SQL queries are executed through the DMS, thus resulting either in an answer for the query, or to the generation of a subquery to be sent to other node. In both cases, the result will be transferred to a *postprocessing* module.
3. Data processing: This module handles data as answers of queries. The corresponding SQL queries are retrieved based on the qId of the received message and the query Id stored in the BookKeeping table. Then, if the corresponding rules contain forall ($\forall$), the SQL queries will not be executed till getting all answers. A local data structure is used to save corresponding received answers. Otherwise, the SQL queries are executed through the DMS and deduced facts are transferred to *postprocessing* module.
4. Postprocessing: This module generates *payloads* in Questlog form by collecting subqueries or facts, fetches their corresponding destinations, encapsulates them in messages, and finally transfers the messages to the router.

The router handles the incoming and outgoing messages through the device wrapper. The specification of the router was described previously in Section 4.

Finally, to facilitate the programming of Questlog programs and to ensure their compilation, we extended the code editor presented in [10] with Questlog syntax coloring and error detection.

6. Simulation Results

The Questlog language is well-adapted to messages with intensional destinations as well as to application queries coming from an API or from external applications running in the network. The queries are on-demand and nodes may enter or leave the network at any time. Our objective here is to monitor the Questlog programs at run time and show their behavior. We thus used a platform that offers these functionalities. The QuestMonitor [10] system is a visualization tool that allows to interact with a network on a 2D graphical interface, and visualize the behavior of declarative protocols. It has three main components:

- The Network Editor: it allows to create groups of nodes, display their status, and install protocols on them;
- The Network Monitor: it allows to visualize different groups of nodes, modify the configuration (e.g. radio range) and interact with the network at run time (e.g. move nodes, delete links, delete nodes);
- The Node Monitor: it exhibits informations about the node selected by the user, allows to monitor the activity, display their data, color nodes and edges, and interact with individual nodes.

We have modified the API of the QuestMonitor system in order to allow a node to send Questlog queries in the network at run time. Figure 7 shows the API where we select a node that sends the query (e.g. $Node\ 1$), the program to be used (e.g. $OnDemandRouting$), and the appropriate query to be sent in the network (e.g. $?Route(1, 10, y, n)$). Figure 6 shows a small network where node source "1" fires a query $?Route(1, 10, y, n)$ to find a route to the destination "10". The parameters y and n are variables corresponding to the next hop and the number of hops respectively.

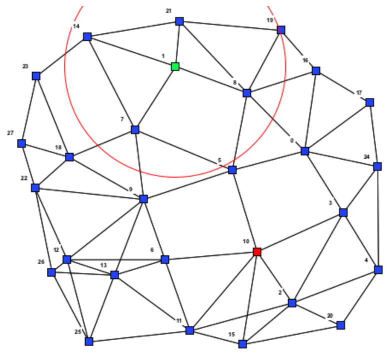
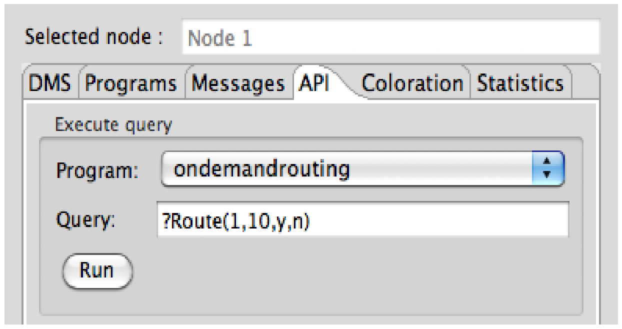


Figure 6. Network topology



Selected node : Node 1

Execute query

Program: ondemandrouting

Query: ?Route(1,10,y,n)

Figure 7. Application programming interface

The Questog programs are installed on each node of the network. Upon running the query $?Route(1, 10, y, n)$ from the API, it is transferred to the engine of node source "1" to be evaluated using, as we have seen in Section 4, Rules (8), (9), and (10). Each node propagates subqueries to neighbors (except neighbor source of the query) if it has no link to the destination. For clarity, we show in Figure 8 a reduced network that demonstrates the propagation of queries (e.g. $?Route(1, 10, y, n)$) in messages.

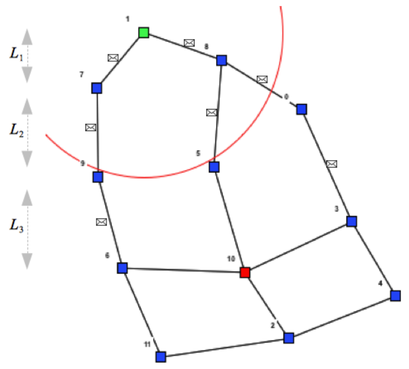


Figure 8. Propagation of queries/answers

The source node sends subqueries to its neighbors (L_1) which in turn repeat the same process (L_i) if no link or route to the destination is found. Intuitively, different routes with different lengths will be received by the source node. The converge-cast of answers by intermediate nodes on the *OnDemandRouting* program follows the same paths of subqueries propagation. Suppose that intermediate nodes have no route to the destination, and that the charge is distributed uniformly over all the nodes in the network, then the first answer received by the source node will be the shortest route. In Figure 8 for instance, node 5 is the first node that answers the query.

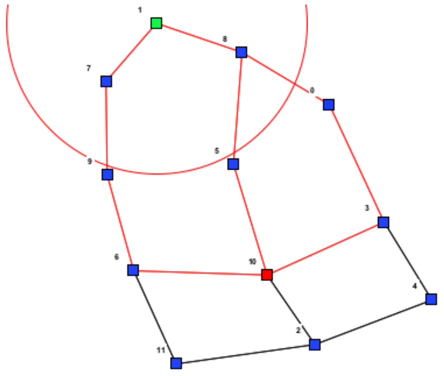
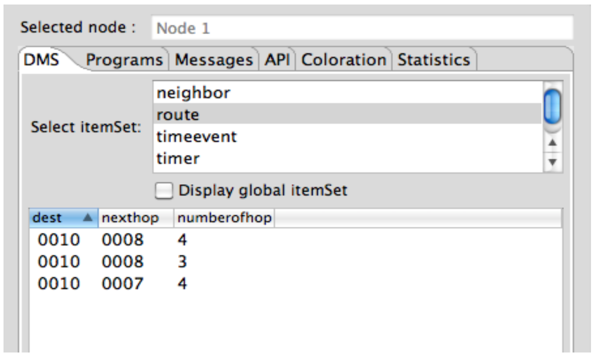


Figure 9. Routes coloration



Selected node : Node 1

Select itemSet: neighbor / route / timeevent / timer

Display global itemSet

dest	nexthop	numberofhop
0010	0008	4
0010	0008	3
0010	0007	4

Figure 10. Visualization of ItemSet route

Intermediate nodes aggregate answers to the source of the query. When receiving the answers, the source node 1 stores their discovered routes in the routing table as seen in Figure 10. Each time a route is built, it will be colored using the coloration feature of QuestMonitor, Figure 9. That allows us to visualize the behavior of declarative network protocols upon link or node failure or departure through direct interaction with the network. In addition, the tab ”Statistics” in Figure 10 calculates the number and the kind of queries executed on a node, and results on an average bound of complexity in communication and computation.

7. Conclusion

We have developed a setting which offers messages with implicit destinations, which can be seen as mobile agents, with limited mobile code. They are solved when meeting local agents which have the corresponding code and data to find the best available destination. They ease programming complex applications where the network is used as an active middleware. We proposed a data-centric language, Questlog, that allows to handle intensional destinations as queries and program complex strategies to evaluate them. We have illustrated the language over classical networking protocols, such as routing, and are currently developing sensor network applications as well as social network functionalities including communication, matching, games, etc. The operational semantics of Questlog has been implemented over the Netquest system, and we ran simple examples over the QuestMonitor platform, whose API has been extended to support interactive queries, and to visualize the execution of programs.

We are currently experimenting with the different programming strategies offered by intensional destinations, as well as studying the resulting overhead. These strategies allow to balance the request between the payload and the destination queries, leading to different evaluation schemes. In particular the use of intensional destination can offer persistence to data sent to nodes which have disappeared, and can be rerouted by reevaluating the intensional destination. We have demonstrated such techniques in another context in [3]. Social networks offer challenging reachability problems that we plan to address using this framework in the near future.

Acknowledgments

The authors would like to thank Eric Bellemon for his contribution to the first implementation prototype of the system [10], as well as Christophe Bobineau, Christine Collet, and Fuda Ma, for fruitful discussions. This work has been supported by the Agence Nationale de la Recherche, under grant ANR-09-BLAN-0131-01.

References

[1] M. Abadi and B. T. Loo. Towards a declarative language and system for secure networking. In *Proc. NETB'07*, pages 1–6. USENIX Association, 2007.

[2] S. Abiteboul, M. Bienvenu, A. Galland, and E. Antoine. A rule-based language for web data management. In *PODS*, pages 293–304, 2011.

[3] A. Ahmad-Kassem, E. Bellemon, and S. Grumbach. Seamless distribution of data centric applications through declarative overlays. *BDA'11*, October 2011.

[4] G. Alonso, E. Kranakis, C. Sawchuk, R. Wattenhofer, and P. Widmayer. Probabilistic protocols for node discovery in ad hoc multi-channel broadcast networks. In *Proc. ADHOC-NOW'03*, 2003.

[5] F. Bancilhon. Naive evaluation of recursively defined relations. In *On knowledge base management systems: integrating artificial intelligence and database technologies*, 1986.

[6] F. Bancilhon, D. Maier, Y. Sagiv, and J. D. Ullman. Magic sets and other strange ways to implement logic programs (extended abstract). In *PODS '86: Proceedings of the fifth ACM SIGACT-SIGMOD symposium on Principles of database systems*, pages 1–15, New York, NY, USA, 1986. ACM. ISBN 0-89791-179-2.

[7] M. Bauderon, S. Grumbach, D. Gu, X. Qi, W. Qu, K. Suo, and Y. Zhang. Programming imote networks made easy. In *The Fourth International Conference on Sensor Technologies and Applications*, pages 539–544. IEEE Computer Society, 2010.

[8] Y. Bejerano, Y. Breitbart, M. N. Garofalakis, and R. Rastogi. Physical topology discovery for large multi-subnet networks. In *Proc. INFOCOM'03*, 2003.

[9] Y. Bejerano, Y. Breitbart, A. Orda, R. Rastogi, and A. Sprintson. Algorithms for computing qos paths with restoration. *IEEE/ACM Trans. Netw.*, 13(3), 2005.

[10] E. Bellemon, V. DuboscIard, S. Grumbach, and K. Suo. Questmonitor: A visualization platform for declarative network protocols. In *MSV 2011: The 8th International Conference on Modeling, Simulation and Visualization Methods, Las Vegas, USA*, 2011.

[11] M. Caesar, M. Castro, E. B. Nightingale, G. O'Shea, and A. Rowstron. Virtual ring routing: network routing inspired by dhts. *SIGCOMM Comput. Commun. Rev.*, 36:351–362, 2006. ISSN 0146-4833.

[12] A. Carzaniga, M. J. Rutherford, and A. L. Wolf. A routing scheme for content-based networking. In *INFOCOM*, 2004.

[13] B. Chen, H. H. Cheng, and J. Palen. Mobile-c: a mobile agent platform for mobile c-c++ agents. *Softw. Pract. Exper.*, 36(15):1711–1733, Dec. 2006. ISSN 0038-0644. doi: 10.1002/spe.v36:15. URL `http://dx.doi.org/10.1002/spe.v36:15`.

[14] A. J. Demers, J. Gehrke, R. Rajaraman, A. Trigoni, and Y. Yao. The cougar project: a work-in-progress report. *SIGMOD Record*, 32(4): 53–59, 2003.

[15] Y. Deng, S. Grumbach, and J.-F. Monin. A framework for verifying data-centric protocols. In *FORTE 2011: The 31th IFIP International Conference on FORmal TEchniques for Networked and Distributed Systems*, Reykjavik, Iceland, 2011.

[16] P. T. Eugster, P. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, 2003.

[17] P. T. Eugster, B. Garbinato, and A. Holzer. Location-based publish/-subscribe. In *NCA*, pages 279–282, 2005.

[18] S. Grumbach and F. Wang. Netlog, a rule-based language for distributed programming. In *PADL'10, Twelfth International Symposium on Practical Aspects of Declarative Languages, Madrid, Spain*, 2010.

[19] A. Holzer, L. Ziarek, K. R. Jayaram, and P. Eugster. Putting events in context: aspects for event-based distributed programming. In *AOSD*, pages 241–252, 2011.

[20] D. Kotz, R. S. Gray, and D. Rus. Mobile agents: Future directions for mobile agent research. *IEEE Distributed Systems Online*, 3(8), 2002.

[21] L. Kulik, E. Tanin, and M. Umer. Efficient data collection and selective queries in sensor networks. In *GSN*, pages 25–44, 2006.

[22] C. Liu, Y. Mao, M. Oprea, P. Basu, and B. T. Loo. A declarative perspective on adaptive manet routing. In *Proc. PRESTO '08*, pages 63–68. ACM, 2008.

[23] B. T. Loo, T. Condie, J. M. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing declarative overlays. In *Proc. SOSP'05*, 2005.

[24] B. T. Loo, J. M. Hellerstein, I. Stoica, and R. Ramakrishnan. Declarative routing: extensible routing with declarative queries. In *Proc. ACM SIGCOMM '05*, 2005.

[25] B. T. Loo, T. Condie, M. N. Garofalakis, D. E. Gay, J. M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica. Declarative networking: language, execution and optimization. In *Proc. ACM SIGMOD'06*, 2006.

[26] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. Tinydb: an acquisitional query processing system for sensor networks. *ACM Trans. Database Syst.*, 30, 2005.

[27] A. Malekpour, A. Carzaniga, F. Pedone, and G. T. Carughi. End-to-end reliability for best-effort content-based publish/subscribe networks. In *DEBS*, pages 207–218, 2011.

[28] P. J. Marron and D. Minder. *Embedded WiSeNts Research Roadmap*. Embedded WiSeNts Consortium, 2006.

[29] J. L. Martins and S. Duarte. Routing algorithms for content-based publish/subscribe systems. *IEEE Communications Surveys and Tutorials*, 01 2010.

[30] C. E. Perkins. Ad-hoc on-demand distance vector routing. In *In Proceedings of the 2nd IEEE Workshop on Mobile Computing Systems and Applications*, pages 90–100, 1999.

[31] R. Ramakrishnan and J. D. Ullman. A survey of research on deductive database systems. *Journal of Logic Programming*, 23:125–149, 1993.

[32] A. Ricci and A. Santi. Designing a general-purpose programming language based on agent-oriented abstractions: the simpal project. In *SPLASH Workshops*, pages 159–170, 2011.

[33] U. Srivastava, K. Munagala, and J. Widom. Operator placement for in-network stream query processing. In *Proc. POCS'05*, pages 250–258, 2005.

[34] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *SIGCOMM Comput. Commun. Rev.*, 31, 2001.

[35] L. Vieille. Recursive axioms in deductive databases: The query/subquery approach. In *Expert Database Conf.*, pages 253–267, 1986.

[36] D. W. Wall. Messages as active agents. In *POPL*, pages 34–39, 1982.

A Decentralized Approach for Programming Interactive Applications with JavaScript and Blockly

Assaf Marron
Weizmann Institute of Science
assaf.marron@weizmann.ac.il

Gera Weiss
Ben-Gurion University
geraw@cs.bgu.ac.il

Guy Wiener
HP Labs
guy.wiener@hp.com

Abstract

We present a decentralized-control methodology and a toolset for developing interactive user interfaces. We focus on the common case of developing the client side of Web applications. Our approach is to combine visual programming using Google Blockly with a single-threaded implementation of behavioral programming in JavaScript. We show how the behavioral programming principles can be implemented with minimal programming resources, i.e., with a single-threaded environment using coroutines. We give initial, yet full, examples of how behavioral programming is instrumental in addressing common issues in this application domain, e.g., that it facilitates separation of graphical representation from logic and handling of complex inter-object scenarios. The implementation in JavaScript and Blockly (separately and together) expands the availability of behavioral programming capabilities, previously implemented in LSC, Java, Erlang and C++, to audiences with different skill-sets and design approaches.

Categories and Subject Descriptors D.3.3 [*Language Constructs and Features*]: Concurrent programming structures; D.1.3 [*Programming Techniques*]: Concurrent programming

General Terms Languages, Design, Human Factors

Keywords Behavioral Programming, JavaScript, Coroutines, HTML 5, Google Blockly, Visual Programming, Client-side, Web application, Browser

1. Introduction

The behavioral programming (BP) approach is an extension and generalization of scenario-based programming, which was introduced in [4, 11] and extended in [17]. In behavioral programming, individual requirements are programmed in a decentralized manner as independent modules which are interwoven at run time. Advantages of the approach include facilitation of incremental development [17], naturalness [10], and facilitation of early detection of conflicting requirements [18]. A review of research and tool development in behavioral programming to date appears in [14]. While behavioral programming mechanisms are available in several languages such as *live sequence charts* (LSC), Java, Erlang and C++, its usage for complex real-world application and development of relevant methodologies are only beginning.

The purpose of the research summarized in this paper was to examine behavioral programming in a specific application domain, and to adjust it to languages, technologies and work methods that are used in this domain. The paper also sheds light on the principles of programming behaviorally and the facilities required of behavioral-programming infrastructure.

The paper describes and demonstrates the implementation of behavioral programming in JavaScript and in Google's Blockly for the client side of Web applications, and then discusses general usability and design principles highlighted by these implementations. Clearly, in addition to the combined Blockly-and-JavaScript implementation shown here, behavioral programming can be used with JavaScript without Blockly, or with Blockly with translation to another language, such as Dart. In this regard, Blockly is a layer above our JavaScript implementation, which can simplify development and facilitate experimenting with a variety of programming idioms. We hope that together with previously described ideas about scenario-based and behavioral programming, this paper will help drive the incorporation of these principles into a wide variety of new and existing environments for development with agents, actors, and decentralized control, and will help add them into the basic set of programming concepts that are understandable by and useful for novice and expert programmers alike.

We propose that decentralized scenario oriented programming techniques offer significant advantages over traditional programming metaphors in this specific domain. Consider, for example, an application with some buttons on a screen where there is a requirement that the software reacts to a sequence of button clicks in a certain way. Using a non-behavioral style with, e.g., JavaScript, one of the standard programming languages in this domain, the programmer would handle each button-click separately, and introduce special code to manage state for recognizing the specific sequence of events. We argue that with behavioral programming such a requirement can be coded in a single imperative script with state management being implicit and natural rather than explicit. See Section 5.1.

Our choice of the domain of interactive technology is influenced also by the current debate about the relative merits of Flash and JavaScript/HTML5 technologies (see, e.g., [28]). We believe that the technological discussion conducted mainly in industry should be accompanied with academic revisiting and analysis of software engineering and methodology aspects.

About the terms *block* and *blocking*

As we are dealing with languages and programming idioms, the reader should note that the term *block* appears in this paper in two different meanings: (a) a brick or a box - referring to programming command drawn as a two-dimensional shape; and (b) a verb meaning *to forbid* or *to prevent*, associated with the behavioral programming idiom for declaring events that must not happen at a given point in time. It is interesting to observe that these meanings are individually commonly used and are appropriate for the intent, that finding alternative terms for the sole purpose of disambiguation, is unnecessary, in the least, and in some cases, artificial and even detrimental to the understandability of the text. In this context, of course, the language name Blockly fits nicely with its proposed use in programming behaviorally. Still to minimize confusion, we avoided using the terms *block* and *blocking* in two other common software-related meanings, namely, (c) stopping a process or a subroutine while waiting for an event or resource; and, (d) a segment of program code which contains all the commands between some end-markers such as curly braces or `begin` and `end`.

2. Behavioral Programming

In this section we outline the technique of Behavioral Programming for the development of reactive systems. Formal definitions and comparisons with other programming techniques appear in [14, 17, 19].

A preliminary assumption in our implementation of behavioral programming is that an application, or a system, is focused on processing streams of events with the goal of identifying and reacting to occurrences of meaningful scenarios. Detected event sequences are then used to trigger abstract, higher level, events, which in turn may trigger other events. Some of these events are translated to actions that the system takes to effect changes in the external world. This cycle results with a reactive system that translates inputs coming from its sensors to actions performed by its actuators.

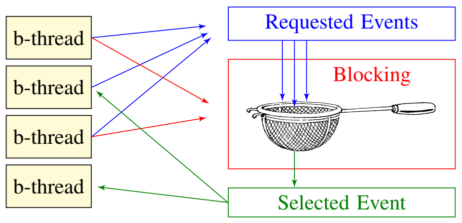


Figure 1. Behavioral programming execution cycle: all b-threads synchronize, declaring requested and blocked events; a requested event that is not blocked is selected and b-threads waiting for it are resumed.

More specifically, in a behavioral program, event sequences are detected and generated by independent threads of behavior that are interwoven at run time in an enhanced publish/subscribe protocol. Each *behavior thread* (abbr. *b-thread*) specifies events which, from its own point of view must, may, or must not occur. As shown in Figure 1, the infrastructure consults all b-threads by interweaving and synchronizing them, and selects the events that will constitute integrated system behavior without requiring direct communication between b-threads. Specifically, all b-threads declare events that should be considered for triggering (called *requested events*) and events whose triggering they forbid (*block*), and then synchronize. An event selection mechanism then triggers one event that is requested and not blocked, and resumes all b-threads that requested the event. B-threads can also declare events that they simply "listen-out for", and they too are resumed when these waited-for events occur.

This facilitates incremental non-intrusive development as outlined in the example of Figure 2. For another example, consider a game-playing program, where each game rule and each player strategy is added in a separate module that is oblivious of other rules and strategies. Detailed examples showing the power of incremental modularity in behavioral programming appear, e.g., in [17–19].

In behavioral programming, all one must do in order to start developing and experimenting with scenarios that will later constitute the final system, is to determine the common set of events that are relevant to these scenarios. While this still requires contemplation, it is often easier to identify these events than to determine objects and associated methods. By default, events are opaque and carry nothing but their name, but they can be extended with rich data and functionality. Further, the incremental traits of BP and the smallness of b-threads (see Section 5) facilitate subsequent adding and changing of event choices.

The behavioral programming technique facilitates new automated approaches for planning in execution [12, 15], program verification and synthesis [13, 18, 22], visualization and debugging of concurrent execution [5], natural language coding of scenarios [9], and program repair [20] .

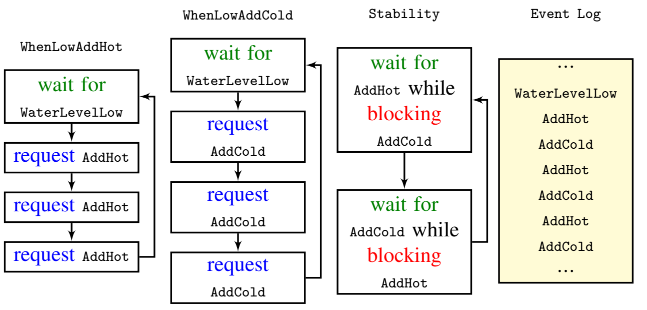


Figure 2. Incremental development of a system for controlling water level in a tank with hot and cold water sources. The b-thread `WhenLowAddHot` repeatedly waits for `WaterLevelLow` events and requests three times the event `AddHot`. `WhenLowAddCold` performs a similar action with the event `AddCold`, reflecting a separate requirement, which was introduced when adding three water quantities for every sensor reading proved to be insufficient. When `WhenLowAddHot` and `WhenLowAddCold` run simultaneously, with the first at a higher priority, the runs will include three consecutive `AddHot` events followed by three `AddCold` events. A new requirement is then introduced, to the effect that water temperature should be kept stable. We add the b-thread `Stability`, to interleave `AddHot` and `AddCold` events using the event-blocking idiom.

3. Infrastructure Implementation

3.1 Coordinating behaviors written in JavaScript

In principle, the concepts of behavioral programming are language independent and indeed they have been implemented in a variety of languages using different techniques. However, certain language facilities are needed in order to control the execution, synchronization and resumption of the simultaneous behaviors. In LSC this is done by the control mechanism which, interpreter-like, advances each chart along its locations (see,e.g. [16]). In Java [17], Erlang [32] and C++ the mechanism is implemented as independent threads or processes and uses language constructs such as `wait` and `notify` for suspension and resumption. When executed in a browser, a JavaScript application is typically executed as a single thread in a single process, hence another mechanism is needed. Note that for the present proof-of-concept, the choice is indeed JavaScript, but the language-independent principles can be implemented also in other technologies, say, Flash ActionScript, if appropriate constructs are available for suspension and resumption.

In JavaScript 1.7 [27] (currently supported in the Firefox browser) the `yield` command was introduced which allows the implementation of generators and coroutines, and we chose to use it for our BP implementation - providing suspension and resumption in single-threaded multi-tasking. Briefly, the `yield` mechanism allows a method to return to its caller, and upon a subsequent call, continue from the instruction immediately after the most recent return. Thus, in the context of coroutines, coordinated behavioral execution can be described as follows:

Behavior threads are implemented as coroutines. For example, the b-thread `Stability` of the water-tap example is

```
function() {
    while (true) {
        yield({
            request: [],
            wait: ["addHot"],
            block: ["addCold"]
        });
        yield({
            request: [],
            wait: ["addCold"],
            block: ["addHot"]
        });
    }
}
```

The infrastructure executes in cycles. In each cycle, the b-threads are called one at a time. The coroutine of each b-thread returns, using the `yield` command, specifying this coroutine's declaration of requested events, blocked events and waited-for events. Once each of the b-threads has been called and has returned in this manner, and its declarations have been recorded, the infrastructure selects for triggering an event that is requested and not blocked. The infrastructure then resumes all b-threads that requested or waited for the event, by calling them (and only them), one by one, as part of the new cycle.

In fact this description summarizes the majority of what was needed for our implementation of behavioral programming in JavaScript. More details appear in Appendix A. The b-threads are, of course, application specific and are provided by the application programmer.

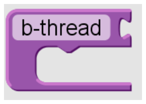
Since JavaScript requires that the `yield` command be visible in the source of the coroutine at least once, in the present implementation we chose not to hide it within a method dedicated to behavioral synchronization and declarations, such as the `bSync` method in the Java implementation. Indeed, we feel that this diversity in command name usage across languages emphasizes that BP benefits, such as ease of state management and incrementality (see Section 5), can be gained by implementing and using BP principles in any language and with a variety of idioms.

3.2 Behavioral blocks for Blockly

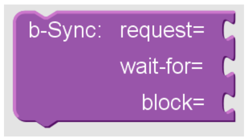
The Google Blockly environment [8] is built along principles similar to those of the popular Scratch language [30]. Other languages and environments in this family include, among others, BYOB/SNAP! [26], MIT App Inventor [1], and Waterbear [6]. In these languages, the programmer assembles scripts from a menu of block-shaped command templates that are dragged onto a canvas. The Blockly blocks contain placeholders for variables and sub-clauses of the commands and can express scope of program-segment containment, relying on the notation of a block containing physically other blocks, with possible nesting. The popularity of the Scratch language suggests that this style of coding is more accessible to children than standard programming languages, and perhaps even other visual languages, such as LSC. However, we also feel that the combination of visualization and language customization make Blockly an excellent platform for demonstrating coding techniques that would otherwise require pseudo-code or abstraction, and it may also provable suitable for complex applications.

While Scratch and BYOB are interpreted (in SmallTalk and now also in Flash), Blockly and Waterbear diagrams are translated into code (we use JavaScript) which can later be manipulated and executed natively without dependency on the development environment.

Our implementation of behavioral programming in Blockly includes new (types of) blocks: the `b-thread` block

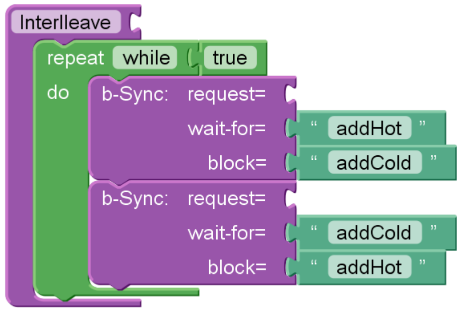


for the b-thread logic (the string `b-thread` in the template can replaced by the programmer with the b-thread's name or description); the `b-Sync` block



for synchronization and bidding of requested, waited-for, and blocked events; and, a `lastEvent` block where b-threads that wait for a number of events can examine the one that indeed happened.

For illustration, the `Stability` b-thread of the water-tap example, is coded in Blockly as



and is automatically compiled into the JavaScript code shown in Section 3.1.

The advantages of programming in this manner are discussed in Section 5. We will only mention here that the visualness of Blockly adds to the usability of BP principles, while behavioral decomposition should simplify the development of complex applications in Blockly .

As the Google Blockly environment is in early development stages, we had to also add some basic capabilities, such as list concatenation, that are not specific to behavioral programming.

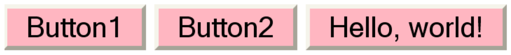
4. Programming an Interactive Application Behaviorally - an Example-driven Tour

In this section we present the underlying design principles for applications coded with Blockly, JavaScript, and HTML, via a review of several small applications. The code and additional explanations are available online at `www.b-prog.org`

4.1 Sensors and Actuators

A key design principle is separating the "real" world from the application logic using appropriate interfaces for translating occurrences in the environment into entities that can be processed by the application, and application decisions into effects on the environment.

We begin our example-driven tour with examination of the sensors and actuators of a simple application which consists of the following three buttons:



The requirements for this application are that when the user clicks Button1 the greeting should be changed to "Good Morning" and when the user clicks Button2 the greeting should be changed to "Good Evening". We first have to create a sensor for the button clicking:

```
<input
   value   = "Button1"
   type    = "button"
   onclick = "bp.event('button1Click');"
/>
```

The clicking is captured by the standard use of the HTML verb `onclick` and the ensuing activation of JavaScript code. The function `bp.event` is part of the behavioral infrastructure in JavaScript, and it creates the behavioral event passed as a parameter. Details about event triggering appear in Section 4.3

To transform application decisions into effects on the environment, an HTML entity can activate JavaScript code using another part of the behavioral infrastructure we added in Blockly/HTML/JavaScript, the verb `when_`*eventName*, as follows:

```
<input
    value    = "Hello, world!"
    type     = "button"
    when_button1Click  = "value='Good Morning'"
    when_button2Click  = "value='Good Evening'"
/>
```

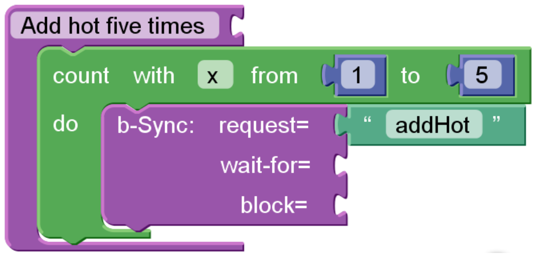
In this simple example, there are no application-logic b-threads and the actuator is driven directly by the behavioral event generated by the sensor.

In the present implementation, events are simply character strings. The semantics of triggering a behavioral event in Blockly is notifying (or resuming) any b-thread or HTML entity that registered interest in the specified event, using the `b-Sync` block or the `when_`*eventName* idiom, respectively.

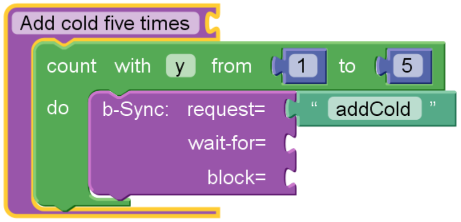
We believe that the design of a reactive behavioral application should start with analysis and determination of the sensors and actuators and the associated events. For example, Figure 3 in Section 4.3 shows such an event list for a richer example. The behaviors can then be added incrementally, as requirements are analyzed. Of course, as needed, sensors and actuators can be modified or replaced. In this approach the role of GUI design can be separated from that of application logic programming, and deciding about sensors and actuators can be seen as a development stage in which negotiation and agreement between individuals acting in these capacities take place.

4.2 Application-logic b-threads

We now move to a slightly richer example - a water-tap application similar to the one described in Section 2. This application's logic is coded in the following b-threads. One b-thread adds five quantities of hot water and terminates:



Another adds five quantities of cold water and terminates:



And, the third b-thread which interleaves the events is the same as the one shown in Section 3.2. The result is of course the interleaved run of ten events alternating repeatedly between `addHot` and `addCold`.

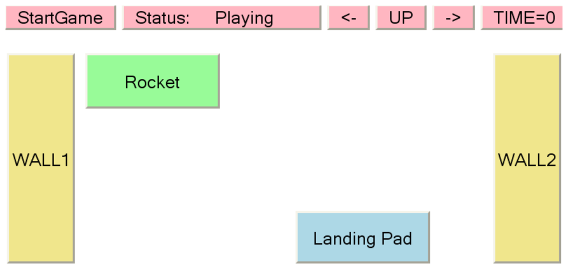
Each b-thread is contained within the Blockly block of `b-thread`. The b-thread can use any Blockly block for implementing the desired logic. To synchronize with other b-threads the `b-Sync` block is used, with the three parameters of requested, waited-for, and blocked events.

In contrast with the BPJ Java implementation, where b-thread priorities were assigned explicitly, in the Blockly implementation, b-thread priority is implied by its physical location on the development canvas, with the topmost b-thread being called first, and the lowest b-thread being called last in every execution cycle. When a new b-thread is added some dragging may be needed in order to insert it at the desired priority.

When starting an application, the Blockly infrastructure also triggers a behavioral event called `start` to activate the JavaScript behavioral programming infrastructure and execution of all b-threads.

4.3 Execution Cycle Details

To show finer points about the interweaving of b-threads in the Blockly/JavaScript environment, we examine an application for a computer game where the player attempts to land a rocket on a landing pad on the surface of a planet, or perhaps a space shuttle on a space station. The rocket moves at a fixed speed in the vertical direction. Using GUI buttons, the player can move the rocket right and left to be positioned directly above the landing pad. The player can also press the `Up` button to suspend the rocket and prevent it from going down in the next time unit. A small challenge is introduced as the landing pad keeps moving right and left either randomly or subject to an unknown plan. Two walls mark the sides of the playing areas, and the rocket cannot move past them (but does not crash when it touches them).



The game is won when the rocket lands safely on the landing pad, and is lost if the rocket either lands on the landing-pad when it is not aligned with it, or if it misses the landing-pad altogether.

As suggested in Section 4.1 we first agree on the events, and the associated sensors and actuators in the game. They are listed in Figure 3.

As described in the infrastructure section, at every synchronization point, the declarations of all b-threads are consulted, an event is selected for triggering, and b-threads that requested or waited for that event are resumed. Events that are generated by sensors are handled as follows. The function `bp.event` dynamically creates a b-thread which requests the event at the next synchronization point, and terminates once the event is triggered.

When an execution cycle ends and no event is triggered, the system is considered to have completed a *superstep*. The next behavioral event, if any, must come from a sensor reporting some external environment event. The sensor-generated event initiates a new superstep which then continues until, again, there are no events to be selected. To force a sensor-generated event to be triggered only at a beginning of a new future superstep, the sensor code should not call the event-triggering function directly, but should set it as a timer-driven JavaScript event. Due to the JavaScript single-threaded non-preemptive events mechanism the code will run as soon as the current function (the super-step) ends. This is shown below where a `RocketLeft` actuator uses a `when_` clause and the function `trigger` to serve as a `RocketTouchedLeftWall` sensor.

Sensor / Actuator	Event	Event Meaning (Description)
Sensor	BtnLeft	User clicked `<-`
Sensor	BtnRight	User clicked `->`
Sensor	BtnUp	User clicked `Up`
Sensor	TimeTick	A unit of time passed
Sensor	RocketAtLeftWall	Rocket started touching left wall
Sensor	RocketAwayFromLeftWall	Rocket stopped touching left wall
Sensor	RocketAtRightWall	Rocket started touching right wall
Sensor	RocketAwayFromLeftWall	Rocket stopped touching right wall
Sensor	TouchDown	Rocket touched launch pad and is aligned with it
Sensor	Missed	Rocket reached or passed launch pad without being aligned with it
Actuator	RocketLeft	Request to redraw rocket 10 pixels to the left
Actuator	RocketRight	Request to redraw rocket 10 pixels to the right
Actuator	RocketDown	Request to redraw rocket 10 pixels down
Actuator	DisplayWin	The application determined that the player won
Actuator	DisplayLose	The application determined that the player lost
Actuator	GameOver	The application determined that the game should be stopped
Actuator	PadLeft	The application wishes the pad to move 10 pixels to the left
Actuator	PadRight	The application wishes the pad to move 10 pixels to the right

Figure 3. The external world in the rocket-landing game is represented to the behavioral application via sensors and actuators.

```
<script>
...
function trigger(exEvent) {
   setTimeout("bp.event('"+exEvent+"')",0);
}
...
</script>

<input
   value = "WALL1"
   type  = "button"
   style = "position:absolute;left:10;top:
            150;width:52;height:500"
   when_RocketLeft =
      "if(rocketX<=(leftWall+1)){
         trigger('RocketTouchedLeftWall');
       }"
 />
```

As shown here, to avoid unnecessary delays, the specified time can be zero.

Note that a separate `RocketLeft` listener is responsible for moving the rocket on the screen, and that multiple listeners can be coded for a given behavioral event. The relevant `when_` clauses may be coded under a wide selection of HTML objects — the approach does not require that the programmer chooses "correctly" HTML entities associated with a given sensor or actuator.

Two of the central questions in real-time system design is whether two events can occur exactly at the same instant, and how much time is required for the processing of all system-generated events that follow a single sensor-generated event. The user should consider the following assumptions and implementation choices as ways to simplify the application, when applicable:

- Always trigger sensor-generated events in a new superstep (using the time-out technique above)
- As in Logical Execution Time [21], assume that a superstep which consists of one sensor-generated event followed by system-generated events takes (practically) zero time.

Note that the second assumption is common, e.g., in real-time interrupt handling and in user interface programming, where event handlers must respond quickly. Thus, the BP semantics is well defined, and when b-threads communicate only through behavioral events also does not allow race conditions. SImplicity emerges partly from the fact that each b-thread can declare the events which affect it at a given state, and then handle the effects of their triggering, and completely ignore the existence of events that should not affect its state.

5. Key Scenarios where BP Benefits Emerge

Below we outline and exemplify some of the advantages and desirable capabilities of behavioral programming techniques , and the software development scenarios in which they appear. For additional comparison of BP with standard programming techniques as well as other publish/subscribe and rule-based environments see [14, 17]

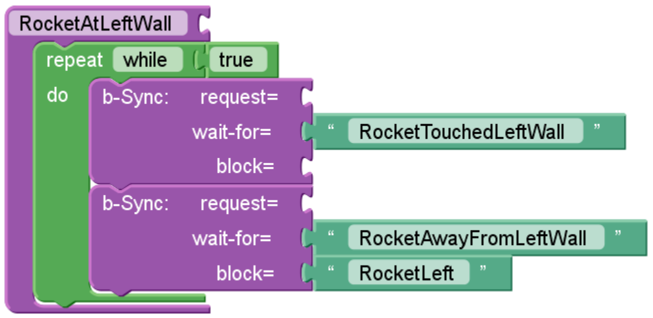
5.1 Incrementality and alignment with the requirements

The first and foremost benefit of programming behaviorally is the ability to structure application modules such that they are aligned with the requirements. As discussed at length in [14, 17], modules can be written to reflect individual requirements with little or no change to existing code. Further, as requirements are added, refined, or merely taken sequentially from a requirements document, the corresponding b-thread code can be developed and added to the application incrementally.

In the rocket-landing game, for example, assume that one first codes the following b-threads without any requirement for walls — hence without the wall-related b-threads and the four wall-related events. The coded b-threads are thus:

- Attempting to move the rocket down in response to the passing of time,
- Attempting to move the rocket right or left in response to a corresponding user click,
- Blocking the rocket's down move in response to clicking `Up`,
- Moving the landing pad right and left, and
- Detecting and announcing user winning or losing.

Then, the developer or the customer realize that walls may be required and describe the desired behavior: no advancement past the wall, but hitting a wall does not mean a crash. The appropriate sensors and b-threads to block rocket movement like



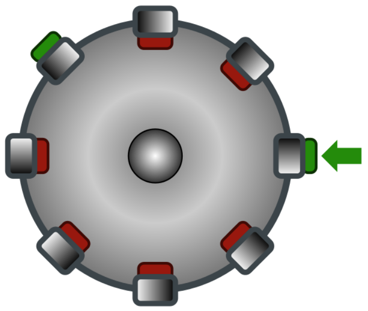
```
RocketAtLeftWall
  repeat while true
  do b-Sync: request=
             wait-for= " RocketTouchedLeftWall "
             block=
     b-Sync: request=
             wait-for= " RocketAwayFromLeftWall "
             block= " RocketLeft "
```

can then be added incrementally. Other capabilities which can be added with no change to the existing b-threads include additional obstacles (with behavior similar to that of the walls), increasingly hard-to-track movements of the landing pad, and a facility for advanced human players to automate their own play by programming strategy b-threads that request prescribed sequences of button-clicking events (this last example is, of course, not needed for this simple game, but is desirable and common in more advanced games).

The independence of behavior threads is also manifested in that scripts and scenarios do not have to communicate with each other directly. In the native Blockly or Scratch, broadcasting and publish/subscribe techniques can suffice for rich processing, relying only on local variables and avoiding global or shared variables. With the addition of behavioral synchronization and event-blocking (i.e., forbidding) the integrated runs are enriched, without adding a burden of peer-to-peer communication. Specifically, any event or condition that a b-thread blocks may be generated by any existing or yet-to-be-developed b-thread or sensor.

5.2 Easy state management for long scenarios

Scenarios are, of course, central to behavioral programming, and go substantially beyond the rule-based capability of waiting for an event and then triggering another event based on the system's state. For example, consider the *nullification game* application[31] — a combinatorial game where the player attempts to push in an entire array of switches placed on a rotating wheel, and where an adversary attempts to reverse the switch settings.



A game move consists of optionally pressing the switch (*Switch*), and then rotating the wheel to the next switch position (*Shift*).

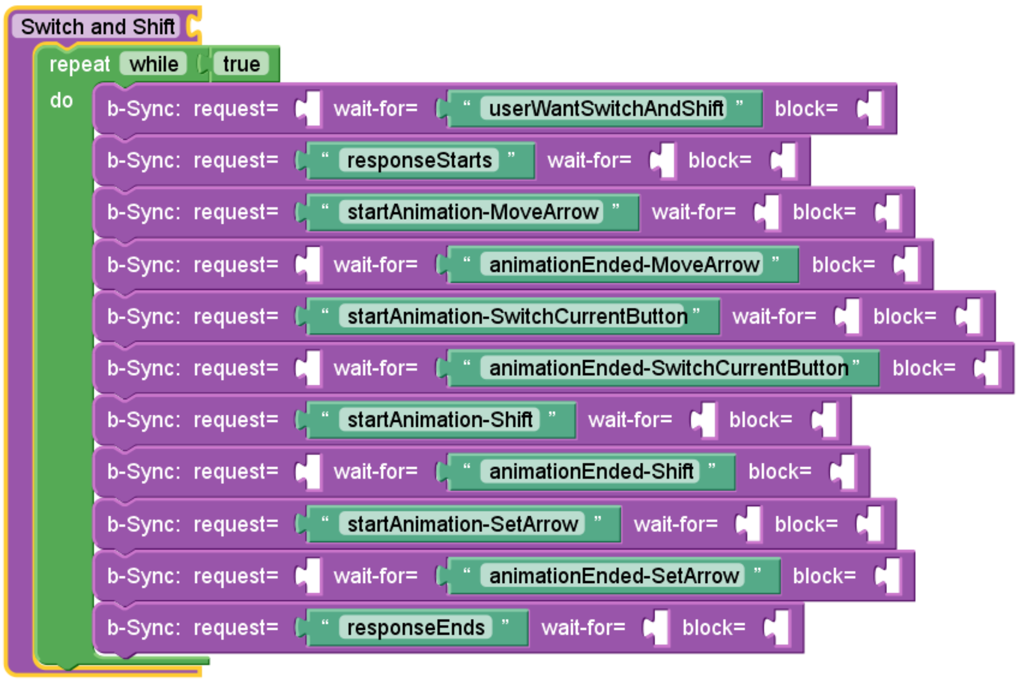
In this example the animated drawing of the rotation of the wheel and the changes in switch position are performed by the GUI-processing JavaScript application package Raphael [2]. Following a game move , multiple animations have to occur, including the moving of an arrow indicating the pressing of a switch, the movement of the switch itself, wheel rotation, and the flyover of the arrow from the human player side to the adversary side and vice versa, in an indication of whose turn it is.

In native JavaScript, without coroutines, this sequence of events would have to be programmed with callbacks and/or independent event handlers, and with variables to keep track of the evolving state and would generally look similar to:

```
when_UserWantsSwitchAndShift=
   state =  SwitchandShift0  ;
   trigger(  ResponseStart  );

when_ResponseStart  =
   if( state =  SwitchAndShift0  ) {
      state =  SwitchAndShift1  ;
      trigger( startAnimation-MoveArrow );
   } else {
      ...
   }

when_AnimationEnded-MoveArrow =
   if( state =  SwitchAndShift1  ) {
      state =  SwitchAndShift2
      trigger( startAnimation-SwitchCurrentButton );
   } else {
      ...
   }
...
```



```
Switch and Shift
  repeat while true
  do
    b-Sync: request= wait-for= " userWantSwitchAndShift " block=
    b-Sync: request= " responseStarts " wait-for= block=
    b-Sync: request= " startAnimation-MoveArrow " wait-for= block=
    b-Sync: request= wait-for= " animationEnded-MoveArrow " block=
    b-Sync: request= " startAnimation-SwitchCurrentButton " wait-for= block=
    b-Sync: request= wait-for= " animationEnded-SwitchCurrentButton " block=
    b-Sync: request= " startAnimation-Shift " wait-for= block=
    b-Sync: request= wait-for= " animationEnded-Shift " block=
    b-Sync: request= " startAnimation-SetArrow " wait-for= block=
    b-Sync: request= wait-for= " animationEnded-SetArrow " block=
    b-Sync: request= " responseEnds " wait-for= block=
```

Figure 4. A scenario of consecutive instructions (shown here in Blockly) facilitates natural and implicit state management.

In our implementation this sequence is handled naturally in a b-thread as consecutive instructions as shown in Figure 5.2.

Thus, our solution facilitates waiting for events in-line and not only by callbacks. It should be noted that several JavaScript pre-compilers, such as NarrativeJS [25], StratifiedJS [29], and others, allow for sequential event handling in JavaScript, similarly to the ability described in this section. These extensions to JavaScript can be viewed as implementing a subset of the complete behavioral protocol, often without events blocking or multiple b-threads. In this context it should be noted that BP is different from rule-based systems is in that blocking in BP is targeted at events, regardless of their originator, as opposed to rule-based system in which blocking is by disabling rules (see, e.g., [7].

5.3 Integration with standard programming

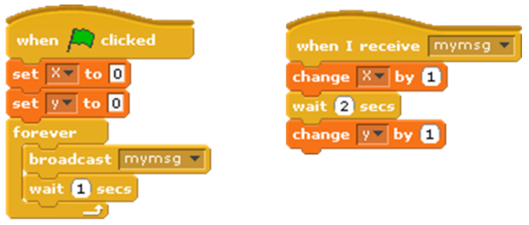
Coding behaviorally does not mean that all calculations and data processing performed by the application must be based on events. A behavioral application can contain substantial pieces that are coded in standard programming languages. In the context of JavaScript and Blockly, JavaScript functions can be called from Blockly blocks, or from the sensors and actuators. In the nullification game example the calculations of the adversary strategy which is based on deBruijn sequences[31], are performed by calling a JavaScript function. Needless to say, the Raphael animations discussed above also demonstrate the power of such integration capabilities.

5.4 Programming with parallel continuous entities with well-defined semantics

As in the LSC language, the basic units of program code (the actors) in the current Blockly and JavaScript environment are scripts that run "all the time". These scripts take desired actions when specific conditions are met, or constantly express their opinion about the global state from a narrow viewpoint based on events that they listen-out for. As observed in [10, 23], this design appears to be "natural" in the sense that it was adopted by children who were not explicitly guided to use it.

In behavioral programming, instantiation, activation and repeated synchronization of such scripts is easy, often "free", i.e., automatic, in comparison to the more elaborate setup commonly needed in other languages and contexts.

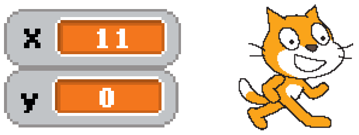
A problem in Scratch pointed out by Ben-Ari and discussed in Scratch forum [3] is that the semantics of interweaving scripts depends on intricate properties of the model whose effects on scheduling are sometimes hidden from the programmer. For a less intricate but illustrative example, consider the scripts



```
when clicked
set x to 0
set y to 0
forever
  broadcast mymsg
  wait 1 secs
```

```
when I receive mymsg
change x by 1
wait 2 secs
change y by 1
```

The programmed flow is that once the green flag is clicked, the first script broadcasts `mymsg`, waits 1 second and then broadcasts the message again. Whenever this message is received by the second script, the variable X is incremented, and after 2 seconds, the variable Y is incremented. However a result of running and stopping the scripts is



suggesting that when the message is broadcast a second time, the first execution of the second script is interrupted and is never resumed, thus Y is incremented only once. When the delay in the first script is set to 5 seconds instead of 1, the final value of Y is 2. We did not find documentation of this semantics of Scratch.

Our approach, in this paper, is to view scripts as global entities with well-defined scheduling, synchronization protocol, and interweaving semantics. Using our Blockly and JavaScript environment, an application similar to the above example will have to be coded differently. Depending on the programmer's solution, when the behavioral event associated with second message is triggered, this event will either: (a) cause no effect, as at the synchronization point when it is triggered no b-thread will declare it as a requested or waited-for event (instead, the second b-thread is only waiting for time-delay to pass), or (b) it will be processed by another running instance of the second b-thread class, which would be explicitly started by the application to catch such events while other instances wait for the time-delay to pass. In either case, the semantics will be well defined and the composite behavior will be readily predictable.

5.5 Priority as a first-class idiom

When multiple simultaneous behaviors are active and vote with their event declarations as votes' with regard to the progress of an application, priority becomes a useful construct. In our implementation, b-thread priorities are based on their easy to manage order on the canvas, i.e., a b-thread laid higher on the canvas has a higher priority. The priorities of b-threads that are perfectly aligned with each other vertically are ordered based on the b-threads' left-to-right horizontal order. In addition, the single-threaded sequential calling of JavaScript coroutines provides for well-defined semantics of the "simultaneous" part of the behavior, and of the corresponding effects on any variables that are shared between b-threads.

5.6 The secondary role of the behaving objects

In Scratch, scripts are anchored on game characters called *sprites* which are perceived as the behaving entities. On one hand, the sprites can be readily thought of as agents or actors in their own right, which in turn rely on scripts as their implementation or as another level in their hierarchy. On the other hand, following the detailed discussion of inter-object versus intra-object behavior in [11], behavior scenarios are not necessarily anchored on a given object. For example, in [23] the researchers observed that when forced to associate scripts with sprites, young programmers split the scripts of the (correct) behavior of one game character on multiple sprites, and game rules were associated with arbitrary sprites. This further puts into question the need to focus on ”the behaving entity” when observing a behavior. The Blockly/JavaScript environment presented here does not force the programmer to associate desired behaviors with behaving objects.

We propose that there is an important distinction between objects in general, as in object oriented programming, and the concept of behaving entities that are tangible in the user's eyes. For example, in an application with a graphical user interface, it is not always best to anchor the code on the elements on the screen. It may be better, instead, to code scenarios that involve multiple tangible entities as standalone modules. Of course, scenarios, events, screen objects, etc., may be coded with object-oriented programming.

6. Discussion and Future Work

We presented an implementation of behavioral programming principles in JavaScript and in Google Blockly. The result is a proof-of-concept for a programming environment which appears to be natural and intuitive, and highlights interesting traits of behavioral programming. An important next step is to show the scalability of the concepts and their applicability to complex systems. As our understanding of BP develops, it will also be interesting to expand the discussion of comparisons of BP to other paradigms, which appear in [14, 17], to additional platforms such as rule-based (e.g., [7]) and functional reactive programming environments (e.g. [24]).

The ease of creating new language constructs in Blockly and the fact that visual block-based programming seems natural to individuals with little computer training, call for using Blockly as a test-bed for investigating the naturalness of new programming idioms. For example, nesting blocks which state things like "while forbidding events `a,b,c` do", or "exit the present block when event `e` is triggered" have the potential of making behavioral programs simpler and more readable than when written with just basic `b-Sync`. Specifically, they can simplify the management of the sets of requested, waited-for and blocked events, and reduce the need to wait, in a single command, for multiple events and then check which of them was indeed triggered.

The availability of Blockly and JavaScript 1.7 for mobile platforms, such as Android smartphones, opens the way for a wide range of applications, and the single-threaded JavaScript implementation may further facilitate running behavioral applications with many b-threads in environments which do not readily accommodate large numbers of concurrent Java threads.

The combination of implementations of behavioral programming principles in popular languages, with IDEs that are particularly user friendly, and with a growing set of natural programming idioms, may further facilitate programming in a decentralized-control mindset by wider communities of beginners and professionals.

Acknowledgments

We would like to thank David Harel for support and valuable discussions and suggestions in the development of this work. The focus on Scratch-like languages is inspired by our ongoing collaboration with Orni Meerbaum-Salant and Michal Gordon and by suggestions by Eran Mizrachi. We thank the anonymous reviewers for their valuable comments and suggestions for improvements, which we tried to incorporate.

The research of Assaf Marron and of Guy Wiener was supported in part by the John von Neumann Minerva Center for the Development of Reactive Systems at the Weizmann Institute of Science, and by an Advanced Research Grant to David Harel from the European Research Council (ERC) under the European Community's FP7 Programme. The research of Gera Weiss was supported by the Lynn and William Frankel Center for Computer Science at Ben-Gurion University and by a reintegration (IRG) grant under the European Community's FP7 Programme.

A. Appendix: Behavioral Programming using Coroutines

This appendix presents a generic algorithm for implementing the behavioral programming event-driven loop using coroutines. The motivation for pursuing coroutines for b-threads is twofold. First, coroutines consume less resources than threads or processes, and therefore can be found in embedded scripting languages that aim at minimizing resource allocation, such as JavaScript and Lua. Second, this implementation does not require concurrent execution, that is not always desirable, e.g., when debugging or verifying the system.

A.1 Introduction to Coroutines

Coroutines provide a mechanism for executing two or more control-flows independently, without requiring a thread scheduler. Instead, the control is passed from one coroutine to the other explicitly, together with a given value. For this reason, coroutines are also referred to as *non-preemptive multi-threading*.

While coroutines are supported by a wide range of programming languages, each language has its own syntax for defining and using them. In this section we will use the following notation.

- A coroutine is a function that instead of the standard `return` statement uses the special `yield` statement. Similarly to `return`, `yield` passes the control flow back to the caller together with a given value. Unlike `return`, when the coroutine is called again, it resumes at the state of the previous `yield`, as if it was paused and resumed. When the coroutine is resumed, the `yield` expression evaluates to the value sent to the coroutine by the caller.
- The `create` statement takes the name of a coroutine and creates a coroutine object. This object acts as a unique identifier of the state of the coroutine. Each subsequent call to the same identifier will continue from the state of the last call to `yield`. Notice that `create` does not run the coroutine itself.
- The `send` statement takes a coroutine object and a value, and resumes the coroutine with the state of the last call to `yield`, returning the given value. If this is the first call after `create`, the coroutine will start, ignoring the given value.
- The `alive` predicate tests if a coroutine object is still running. It returns `true` after the coroutines is created, and as long as the coroutine function is not completed. It returns `false` otherwise.

A.2 The Algorithm

In this implementation, the programmer codes each application b-thread as a coroutine. We use a JavaScript-like notation for lists and records.

```
// Queues of b-threads and their bids
running = []
pending = []

// -------------------------------------------
// Adding a b-thread translates to pushing it
// to the running queue and creating a coroutine
// for it
// -------------------------------------------
function addBThread(prio, func) {
   push running, {priority: prio,
                  bthread: create func}
}

// -------------------------------------------
// Run is called to begin a superstep. It invokes
// the coroutines sequentially, collects the bids,
// selects the next event, and calls itself
// recursively
// -------------------------------------------
function run() {
   while (running is not empty) {
      bid = unqueue running
      bt  = bid.bthread
      newbid = send bt, lastEvent
      if (bt has not finished) {
         newbid.bthread = bt
         newbid.priority = bid.priority
         queue pending, newbid
      }
   }

   lastEvent = the first event that some
               b-thread requested and
               no b-thread blocked

   if (lastEvent is not equal to undefined) {
      temp = []
      while (pending not empty) {
         bid = unqueue pending
            if (bid specifies waiting-for
                or requesting lastEvent)
             queue running, bid
          else
             queue temp, bid
       }
       pending = temp
       run()
    }
}
```

References

[1] H. Abelson and M. Friedman. MIT App Inventor. URL: http://appinventor.mit.edu, accessed Aug. 2012.

[2] D. Baranovskiy. Raphael. URL: http://raphaeljs.com/, accessed Aug. 2012.

[3] M. Ben-Ari and J. Maloney. Scratch project forum discussion. URL: http://scratch.mit.edu/forums/viewtopic.php?id=8130, accessed Aug. 2012.

[4] W. Damm and D. Harel. LSCs: Breathing Life into Message Sequence Charts. *J. on Formal Methods in System Design*, 19 (1):45–80, 2001.

[5] N. Eitan, M. Gordon, D. Harel, A. Marron, and G. Weiss. On Visualization and Comprehension of Scenario-Based Programs. *Int. Conf. on Program Comprehension (ICPC)*, 2011.

[6] D. Elza. Waterbear language web site. URL: http://waterbearlang.com/, accessed Aug. 2012.

[7] J. Fenton and K. Beck. Playground: an object-oriented simulation system with agent rules for children of all ages. *ACM SIGPLAN Notices*, 24(10):123–137, 1989.

[8] N. Fraser. Google blockly - a visual programming editor. URL: http://code.google.com/p/blockly, accessed Aug. 2012.

[9] M. Gordon and D. Harel. Generating executable scenarios from natural language. *Computational Linguistics and Intelligent Text Processing*, pages 456–467, 2009.

[10] M. Gordon, A. Marron, and O. Meerbaum-Salant. Spaghetti for the main course? observations on naturalness of scenario-based programming. *17th Annual Conference on Innovation and Technology in Computer Science Education*, 2012.

[11] D. Harel and R. Marelly. *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine*. Springer, 2003.

[12] D. Harel and I. Segall. Planned and Traversable Play-Out: A Flexible Method for Executing Scenario-Based Programs. In *Proc. 13th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 485–499, 2007.

[13] D. Harel and I. Segall. Synthesis from live sequence chart specifications. *Jour. Computer System Sciences*, 78:3:970–980, 2012.

[14] D. Harel, A. Marron, and G. Weiss. Behavioral Programming. *Communications of the ACM*, 55(7):90–100.

[15] D. Harel, H. Kugler, R. Marelly, and A. Pnueli. Smart Play-Out of Behavioral Requirements. In *Proc. 4th Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD)*, pages 378–398, 2002.

[16] D. Harel, S. Maoz, S. Szekely, and D. Barkan. PlayGo: towards a comprehensive tool for scenario based programming. In *ASE*, 2010.

[17] D. Harel, A. Marron, and G. Weiss. Programming Coordinated Scenarios in Java. In *Proc. 24th European Conf. on Object-Oriented Programming (ECOOP)*, pages 250–274, 2010.

[18] D. Harel, R. Lampert, A. Marron, and G. Weiss. Model-Checking Behavioral Programs. In *Proc. 11th Int. Conf. on Embedded Software (EMSOFT)*, pages 279–288, 2011.

[19] D. Harel, A. Marron, G. Weiss, and G. Wiener. Behavioral programming, decentralized control, and multiple time scales. In *Proc. of the SPLASH Workshop on Programming Systems, Languages, and Applications based on Agents, Actors, and Decentralized Control (AGERE!)*, pages 171–182, 2011.

[20] D. Harel, G. Katz, A. Marron, and G. Weiss. Non-intrusive repair of reactive programs. In *International Conference on Engineering of Complex Computer Systems (ICECCS)*, 2012.

[21] T. A. Henzinger, C. M. Kirsch, M. Sanvido, and W. Pree. From control models to real-time code using Giotto. *IEEE Control Systems Magazine*, 23(1):50–64, 2003.

[22] H. Kugler, C. Plock, and A. Roberts. Synthesizing biological theories. In *CAV*, 2011.

[23] O. Meerbaum-Salant, M. Armoni, and M. Ben-Ari. Habits of Programming in Scratch. In *Proc. of the 16th Annual Joint Conf. on Innovation and Technology in Computer Science Education (ITICSE)*, pages 168–172. ACM, 2011.

[24] L. Meyerovich, A. Guha, J. Baskin, G. Cooper, M. Greenberg, A. Bromfield, and S. Krishnamurthi. Flapjax: A programming language for ajax applications. In *ACM SIGPLAN Notices*, volume 44, pages 1–20. ACM, 2009.

[25] N. Mix. Narrativejs. URL: http://www.neilmix.com/narrativejs/, accessed Aug. 2012.

[26] J. Moenig and B. Harvey. BYOB Build your own blocks (a/k/a SNAP!). URL: http://byob.berkeley.edu/, accessed Aug. 2012.

[27] Mozilla Foundation. FireFox JavaScript 1.7 -. URL: http://developer.mozilla.org/en/New_in_JavaScript_1.7, accessed Aug. 2012.

[28] E. Naone. HTML 5 could challenge Flash. URL: http://www.technologyreview.com/news/418130/html-5-could-challenge-flash/, accessed Aug. 2012.

[29] Oni Labs. Stratifiedjs. URL: http://onilabs.com/stratifiedjs/, accessed Aug. 2012.

[30] M. Resnick, J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman, et al. Scratch: Programming for All. *Comm. of the ACM*, 52(11):60–67, 2009.

[31] G. Weiss. A combinatorial game approach to state nullification by hybrid feedback. In *46th IEEE Conference on Decision and Control*, pages 4643–4647. IEEE, 2007.

[32] G. Wiener, G. Weiss, and A. Marron. Coordinating and Visualizing Independent Behaviors in Erlang. In *Proc. 9th ACM SIGPLAN Erlang Workshop*, 2010.

Optimized Distributed Implementation of Multiparty Interactions with Observation *

Saddek Bensalem[1] Marius Bozga[1] Jean Quilbeuf[1] Joseph Sifakis[1,2]

[1] UJF-Grenoble 1 / CNRS VERIMAG UMR 5104, Grenoble, F-38041, France
[2] RISD Laboratory, EPFL, Lausanne, CH-1015, Switzerland
{bensalem,bozga,quilbeuf,sifakis}@imag.fr

Abstract

Using high level coordination primitives allows enhanced expressiveness of component-based frameworks to cope with the inherent complexity of present-day systems designs. Nonetheless, their distributed implementation raises multiple issues, regarding both the correctness and the runtime performance of the final implementation. We propose a novel approach for distributed implementation of multiparty interactions subject to scheduling constraints expressed by priorities. We rely on new composition operators and semantics that combine multiparty interactions with observation. We show that this model provides a natural encoding for priorities and moreover, can be used as an intermediate step towards provably correct and optimized distributed implementations.

Categories and Subject Descriptors F.1.1 [*Theory of Computation*]: COMPUTATION BY ABSTRACT DEVICES; C.5 [*Computer Systems Organization*]: COMPUTER SYSTEM IMPLEMENTATION; C.2.4 [*Coputer Systems Organization*]: COMPUTER-COMMUNICATION NETWORKS

Keywords multiparty interaction, priority, observation, conflict resolution, distributed systems

1. Introduction

Correct design and implementation of computing systems has been an active research topic over the past three decades. This problem is significantly more challenging in the context of distributed systems due to a number of factors such as non-determinism, asynchronous communication, race conditions, fault occurrences, etc. Model-based development of such applications aims to ensure correctness through the usage of explicit model transformations.

In this paper, we focus on distributed implementation for models defined using the BIP framework [3]. BIP (Behavior, Interaction, Priority) is based on a semantic model encompassing composition of heterogeneous components. The *behavior* of components is described as an automaton extended by arbitrary data and associated functions written in C. BIP uses an expressive set of composition operators for obtaining composite components from a set of components. The operators are parameterized by a set of *multiparty interactions* between the composed components and by *priorities*, used to specify different scheduling mechanisms between interactions[1].

Transforming a BIP model into a distributed implementation consists in addressing three fundamental issues:

1. *Enabling concurrency.* Components and interactions should be able to run concurrently while respecting the semantics of the high-level model.
2. *Conflict resolution.* Interactions that share a common component can potentially conflict with each other.
3. *Enforcing priorities.* When two interactions can execute simultaneously, the one with higher priority must be executed.

We developed a general method based on source-to-source transformations of BIP models with multiparty interactions leading to distributed models that can be directly implemented [8, 9]. This method has been later extended to handle priorities [10] and optimized by exploiting knowledge [6]. The target model consists of components representing processes and Send/Receive interactions representing asynchronous message passing. Correct coordination is achieved through additional components implementing conflict resolution and enforcing priorities between interactions.

In particular, the conflict resolution issue has been addressed by incorporating solutions to the *committee coordination problem* [12] for implementing multiparty interactions. Bagrodia [1] proposes solutions to this problem with different degrees of parallelism. The most distributed solution is based on the drinking philosophers problem [11], and has inspired the approaches of Pérez et al. [18] and Parrow et al. [17]. In the context of BIP, a transformation addressing all the three challenges through employing *centralized scheduler* is proposed in [2]. Moreover, in [8], we propose transformations that address both the concurrency issue by breaking the atomicity of interactions and the conflict resolution issue by embedding a solution to the committee coordination problem in a distributed fashion.

Distributed implementation of priorities is usually considered as a separate issue, and solved using completely different approaches. For example, in [10], priorities are eliminated by adding explicit scheduler components and more multiparty interactions. This transformation leads to potentially more complex models, having def-

* The research leading to these results has received funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under grant agreement no. 248776 (PRO3D) and no 257414 (ASCENS) and from ARTEMIS JU grant agreement ARTEMIS-2009-1-100230 (SMECY)

[1] Although our focus is on BIP, all results in this paper can be applied to any model that is specified in terms of a set of components synchronized by interactions with priorities.

initely more interactions and conflicts than the original one. In [4, 5, 7], the focus is on reducing the overhead for implementing priorities by exploiting knowledge. Yet, these approaches make the implicit assumption that multiparty interactions are executed atomically and do not consider conflict resolution. In a similar line of work, [6] aims at detecting false conflicts, that is, statically detected but never occurring during execution. However, this method still relies on conflict resolution protocols, at least for states where no false conflicts exist.

In this paper, we propose a combined implementation of the two coordination mechanisms, that is, multiparty interactions and priorities. We propose an appropriate intermediate model and transformations towards fully distributed models dealing adequately with both of them. The contribution is twofold:

1. First, we introduce an alternative observation-based semantic model for BIP. We show that this model is general enough to encompass priorities and multiparty interactions and, moreover, to capture knowledge-based optimization as in [6]. Observation-based semantics reveals two types of conflicts occuring between interactions, that can be handled using different conflict resolution mechanisms (see below).
2. Second, this model is used in an intermediate step of a transformation leading to a distributed implementation. We show that *observation conflicts*, that usually follow from encoding of priorities, can be dealt more efficiently than *structural conflicts*, due to sharing of components between multiparty interactions. We extend the counter-based conflict resolution protocols of Bagrodia in order to handle these types of conflicts. These extensions have been fully implemented. We report some preliminary results on benchmarks.

The paper is organized as follows. Section 2 introduces the main concepts of the BIP framework together with the alternative observation-based composition semantics. Section 3 recalls the principles for distributed implementation of BIP models, focusing on conflict resolution by using counter-based protocols. Section 4 defines the method for distributed implementation of BIP models with observation and in particular, the necessary adaptation of the conflict resolution protocols. Experiments are reported in Section 5. Section 6 provides conclusions and perspectives for future work.

2. Semantic Models of BIP

In this section, we present BIP[3], a component framework for building systems from a set of atomic components by using two types of composition operators: Interaction and Priority. We then present an alternative model based on Observation that can express Priority. Finally we present a transformation from a component with Observation into a equivalent component with only Interaction.

Atomic Components. An *atomic component* B is a labelled transition system represented by a tuple (Q, P, T) where Q is a set of *control locations* or *states*, P is a set of *communication ports* and $T \subseteq Q \times P \times Q$ is a set of *transitions*.

Interactions. In order to compose a set of n atomic components $\{B_i = (Q_i, P_i, T_i)\}_{i=1,n}$, we assume that their respective sets of control locations and ports are pairwise disjoint; i.e., for any two $i \neq j$ in $\{1..n\}$, we require that $Q_i \cap Q_j = \emptyset$ and $P_i \cap P_j = \emptyset$. We define the global set $P \overset{def}{=} \bigcup_{i=1}^{n} P_i$ of ports. An *interaction* a is a set of ports such that a contains at most one port from each atomic component. We take $a = \{p_i\}_{i \in I}$ with $I \subseteq \{1..n\}$ and $p_i \in P_i$. If a is an interaction, we denote by $support(a)$ the set of atomic components that participate in a. This notation is extended to sets of interactions γ, that is, $support(\gamma) \overset{def}{=} \bigcup_{a \in \gamma} support(a)$.

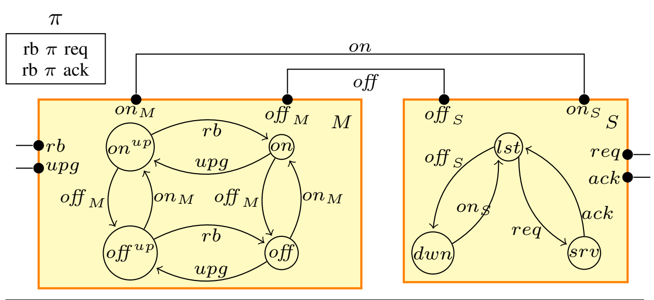


Figure 1. BIP component. Initial state is (off, dwn).

Priorities. Given a set γ of interactions, we define a priority as a strict partial order $\pi \subseteq \gamma \times \gamma$. We write $a \pi b$ for $(a, b) \in \pi$ to express that a has lower priority than b.

Composite Components. A *composite component* $\pi\gamma(B_1, \dots, B_n)$ (or simply *component*) is defined by a set of atomic components $\{B_i = (Q_i, P_i, T_i)\}_{i=1,n}$ composed by a set of interactions γ and a priority $\pi \subseteq \gamma \times \gamma$. If π is the empty relation, then we omit π and simply write $\gamma(B_1, \dots, B_n)$. A global state q of $\pi\gamma(B_1, \dots, B_n)$ is defined by a tuple of control locations $q = (q_1, \dots, q_n)$. The behavior of $\pi\gamma(B_1, \dots, B_n)$ is a labelled transition system $(Q, \gamma, \rightarrow_{\pi\gamma})$, where $Q = \bigotimes_{i=1}^{n} Q_i$ and $\rightarrow_\gamma$, $\rightarrow_{\pi\gamma}$ are the least sets of transitions satisfying the rules:

$$\frac{\begin{array}{c} a = \{p_i\}_{i \in I} \in \gamma \\ \forall i \in I.\, (q_i, p_i, q_i') \in T_i \qquad \forall i \notin I.\, q_i = q_i' \end{array}}{(q_1, \dots, q_n) \xrightarrow{a}_\gamma (q_1', \dots, q_n')} \; [\text{INTER}]$$

$$\frac{q \xrightarrow{a}_\gamma q' \qquad \forall a' \in \gamma.\, a \pi a' \implies q \overset{a'}{\not\rightarrow}_\gamma}{q \xrightarrow{a}_{\pi\gamma} q'} \; [\text{PRIO}]$$

Intuitively, transitions $\rightarrow_\gamma$ defined by rule [INTER] specify the behavior of the component without considering priorities. A component can execute an interaction $a \in \gamma$ iff for each port $p_i \in a$, the corresponding atomic component B_i can execute a transition labelled by p_i. If this happens, a is said to be *enabled*. Execution of a modifies atomically the state of all interacting atomic components whereas all others stay unchanged. The behavior of the component is defined by transitions $\rightarrow_{\pi\gamma}$ defined by rule [PRIO]. This rule restricts execution to interactions which are maximal with respect to the priority order. An enabled interaction a can execute only if no other interaction a' with higher priority is enabled.

Example 1. A BIP component is depicted in Figure 1 using a graphical notation. It consists of two atomic components named M and S. Component S is a server, that may receive requests (req) and acknowledge them (ack). Component M is a manager that may perform upgrades (upg) and needs to reboot (rb) the server for the upgrade to be done. Interactions are represented using connectors between the interacting ports. There are 4 unary interactions and 2 binary interactions. The component goes up and down through the binary interactions on and off respectively. Priority $rb\ \pi\ req$, $rb\ \pi\ ack$ is used to prevent a reboot whenever a request or an acknowledgement are possible.

2.1 Replacing Priority by Observation

According to BIP semantics, a low priority interaction is executed only if all higher priority interactions are not enabled. In general, detecting such situations requires information about components that are not involved in the low priority interaction. We propose here an alternative semantics of BIP parameterized by *Observation*. This semantics makes explicit the sets of components to be

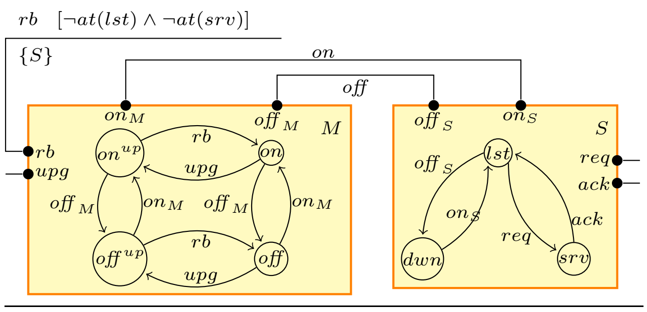


Figure 2. Example of a component with observation.

observed and the global state condition to be met for authorizing execution of each interaction.

Observation. Given a BIP component $\gamma(B_1, \ldots, B_n)$, we define an observation as a pair of functions $\mathcal{O} = (obs, pred)$, that are both defined over γ. Let $a \in \gamma$ be an interaction ; $obs(a)$ is a subset of $\{B_1, \ldots, B_n\}$ including the set of components *observed* by the interaction a. We require that $obs(a) \cap support(a) = \emptyset$. The observed components and the support of a are the components visible to a, that is $V_a = support(a) \cup obs(a)$. For $a \in \gamma$, $pred(a)$ is a predicate defined on the states of components in V_a.

Composite Component with Observation. A composite component with observation $\mathcal{O}\gamma(B_1, \ldots, B_n)$ is defined by a component $\gamma(B_1, \ldots, B_n)$ and an observation $\mathcal{O}$ over this component. The behavior of $\mathcal{O}\gamma(B_1, \ldots, B_n)$ is the labeled transition system $(Q, \gamma, \longrightarrow_{\mathcal{O}\gamma})$, where $Q = \bigotimes_{i=1}^{n} Q_i$ is the set of global states, and $\longrightarrow_{\mathcal{O}\gamma}$ is the least set of transitions satisfying the rule:

$$\frac{(q_1, \ldots, q_n) \xrightarrow{a}_{\gamma} (q'_1, \ldots, q'_n) \qquad pred(a)\,((q_i)_{B_i \in V_a})}{(q_1, \ldots, q_n) \xrightarrow{a}_{\mathcal{O}\gamma} (q'_1, \ldots, q'_n)} \text{[OBS]}$$

The rule [OBS] states that a transition a can take place in the component with observation if it is already a valid transition in the component $\gamma(B_1, \ldots, B_n)$ and if the predicate $pred(a)$ holds for the current state of components in V_a. The predicate $pred(a)$, is a boolean expression involving atomic predicates $at(q)$ for each state $q \in \bigcup_{i=1}^{n} Q_i$. The atomic predicate $at(q)$ evaluates to true whenever the corresponding atomic component is at state q and to false otherwise. The rule [OBS] requires that $pred(a)$ depends only on states of components that are visible to a, that is $pred(a)$ is a boolean expression on $at(q)$ predicates for $q \in \bigcup_{B_i \in V_a} Q_i$.

Example 2. Figure 2 depicts a composite component with observation. Each interaction is labeled by the set of observed components and the corresponding predicate. Here, the only interaction with additional observation is rb, with $obs(rb) = \{S\}$. The predicate for executing rb is written between square brackets.

Observation-based semantics violates the component encapsulation principle as it needs access to inner states of components. We use components with observation as an intermediate model towards a distributed implementation where we exploit the locality of observation: observing only the components visible to an interaction is sufficient to decide whether the interaction can take place.

Priority vs. observation. In Figure 2, we presented an example of composite components with observation. Note that the predicate associated to rb actually encodes the priority rule of Figure 1, since it guarantees that nor req neither ack are enabled when executing rb. We show that given a priority π one can obtain an observation $\mathcal{O}_\pi$ such that the behaviors of the components with priority and observation are identical.

Using $at(q)$ predicates, we define the predicate EN_a stating whether the interaction a is enabled. First, we define the predicate $EN^i_{p_i}$ characterizing enabledness of port p_i in a component $B_i = (Q_i, P_i, T_i)$, that is $EN^i_{p_i} = \bigvee_{(q_i, p_i, -) \in T_i} at(q_i)$. Then, the predicate EN_a can be defined by: $EN_a = \bigwedge_{p_i \in a} EN^i_{p_i}$. Note that this predicate depends only of components in $support(a)$.

Definition 1 (Priority Observation)**.** Given a prioritized BIP component $\pi\gamma(B_1, \ldots, B_n)$, we define the *priority observation* $\mathcal{O}_\pi = (obs, pred)$ for the component $\gamma(B_1, \ldots, B_n)$, for each interaction $a \in \gamma$:

- $obs(a)$ contains all components involved in an higher priority interaction b that do not participate in a. Formally: $obs(a) = \left(\bigcup_{a \pi b} support(b)\right) \setminus support(a)$.
- $pred(a)$ ensures that each higher priority interaction b is not enabled. Formally, $pred(a) = \bigwedge_{a\,\pi\,b} \neg EN_b$. Obviously, this predicate depends only on components in $support(a) \cup obs(a)$.

For the example in Figure 1, the only low-priority interaction is rb. For all other interactions, $obs(a)$ and $pred(a)$ are respectively $\emptyset$ and $True$. The component with observation obtained from the component with priority is exactly the one depicted in Figure 2. Indeed, rb observes the component S and the predicate on this interaction is $\neg at(lst) \wedge \neg at(srv) = \neg EN_{req} \wedge \neg EN_{ack}$.

Proposition 1. *Given a component with priority $\pi\gamma(B_1, \ldots, B_n)$ and the component with observation $\mathcal{O}_\pi\gamma(B_1, \ldots, B_n)$, where $\mathcal{O}_\pi$ follows Definition 1, we have $\longrightarrow_{\pi\gamma} = \longrightarrow_{\mathcal{O}_\pi\gamma}$.*

Proof. For each interaction a, the predicate $pred(a) = \bigwedge_{a\,\pi\,b} \neg EN_b$ is equivalent to $\forall b \in \gamma \;\; a \,\pi\, b \implies q \overset{b}{\not\rightarrow}_{\gamma}$. Thus the rules [PRIO] and [OBS] define exactly the same set of transitions. □

In [6], we provided a heuristic to reduce the scope of observation while preserving behavior equivalence. More precisely, this heuristic takes an observation $\mathcal{O}_\pi = (obs, pred)$ and returns another observation $\mathcal{O}' = (obs', pred')$, such that

- $\forall a \in \gamma \;\; |obs'(a)| \leq |obs(a)|$ the scope of the observation is reduced, and
- $\longrightarrow_{\mathcal{O}'\gamma} \subseteq \longrightarrow_{\mathcal{O}_\pi\gamma}$ the obtained behavior using observation $\mathcal{O}'$ is correct with respect to the original one.

Furthermore, the heuristics ensures either that if the inclusion is strict, no deadlocks are introduced or otherwise, that the obtained component has precisely the same behavior as the original one.

2.2 Implementing Observation with Interactions

We start from a component with observation $\mathcal{O}\gamma(B_1, \ldots, B_n)$ and translate it into an equivalent observable BIP component $\gamma'(B'_1, \ldots, B'_n)$. In order to implement observation, each atomic component has to make explicit its current state, both for interactions where it is involved and for interactions where it is observed. Observation is therefore encoded by extending interactions to observed components.

Transforming Atomic Components. Given an atomic component $B = (Q, P, T)$, we define the corresponding atomic observable component as a labeled transition system $B' = (Q', P', T')$, where:

- $Q = Q'$ the states are the same than in the original component.
- $P' = (P \cup \{obs\}) \times Q$: we add a new port denoted obs, that will be used for observation. All ports contain the information of the current state. We denote by $p(q)$ the port $(p, q) \in P'$.
- For each transition $(q, p, q') \in T$, T' contains the transition $(q, p(q), q')$ where the current state of the component is explicit

in the offered port. For $q \in Q$, T' contains the loop transition $(q, obs(q), q)$ that is used when the component is observed.

Transforming Interactions. Given a set γ of interactions and an observation $\mathcal{O} = (obs, pred)$, we define the new set of interactions γ' as follows. For each interaction $a \in \gamma$, where $a = \{p_i\}_{i \in I}$, we extend its support to the components $support(a) \cup obs(a) = \{B'_{j_1}, \ldots, B'_{j_k}\}$, and we denote by J the set of indices $\{j_1, \ldots, j_k\}$. For each state of this set of components $(q_{j_1}, \ldots, q_{j_k})$ such that $pred(a)(q_{j_1}, \ldots, q_{j_k})$ holds, γ' contains the interaction $a(q_{j_1}, \ldots, q_{j_k}) = \{p'_j(q_j)\}_{j \in J}$, where $p'_j = obs_j$ if $B_j \in obs(a)$, that is B_j is observed by a, and $p'_j = p_j$ otherwise. This transformation associates to any interaction a of $\mathcal{O}\gamma(B_1, \ldots, B_n)$ a set of interactions $a(q_{j_1}, \ldots q_{j_k})$ of $\gamma'(B'_1, .., B'_n)$, each interaction of γ' being enabled by states $(q_{j_1}, \ldots, q_{j_k})$ satisfying $pred(a)$.

Proposition 2. *We have $\longrightarrow_{\gamma'} = \longrightarrow_{\mathcal{O}\gamma}$ by mapping the interactions $a(q_{j_1}, \ldots, q_{j_k})$ of γ' to a.*

Proof. The states of $\mathcal{O}\gamma(B_1, \ldots, B_n)$ and $\gamma'(B'_1, \ldots, B'_n)$ are the same. The transition $q \xrightarrow{a}_{\mathcal{O}\gamma} q'$ can be fired if and only if the components visible to a, namely $\{B_j\}_{j \in J}$, are in a state $(q_{j_1}, \ldots, q_{j_k})$ satisfying the predicate $pred(a)$. In that case γ' contains an interaction $a(q_{j_1}, \ldots, q_{j_k})$. This interaction only changes the state of participants in a, thus we have $q \xrightarrow{a}_{\gamma'} q'$. $\square$

Note that the duplication of interactions can be avoided by using models extended with variables and guards on interactions, In that case, instead of creating a new port $p(q)$ for any pair in $P \times Q$, each port exports a state variable q. Then $pred(a)$ is the guard associated with the interaction a, and depends only on variables exported by the ports involved in a.

3. Decentralized Implementation of BIP

We provide here the principle of the method for distributed implementation of BIP presented in [8, 9]. This method relies on a systematic transformation from arbitrary BIP components[2] into distributed BIP components with Send/Receive interactions. These are binary point-to-point and directed interactions from one sender component (port), to one receiver component (port) implementing message passing, from the sender to the receiver. The transformation guarantees that the receive port is always enabled when the corresponding send port becomes enabled, and therefore Send/Receive interactions can be safely implemented using any asynchronous message passing primitives (e.g., MPI send/receive communication, TCP/IP network communication, etc...).

In a distributed setting, each atomic component executes independently and thus has to communicate with other atomic components in order to ensure correct execution with respect to the original semantics. Thus, a reasonable assumption is that each component will publish its offer, that is the list of its enabled ports, and then wait for a notification indicating which interaction has been chosen for execution. This is achieved by splitting each transition in atomic components: one part sends the offer, the other part is triggered by the notification and executes the chosen interaction.

The main difficulty when transforming a BIP component into a distributed Send/Receive BIP component is to resolve conflicts between simultaneously enabled interactions. In a centralized execution, only one entity is responsible for executing interactions, and has exclusive access to all components. In contrast, in a distributed setting, several entities may be responsible for executing interactions. A conflict occurs if two different entities try to execute two interactions involving a common component. If both entities send a notification to this component, then the original semantics is jeopardized, since a component cannot participate in two concurrently enabled interactions. For conflict resolution, a protocol must be used in order to ensure that conflicting interactions are not executed concurrently. This protocol takes into account the offers from components and sends back notifications so that the distributed execution is correct with respect to the original semantics.

Distributed conflict resolution boils down to solving the *committee coordination problem* [12], where a set of professors organize themselves in different committees, a meeting requires the presence of all professors to take place and two committees that have a professor in common cannot meet simultaneously. Different solutions have been provided, using managers [1, 12, 17, 18], circulating tokens [15], or randomized algorithms without managers [14] to implement the conflict resolution.

We first describe how atomic components are modified to send offers and receive notifications. Then, we focus on the Bagrodia's solutions from [1], that use managers and counters to implement conflict resolution. Finally, we recall how these protocols are used for building a 3-layer distributed component.

3.1 Distributed Atomic Components

The transformation of atomic components consists in splitting each transition into two consecutive transitions: (i) an *offer* that publishes the current state of the component, and (ii) a *notification* that triggers the transition corresponding to the chosen interaction. The offer transition publishes its enabled ports through a set of special ports, labeled $o(\mathit{Off})$ where Off is the subset of enabled ports.

Definition 2 (Distributed atomic components)**.** Let $B = (Q, P, T)$ be an atomic component. The corresponding transformed atomic component is $B^{\perp} = (Q^{\perp}, P^{\perp}, T^{\perp})$, such that:

- $Q^{\perp} = Q \cup \{\perp_q \mid q \in Q\}$ is the union of *stable* states Q and *busy* states $\{\perp_q \mid q \in Q\}$.
- $P^{\perp} = P \cup \{o(\mathit{Off}) | \mathit{Off} \subseteq P\}$, where $o(\mathit{Off})$ is a port indicating that ports in $\mathit{Off} \subseteq P$ are enabled.
- the set of transitions $T^{\perp}$ include, for every transition $\tau = (q, p, q') \in T$:
  1. an *offer* transition $\left(\perp_q, o(\{p | q \xrightarrow{p}\}), q\right)$ that goes from a busy to a stable state and publishes the offer.
  2. a *notification* transition $q \xrightarrow{p} \perp_{q'}$ that goes from a stable to a busy state and executes the transition from the original component.

Notice that we introduced a new port for each possible offer. This allows us to using the same model as for non-distributed atomic components. However, as the notation suggests, we can use a single port o with exported variables as described in [9].

3.2 Bagrodia's Counter-based Conflict Resolution

In Bagrodia's solutions, the protocol is made of one or several managers that receive offers from the atomic components and reply with notifications.

Centralized (Single) Manager. The first solution consists of a single manager. In order to ensure mutual exclusion of conflicting interactions, the protocol maintains two counters for each atomic component B_i:

- The *offer-count* n_i which counts the number of offers sent by the component. This counter is initially set to 0 and is incremented each time an offer from B_i is received.
- The *participation-count* N_i which counts the number of times the component participated in an interaction. This counter is

[2] with or without priorities

initially set to 0 and is incremented each time the manager selects an interaction involving B_i for execution.

Intuitively, the offer-count n_i associated to an offer from a component B_i correspond to a time stamp. The manager maintains the last used time stamp (N_i) for each component. If the time stamp (n_i) of an offer is greater than the last used time stamp (N_i), then the offer from B_i has not been consumed yet. Otherwise, some interaction has taken place and the manager has to wait for a new offer from this component.

Furthermore, the manager recalls the last offer sent by each component. Thus in order to schedule an interaction, it must check that (1) the interaction is enabled according to the last offers received and (2) these offers are still valid according to the n_i and N_i counters. We define formally the behavior of the centralized protocol as a composition operator over distributed atomic components.

Definition 3 (Centralized Counter-based Implementation)**.** Given a BIP component $\gamma(B_1, \dots, B_n)$ we define the behavior of the counter-based centralized implementation as an infinite state LTS $(Q^\perp, \gamma^\perp, T^\perp)$ where:

- The set of states $Q^\perp$ is the product of the states of the atomic components with the state of the protocol:

$$Q^\perp = \bigotimes_{i=1}^{n} Q_i^\perp \times \bigotimes_{i=1}^{n} \left(\mathbf{N} \times \mathbf{N} \times 2^{P_i}\right)$$

  The state of the manager is defined by n triplets $m_i = (n_i, N_i, \mathit{Off}_i)$, one for each component B_i, where n_i and N_i are the values of the corresponding counters and Off_i is the last offer from B_i. We denote by (q, m) a state of $Q^\perp$, $q[i]$ and $m[i]$ represent the ith component of the tuples q and m.
- The interactions $\gamma^\perp$ consists of interactions of the original component and the offers:

$$\gamma^\perp = \gamma \cup \bigcup_{i=1}^{n} \bigcup_{\mathit{Off} \in 2^{P_i}} o_i(\mathit{Off}_i)$$

- There are two types of transitions in $T^\perp$:
  (1) offer transitions: From state $(q, m) \in Q^\perp$, there is an offer transition in $T^\perp$ if for some component B_i an offer is enabled: $(q[i], o_i(\mathit{Off}), q_i') \in T_i^\perp$. Then $T^\perp$ contains the transition $(q, m) \xrightarrow{o_i(\mathit{Off})} (q', m')$, where :
  - $q'[i] = q_i'$,
  - $m'[i] = (n_i + 1, N_i, \mathit{Off})$, with $m[i] = (n_i, N_i, \mathit{Off}_i)$,
  - for all $j \neq i$, $q'[j] = q[j]$ and $m'[j] = m[j]$.

  (2) execute transitions: From state $(q, m) \in Q^\perp$, there is an execute transition in $T^\perp$ if for some interaction $a = \{p_i\}_{i \in I}$, we have, for all $i \in I$ (with $m[i] = (n_i, N_i, \mathit{Off}_i)$):
  - $p_i \in \mathit{Off}_i$: the interaction is enabled according to the last offers,
  - $n_i > N_i$: the last offers are still valid.

  Then, the transition $(q, m) \xrightarrow{a} (q', m')$ is in $T^\perp$, with:
  - $\forall i \in I$, $q'[i]$ is the state such that $(q[i], p_i, q'[i]) \in T_i^\perp$,
  - $\forall i \in I$, $m'[i] = (n_i, N_i{+}1, \mathit{Off}_i)$: counters of participants are incremented.
  - $\forall j \notin I$, $q'[j] = q[j] \wedge m'[j] = m[j]$

We show that the component $\gamma(B_1, \dots, B_n)$ and the corresponding counter-based implementation are observationally equivalent in the sense of Milner [16]. We first prove the following lemma.

Lemma 1. *If $n_i > N_i$, then the component $B_i^\perp$ is in a stable state q_i and $\mathit{Off}_i = \{p | q_i \xrightarrow{p}_i\}$.*

Proof. The construction of $B_i^\perp$ implies that it alternates offer and execute transitions. Initially, $n_i = N_i$ and $B_i^\perp$ is in a busy state. The only possible transition is an offer, which brings the system to a state where $n_i = N_i + 1 > N_i$ is true and the offer transition ensures the property to prove. Next possible step in $B_i^\perp$ is an execute action, after which again $n_i = N_i$ and $B_i^\perp$ is a busy state. This behavior repeats forever. ☐

In order to show observational equivalence, we have to define the observable actions of both systems. For the component $\gamma(B_1, \dots, B_n)$ the observable actions are the interactions γ. For the counter-based implementation, the visible actions are the execute actions γ. We denote by β the offer actions.

We define a relation between states Q of the centralized component and states $Q^\perp$ of its distributed implementation. To each state $q^\perp \in Q^\perp$ of the distributed implementation, we associate a state $e(q^\perp) \in Q$ of the original component. For each component $B_i^\perp$, $q^\perp[i]$ is either a stable state q_i or a busy state $\perp_{q_i}$. In both cases, we take $e(q^\perp)[i] = q_i$. We say that a state $q \in Q$ and $q^\perp \in Q^\perp$ are equivalent, denoted by $q^\perp \sim q$, if $q = e(q^\perp)$.

Proposition 3 (Correctness of Centralized Counter-based Implementation)**.** *Given a component $\gamma(B_1, \dots, B_n)$, the labeled transitions systems (Q, γ, T) and $(Q^\perp, \gamma^\perp, T^\perp)$ of its distributed implementation are observationally equivalent.*

Proof. We have to prove that:

1. If $q^\perp \xrightarrow{\beta} r^\perp$, then $\forall q \sim q^\perp,\ r \sim q^\perp$.
2. If $q^\perp \xrightarrow{a} r^\perp$, then $\forall q \sim q^\perp,\ \exists r \in Q\ \ q \xrightarrow{a}_\gamma r \wedge r \sim r^\perp$.
3. If $q \xrightarrow{a} r$, then $\forall q^\perp \sim q,\ \exists r^\perp \in Q^\perp\ \ q^\perp \xrightarrow{\beta^* a} r^\perp \wedge r \sim r^\perp$.

1. This is a consequence of the definition of $\sim$.
2. The transition $(q^\perp, a, r^\perp)$ is possible at state $q^\perp \in Q^\perp$ if for each participant B_i in the interaction, the counters verify $n_i > N_i$, and for each port $p_i \in a$, we have $p_i \in \mathit{Off}_i$. The Lemma 1 ensures that in the equivalent state $q \in Q$, we have as well $q \xrightarrow{a} r$. The construction of distributed atomic components ensures that $r \sim r^\perp$.
3. If $q \xrightarrow{a} r$, then for each state $q^\perp \sim q$, each participant B_i in a is either in a busy or in a stable state. In the first case, it can perform an offer transition, labeled β, and reach a stable state. By point 1., the stable state $q'^\perp$ such that $q^\perp \xrightarrow{\beta^*} q'^\perp$ is also equivalent to q. At state $q'^\perp$, all offers transitions for a have been executed and we have $q^\perp \xrightarrow{\beta^*} q'^\perp \xrightarrow{a} r^\perp$, with $r^\perp \sim r$. ☐

In Definition 3, the enabling of offer transitions depends exclusively on the state of the component sending the offer. Similarly, the enabling of execute transitions is decided by the manager alone. Thus we can assume an asynchronous execution where an offer transition is executed first by the atomic component, by sending a message and then by the manager when receiving the message. Similarly, the execute transitions are performed after the manager sends messages to components involved in the interaction.

Decentralized (Multiple) Manager(s). In [1], Bagrodia decentralizes the manager into a set of distributed managers, also relying on counters to ensure correct execution of the interactions. The correctness is guaranteed as long as each manager can check and modify atomically all the N_i counters corresponding to an interaction. Bagrodia proposes two protocols guaranteeing this atomicity:

- The token ring protocol, where a token circulates through all managers. This token stores the N_i counters for the whole system, which guarantees atomic access for each manager.
- The dining philosophers protocol, where two interactions that involve a common component share a fork with a copy of the N_i counter on it. In order to execute an interaction, the manager needs to acquire all forks and can then check and update if necessary all N_i values simultaneously.

It can be shown that these protocols are trace equivalent with the centralized implementation [9]. However, they are not observationally equivalent with the centralized implementation, since the position of the token or of the forks may prevent some choices to be made (see [9] for details).

3.3 3-layer Distributed Architecture

The obtained distributed components must meet the following three properties: (1) preserve the behavior of each atomic component, (2) preserve the behavior of interactions, and (3) resolve conflicts in a distributed manner. To ensure these properties, we structure distributed components according to a hierarchical architecture with three layers. The lower layer includes the transformed atomic components. The second layer deals with distributed interaction execution by implementing interaction protocols (IP). The third layer deals with conflict resolution. Since several distributed algorithms exist for conflict resolution, this layer is generic with appropriate interfaces. An example of 3-layer architecture obtained from the component presented in Figure 1 is depicted in Figure 3.

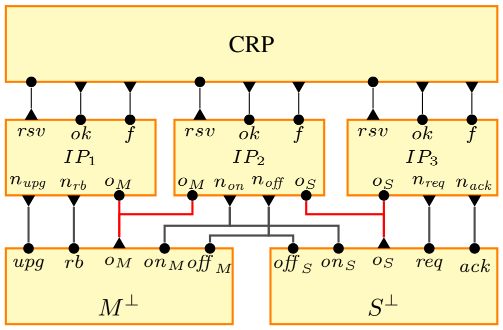


Figure 3. 3-layer distributed implementation of component from Figure 1.

Components Layer. This layer contains the distributed version of the atomic components, as described in section 3.1. In Figure 3, it corresponds to components $M^\perp$ and $S^\perp$.

Interaction Protocol. This layer consists of a set of interaction protocols each hosting a set of interactions from the original BIP component. Conflicts between interactions included in the same interaction protocol are resolved by that component locally. On Figure 3, IP_1 handles interaction upg and rb, IP_2 handles on and off, and IP_3 handles req and ack.

The interaction protocol evaluates the guard of each interaction and executes the code associated with an interaction that is selected locally or by the upper layer. The interface between this layer and the component layer provides ports for receiving offers from each component (through ports such as o_M) and notifying the components on permitted port for execution (through ports such as n_{on}). Sender ports are denoted by triangles and receiver ports by bullets. Interactions with one sender and multiple receivers means that the sender sequentially sends a message to each receiver.

Conflict Resolution Protocol. This algorithm embeds one of the Bagrodia's counter-based protocols as presented in the previous section. The protocols have been slightly modified since managers do not receive offers one by one from components but instead receive the set of offers corresponding to an interaction sent by one of the interaction protocols. The protocol can either be centralized, or distributed e.g. token ring or dining philosophers. The interface between this layer and the Interaction Protocol involves ports for receiving requests to *reserve* an interaction (labelled rsv) and responding by either success (labelled ok) or failure (labelled f).

4. Distributed Implementation of Observational Semantics

Applying the transformation presented in Subsection 2.2 followed by the distribution method presented in 3 allows to obtain a distributed model from a component with observation. This method leads to a *multiparty-based* implementation. We show here that a multiparty-based implementation is costly, as it treats all observation conflicts as structural conflicts. We propose an optimized version of Bagrodia's counter-based protocol presented in the previous section, that allows us to build an *observation-aware* implementation.

4.1 Observation Conflicts

Using the transformation presented in 2.2, we can transform a component with observation into a observable component. This transformation implements observation of components through new ports denoted obs. However, it introduces new structural conflicts between interactions on the observation ports obs.

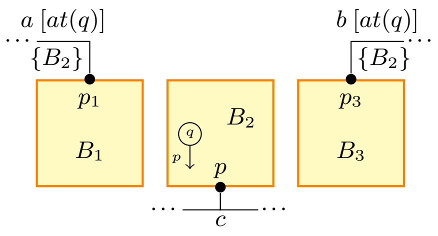


Figure 4. Model with observation.

As an example, consider the model depicted in Figure 4. It contains three atomic components and three fragments of interaction. Interactions a and b observe the atomic component B_2. Execution of a or b will not change the state of B_2 since none of its transitions is involved. Intuitively, a and b can be executed in parallel, they do not really conflict. However, execution of c changes the state of the atomic component B_2 and may disable the predicate associated to a or b. Thus a and c cannot be executed simultaneously. They are conflicting.

This type of conflicts also appears in transactional memories [13]. In this context, different transactions (interactions) can simultaneously read (observe) a variable (an atomic component), but writing on a variable (executing a transition) requires exclusive access to the variable.

When we transform such a model with observation into a observable model, as described in subsection 2.2, we obtain the model depicted in the Figure 5. The observation is implemented by adding a new port $obs(q)$ and extending interactions a and b to that new port. In this model, B_2 becomes a participant in the interactions a and b by executing a loop transition. This results in a structural conflict between a and b.

The 3-layer distributed implementation generated from a component obtained with the transformation presented in Subsec-

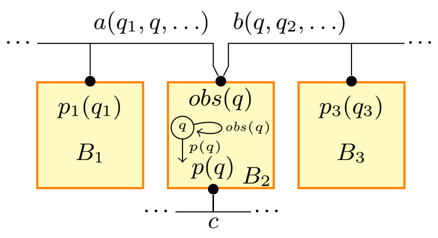


Figure 5. Observable model obtained from the model with observation in Figure 4.

tion 2.2 involves an unnecessarily high number of exchanged messages. Consider the model presented in Figure 5. Execution of interaction a followed by interaction b requires at least 4 messages between the component B_2 and the protocol. Indeed, each interaction requires at least one offer and one notification. These four messages could be replaced by a single one, indicating that B_2 is at state q to the protocol, since the component B_2 does not need to be notified when it is observed.

4.2 Counter-based Conflict Resolution for Observation

The transformation from a component with observation to an observable component adds new conflicts and results in a message-inefficient distributed implementation. In order to avoid this, we modify the conflict resolution protocol to take observation into account. The particularity of observation is checking that a component is at a particular state, without state change. This differs from multiparty interactions, where observation is combined with state change.

The proposed adaptation of the counter-based protocol presented in Definition 3 can be reused in the 3-layer BIP model to encompass observation and thus priority.

This adaptation relies on the following key facts:

- *Observation of a component does not imply state change*. Freshness of the offer from a component (the observation) is still validated by checking $n_i > N_i$. However, upon execution of an interaction, the N_i counters corresponding to the observed components are not incremented. Thus $n_i > N_i$ still holds and another interaction observing the same component can still take place.
- *The state predicates need to be checked*. This assumes that every component sends its local state with its offer and that the manager knows the state predicate for each interaction.

Definition 4. Given a BIP component with observation $\mathcal{O}\gamma(B_1, \ldots, B_n)$ we define the behavior of the adapted counter-based centralized implementation as an infinite state LTS $(Q^\perp, \gamma^\perp, T^\perp)$ where:

- The set of states $Q^\perp$ is the product of the states of the atomic components with the state of the protocol:
$$Q^\perp = \bigotimes_{i=1}^{n} Q_i^\perp \times \bigotimes_{i=1}^{n} \left(\mathbf{N} \times \mathbf{N} \times 2^{P_i} \times Q_i\right)$$
The state of the manager is defined by n quadruples $m_i = (n_i, N_i, \mathit{Off}_i, q_i)$, one for each component B_i, where n_i and N_i are the values of the corresponding counters, Off_i is the last offer from B_i and q_i is the last known state from B_i. We denote by (q, m) a state of $Q^\perp$, $q[i]$ and $m[i]$ represent the ith element of the tuples q and m.
- The interactions of $\gamma^\perp$ include the interactions from the original component and the offers:
$$\gamma^\perp = \gamma \cup \bigcup_{i=1}^{n} \bigcup_{\mathit{Off} \in 2^{P_i}} o_i(\mathit{Off}_i)$$
- There are two types of transitions in $T^\perp$:
(1) offer transitions: From state $(q, m) \in Q^\perp$, there is an offer transition in $T^\perp$ if for some component B_i an offer is enabled: $(q[i], o_i(\mathit{Off}), q_i') \in T_i^\perp$. That is, $T^\perp$ contains the transition $(q, m) \xrightarrow{o_i(\mathit{Off})} (q', m')$, where:
  - $q'[i] = q_i'$,
  - $m'[i] = (n_i{+}1, N_i, \mathit{Off}, q[i])$ (with $m[i] = (n_i, N_i, \mathit{Off}_i, q_i)$,
  - for all $j \neq i$, $q'[j] = q[j]$ and $m'[j] = m[j]$.

  (2) execute transitions: From state $(q, m) \in Q^\perp$, there is an execute transition in $T^\perp$ if for some interaction $a = \{p_i\}_{i \in I}$, we have, for all $i \in I$ (with $m[i] = (n_i, N_i, \mathit{Off}_i, q_i)$):
  - $p_i \in \mathit{Off}_i$: the interaction is enabled according to the last offers,
  - $n_i > N_i$: the last offers are still valid.

  Furthermore, we require that $pred(a)((q_i)_{B_i \in V_a})$ holds.
  Then, the transition $(q, m) \xrightarrow{a} (q', m')$ is in $T^\perp$, with:
  - $\forall i \in I$, $q'[i]$ is the state such that $(q[i], p_i, q'[i]) \in T_i^\perp$,
  - $\forall i \in I$, $m'[i] = (n_i, N_i + 1, \mathit{Off}_i, q_i)$: counters of participants are incremented.
  - $\forall j \notin I$, $q'[j] = q[j] \wedge m'[j] = m[j]$

As for the counter-based implementation, we prove the correctness of the adapted version using Milner's observational equivalence.

Proposition 4 (Correctness of adapted Counter-based Implementation). *Given a component $\mathcal{O}\gamma(B_1, \ldots, B_n)$, the labeled transitions systems (Q, γ, T) and $(Q^\perp, \gamma^\perp, T^\perp)$ of its distributed implementation are observationally equivalent.*

The proof has the same structure as for the Proposition 3, and uses the same equivalence relation. The only difference is in points 2. and 3. where we have to take into account the additional enabling condition. More precisely, we have to show that the truth value of the enabling condition is preserved by the equivalence relation restricted to *stable* states. This is obtained by considering the counters of observed components.

The correctness is guaranteed through the fact that checking the freshness of offers sent by visible components and incrementing the counters of participant components is an atomic action. Thus as for Bagrodia's original version, the manager can be distributed provided this atomicity is ensured, either by the token ring or by the dining philosophers solutions.

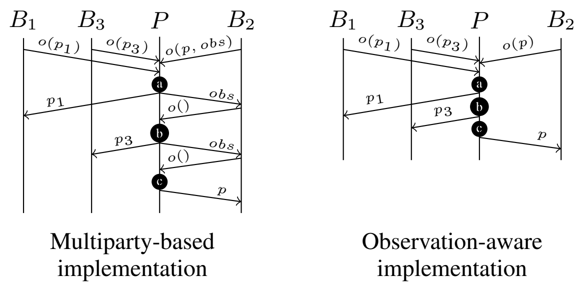


Figure 6. Exchanges of messages to execute the sequence a, b, c in the model of Figure 4, for the two implementations.

Example 3. To illustrate the behavior of this new protocol, consider again the model depicted in Figure 4. We obtain a multiparty-based implementation by transforming it into the model of Figure 5 and then using the original protocol from Bagrodia. The modified protocol presented here allows to obtain an observation-aware implementation directly from the model in Figure 4. In Figure 6, we compare the behavior of the two approaches, when executing the interaction sequence a, b, c. On the left, we show the messages exchanged in the multiparty-based implementation. On the right we show the messages exchanged in the observation-aware implementation. For each process (the distributed components B_i and the protocol P) Figure 6 presents the sequence of messages received and sent. The black circles indicate that an interaction is scheduled by the Protocol. Note that the component B_2 is observed by a and b and is participant in c. With the multiparty-based implementation, the observation is treated as a participation. Both execution of a and b trigger the emission of a notification (obs) to B_2 followed by a new offer ($o()$). With the observation-aware implementation, the first offer sent by B_2 is observed but not consumed by a and b. So, there is no need to send notifications and wait for corresponding offers. Only the execution of c consumes the offer. For this particular configuration, the new protocol spares 4 messages and increases parallelism since b and c can be launched directly after a, without waiting for a new offer.

The observation-aware implementation is more message-efficient than the multiparty-based implementation. If there is no observation, both implementations behave exactly the same. If there is an observation, executing the observing interaction results in the emission of a notification to each observed component in the multiparty-based implementation. This notification is not generated in the observation-aware implementation. Moreover, in the observation-aware implementation, an offer may be shared between several interactions observing the same component, reducing further the overall number of messages.

5. Experiments

We compare the execution time and the number of exchanged messages for several distributed implementations of a component with priority. The first step involves transformation of this component into a component with observation. Then we consider the two following sequences of transformations.

- Transform the component with observation into an observable component as explained in Subsection 2.2. Then generate a 3-layer distributed model embedding Bagrodia's conflict resolution protocol described in Subsection 3.2. This method results in a multiparty-based implementation.
- Directly transform the component with observation into a 3-layer distributed model embedding the modified conflict resolution protocol described in Subsection 4.2. This method results in a observation-aware implementation.

For both implementations, we used the centralized version of the conflict resolution protocol.

5.1 Dining Philosophers

We consider a variation of the dining philosophers problem, denoted by Philo$N$ where N is the number of philosophers. A fragment of this composite component is presented in Figure 7. In this component, an "eat" interaction eat_i involves a philosopher and the two adjacent forks. After eating, philosopher P_i cleans the forks one by one ($cleanleft_i$ then $cleanright_i$). We consider that each eat_i interaction has higher priority than any $cleanleft_j$ or $cleanright_j$ interaction.

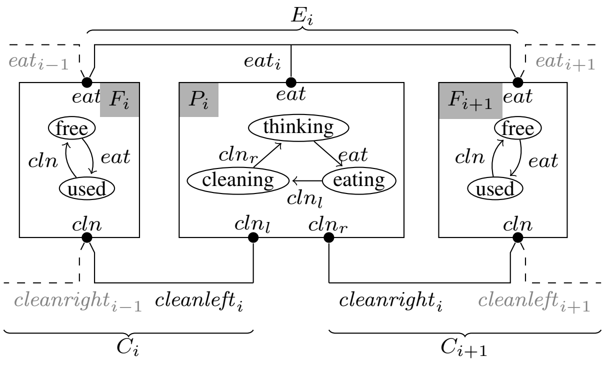


Figure 7. Fragment of the dining philosopher component. Braces indicate how interactions are grouped into interaction protocols.

This example has a particularly strong priority rule. Indeed, executing one "$clean$" interaction requires to check that *all* "eat" interactions are disabled, that is to observe all components. This example allows to compare both implementations under strong priority constraints.

As explained in Section 3.3, the construction of our distributed implementation is structured in 3 layers. The second layer is parameterized by a partition of the interactions. For this example, the partition is built as follows. There is one interaction protocol E_i for every eat_i interaction and one interaction protocol C_i for every pair $cleanright_{i-1}$, $cleanleft_i$. Only the latter deals with low priority interactions that need to observe additional atomic components.

We compare multiparty-based and observation-aware implementations. For both, once we have built the distributed components, we use a code generator that generates a standalone C++ program for each atomic component. These programs communicate by using Unix sockets.

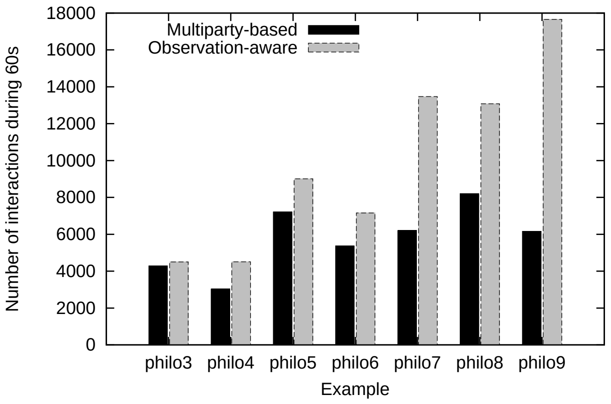


Figure 8. Number of interactions executed in 60s for the dining philosophers example.

The obtained code has been run on a UltraSparc T1 that allows parallel execution of 24 threads. For each run, we count the number of interactions executed and messages exchanged in 60 seconds, not including the initialization phase. For each instance we consider the average values obtained over 10 runs. The number of interactions executed by each implementation is presented in Figure 8. The total number of messages exchanged for the execution of each implementation is presented in Figure 9.

The comparison of the two implementations shows a huge difference both in performance (number of interactions executed) and

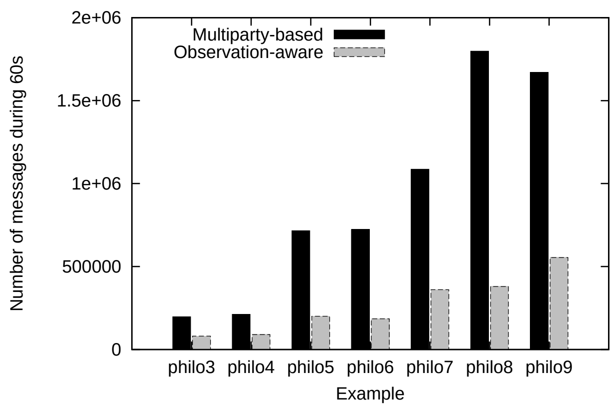


Figure 9. Number of messages exchanged in 60s for the dining philosophers example.

communications needed (total number of messages exchanged). The observation-aware implementation is fastest and needs less messages than multiparty-based implementation. This can be explained as follows. In both cases, eat_i interactions can execute in parallel, provided they do not involve a common fork. However, resolving priority conflicts requires to observe all components for executing a $cleanleft_i$ or a $cleanright_i$ interaction. In the multiparty-based implementation, observed components must synchronize to execute some interaction $cleanleft_i$ or $cleanright_i$. Between two "$clean$" executions, each component has to receive a notification and to send a new offer. This strongly restricts the parallelism. In the observation-aware implementation, a component offer is still valid after execution of an interaction observing that component. For a "$clean$" interaction, only two components will need to send a new offer before another "$clean$" interaction can be executed. This explains the speedup.

5.2 Jukebox

The second example is a jukebox depicted in Figure 10. It represents a system, where a set of readers $R_1 \ldots R_4$ access data located on N disks $D_1 \ldots D_N$. Readers may need to access any disk. We denote by jukebox$N$ the jukebox component with N disks. Access to disks is managed by jukeboxes J_1, J_2 that can load any disk to make it available to the connected readers. The interaction $load_{i,k}$ (respectively $unload_{i,k}$) allows loading (respectively unloading) the disk D_i in the jukebox J_k. Each reader R_j is connected to a jukebox through the $read_j$ interaction. Once a jukebox has loaded a disk, it can either take part in a "read" or "unload" interaction. Each jukebox repeatedly loads all N disks in a random order.

If unload interactions are always chosen immediately after a disk is loaded, then readers may never be able to read data. Therefore, we add the priority $unload_{i,k} \, \pi \, read_j$, for all i, j, k. This ensures that "read" interactions will take place before corresponding disks are unloaded. Furthermore, we assume that readers connected to J_1 need more often disk 1 and that readers connected to J_2 need more often disk 2. Therefore, loading these disks in the corresponding jukeboxes is assigned higher priority: $load_{i,1} \, \pi \, load_{1,1}$ for $i \in \{2, 3\}$ and $load_{i,2} \, \pi \, load_{2,2}$ for $i \in \{1, 3\}$. Each interaction is handled by a dedicated interaction protocol.

Compared to the Dining Philosopher example, this one has more localized priorities, in the sense that they do not require to observe the global state of the system. Here a priority rule is used to express a scheduling policy that aims to improve the efficiency of the system, in terms of "$read$" interactions. Generating the

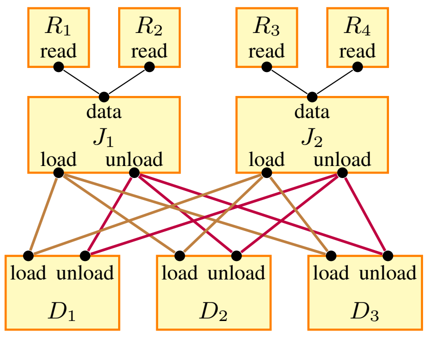


Figure 10. Jukebox component with 3 discs.

same example without taking priority into account results in an implementation that does less "$read$" interactions.

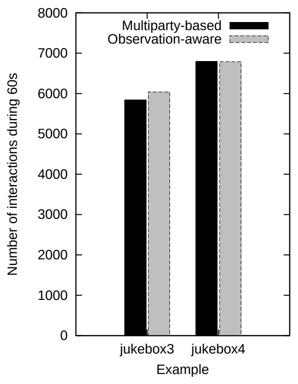


Figure 11. Number of interaction executed in 60s for the jukebox example.

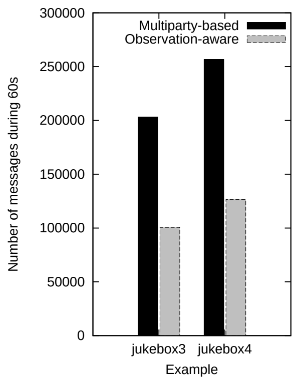


Figure 12. Number of messages exchanged in 60s for the jukebox example.

We performed the same measurements, in the same conditions as for the previous example. The number of interactions executed in 60s is presented in Figure 11. Here performance of both versions is the same. The main reason is that no or few parallelism is allowed between low priority interactions, i.e. two "$unload$" interactions from the same jukebox cannot be launched sequentially and run in parallel since they involve the same jukebox. However, Figure 12 shows that fewer messages are exchanged, with the observation-aware implementation. Intuitively, this difference corresponds to the notifications and subsequent offers to and from observed components, that are not necessary with the observation-aware implementation.

6. Conclusion

We proposed different methods of generating a distributed implementation for multiparty interactions with observation. The proposed model ensures enhanced expressiveness as the enabling conditions of an interaction can be strengthened by state predicates of components non participating in that interaction. It directly encompasses modeling of priorities which are essential for modeling scheduling policies. We have proposed a transformation leading from a model with observation into an equivalent model with interactions. The transformation consists in creating events making visible state-dependent conditions.

Expressing observation by interactions allows the application of existing distributed implementation techniques, such as the one presented in [9]. We have proposed an optimization of the conflict resolution algorithm from [1] that takes into account the fact that an observed component does not participate in the observing interaction. Preliminary experiments show significant performance improvement of this optimized implementation method.

Future work directions include the study of knowledge-based techniques [6] for efficient conflict resolution, in particular by minimizing the set of the observed components for each interaction. We also plan to study optimized implementations of systems with multiparty interaction and observation, for other implementations based on other conflict resolution protocols, such as α-core [18].

References

[1] R. Bagrodia. Process synchronization: Design and performance evaluation of distributed algorithms. *IEEE Transactions on Software Engineering (TSE)*, 15(9):1053–1065, 1989.

[2] A. Basu, P. Bidinger, M. Bozga, and J. Sifakis. Distributed semantics and implementation for systems with interaction and priority. In *Formal Techniques for Networked and Distributed Systems (FORTE)*, pages 116–133, 2008.

[3] A. Basu, M. Bozga, and J. Sifakis. Modeling heterogeneous real-time components in BIP. In *Software Engineering and Formal Methods (SEFM)*, pages 3–12, 2006.

[4] Ananda Basu, Saddek Bensalem, Doron Peled, and Joseph Sifakis. Priority scheduling of distributed systems based on model checking. *Formal Methods in System Design*, 39(3):229–245, 2011.

[5] S. Bensalem, M. Bozga, S. Graf, D. Peled, and S. Quinton. Methods for knowledge based controlling of distributed systems. In *Automated Technology for Verification and Analysis - 8th International Symposium, ATVA 2010, Proceedings*, volume 6252, pages 52–66. Springer, September 2010.

[6] S. Bensalem, M. Bozga, J. Quilbeuf, and J. Sifakis. Knowledge-based distributed conflict resolution for multiparty interactions and priorities. In *FMOODS/FORTE*, pages 118–134, 2012.

[7] Saddek Bensalem, Doron Peled, and Joseph Sifakis. Knowledge based scheduling of distributed systems. In *Essays in Memory of Amir Pnueli*, pages 26–41, 2010.

[8] B. Bonakdarpour, M. Bozga, M. Jaber, J. Quilbeuf, and J. Sifakis. From high-level component-based models to distributed implementations. In *EMSOFT*, 2010.

[9] B. Bonakdarpour, M. Bozga, M. Jaber, J. Quilbeuf, and J. Sifakis. A framework for automated distributed implementation of component-based models. *Distributed Computing*, to appear.

[10] B. Bonakdarpour, M. Bozga, and J. Quilbeuf. Automated distributed implementation of component-based models with priorities. In *EMSOFT*, pages 59–68, 2011.

[11] K. M. Chandy and J. Misra. The drinking philosophers problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 6(4):632–646, 1984.

[12] K. M. Chandy and J. Misra. *Parallel program design: a foundation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1988.

[13] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. *SIGARCH Comput. Archit. News*, 21(2):289–300, May 1993.

[14] Y.-J. Joung and S. A. Smolka. Strong interaction fairness via randomization. *IEEE Trans. Parallel Distrib. Syst.*, 9(2):137–149, 1998.

[15] D. Kumar. An implementation of n-party synchronization using tokens. In *ICDCS*, pages 320–327, 1990.

[16] R. Milner. *Communication and concurrency*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, 1995.

[17] J. Parrow and P. Sjödin. Multiway synchronizaton verified with coupled simulation. In *International Conference on Concurrency Theory (CONCUR)*, pages 518–533, 1992.

[18] J. A. Pérez, R. Corchuelo, and M. Toro. An order-based algorithm for implementing multiparty synchronization. *Concurrency and Computation: Practice and Experience*, 16(12):1173–1206, 2004.

Distributed Priority Synthesis using Knowledge

Chih-Hong Cheng[1] Rongjie Yan[2] Harald Ruess[1] Saddek Bensalem[3]

[1] Fortiss - An-Institut der TU München, Munich, Germany
[2] State Key Laboratory of Computer Science, Beijing, China
[3] Verimag Laboratory, Grenoble, France

cheng@fortiss.org yrj@ios.ac.cn ruess@fortiss.org saddek.bensalem@imag.fr

Abstract

For distributed computing, orchestrations along predefined communication paths are used to obtain agreement between system components on the next chosen transition. Although the communication overhead can be high, it can be efficiently reduced by the introduction of knowledge, which provides each local component imperfect view on the global state during run-time. In this paper, given a safety criterion, we formulate the problem how to automatically orchestrate components in a system using stateless precedences between actions under the assist of statically computed knowledge. If the system is diagnosed as unsafe, the use of knowledge can be integrated in the synthesis process to enlarge the set of legal fixing candidates. These new solution candidates may disrespect predefined communication paths but their defined priorities are still guaranteed to be deployable.

Keywords component-based systems, knowledge, synthesis

1. Introduction

In distributed computing, knowledge — algorithmic methods for each component to reason global execution from a local view — is essential to reduce communication overhead in distributed orchestration of system components. Consider, for example, the situation where component A is prohibited from executing action τ whenever component B is able to execute action σ. These kinds of dependencies between actions are called *priorities* and each priority is of the form $\tau \prec \sigma$. In situations where A intends to execute τ, it needs to be informed by B that B does not intend to execute σ. This requires an explicit communication from B to A. In general, these kinds of communication and synchronization overhead at runtime may lead to inefficient execution of the distributed systems. In many interesting cases, however, component A may already infer from, a priori, knowledge about certain aspects of the global behavior. For example, if it is known by A that it is impossible for B to execute σ in the current configuration, then it is safe for A to execute τ without initiating any further communication. We refer the above use of knowledge *posterior* to design, i.e., it is used at *run-time* to reduce the communication. The problem of using knowledge to assist system execution can be found in several recent works [2, 6, 7].

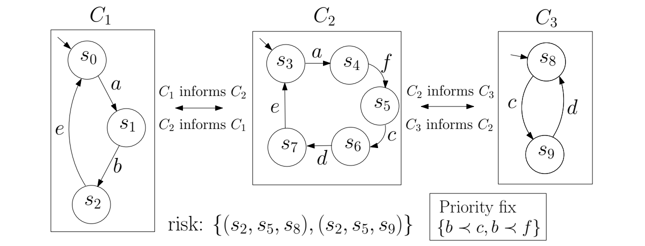


Figure 1. A simple component-based system.

In this paper, we investigate the dual problem, i.e., how can the use of knowledge assist system design, where the knowledge is computed *statically* and *prior* to the design process. The importance of this problem is that the integration of knowledge allows to automatically find distributed controllers, where in some cases the synthesis process without knowledge can not provide a solution (shown in later sections). Here we constrain "design" to the process of synthesizing appropriate orchestration mechanisms on a system from a given set of components, together with fixed interactions between components and pre-defined communication topologies. More precisely, our developments are based on generating distributed controllers in terms of stateless precedences (i.e., priorities) between transitions to control a system such that the controlled system is deadlock-free and respects given safety properties, as in *distributed priority synthesis* [4]. We show that this framework of distributed priority synthesis can be extended, such that under the assistance of knowledge, we can synthesize a set of priorities which may disrespect the defined communication paths but such a priority set is still guaranteed to be deployable due to the precomputed knowledge. Nevertheless, knowledge is used differently apart from the example above: for such examples we consider that the knowledge is used *negatively* by guaranteeing the absence of certain actions. In synthesis, as introducing a priority $\tau \prec \sigma$ is used to block τ under the condition that σ is also available, the knowledge shall be used *positively* by guaranteeing the presence of certain actions. Unfortunately, due to the interleaving semantics and nondeterminism, methods which compute knowledge might only return a *weak* form, i.e., for a given local state s, a set Σ of actions where at least one of them is enabled. This implies that when a component is executing τ at its local state s, the use of weak knowledge requires to introduce a complete set of priorities $\{\tau \prec \sigma \mid \sigma \in \Sigma\}$, such that the local component can use the inferred knowledge and block executing τ.

2. Knowledge and Distributed Priority Synthesis

In this section, we formulate the problem of synthesizing priority-based distributed controllers using knowledge. The underlying system is a simplified form of the Behavior-Interaction-Priority (BIP) framework [1], where we omit many syntactic features of BIP such as hierarchies of interactions and disallow the use of variables. In addition, uncontrollability from the environment is not modelled. We use the system in Figure 1 as a running example.

2.1 Component-based Systems and Global Semantics

Let Σ be the set of *interactions*. A *component* C_i of the form $(L_i, \Sigma_i, T_i, l_i^0)$ is a *transition system*, where L_i is a nonempty, finite set of *control locations*, $\Sigma_i \subseteq \Sigma$ is a nonempty subset of interaction labels used in C_i. T_i is the set of *transitions* of the form (l, σ, l'), where $l, l' \in L_i$ respectively are the source and target locations, and $\sigma \in \Sigma_i$ is an interaction label (specifying the event triggering the transition). Finally, $l_i^0 \in L_i$ is the *initial location*.

A system $\mathcal{S}$ of *interacting components* is of the form $(C = \bigcup_{i=1}^m C_i, \Sigma, \mathcal{P})$, where $m \geq 1$, all the C_i's are components, the set of *priorities* $\mathcal{P} \subseteq 2^{\Sigma \times \Sigma}$ is irreflexive and transitive [5]. The notation $\sigma_1 \prec \sigma_2$ is usually used instead of $(\sigma_1, \sigma_2) \in \mathcal{P}$, and we say that σ_2 has higher priority than σ_1. A *configuration (or state)* c of a system $\mathcal{S}$ is of the form $(l_1, \ldots, l_m)$ with $l_i \in L_i$ for all $i \in \{1, \ldots, m\}$, and let $\mathcal{C}_\mathcal{S}$ be the set of all configurations. The *initial configuration* c_0 of $\mathcal{S}$ is of the form $(l_1^0, \ldots, l_m^0)$. An interaction $\sigma \in \Sigma$ is *(globally) enabled* in a configuration c if, first, *joint participation* holds for σ, that is, for all $\sigma \in \Sigma_i$ with $i \in \{1, \ldots, m\}$, there exists a transition $(l_i, \sigma, l_i') \in T_i$, and, second, there is no other interaction of higher priority for which joint participation holds. Σ_c denotes the set of (globally) enabled interactions in a configuration c. For $\sigma \in \Sigma_c$, a configuration c' of the form $(l_1', \ldots, l_m')$ is a *σ-successor* of c, denoted by $c \xrightarrow{\sigma} c'$, if, for all i in $\{1, \ldots, m\}$: If $\sigma \notin \Sigma_i$, then $l_i' = l_i$, otherwise (i.e., $\sigma \in \Sigma_i$) there exists transition of the form $(l_i, \sigma, l_i') \in T_i$.

A *trace* is of the form $c_0 \ldots c_k$ with c_0 the initial configuration and $c_j \xrightarrow{\sigma_{j+1}} c_{j+1}$ for all $j : 0 \leq j < k$. In this case, c_k is reachable, and $\mathcal{R}_\mathcal{S}$ denotes the set of all reachable configurations from c_0. The system is *deadlocked* in configuration c if there is no $c' \in \mathcal{R}_\mathcal{S}$ and no $\sigma \in \Sigma_c$ such that $c \xrightarrow{\sigma} c'$, and the set of deadlocked states is denoted by $\mathcal{C}_{dead}$. A configuration c is *safe* if, given a set of risk configurations $\mathcal{C}_{risk}$, $c \notin \mathcal{C}_{dead} \cup \mathcal{C}_{risk}$, and a system is safe if no reachable configuration is unsafe.

DEFINITION 1 (Priority Synthesis). *Given a system $\mathcal{S} = (C, \Sigma, \mathcal{P})$ together with a set $\mathcal{C}_{risk} \subseteq \mathcal{C}_\mathcal{S}$ of risk configurations, $\mathcal{P}_+ \subseteq \Sigma \times \Sigma$ is a solution to the* (centralized) priority synthesis problem *if the extended system $(C, \Sigma, \mathcal{P} \cup \mathcal{P}_+)$ is safe, and the defined relation of $\mathcal{P} \cup \mathcal{P}_+$ is also irreflexive and transitive.*

(Example in Figure 1) The system $\mathcal{S}$ has three components C_1, C_2, C_3, uses interactions $\Sigma = \{a, b, c, d, e, f\}$, and has no predefined priorities. The initial configuration is (s_0, s_3, s_8). Define the set of risk states to be $\{(s_2, s_5, s_8), (s_2, s_5, s_9)\}$. Then priority synthesis introduces the set $\{b \prec c, b \prec f\}$ to avoid deadlock and risk states. E.g., in state (s_1, s_5, s_8) (see Figure 2), interaction b is not enabled due to priority $b \prec c$, and the subsequent state can only be (s_1, s_6, s_9).

2.2 Communication Architecture and Knowledge

We now use the notion of communication architecture to define distributed execution. Intuitively, a communication architecture is defined as a set of communication paths such that intentions of executions are properly transmitted. Formally, a *communication architecture Com* for a system $\mathcal{S}$ of interacting components is a set of ordered pairs of components of the form (C_i, C_j) for $C_i, C_j \in C$. In this case we say that C_i *informs* C_j and we use the notation $C_i \rightsquigarrow C_j$. Furthermore, the communication architecture defines, for each $i, j \in \{1, \ldots, m\}$, a function $\mathsf{avail}_i^j : L_i \rightarrow 2^{\Sigma_j}$ such that given a configuration $(l_1, \ldots, l_m)$: If $C_j \rightsquigarrow C_i$ then $\mathsf{avail}_i^j(l_i) = \{\sigma \mid \exists t = (l_j, \sigma, l_j')$ such that $t \in T_j\}$. Else $\mathsf{avail}_i^j(l_i) = \emptyset$. Such

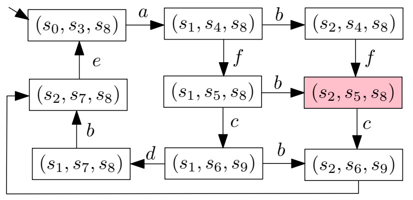


Figure 2. The state transition graph (reachable from the initial state) of Figure 1.

a communication architecture Com is *deployable* if the following conditions hold for all $\sigma, \tau \in \Sigma$ and $i, j \in \{1, \ldots, m\}$:

- *(Self-transmission)* $C_i \rightsquigarrow C_i \in Com$.
- *(Group transmission)* If $\sigma \in \Sigma_i \cap \Sigma_j$ then $C_j \rightsquigarrow C_i$, $C_i \rightsquigarrow C_j \in Com$.
- *(Priority transmission)* If $\sigma \prec \tau \in \mathcal{P}$, $\sigma \in \Sigma_j$, and $\tau \in \Sigma_i$ then $C_i \rightsquigarrow C_j \in Com$.

Therefore, components that possibly participate in a joint interaction exchange information about next intended moves (group transmission), and components with a high priority interaction τ need to inform all components with an interaction of lower priority than τ (priority transmission). In this paper, we make an explicit assumption that a system under synthesis is deployable on the given communication architecture.

(Example in Figure 1) The communication architecture Com in Figure 1 is $\{C_1 \rightsquigarrow C_1, C_2 \rightsquigarrow C_2, C_3 \rightsquigarrow C_3, C_1 \rightsquigarrow C_2, C_2 \rightsquigarrow C_1, C_2 \rightsquigarrow C_3, C_3 \rightsquigarrow C_2\}$, which disallows C_1 and C_3 to communicate to each other. At configuration (s_1, s_4, s_8), we have $\mathsf{avail}_1^2(s_1) = \{f\}$ (later we write it as $\{C_2.f\}$ to clarify that such an information is obtained from C_2) and $\mathsf{avail}_1^3(s_1) = \emptyset$. Intuitively, the avail function indicates that C_1 is able to see what are the possible moves of C_2 but not C_3. Concerning deployability, the original system $\mathcal{S}$ under Com is deployable, as it satisfies self-transmission and group transmission. However, the modified system which includes the synthesized priorities $\{b \prec c, b \prec f\}$ is not deployable, as it requires $C_3 \rightsquigarrow C_1$ to support the use of priority.

Given a communication architecture Com for a system $\mathcal{S}$, an interaction σ is *visible* by C_j if $\forall i$ where $\sigma \in \Sigma_i$, $C_i \rightsquigarrow C_j$ (such information can be computed statically). Now we present the notion of knowledge of every component, which is used to define its partial view on enabled interactions in the system. Here for simplicity we restrict the use of knowledge to be *history-unaware*, i.e., the knowledge helps a local controller to decide the global information based on (1) its current control location and (2) the communicated intention from other components.

DEFINITION 2 (Knowledge). *Consider a system $\mathcal{S} = (C, \Sigma, \mathcal{P})$, for component $C_i = (L_i, \Sigma_i, T_i, l_i^0)$, define two functions $\mathcal{K}_i^{\text{STR}}$ (strong knowledge), $\mathcal{K}_i^{\text{WK}}$ (weak knowledge) which map from $\Sigma_i \times L_i \times \bigwedge_{j=1}^m 2^{\Sigma_j}$ to 2^Σ,*

- *strong knowledge: $\forall \tau \in \Sigma^{\text{STR}} = \mathcal{K}_i^{\text{STR}}(\sigma, l_i, \bigwedge_{j=1}^m \mathsf{avail}_i^j(l_i))$, joint participation holds for τ in* ***all states*** *of $S = \{(l_1, \ldots, l_i, \ldots, l_k, \ldots, l_m)\} \subseteq \mathcal{R}_\mathcal{S}$ where $\forall k \in \{1, \ldots, m\}$: if $\sigma_k \in \mathsf{avail}_i^k(l_i)$ then $(l_k, \sigma_k, \_) \in T_k$.*
- *weak knowledge: $\forall \tau \in \Sigma^{\text{WK}} = \mathcal{K}_i^{\text{WK}}(\sigma, l_i, \bigwedge_{j=1}^m \mathsf{avail}_i^j(l_i))$, joint participation holds for τ in* ***at least one state*** *of $S = \{(l_1, \ldots, l_i, \ldots, l_k, \ldots, l_m)\} \subseteq \mathcal{R}_\mathcal{S}$ where $\forall k \in \{1, \ldots, m\}$: if $\sigma_k \in \mathsf{avail}_i^k(l_i)$ then $(l_k, \sigma_k, \_) \in T_k$.*
- *$\Sigma^{\text{STR}} \cap \{\sigma\} = \Sigma^{\text{WK}} \cap \{\sigma\} = \Sigma^{\text{STR}} \cap \Sigma^{\text{WK}} = \emptyset$. No element in Σ^{STR} or Σ^{WK} is visible by C_i.*

Intuitively, the above definition is used for component C_i to understand whether there exists another available interaction τ when C_i intends to execute σ: Based on the global execution semantics, if there exists predefined priorities of the form $\sigma \prec \tau$, σ shall be suspended when realizing the knowledge. The knowledge is computed according to the available interactions from the current view of a component and the filtered locations that can fire transitions labeled by these interactions. Strong knowledge provides a guarantee that *every* interaction from the returned set satisfies joint participation, and weak knowledge only ensures that *at least one* of them satisfies joint interaction. We perform explicit partition on the strong and weak knowledge, and differentiate knowledge from what can be summarized from the observation (no element in $\Sigma^{\mathtt{STR}}$ or $\Sigma^{\mathtt{WK}}$ shall be visible by C_i).

(Example in Figure 1) As stated previously, for priorities $\{b \prec c, b \prec f\}$, $b \prec c$ can not be deployed. However, starting from the initial configuration, the system first proceeds with interaction a. Then for component C_1 at s_1, when it intends to execute b:

- As $C_2 \rightsquigarrow C_1$ on its intentions, C_1 is only unaware on the situation of C_3.
- It can be observed from the reachable states that when C_2 raises its intention on c, C_3 also raises its intention on c. The similar case also appears for d. Therefore, we can derive the following strong memoryless knowledge $\mathcal{K}_1^{\mathtt{STR}}(b, s_1, \{C_1.b\}, \{C_2.d\}, \emptyset) = \{d\}$, $\mathcal{K}_1^{\mathtt{STR}}(b, s_1, \{C_1.b\}, \{C_2.c\}, \emptyset) = \{c\}$. Notice that $\mathcal{K}_1^{\mathtt{STR}}(b, s_1, \{C_1.b\}, \{C_2.f\}, \emptyset) = \emptyset$. It is not $\{f\}$, as f is visible by C_1 due to the communication path $C_2 \rightsquigarrow C_1$.

Computing Knowledge Statically We explain how knowledge can be computed statically, following the definition. For intuition, we use the example in Figure 1. The base step is to compute the set of all reachable states, i.e., our knowledge is derived from the invariant of the state space. Figure 2 shows the corresponding state-transition graphs. Consider we want to derive $\mathcal{K}_1^{\mathtt{STR}}(b, s_1, \{C_1.b\}, \{C_2.d\}, \emptyset)$, i.e., the knowledge of C_1 at location s_1 with C_2 informing its availability on d.

- The first step is to collect all possible control locations of C_2 that can signal d. This is done by checking component C_2. In this example, we derive the set $\{s_6\}$.
- Then perform an intersection with the set of all reachable states whose control location of C_1 is s_1 and control location of C_2 is within set $\{s_6\}$ (derived from the first step). In this example, we produce the set $S = \{(s_1, s_6, s_9)\}$.
- By checking the enabled interactions at $\{(s_1, s_6, s_9)\}$, we derive that, other than b, d is guaranteed to be available. Overall, if an interaction σ appears in all states in S, σ can be placed in the strong knowledge. Otherwise, the set of all interactions that are available in S constitutes the weak knowledge.

Algorithm 1 describes the above procedure. Line 1 and 2 perform filtering of states, and line 3 adds an interaction τ to the strong knowledge only when the enableness of τ is an invariant of know. Recall that knowledge is used to detect whether another interaction is possible to be enabled. Therefore, if the condition in line 4 holds, we can not guarantee that within weak, at least one interaction is enabled to block σ. In this case, the previously computed weak knowledge is not valid and shall be replaced by $\emptyset$ (line 5). The method can be implemented symbolically using binary decision diagrams [3], enabling efficient generation of knowledge.

2.3 Distributed Execution

In the following, we define distributed notions of enabled interactions and behaviors based on knowledge.

DEFINITION 3. *Given a system $\mathcal{S} = (C, \Sigma, \mathcal{P})$ under communication architecture Com, for configuration $c = (l_1, \ldots, l_m)$, an interaction $\sigma \in \Sigma$ is* distributively-enabled (at c) *if ($i \in \{1, \ldots, m\}$):*

- *(Joint participation: distributed version) $\forall i$ with $\sigma \in \Sigma_i$, σ is visible by C_i, there exists $(l_i, \sigma, \_) \in T_i$.*
- *(No higher priorities enabled: distributed version) for all $\tau \in \Sigma$ with $\sigma \prec \tau$, τ is visible by C_i, and there is a $j \in \{1, \ldots, m\}$ such that $\tau \in \Sigma_j$ and $(l_j, \tau, \_) \notin T_j$.*
- *(No higher priorities enabled: strong knowledge version) $\forall \tau \in \Sigma$ with $\sigma \prec \tau \in \mathcal{P}$: $\tau \notin \mathcal{K}_i^{\mathtt{STR}}(\sigma, l_i, \bigwedge_{j=1}^{m} \mathsf{avail}_i^j(l_i))$.*
- *(No higher priorities enabled: weak knowledge version) $\forall \tau \in \Sigma$ with $\sigma \prec \tau \in \mathcal{P}$: $\tau \notin \mathcal{K}_i^{\mathtt{WK}}(\sigma, l_i, \bigwedge_{j=1}^{m} \mathsf{avail}_i^j(l_i))$.*

Algorithm 1: Knowledge generation (sketch)

```
input : System $\mathcal{S} = (C, \Sigma, \mathcal{P})$, reachable states $\mathcal{R}_{\mathcal{S}}$, query
        parameter $(\sigma, l_i, \bigwedge_{j=1}^{m} \Sigma_{ij})$ where $\Sigma_{ij} \subseteq \Sigma_j$
output: $\langle \mathcal{K}_i^{\mathtt{STR}}(\sigma, l_i, \bigwedge_{j=1}^{m} \Sigma_{ij}), \mathcal{K}_i^{\mathtt{WK}}(\sigma, l_i, \bigwedge_{j=1}^{m} \Sigma_{ij}) \rangle$
begin
      let strong := $\emptyset$, weak := $\emptyset$
1     let know := $\mathcal{R}_{\mathcal{S}} \cap \{s = (l'_1, \ldots, l'_m) | s \in \mathcal{R}_{\mathcal{S}}, l'_i = l_i\}$
      for $j = 1 \ldots m$ do
          let $L'_j \subseteq L_j$ be locations where all $\sigma_j \in \Sigma_{ij}$ is locally
          enabled at locations in $L'_j$
2         know := know $\cap \{s = (l'_1, \ldots, l'_m) | s \in \mathcal{R}_{\mathcal{S}}, l'_j \in L'_j\}$
      for $\tau \in \Sigma, \tau \neq \sigma$ and $\tau \notin \{\kappa | \kappa$ is visible by $C_i\}$ do
          if $\tau$ is globally enabled in every state of know then
3             str := strong $\cup \{\tau\}$
          else if $\tau$ is globally enabled in one state of know then
              weak := weak $\cup \{\tau\}$
4     if exists a state of know where no interaction or only
      interactions in str $\cup \{\sigma\} \cup \{\kappa | \kappa$ is visible by $C_i\}$ is enabled
      then
5         return $\langle$str, $\emptyset\rangle$
6     else return $\langle$strong, weak$\rangle$
```

PROPOSITION 1. *Consider a system $\mathcal{S} = (C, \Sigma, \mathcal{P})$ under a deployable communication architecture Com.* (a) *If $\sigma \in \Sigma$ is globally enabled at configuration c, then σ is distributively-enabled at c.* (b) *The set of distributively-enabled interactions at configuration c equals Σ_c.* (c) *If configuration c has no distributively-enabled interaction, it has no globally enabled interaction.*

A configuration $c' = (l'_1, \ldots, l'_m)$ is a *distributed σ-successor* of c if σ is distributively-enabled and c' is a σ-successor of c. *D*istributed runs are runs of system $\mathcal{S}$ under communication architecture Com. Any move from a configuration to a successor configuration in the distributed semantics can be understood as $|C|$ processes, where each local controller Ctrl_i works on component C_i. In contrast to the global semantics, Ctrl_i now is only informed on the intended next moves of the components defined by the knowledge and the communication architecture (notice: such a knowledge has no relation with the visible region as defined by the communication architecture). The semantics of distributed execution is that local controllers agree (cmp. Assumption 1 below) on an interaction $\sigma \in \Sigma_c$ and perform a joint move.

ASSUMPTION 1 (Runtime Assumption). *For a configuration c with $|\Sigma_c| > 0$, the distributed controllers Ctrl_i agree on a distributively-enabled interaction $\sigma \in \Sigma_c$ for execution.*

With the above assumption, we then define, given a system $\mathcal{S} = (C, \Sigma, \mathcal{P})$ under a communication architecture Com, the set of deadlock states of $\mathcal{S}$ in distributed execution to be $\mathcal{C}_{dist.dead} = \{c\}$ where no interaction is distributively-enabled at c. We immediately derive $\mathcal{C}_{dist.dead} = \mathcal{C}_{dead}$, as the left inclusion ($\mathcal{C}_{dist.dead} \subseteq \mathcal{C}_{dead}$) is the consequence of Proposition 1, and the right inclusion is trivially true. With such an equality, given a risk configuration $\mathcal{C}_{risk}$ and global deadlock states $\mathcal{C}_{dead}$, we say that system S under the distributed semantics is *distributively-safe* if there is no

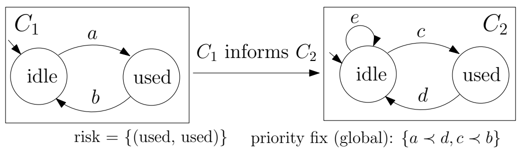


Figure 3. Another example with uni-directional communication.

distributed run $c_0, \ldots, c_k$ such that $c_k \in \mathcal{C}_{dead} \cup \mathcal{C}_{risk}$; a system that is not safe is called *distributively-unsafe*.

2.4 Knowledge-based Distributed Priority Synthesis

We define distributed priority synthesis using knowledge.

DEFINITION 4 (Knowledge-based Distributed Priority Synthesis). *Given a system $\mathcal{S} = (C, \Sigma, \mathcal{P})$ together with a deployable communication architecture Com, the set of risk configurations $\mathcal{C}_{risk} \subseteq \mathcal{C}_{\mathcal{S}}$ and knowledge functions $\mathcal{K}_1^{\text{STR}}, \mathcal{K}_1^{\text{WK}} \ldots, \mathcal{K}_m^{\text{STR}}, \mathcal{K}_m^{\text{WK}}$, a set of priorities $\mathcal{P}_{d+}$ is a solution of the* knowledge-based distributed priority synthesis (KDPS) *problem if the following holds:*

1. *$\mathcal{P} \cup \mathcal{P}_{d+}$ is transitive and irreflexive.*
2. *$(C, \Sigma, \mathcal{P} \cup \mathcal{P}_{d+})$ is distributively-safe.*
3. *For all $i, j \in \{1, \ldots, m\}$ s.t. $\sigma \in \Sigma_i$, $\tau \in \Sigma_j$, if $\sigma \prec \tau \in \mathcal{P} \cup \mathcal{P}_{d+}$ then*
   - *(a) (Priority by communication) $C_j \rightsquigarrow C_i \in Com$, or*
   - *(b) (Priority by knowledge) if $C_j \not\rightsquigarrow C_i$, and C_i is at location l_i, then*
     - *either $\tau \in \mathcal{K}_i^{\text{STR}}(\sigma, l_i, \bigwedge_{j=1}^{m} \mathsf{avail}_i^j(l_i))$,*
     - *or $\tau \in \Sigma^{\text{WK}} = \mathcal{K}_i^{\text{WK}}(\sigma, l_i, \bigwedge_{j=1}^{m} \mathsf{avail}_i^j(l_i))$, and $\forall \gamma \in \Sigma^{\text{WK}}$, $\sigma \prec \gamma \in \mathcal{P} \cup \mathcal{P}_{d+}$.*

We call $\mathcal{P}_{d+}$ a solution of the (normal) distributed priority synthesis (DPS) *problem when 3.(b) can not be used.*

The 3rd condition in KDPS states that every newly introduced priority $\sigma \prec \tau$ is either deployable, or the derived knowledge from C_i is sufficient to assist blocking σ under the case where τ is enabled. As overall for weak knowledge, Σ^{WK} only guarantees that one of the interaction is enabled, to successfully block σ, we have to conservatively include every $\sigma \prec \gamma$, where $\gamma \in \Sigma^{\text{WK}}$, to $\mathcal{P} \cup \mathcal{P}_{d+}$, if components executing γ is also unable to inform its intention to components executing σ. If only $\sigma \prec \tau$ is included, then the knowledge Σ^{WK} is unable to give definite answer concerning the availability of τ. From this C_i needs information from C_j (which is impossible due to architectural constraints), making the newly synthesized system $(C, \Sigma, \mathcal{P} \cup \mathcal{P}_{d+})$ undeployable. For DPS, the last condition states that knowledge only faithfully reflects the passed intention from the communication architecture.

(Example in Figure 1) We can observe that using the previously stated strong knowledge is sufficient to block the execution of b when c is distributively enabled, as the enableness of c locally in C_2 can be used to deduce the enableness of c in global sense.

(Example in Figure 3) Consider the scenario in Figure 3, where no communication exists from C_2 to C_1. Therefore, C_1 only processes a weak knowledge $\{c, d, e\}$. Admittedly, $\{a \prec d, c \prec b\}$ is a solution for global priority synthesis, but the weak knowledge is not informative enough to support distributed execution (C_1 can freely execute a if $\{c, e\}$ is enabled, but not when d is enabled). Then during execution, C_1 needs to acquire information on C_2, violating the communication architecture. When we additionally add $a \prec c$ and $a \prec e$ (following KDPS 3.(b)-ii), as at least one of the interaction is distributively enabled, at least one priority is activated to block b.

In the following, we prove that when distributively executing a system with new priorities from KDPS, the system under execution has the same *trace behavior* as the system under centralized execution. By doing so, a solution under distributed priority synthesis is also a solution of centralized priority synthesis.

PROPOSITION 2. *Given a system $\mathcal{S} = (C, \Sigma, \mathcal{P})$ together with a deployable communication architecture Com, the set of risk configurations $\mathcal{C}_{risk} \subseteq \mathcal{C}_{\mathcal{S}}$ and knowledge functions $\mathcal{K}_1^{\text{STR}}, \mathcal{K}_1^{\text{WK}} \ldots$, $\mathcal{K}_m^{\text{STR}}, \mathcal{K}_m^{\text{WK}}$. Let the set of priorities $\mathcal{P}_{d+}$ be a solution to KDPS. Then for $(C, \Sigma, \mathcal{P} \cup \mathcal{P}_{d+})$, the set of traces for centralized execution is the same as that of distributed execution using knowledge.*

Our second conclusion is that KDPS is more powerful than DPS.

PROPOSITION 3. *KDPS is strictly more powerful than DPS, i.e., (i) for every system $\mathcal{S}$ under architecture Com, every solution from DPS is a solution to KDPS, while (ii) there exists a system under a communication architecture, such that only KDPS returns a solution.*

3. Concluding Remarks

The contributions of this work are three-fold: (1) we have defined a history-unaware knowledge and its integration to the synthesis process to create priority-based distributed controllers (the KDPS process). The key ingredient relies on the fact that the knowledge guarantees the presence of actions. (2) We have presented an algorithm to compute such knowledge to be used in synthesis. (3) We have proven properties of the integrated synthesis process, such as the preservation of global semantics and its superiority over DPS.

In [4], we present algorithms and tools for DPS, together with an NP-completeness proof: an intuitive idea is to non-deterministically pick some priorities from all possible candidates, and check if (1) the set of priorities is transitive, irreflexive, and contains existing priorities, (2) all priorities are supported by the communication architecture, and (3) the system augmented with new priorities is safe. In [4] we further developed an algorithm that performs fault-localization (using game solving) and fault-fixing (using SAT solver). Due to space limit, we do not present details how to integrate knowledge into our previous work, but the integration is not difficult to achieve: Based on the statically computed knowledge, the synthesis algorithm can freely use the strong knowledge, but shall dynamically decide whether to use weak knowledge or not, as using a weak knowledge can raise an additional constraint on the synthesized artifact (KDPS 3.(b)-ii: $\forall \gamma \in \Sigma^{\text{WK}}$, $\sigma \prec \gamma \in \mathcal{P} \cup \mathcal{P}_{d+}$).

References

[1] A. Basu, M. Bozga, and J. Sifakis. Modeling heterogeneous real-time components in BIP. In *SEFM'06*, pages 3–12. IEEE, 2006.

[2] S. Bensalem, M. Bozga, S. Graf, D. Peled, and S. Quinton. Methods for knowledge based controlling of distributed systems. In *ATVA'10*, volume 6252 of *LNCS*, pages 52–66. Springer-Verlag, 2010.

[3] R. Bryant. Graph-based algorithms for boolean function manipulation. *Computers, IEEE Transactions on*, 100(8):677–691, 1986.

[4] C.-H. Cheng, S. Bensalem, R. Yan, H. Ruess, C. Buckl, and A. Knoll. Distributed Priority Synthesis and its Applications. *ArXiv e-prints (cs.LO 1112.1783)*, 2011.

[5] G. Gößler and J. Sifakis. Priority systems. In *FMCO'03*, volume 3188 of *LNCS*, pages 314–329. Springer-Verlag, 2003.

[6] G. Katz, D. Peled, and S. Schewe. The buck stops here: Order, chance, and coordination in distributed control. In *ATVA'11*, volume 6996 of *LNCS*, pages 422–431. LNCS, 2011.

[7] G. Katz, D. Peled, and S. Schewe. Synthesis of distributed control through knowledge accumulation. In *CAV'11*, volume 6806 of *LNCS*, pages 510–525. Springer-Verlag, 2011.

Parallel Gesture Recognition with Soft Real-Time Guarantees

Thierry Renaux Lode Hoste Stefan Marr Wolfgang De Meuter

Software Languages Lab
Vrije Universiteit Brussel, Belgium
{trenaux,lhoste,smarr,wdemeuter}@vub.ac.be

Abstract

Applying imperative programming techniques to process event streams, like those generated by multi-touch devices and full-body motion detection, has significant engineering drawbacks. Declarative approaches solve these problems but have not been able to scale on multicore systems while providing guaranteed response times.

We propose PARTE, a parallel scalable complex event processing engine which allows a declarative definition of patterns and provides soft real-time guarantees for their recognition. It extends the state-saving Rete algorithm and maps the event matching onto a graph of actor nodes. Using a tiered event matching model, PARTE provides upper bounds on the detection latency. Based on the domain-specific constraints, PARTE's design relies on a combination of lock-free data structures, safe memory management techniques, and message passing between Rete nodes. In our benchmarks, we measured scalability up to 8 cores, outperforming highly optimized sequential implementations.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent programming; D.3.4 [*Programming Techniques*]: Processors; I.5.5 [*Pattern Recognition*]: Implementation

General Terms Algorithms, Design, Performance

Keywords multimodal interaction, gesture recognition, Rete, actors, soft real-time guarantees, nonblocking, complex event processing, multicore

1. Introduction

To improve the quality of interactions between users and computers, interest in multi-touch input, gesture recognition, and speech processing on consumer hardware has recently emerged. To power natural user interfaces, primitive sensor readings, which are collected by devices for multimodal input, need to be correlated to create higher-level events.

Hard-coding these complex correlations in imperative programming languages is cumbersome, error-prone, and lacks flexibility [14]. On the other hand, the domain of machine learning requires a lot of training data to build a statistical model of the gesture. Gathering and manually annotating this data (in positive and negative examples) and additionally parameterising important features, is usually time intensive. Hammond and Davis [12], Scholliers et al. [20], and Hoste et al. [14] demonstrate that declarative definitions for sketch recognition, multi-touch gestures, or multimodal correlation have important benefits on multiple levels. Firstly, they provide important software engineering abstractions to help the programmer to express their intended event patterns. Secondly, they offer an alternative solution compared to ad-hoc implementations when training data is lacking or hard to gather. Finally, expert programmers are able to refine their event correlations with explicit programming code. These declarative approaches all require an inference engine, which compares sensor events with declarative rules describing the gestures.

The Rete algorithm [7] is one possible foundation for such inference engine. It is a forward-chaining, state-saving algorithm that is used to build rule-based expert systems. More concretely, declarative gesture approaches benefit from it as the execution engine incrementally interprets the events of various input sources based on predefined patterns, i. e., rules defining the possible interactions of a human with a computer. Since the majority of the information is constant, the Rete algorithm minimizes the necessary computation that has to be performed whenever a new fact is asserted to the knowledge base, and thus reduces the computational overhead of continuous pattern matching.

In a multimodal system with many possible interactions, the required computational power outgrows easily what today's processors provide in terms of sequential performance.

This is problematic for server-sided pattern recognition, for instance to process surveillance camera-input, as well as for embedded devices and mobile phones with various sensors such as an accelerometer, gyroscope, multi-touch, and a microphone. A wide range of applications has been proposed to utilize such sensors by extracting meaningful information from the raw data. Examples are discovering a phone drop, detecting whether the user is "throwing" data to another device[1], or performing multi-touch gestures to quickly access information[2]. Additionally, in certain multimodal applications, this information must be correlated to speech-based input. Fusing these primitive and higher-order events easily becomes excessive for a single processing unit, such that utilizing the steadily rising degree of available parallelism becomes a necessity to provide the required degree of interactivity to the users.

We present here a variation of the Rete algorithm called PARTE, built on a Rete network represented by a set of actors, that provides both scalability and responsiveness. The contributions of our work are:

Design and implementation of PARTE, a parallel Rete engine tailored towards recognition of user interaction patterns with soft real-time [3] guarantees.

Validation of PARTE's real-time guarantees by characterizing the execution properties of the implemented algorithm, ensuring freedom of unbounded loops and freedom of blocking concurrent interactions.

Validation of PARTE's practicality by showing the scalability of the parallel implementation and demonstrating that the sequential overhead compared to CLIPS[4], a highly optimized sequential implementation, is acceptable.

The remainder of this paper is structured as follows: First, we will discuss in section 2 in more detail the context of multimodal input systems, their requirements, and constraints and assumptions we can make for a solution. Then, in section 3, we will describe our solution PARTE in detail and discuss the parallel Rete algorithm used. Afterwards, we will evaluate the resulting system in section 4, characterizing its execution semantics both with respect to non-blocking behavior and with respect to unbounded loops, as well as presenting the performance evaluation. Finally, we will contrast our approach with the related work in section 5 and summarize our conclusions and future work in section 6.

2. Context and Requirements

The domain of gesture recognition comes with a set of properties that is different from many domains in which Rete-like inference engines are commonly used. Since we utilize these particularities of the problem domain in the design of PARTE, we will sketch the application domain briefly and detail a list of requirements for inference engines in this domain.

2.1 Inference Engines for Gesture Detection

To provide a high-quality user experience, an inference engine used for gesture recognition has to correlate events in a timely manner: When a user for instance interacts with a system through a multi-touch interface, changes should be reflected immediately and with a predictable delay to give the user a natural feedback. The same is true for multimodal interaction: When a user gives a series of voice and gesture commands, the right action should be performed without random delays that confuse the user about whether the command has been accepted or not.

Systems such as Mudra [14] embed inference engines which only tap the computational power of a single processing unit. However, the rise in sequential processing power offered by single processing units is stagnating, because efforts for instance to increase clock-speed, instruction-pipeline depth, memory-bus width, and cache size, offer diminishing returns. This severely limits the possible number of rules, their complexity, and the rate of events the system can handle. The only way to recognize more complex user interaction patterns without undermining the user experience by increased delays, is to embrace parallel processing power.

In addition to recognizing patterns in a timely manner, the system also needs to guarantee predictable response times. This ensures that the system feels interactive and responsive. Akscyn et al. [2] show that long delays in interactive systems can distract users, and even cause them to stop using the system altogether. Consider for instance a user of a multi-touch gesture recognition system, who taps a certain location. If the user interface does not reflect this change within the timeframe users have grown to expect, they will assume the command was not received, and may tap again. When the system then finishes processing the overdue gestures, the action will be executed twice. Users will rightfully blame the gesture detection system for this mistake. To prevent such errors, the detection of complex user interaction patterns should happen within a timeframe that can be predicted reliably up front.

However, the requirements of responsiveness and predictable runtime conflict: To offer the best performance on current hardware, the rule engine needs to use the available

[1] Hoccer, exchanging data using gestures.
Youtube: `http://www.youtube.com/watch?v=eqv8Q6M1O6Y`

[2] Gesture Search for Android
`http://www.google.com/mobile/gesture-search/`

[3] In *soft* real-time systems, the usefulness of results degrades past their deadline, while in *hard* real-time systems the usefulness drops to zero on a missed deadline. Hence, delays in a soft real-time system undermine the system's quality of service, where delays in hard real-time systems undermine the system's correctness

[4] *CLIPS: A Tool for Building Expert Systems*, Gary Riley, 13 March 2011
`http://clipsrules.sourceforge.net/`

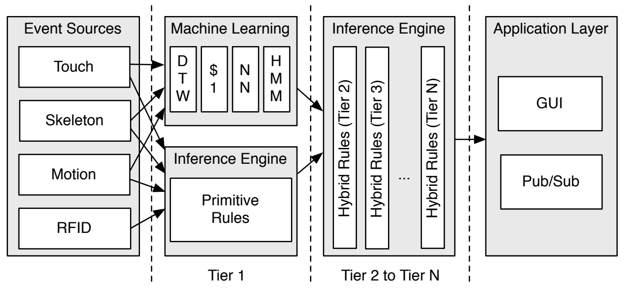


Figure 1. Contextual Framework

parallelism, and should provide soft real-time guarantees to ensure responsiveness. Current rule engines do not combine both requirements. They either are single-threaded in nature, or do not guarantee predictable worst-case execution times.

To give an example, Figure 1 visualizes the data-flow of the multimodal approach presented by Hoste et al. [14]. Event sources such as multi-touch displays, skeletal tracking [21], accelerometer readings or the history of RFID tags contain potential valuable patterns that need to be processed. This unified architecture allows for low-level events to be processed using machine learning-based approaches, as well as declarative definitions. Fusion and refinement of resulting higher-level events can be handled by consecutive declarative rules.

Given these observations, we will use a tiered architecture for event processing. In this architecture, rules of $tier\ N$ can consume only events that were generated by lower-level tiers (1 to $N-1$). It enables developers to easily modularize and compose their rules. For instance, a *wave* gesture can be composed of two lower-level gestures *flick right* and *flick left*, which themselves where extracted from the low-level skeletal data provided by a Kinect system. A declarative approach enables this kind of efficient composition and helps developers to improve gesture recognition code. Enforcing tiering however, implies that a rule of $tier\ N$ should not insert additional lower-level events to help pattern classification on lower levels. Although certain multimodal use-cases benefit from using high-level event information to improve the accuracy of lower-level event detection [14], avoiding feedback loops is required for computational predictability.

Finally, the application layer uses a publish-subscribe model to register for high-level events processed by the inference engine. Depending on the application, it is useful to support different subscription modi. Topic-based subscriptions are used to filter by the type of the event and are the most common ones. However, for instance GUI components use content-based subscriptions to only react on events that happen at a specific spatial location. To enable such application-specific usage, the system needs to provide the necessary extensibility.

This contextual framework is tailored to the domain of multimodal interaction and gesture recognition and guided the design of PARTE. However, similar properties can be found in the broader context of *Time Series Analysis* and *Complex Event Processing*, including domains such as algorithmic stock trading and monitoring security breaches.

2.2 Requirements and Assumptions

By restricting the generality of the Rete algorithm and tailoring it to our application domain, we can make design decisions that simplify the implementation and enable us to achieve the desired properties.

The main target for the system will be commodity multicore hardware. Thus, we will assume shared memory between cores and the presence of a cache hierarchy with memory coherency guarantees.

An important requirement to achieve bounded execution time is that rules are constructed without feedback loops. Based on the practice of tiering, which we outlined in subsection 2.1, we will disallow direct feedback loops and assume that the results always represent higher-level events for a higher tier in the system. Those events may be asserted back into the same inference engine, but will activate a disjoint subgraph of the Rete network.

Since the ordering of events is an application specific issue, it needs to be handled explicitly as part of the rules. A higher-level event might require the timestamp of the first low-level event in a sequence, the last one, or the time span in which all the related lower-level events occurred. The choice of this timestamp or time span depends on the semantics of the declarative rule, so this choice cannot be automated. Such information therefore needs to be constructed and provided to the next tier explicitly, if temporal order between higher-level events needs to be known.

Related to this assumption is our interpretation of the semantics of events as being permanent. Thus, for the intended use case, it is not necessary to enable retraction of facts, i. e., events will not be removed from the system as part of the action of a rule. Instead, we assume that a higher level rule can always subsume events if necessary. This enables us to avoid the need for *conflict resolution*: Conflict resolution is commonly used in rule-based systems to order the execution of rule activations, and enable retraction of facts and subsumption of rule activations. For the intended use case, however, it is desirable that all rules will always be triggered and subsumption is deferred to a higher-level tier. The ordering hence does not determine the result, and conflict resolution serves no purpose.

The semantically indefinite validity of events entails that the data structures representing events are not removed from working memory by the rules themselves, hence must be removed automatically by the system to prevent the working memory from growing unboundedly. For this, a sliding window of events which are relevant to the current reasoning process can be used. We will require events to be correlated by timestamp, so that the temporal dependencies between events can be used to compute their maximum useful lifespan: At any point in time, only those events can be part of a new pattern, for which there exist rules correlating them with other events that occurred within the lifespan of the first event.

Classic rule-based engines are employed in business environments in long-running systems that need to be adaptable and allow changes to the rule set at runtime to avoid downtime. However, this leads to additional complexity and is not required for the given scenario. Thus, we assume that in games and user interface applications only static sets of rules are used and that it is sufficient to determine the set of rules at startup time.

A final requirement is that all event sources have an upper bound on the rate with which they emit events. This upper bound is necessary to enable an estimation of the maximal load of the system.

Summarized, the important assumptions are:

- Rules are free of feedback cycles and produce results for a higher-level tier only.
- Activations of different rules do not require ordering.
- Temporal dependencies are solved in an application specific way by the rules.
- Events never need to be retracted from the system. Event subsumption is done on a higher tier. Preventing memory leaks is handled by making events expire when they are no longer useful for the reasoning process.
- The set of rules is known and fixed at startup time.
- All event sources have an upper bound on the rate with which they emit events.

Based on these assumptions and the previously given context, the requirements for a parallel gesture recognition engine are the following:

Soft Real-Time Guarantees The detection of user-interaction patterns has to complete in a predictable amount of time to give the user appropriate feedback.

Efficiency Beside providing predictability, the rule processing has to achieve sufficient efficiency to satisfy constraints on the response time required for interaction with humans. Miller [17] identified three threshold levels in human attention, based on the order of magnitude of seconds that one has to wait. Response times in the order of tenths of seconds are perceived as instantaneous and response times of around one second are perceived as a fluent interaction. For a system detecting user-interaction patterns, the interaction should at the very least be fluent, and preferably instantaneous, i. e., in the sub-second range.

Scalability on Multicore Hardware The performance of the system needs to improve with an increase in the number of available processing cores, relying on a shared-memory architecture.

Optimized for Continuous Event Streams The production system has to be tailored for complex event processing on event streams with a bounded event rate. The event streams are assumed to be infinite and processing has to be online (in contrast to off-line batch processing systems).

Extensibility and Embeddability The system needs to support user-defined functions to process and correlate events, and to produce results on rule activation to be extensible. Domain specific tests are required to facilitate for instance testing of spatial properties of coordinates. For embedding into existing systems, it is necessary to produce the expected result format by invoking callbacks or sending messages to the consuming tier.

3. PARTE

PARTE is a production system using a variant of the Rete algorithm to detect user-interaction patterns. To that end, it transforms a set of *if-then* rules into a directed acyclic graph, and uses this graph to match facts. PARTE is designed to be scalable on parallel systems, as well as to satisfy the requirements and assumptions described in subsection 2.2.

This section first describes how the solution is embedded into the context of gesture recognition, and how it interacts with the main components in such an environment. Then, the architecture of PARTE is described and a high-level overview over the solution strategy is given. Finally, we detail the solution and discuss implementation decisions that are essential to satisfy the posed requirements.

3.1 Architecture and Embedding into Gesture Recognition Context

As outlined in subsection 2.1, inference engines such as PARTE are mediating between the raw input devices and high-level consumers such as an application. For engineering reasons, such systems use tiered architectures to gradually enrich the semantics of the events. PARTE can be applied at multiple tiers in such an architecture. In such a scenario, PARTE would process incoming lower-level events from the a set of event sources, based on a given set of rules which describe relevant patterns that need to be recognized in these event streams. Thus, PARTE is part of the middleware for building such applications.

Figure 2 depicts the architecture of PARTE with potential input sources on the left, and potential consumers at the right hand side. Rules, the templates describing the facts' structure, and the facts themselves are to be encoded as S-expressions and get converted to the internal representation by a set of parsers. A pool of threads is maintained by the Rete engine, as well as a task queue on which the actors are scheduled. Finally, reentrant evaluator functions are provided to evaluate the test-expressions specified by the rules.

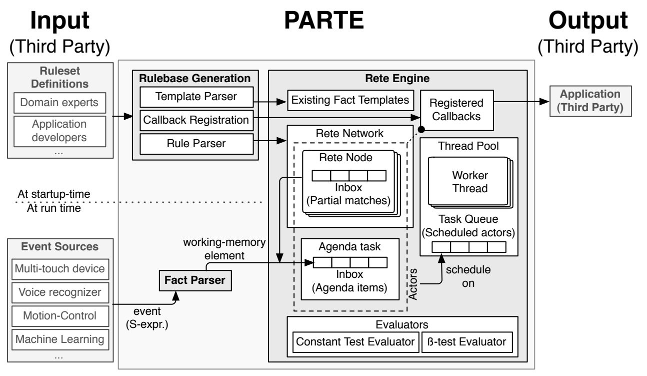


Figure 2. The architecture of PARTE

```
(defrule detectZShape
   ?hA <- (horizontal-drag)
   ?hB <- (horizontal-drag)
   ?diagonal <- (down-left-drag)
   (test (endMeetsStart ?hA ?diagonal))
   (test (endMeetsStart ?diagonal ?hB))
   (test (chronologically
                ?hA ?diagonal ?hB))
=> (reportZShapeCenteredOn
       (avg ?hA.startX ?hA.endX
            ?hB.startX ?hB.endX)
       (avg ?hA.y ?hB.y)))
```

Listing 1. A possible rule for gesture recognition

The Rete network itself is constructed from the set of rules inserted into PARTE at startup time. The rule parser converts the rules to an abstract syntax tree, validating their semantics while doing so. From the AST, it then builds the directed acyclic graph as prescribed by the Rete algorithm. When multiple nodes are required which select for the same type of event, the parser reuses the first node selecting for that type it created. More involved node-reuse and optimization strategies are not yet implemented. After computing the Rete graph, PARTE computes the indices ('lexical addresses') of slots within facts and of facts within partial matches such that, once the system is running, their lookup can be replaced with a constant-time indexed memory access. Finally, PARTE creates a set of actors and links them up to each other to constitute the Rete network.

Apart from the nodes in the Rete network themselves, the Rete algorithm requires another data structure: the agenda. This agenda reifies the FIFO queue of I/O actions to and from PARTE that have to be performed. The only form of input which PARTE accepts at runtime comes in the form of the assertion of new facts, for which *assert* agenda items are used. When the agenda task processes an *assert* agenda item, the event specified in that item is propagated to the inboxes of the Rete network's entry nodes. Inversely, to communicate results to the outside world, PARTE supports user-defined functions, which result in the scheduling of *user function* agenda items on the agenda. When such an item is processed, the corresponding callback function is called.

Since user-defined functions are plain C functions, they *can* technically perform whatever I/O or other time-consuming and/or blocking operation they want, but *are presumed* not to do so. If a rule should require something which is not normally considered a good match to event-processing, such as reading from a file, the user-defined function should dispatch the job of reading the file to a worker thread provided by the application hosting the PARTE engine, in a non-blocking way. That thread can then read the file and assert an event into the systems with the contents of the file.

In addition to the *assert* and *user function* agenda items, PARTE currently supports *print* and *terminate* agenda items which respectively print a string to the console and halt the engine, and could technically have been implemented in terms of *user function* agenda items. PARTE does not support *retract* agenda items, since in our event processing context, facts get removed from the fact base automatically when they expire.

Items on the agenda are processed sequentially, but in parallel with the processing of network nodes. Because of its interaction model, the agenda can be represented by an actor as well.

The S-expression in Listing 1 gives an example of a high-level motion gesture rule that can be processed by PARTE. The expression defines the rule `detectZShape`, which describes how two horizontal drags and a down-left drag can combine into a Z-shape. Line by line, the rule binds two events of type `horizontal-drag` to the variables `?hA` and `?hB`, binds an event of type `down-left-drag` to the variable `?diagonal`, and uses the user-defined functions `endMeetsStart` and `chronologically` to verify that the shapes follow each other both spatially and temporally. As a consequent to the detection of the Z-shape, the rule specifies that the callback `reportZShapeCenteredOn` should be called, passing the average `x` and `y` slots of the shape's points. Figure 3 shows a Rete graph and the flow of the facts along the edges when the facts at the lower-left corner are asserted into the Rete network. Lists delimited by square brackets denote tokens, the units of communication between the nodes in the network.

3.2 Parallel Execution Model

For the description of the execution model, we will use the metaphor of the actor model [1] to map each node of the Rete network to its own actor, executing independently. While such an approach does not enable us to utilize potential data parallelism for the matching inside a node, it enables a high degree of parallelism throughout the network. Even in situations where only parts of an actor network are used fre-

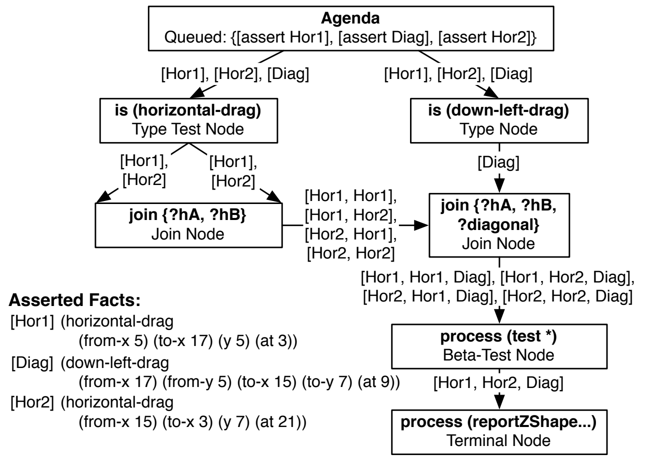

Agenda
Queued: {[assert Hor1], [assert Diag], [assert Hor2]}
[Hor1], [Hor2], [Diag]
[Hor1], [Hor2], [Diag]
is (horizontal-drag)
Type Test Node
is (down-left-drag)
Type Node
[Hor1], [Hor2]
[Hor1], [Hor2]
[Diag]
join {?hA, ?hB}
Join Node
[Hor1, Hor1], [Hor1, Hor2], [Hor2, Hor1], [Hor2, Hor2]
join {?hA, ?hB, ?diagonal}
Join Node
[Hor1, Hor1, Diag], [Hor1, Hor2, Diag], [Hor2, Hor1, Diag], [Hor2, Hor2, Diag]
Asserted Facts:
[Hor1] (horizontal-drag (from-x 5) (to-x 17) (y 5) (at 3))
[Diag] (down-left-drag (from-x 17) (from-y 5) (to-x 15) (to-y 7) (at 9))
[Hor2] (horizontal-drag (from-x 15) (to-x 3) (y 7) (at 21))
process (test *)
Beta-Test Node
[Hor1, Hor2, Diag]
process (reportZShape...)
Terminal Node

Figure 3. Flow of data through the Rete network specified by the rule in Listing 1

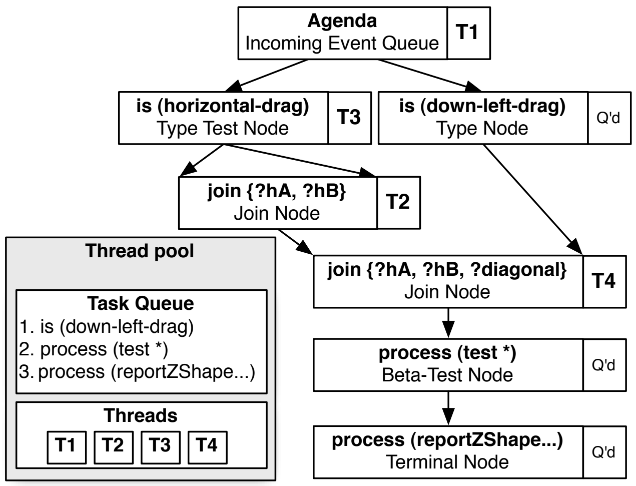


Figure 4. A potential runtime snapshot of a PARTE system detecting the pattern specified in Listing 1

quently, this approach enables pipeline parallelism, enabling scaling on multicore processors.

Furthermore, the directed acyclic graph (DAG) structure of a Rete network, and its structural properties in terms of edges between nodes provide a ideal foundation to apply non-blocking data structures to gain predictable upper bounds for execution time. We utilize these characteristics to provide the desired real-time properties.

Execution Model As indicated above, the individual actor nodes of the Rete network are the parallel units of computation. The DAG of the Rete network thereby forms a task-dependency-graph for the match phase of the fact processing. This means that every actor node needs only wait for information from their predecessors in the Rete graph, and only send data to their successors in the Rete graph. This entails that the same spatial and temporal efficiency that the Rete algorithm offers for matching facts to rules, also ensures low contention for shared resources: Every node's communication channel is contended for by at most two predecessors and its own thread of control.

The Rete algorithm's approach of passing tokens between nodes makes it map very well on message-passing between actors. Especially, since the step of processing an incoming token can be seen as an atomic operation by the rest of the system that does not require the notion of shared memory. In the implementation of PARTE, the nodes of the Rete network have an inbox, which is realized with a nonblocking queue, in which predecessors put the incoming tokens. A node dequeues tokens from its inbox one-by-one for processing. Because of the nonblocking nature of the inboxes, we avoid the potential for deadlocks and livelocks.

To prevent starvation, every actor, i. e., every Rete node and the agenda, are scheduled on a thread pool using a round-robin scheduler. Figure 4 shows a possible snapshot of a running PARTE system which contains only the rule specified in Listing 1. Four threads are allocated in the thread pool, meaning four of the seven actors can be active at the same time. The other three actors remain queued on the task-queue.

Non-Blocking Data Structures The only form of inter-thread communication on which PARTE depends comes in the form of messages sent to actors' inboxes. Since some nodes and the agenda have multiple predecessors in the Rete graph, we chose the n-producer/m-consumer FIFO queue design by Harris [13]. The list offers lock-free inserts at the front and deletes at the back, and contention is localized to only the element that is inserted or removed, meaning that in lists with two elements or more, enqueueing and dequeueing can happen simultaneously without interfering with each other. Each node of the list consists of a pointer to the data and a next-pointer. The queue always contains at least one such node, with NULL-ed out data: the 'dummy' node.

To enqueue an element in the non-blocking queue, a new list node is created, and its data pointer filled in. The algorithm then enters a loop in which it tries to enqueue the newly created node. To this effect, it grabs the current tail-pointer of the queue, looks at its next-pointer, and if it is not NULL, help the other thread which must have been responsible for adding a new node past the tail, by attempting to atomically compare-and-swap the next-pointer to the queue's tail slot. If the tail's next-pointer is NULL, then the algorithm attempts to compare-and-swap its own newly created node to the tail's next-pointer. If that succeeds, the algorithm breaks out of the loop; otherwise it keeps looping. After breaking out of the loop, the algorithm still has to compare-and-swap the newly created node to queue's tail-pointer, but if that fails (i. e., when another thread has overwritten the tail-pointer since), no correcting action has to be performed: The other thread will have set the correct tail-pointer. Because of the construction of the Rete network with its limited number of producers and consumers, as well as the semantics of how the lists are used, i. e., how items are inserted, the retry-loop is bounded and local as well as global progress are guaranteed.

Dequeuing works similarly, however, since all lists have only a single consumer, a simple compare-and-swap on the head-pointer to the head-pointer's next-pointer is sufficient. The case of the empty list is implicitly handled with the dummy node.

Memory Management Since PARTE is implemented in C++, memory management becomes an issue that needs to be handled carefully. In our implementation, all facts, i. e., tokens in the Rete network are handled by value and the consumer is responsible for freeing them when they expire. Expiration of events can be determined from the data local to the actor in which the tokens are stored, and therefore does not require a global synchronization effort as is the case in systems where events are only removed after a request for retraction has percolated through the Rete network. PARTE makes use of the timestamps carried by tokens to determine not only how long those tokens still have to be preserved, but also to implement a logical clock with which nodes can know what the oldest point in time is from which their predecessors still may have unpropagated events. By maintaining the invariant that tokens are always propagated in-order, the node actors can know whether new tokens that still can correlate with stored tokens can be expected, and discard tokens for which this is not the case.

More complex is the situation of managing the list elements for the lock-free list implementation. Since they are inherently subject to race conditions, we use a solution similar to reference counting proposed by Michael [16]. All operations on list elements must maintain a set of *hazard pointers*. The hazard pointers are kept in a contiguous piece of memory, and their number depends on the number of threads as well as the use of the data structure. Each thread is associated with a subset of these pointers for its own use. Whenever an operation on a list takes place, a thread explicitly stores pointers to the elements which are *in use* in its thread's section of the hazard pointer array. When an element is deleted from a list, a memory reclamation operation is triggered, which iterates over the hazard pointers and conservatively does not delete any list element which is referenced by a hazard pointer. Since the hazard pointers are visited in ascending order of index in the hazard pointer array, any interleaving of threads using the non-blocking linked-list and the thread reclaiming the linked-list's elements will encounter *at least* all elements that are *in use* at that moment. To this effect, the list's operations may only shift hazard pointers in increasing order of the index.

Because the number of hazard pointers per thread is a small constant, and the number of threads is constant, the amount of memory that is no longer in use, yet not yet reclaimed, is bounded by a small constant per thread, and a reasonably small constant in total.

4. Evaluation

In subsection 2.2, we outlined the requirements for a parallel inference engine used in the context of parallel gesture recognition. This section will discuss how PARTE satisfies these requirements. First, we will discuss the general requirements of how PARTE is optimized for complex event processing (CEP) and how it is embeddable into existing systems. The second part of the evaluation will evaluate the soft real-time properties of PARTE by arguing that the general algorithmic properties and the boundedness of the evaluation process provide the desired guarantees. Finally, we will discuss the performance aspects by comparing the single threaded performance of PARTE and CLIPS, as well as demonstrating scalability of commodity multicore hardware.

4.1 Extensibility and Embeddability

PARTE is designed to be a middleware mediating between the low level of event sources and the application level. To that effect it is a self-contained system, managing its own memory and interacting with other systems via a simple API and callback methods. Applications using PARTE need provide a ruleset and register user functions and callbacks. Event sources need only inform PARTE of new events. The inference engine is continuously running and processes the incoming events as soon as they arrive, maximizing throughput. Through this low coupling between PARTE and the remainder of the system, PARTE is a reusable, easily embeddable software component. The notion of custom user-defined test functions and actions enables PARTE to process arbitrary events and produce output in whatever format the application requires.

4.2 Continuous Event Streams

Since the input devices are assumed to continuously produce events, PARTE was designed to handle expiration of facts automatically to avoid unboundedly growing fact bases. The sliding window mechanism explained in section 3.2 causes the expiration of events which are no longer relevant. This removes not only the burden of manual memory management from the application-level; it also reduces synchronization overhead as *retract* messages need not be sent and processed separately.

4.3 Soft Real-Time

For the assessment of the real-time properties of PARTE, we will rely on the algorithmic properties only. Thus, we will disregard architectural issues such as microprocessor architectures [6] and operating system aspects [15]. Instead, we will give an informal argument to demonstrate that the pattern matching is done in a bounded number of steps, which all complete in a bounded amount of time.

We will therefore demonstrate the soft real-time properties of our system by showing that a predictable bound exists on *a*) the time every *turn* of an actor requires; on *b*) the time required for scheduling actors; on *c*) the time required for passing messages between actors; on *d*) the amount of actors; and on *e*) the amount of turns per actor that are required to detect a pattern .

For this first requirement, more information about the inner workings of the actors is required. We introduced five types of actors in Figure 3: the agenda, type test nodes, join nodes, test nodes, and terminal nodes. In the case of the agenda, type test nodes and terminal nodes, this requirement is trivially met. They all perform a constant amount of work, respectively executing one agenda item, performing a single equality test, and scheduling a number of actions which cannot change at run-time. For test nodes, the situation is barely different: The arithmetic expressions and (in-)equality tests they evaluate are fixed at compile-time, so an upper bound on their runtime can be computed. For the join nodes, the argumentation is a little more involved. The only variable factors in a join node's runtime, however, are the number of variables to unify and the number of fields that have to be bound to those variables. Both are explicitly specified in the ruleset, and therefore known at the time the Rete graph is being compiled. Thus, an upper bound on join nodes' runtimes can be computed.

The next two requirements are closely related. Both the task-queue and the actors' inboxes are implemented as non-blocking data structures. Since the structure of the Rete algorithm limits contention on the actors' inboxes, and Rete nodes cannot generate an arbitrary amount of tokens before having to wait for new incoming tokens, an upper bound on the time required to enqueue and dequeue exists.

The requirement for an upper bound on the number of actors is trivially met, as all actors are statically allocated at startup time.

The last requirement is fulfilled thanks to tiering. By definition, the Rete DAG is acyclic, and by enforcing tiering, we prevent the possibility to make cyclic structures via the feedback loop constituted by the agenda. As such, not only the width but also the depth of the loop-unrolled graph is bounded and known for a given ruleset. Since we require a known upper bound on the rate at which events can be asserted into the system, and communication happens via FIFO queues, the maximum number of turns required before all events currently in the system are processed can be computed.

By showing that an upper bound can be computed on the amount of time PARTE requires to detect gestures, we have demonstrated that PARTE offers soft real-time guarantees.

4.4 Performance Evaluation

To evaluate the performance of PARTE, we follow the methodology proposed by Georges et al. [8]. The benchmarks were executed on a Mac OS X 10.6.8 workstation with two Xeon E5520 processors at 2.26 GHz. Neither TurboBoost nor hyperthreading could be disabled. Thus, TurboBoost can lead to up to 12% higher sequential than parallel performance. However, the measurements of sequential performance remain directly comparable. Both CLIPS and PARTE were compiled with maximum optimizations (-O3) using the GCC 4.2.1 compiler shipped with OS X.

Every benchmarked configuration is run at least 30 times, and is automatically run additionally until a confidence level of 95% is reached. The benchmark results are visualized with beanplots to show the distribution of measurements instead of simple overgeneralizing averages.

We used 13 different benchmarks for the evaluation. Each benchmark consists of a set of rules and a set of pregenerated events to be fed into the system. The benchmarks include microbenchmarks to measure the performance of variable binding, different fact sizes, unification, and β-tests. Furthermore, we used a number of kernel benchmarks designed after common gesture rules to assess the performance of rules with complex tests, and tests that use computational intensive user functions. For motion detection such tests are typically trying to find the spacial relations of a group of points and movements.

We will first look into the efficiency of our system by comparing the runtime performance of PARTE running on a single thread with the runtime performance of CLIPS. CLIPS is an open-source and highly tuned sequential implementation of the Rete algorithm and forms the basis of multiple other production systems, such as PRAIS [9] and Lana [3]. After our evaluation of the efficiency, we will look into PARTE's scalability by investigating the effect of increasing the number of threads allocated for the system.

Efficiency To assess the sequential efficiency of our implementation, we compare PARTE to CLIPS. Our goal is to demonstrate that PARTE, in its current unoptimized state has acceptable sequential performance characteristics in direct comparison. Thus, the performance in kernel benchmarks based on the gesture recognition use case as well as computational intensive workload should be in the same order of magnitude.

Figure 5 depicts the results in form of asymmetric beanplots. For each benchmark, the results have been normalized to the average of the CLIPS results. The distribution of the CLIPS results are depicted in gray, while the results for PARTE are shown in black.

The first graph shows the results for the microbenchmarks. They demonstrate the overhead of message-passing and the performance cost compared to CLIPS. This overhead comes partially from the *by-value* semantics used for the messages and partially from the lock-free queues. Both still have optimization potential, but such pathologic microbenchmarks will always point out the higher overhead compared to a direct sequential implementation as employed by CLIPS.

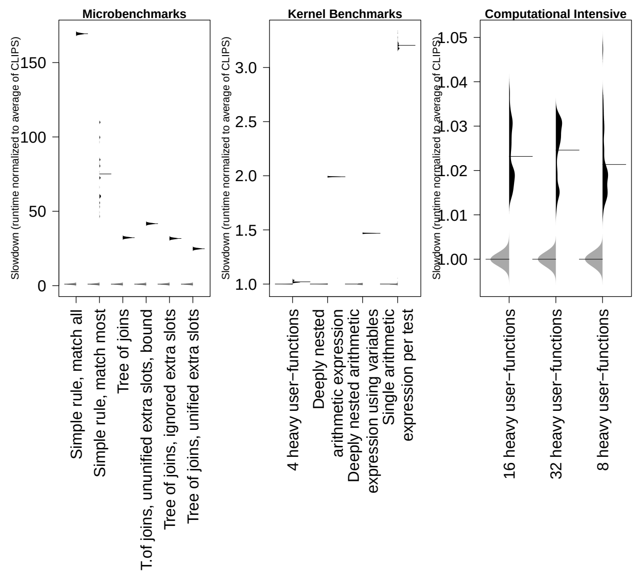


Figure 5. Beanplot comparing CLIPS to PARTE in single threaded execution. Microbenchmarks show architectural overhead. Kernel benchmarks representing typical gesture rules show competitive performance. Computational intensive rules, typical for motion detection rules, show less than 5% overhead.

The second graph shows the results for kernel benchmarks with characteristics found in the gesture recognition rules used by Hoste et al. [14]. Here we see that PARTE delivers comparable single threaded performance on the same order of magnitude as CLIPS, without its a highly tuned implementation.

The third graph shows results of computational intensive rules, as they are found in complex motion recognition rules that need to correlate spatial coordinates of many events. These kind of rules are the most relevant for advanced uses cases based on input devices such as Microsoft's Kinect. Here, the evaluation of the test expressions is the most intensive part, and PARTE has comparable performance to CLIPS.

Scalability A beanplot of the combined results of all benchmarks is depicted in Figure 6. This graph gives an impression of the distribution of all benchmark results for a varying number of worker threads. The little accumulation points along the runtime axis indicate the increased number of measurements at that point, which coincide with specific benchmarks. The horizontal bar indicates the average.

Figure 7 depicts results as a speedup graph to emphasize the scalability of the system. The dotted line indicates the ideal speedup—a speedup which increases linearly with the amount of processors. The graphs show that PARTE scales well in the average case. The system's design offers the significant advantage of spreading the load over multiple threads. When that load gets heavy, PARTE can effectively

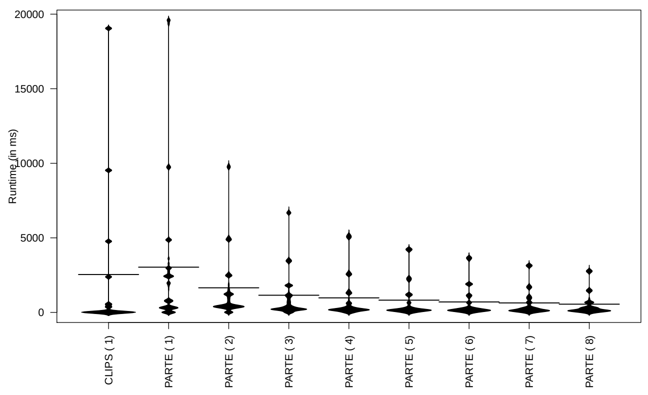


Figure 6. Beanplot showing distribution of benchmark results for 1 to 8 worker threads.

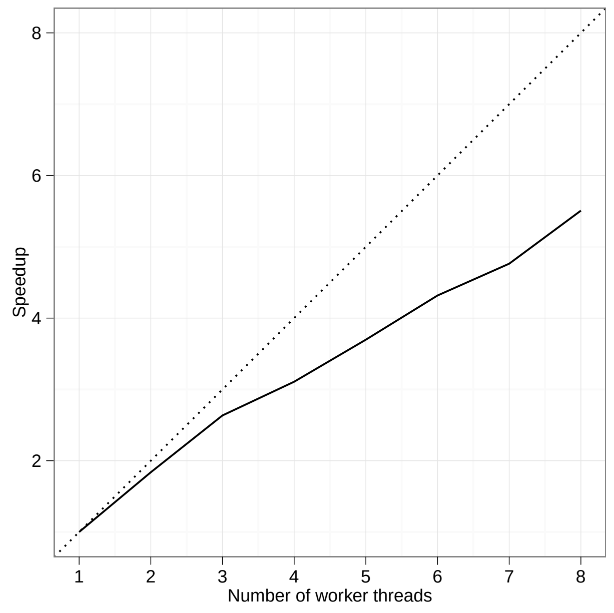


Figure 7. Speedup graph, showing PARTE's scalability with number of worker threads compare to ideal speedup (dotted line).

benefit from the processing power of different processing units. This form of pipeline parallelism is ideal for rules that rely on complex tests such as used in motion detection. For the intended use case, the parallel decomposition created by PARTE enables the system to benefit from a close to ideal scaling up to 8 worker threads.

4.5 Discussion

Our choice of actors as the unit of parallelization stems from the strong similarity between the passing of tokens between nodes in the Rete algorithm on the one hand, and message passing between actors on the other hand. The actor model provides a nice metaphor for the construction of a graph of interlinked nodes which share data only by explicitly passing it to their successors in the DAG.

Some limitations exist in the current version of PARTE. Since test-expressions are separated from the nodes that join two branches, negation-as-failure is not supported in PARTE. Because of the parallel execution model, the implementation of negation requires additional communication and does not fit into the current model. However, gesture recognition systems can work without a notion of negation, but the addition of it would be worthwhile addition for other use cases.

Furthermore, the coarseness of parallelization can further be tuned, as in some situations the actor-based approach does not expose all options for parallelism. Computational intensive test functions such as required for motion pattern detection could benefit from parallelizing the match on facts inside a node/actor. Inversely, in other situations the actor-based implementation using message-passing and lock-free queues imposes a high overhead which could be reduced by merging nodes appropriately.

Another area that could be improved is PARTE's scheduler for the actors, which focusses on correctness and real-time properties. Currently, actors are scheduled although their inbox is empty. Improved scheduling which preserves soft real-time guarantees is planned for future work.

5. Related Work

The Rete algorithm is widely used, and has had many adaptations to both fit real-time requirements and to fit parallel processing.

5.1 Parallel Rete

Gupta et al. [11] measure that the matching is the most computationally expensive task and takes up to 90% of the execution time in production systems. Consequently, most effort has been dedicated to parallelizing the match-phase.

One of the best known Rete derivates focussing on parallelism is the TREAT algorithm by Miranker [18]. In TREAT, for every condition element, the matching facts are stored. This makes TREAT a state-saving algorithm, but less so than Rete, which in addition to those *alpha memories* stores the matching sets of facts for combinations of condition elements that appear in the rules, in *beta memories*. At the other end of the spectrum is the production system proposed by Oflazer [19], which stores the matching sets of facts for *every* combination of condition elements, regardless of whether they appear in the rules. Both TREAT and Oflazer's machine diverged from the traditional Rete algorithm to reduce the need for synchronization, thereby opening options for parallelism. Both approaches had the foreseeable drawbacks: TREAT spends a lot of time recomputing matches for entire patterns, and the combinatorial explosion made Oflazer's machine consume a lot of working memory, in addition to spending a large amount of time computing combinations of facts which may never get used. PARTE, which sticks to the traditional Rete algorithm has none of these drawbacks. It does not decouple the different threads of execution by performing too little or too much work to be able to skip synchronizing, but instead focusses on reducing the overhead of the synchronization. In addition, it uses automatic expiration of facts, which halves the number of inter-node communication that has to take place compared to Miranker's and Oflazer's system with manual retraction.

A different approach is taken by Aref and Tayyib [3], whose distributed Rete algorithm Lana is an optimistic algorithm, allowing the different processing elements to run with minimal synchronization, informing a single central *Master Fact List* of changes to the working memory, and backtracking when the updates of the multiple replicated Rete engines conflict. Unlike in PARTE, the different entities running in parallel in Lana are fixed, allowing less flexibility to redistribute workload among the available processing units, and by splitting up the Rete graph, common subgraphs cannot be shared by multiple rules, requiring Lana to duplicate work where PARTE could reuse computations. Moreover, the optimistic approach generates a degree of nondeterminism with respect to run time which a real-time system like PARTE cannot risk to incur.

Yet other systems use hierarchical blackboard systems on which multiple agents concurrently work, and where every 'row' of nodes in the Rete network is reified as a different blackboard in the knowledge base's hierarchy. Examples of such systems are the Parallel Real-time Artificial Intelligence System (PRAIS) of Goldstein [9] and the Hierarchically Organized Parallel Expert System (HOPES) by Dai et al. [5]. Semantically similar, but not using the blackboard metaphor are for instance Gupta's parallel Rete: They also acknowledged that having more than one token flowing through the graph at any one time could be opportune for the execution speed. Gupta et al. [10] proposed to give every node one or more internal threads of control. Their approach was conceptually very close to ours, was not constrained to event-processing, and supported conflict-resolution strategies like sequential Rete implementations. Because of this, they had to omit all conceptual optimizations which depend on temporal reasoning and many options for parallelism in the evaluation of the rules' consequents. Furthermore, their approach depended on hardware task schedulers to enqueue and dequeue node activations in a timely manner.

In general, previous approaches did not use the abstraction of actors like PARTE does. Their scheduling algorithms could not transparently handle nodes and the agenda, and they did not consider the nodes as self-contained elements, solving data-locality in an ad-hoc manner, in Gupta's case expecting the presence of processor-local private memory in addition to the caches and main memory.

5.2 Real-Time Rete

Real-time execution characteristics can be achieved in two ways: By guaranteeing for every task that it completes in a known timeframe, and scheduling them such that they all complete before their deadline, or by assigning priorities to tasks, and enabling the system to preempt lower-priority tasks to make sure high-priority tasks complete in time.

PARTE takes this first approach, and ensures that every action taken by the inference engine completes in time – unless unexpected load is put on the system by entities other than PARTE. The Parallel Real-time Artificial Intelligence System (PRAIS) of Goldstein [9] instead takes the second approach, dropping the matching of lower-priority rules to allow more important rules to be matched in time. Despite being considered a real-time system, PRAIS can only offer best-effort guarantees, as it is a distributed system depending on TCP/IP for communication.

Sequential implementations such as the self-stabilizing OPS5 production system by Cheng and Fujii [4] do offer actual hard real-time guarantees, but their approach is not viable for our problem domain: It requires lots of effort in the generation of the ruleset to provide fault-tolerance on top of the real-time guarantees. Their system is aimed at situations where failure is catastrophic and must hence be avoided at all costs. PARTE does not pose such severe restrictions on the ruleset, only requiring *tiering*.

6. Conclusions and Future Work

The presented PARTE inference engine implements a variation of the Rete algorithm using actor semantics for Rete nodes to achieve parallel execution. The system is designed for continuous gesture recognition which requires soft real-time execution guarantees and scalability on parallel systems. While PARTE utilizes the constraints of the domain to achieve these properties, it remains applicable to the broader domain of *Complex Event Processing*. This includes applications such as algorithmic stock trading and monitoring network security.

PARTE achieves the desired scalability and soft real-time guarantees by using a tiered architecture, lock-free queues, and an actor execution model to provide upper bound guarantees on the event matching in a Rete network.

PARTE is compatible with existing single threaded inference engines such as CLIPS from NASA. It has been used as a replacement for CLIPS in the core infrastructure of the multimodal Mudra framework [20]. PARTE provides the benefits of transparent parallelization of declarative rules as well as automatic event expiration.

Our preliminary performance evaluation used a number of microbenchmarks, kernel, and computational intensive benchmarks. The benchmark characteristics are representative for the gesture recognition and multimodal event processing context. In the current unoptimized state of PARTE, we achieve comparable performance to CLIPS. Furthermore, PARTE was demonstrated to scale on multicore systems with up to 8 cores, outperforming the inherently sequential implementation of CLIPS.

In future work, we will approach more efficient scheduling of the Rete nodes as well as exposing more parallelism opportunities by optimizing the Rete network. Support for an efficient implementation of a negation operator is planned as well.

Acknowledgments

Lode Hoste and Stefan Marr are supported by a doctoral scholarship of the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen), Belgium.

References

[1] G. Agha. Concurrent object-oriented programming. *Commun. ACM*, 33(9):125–141, Sept. 1990. ISSN 0001-0782. doi: 10.1145/83880.84528.

[2] R. M. Akscyn, D. L. McCracken, and E. A. Yoder. KMS: a distributed hypermedia system for managing knowledge in organizations. *Communications of the ACM*, 31(7):820–835, July 1988. ISSN 00010782. doi: 10.1145/48511.48513.

[3] M. M. Aref and M. A. Tayyib. Lanamatch algorithm: a parallel version of the retematch algorithm. *Parallel Computing*, 24(5-6):763–775, June 1998. ISSN 0167-8191. doi: 10.1016/S0167-8191(98)00003-9.

[4] A. Cheng and S. Fujii. Bounded-response-time self-stabilizing ops5 production systems. In *Proceedings 14th International Parallel and Distributed Processing Symposium. IPDPS 2000*, pages 399–404. IEEE Comput. Soc, 2000. ISBN 0-7695-0574-0. doi: 10.1109/IPDPS.2000.846012.

[5] H. Dai, T. Anderson, and F. Monds. On the implementation issues of a parallel expert system. *Information and Software Technology*, 34(11):739–755, November 1992. ISSN 09505849. doi: 10.1016/0950-5849(92)90169-P.

[6] A. A. El-Haj Mahmoud. *Hard-Real-Time Multithreading: A Combined Microarchitectural and Scheduling Approach*. PhD thesis, 2006. AAI3223132.

[7] C. L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19(1): 17–37, 1982. ISSN 0004-3702. doi: 10.1016/0004-3702(82) 90020-0.

[8] A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous java performance evaluation. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, OOPSLA '07, pages 57–76. ACM, 2007. ISBN 978-1-59593-786-5. doi: 10.1145/ 1297027.1297033.

[9] D. Goldstein. Extensions to the Parallel Real-Time Artificial Intelligence System(PRAIS) for fault-tolerant heterogeneous cycle-stealing reasoning. In *Proceedings of the 2nd Annual CLIPS ('C' Language Integrated Production System) Conference, NASA. Johnson Space Center*, pages 287–293, Houston, September 1991.

[10] A. Gupta, C. Forgy, A. Newell, and R. Wedig. Parallel algorithms and architectures for rule-based systems. In *Proceedings of the 13th annual international symposium on Computer architecture*, ISCA '86, pages 28–37. IEEE Computer Society Press, 1986. ISBN 0-8186-0719-X. doi: 10.1145/17407.17360.

[11] A. P. Gupta, C. L. Forgy, D. Kalp, and A. Newell. Parallel OPS5 on the Encore Multimax. *Proceedings of the International Conference on Parallel Processing*, pages 271–280, 1988.

[12] T. Hammond and R. Davis. LADDER, A Sketching Language for User Interface Developers. *Computers and Graphics*, 29 (4), August 2005.

[13] T. L. Harris. *A pragmatic implementation of non-blocking linked-lists*. Springer, 2001. ISBN 3-540-42605-1. doi: 10.1007/3-540-45414-4_21.

[14] L. Hoste, B. Dumas, and B. Signer. Mudra: A Unified Multimodal Interaction Framework. In *Proceedings of ICMI 2011, 13th International Conference on Multimodal Interaction*, Alicante, Spain, Nov. 2011.

[15] J. W. S. Liu. *Real-Time Systems*. Prentice Hall, Upper Saddle River, NJ, 2000. ISBN 0130996513 9780130996510.

[16] M. M. Michael. Safe memory reclamation for dynamic lock-free objects using atomic reads and writes. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing - PODC '02*, page 21, New York, New York, USA, 2002. ACM Press. ISBN 1581134851. doi: 10.1145/571826.571829.

[17] R. B. Miller. Response time in man-computer conversational transactions. In *Proceedings of the December 9-11, 1968, fall joint computer conference, part I on - AFIPS '68 (Fall, part I)*, page 267, New York, New York, USA, 1968. ACM Press. doi: 10.1145/1476589.1476628.

[18] D. P. Miranker. Treat: a better match algorithm for ai production systems. In *Proceedings of the sixth National conference on Artificial intelligence - Volume 1*, AAAI'87, pages 42–47. AAAI Press, 1987. ISBN 0-934613-42-7.

[19] K. Oflazer. *Partitioning in ParalleI Processing of Production Systems*. PhD thesis, Carnegie Mellon University, 1985.

[20] C. Scholliers, L. Hoste, B. Signer, and W. De Meuter. Midas: A declarative multi-touch interaction framework. In *Proceedings of the Fifth International Conference on Tangible, Embedded, and Embodied Interaction*, TEI '11, pages 49–56, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0478-8. doi: 10.1145/1935701.1935712.

[21] J. Shotton, A. Fitzgibbon, M. Cook, T. Sharp, M. Finocchio, R. Moore, A. Kipman, and A. Blake. Real-time human pose recognition in parts from single depth images. In *CVPR*, volume 2, page 7, 2011.

A Relational Trace Logic for Simple Hierarchical Actor-Based Component Systems

Ilham W. Kurnia
University of Kaiserslautern
ilham@cs.uni-kl.de

Arnd Poetzsch-Heffter
University of Kaiserslautern
poetzsch@cs.uni-kl.de

Abstract

We present a logic for proving functional properties of concurrent component-based systems. A component is either a single actor or a group of dynamically created actors. The component hierarchy is based on the actor creation tree. The actors work concurrently and communicate asynchronously. Each actor is an instance of an actor class. An actor class determines the behavior of its instances. We assume that specifications of the behavior of the actor classes are available. The presented logic allows deriving properties of larger components from specifications of smaller components in a hierarchical manner.

The behavior of components is expressed in terms of traces where a trace is a sequence of events. A component specification relates traces of input events to traces of output events. Generalizing Hoare-like logics from states to traces and from statements to components, we write $\{p\}\ C\ \{q\}$ to mean that if an input trace satisfies p, component C produces output traces satisfying q; that is, p and q are assertions over traces. Such specifications are partial in that they only specify the reaction of C to input traces satisfying p.

This paper develops the trace semantics and specification technique for actor-based component systems, presents important proof rules, proves soundness of the rules, and illustrates the interplay between the trace semantics, the specification technique and the proof rules by an example derived from an industrial Erlang case study.

Categories and Subject Descriptors F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs — generalized Hoare logics

General Terms Design, Theory, Verification

Keywords actors, specification techniques, relational reasoning

1. Introduction

In this paper, we develop a specification and reasoning technique for component-based open distributed systems. Distributed systems are realized by dynamically growing collections of actors ([1]) that communicate with other actors via asynchronous messages. In particular, we want to enable reasoning about functional properties of *open* systems, that is, about systems working in an environment for which we do not have an implementation or a precise specification.

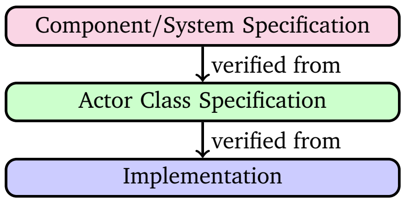


Figure 1. Two-tier verification

The basis of our approach is a two-tier verification as shown in Fig. 1. The two-tier verification avoids the complex task of directly reasoning about system properties on the implementation level. Following the approach of Creol [19] and ABS [20], we assume that actors are implemented using the object-oriented concept of *classes*. A class determines how all instances of the class behave. To reflect the concept of classes at the specification level, we use specifications of actor classes, called *actor class specifications*, which allow specifying properties about the behavior of all instances of a class implementation. In the first tier of our verification approach, the actor implementation is then verified against a specification of the actor. This task is not considered in this paper, but addressed by the work of Din et al. [15] and Ahrendt and Dylla [3].

The aim of this paper lies in the second tier, i.e., to use actor class specifications to verify properties of small components and to use these component specifications to verify larger components and open systems. A component is formed hierarchically by following the actor creation tree. Starting from the initial actor, a component consists of all actors transitively created by the initial actor. The exact formation of a component is influenced by how the unknown environment interacts with the component. Hence, an open system can be considered as a component.

To achieve the aforementioned goal, we characterize open actor systems in terms of a trace semantics, introduce a specification technique based on the traces that does not refer to any implementation, and derive a logic that utilizes this specification technique. An execution of an actor system can be represented by a trace of observable events [9, 18]. The advantage of dealing only with traces is that it abstracts from the actual state representation of the system. The semantics of actors and components can be expressed in terms of trace sets. When the actor or the component represented by the trace set is known, each trace of the set can be split into an input and output trace representing the events it receives and produces, respectively.

Based on this semantics, we develop a specification technique relating input traces to output traces. Formally, a specification consists of a finite set of Hoare-like triples [17]. A triple $\{p\}\ D\ \{q\}$ denotes that if an input trace satisfies p, component D produces output traces satisfying q. The component D either denotes the behavior of a single actors of class C, or denotes the (external) behavior of groups of actors with an initial actor of class C. It is important to notice that a triple specifies the behavior of a component only for inputs satisfying p. These input conditions express assumptions about the usage of the component and help to focus the reasoning.

We show the usage of the specification technique by means of a proof system for a simple form of composition that we call *daisy chain* composition. It allows a component to dynamically create a new component, but forbids the new component to call back to its creator. We show through an example taken from an Erlang case study [6] how to verify properties of larger components by only using the specifications of smaller components.

Paper Structure. The following section describes the language background and our running example. Section 3 gives the definitions of actor classes and components and how their trace sets are characterized and composed. Sections 4 and 5 present the specification technique and the proof system, along with their application to the running example. Section 6 discusses the related work. Section 7 concludes and describes future work.

2. Language Background and Example

To have a sufficiently clear background for the following discussion on specification and verification, we informally introduce a core actor language AJ together with an example for illustrating our approach. Following the design of the modeling languages Creol [19] and ABS [20], AJ uses classes to describe actors (we use the keyword **actor class**). Actors can be dynamically created, implement interfaces, have an actor-local state expressed in terms of instance variables, and are addressed via a typed reference.

As a running example, we use a variant of the client-server setting treated in an industrial Erlang case study by Arts and Dam [6]. The server receives requests from the client, where each of these requests contains a task. The goal of the server system is to respond to the requests with the appropriate task computation results. To serve each request, the server creates a worker and pass on the task to be computed. As a computation task can be divided into multiple chunks, more concurrency can be introduced in the following way. Before each worker processes the first chunk of the task, it creates another worker to which the rest of the task is passed on. When the computation of the first task chunk is finished, the worker merges the previous result with this computation result and passes on the merged result to the next worker. Eventually all chunks of the task are processed, and the last worker sends back the final result to the client. The structure of the request processing forms a daisy chain as illustrated in Fig. 2.[1] In the actor setting (where all clients, servers and workers are actors), the client name needs to be passed around as well so that the last worker can return the task computation result to the client. This example illustrates unbounded actor creation and non-trivial concurrency.

Figures 3 and 4 illustrate how the server scenario can be implemented in AJ, which uses a Java-like syntax. The central actor of our example is `AServer` implementing the interface `Server`: receiving message `serve(c, t)`, it deals with the computation task `t` and makes sure that a response is sent to the

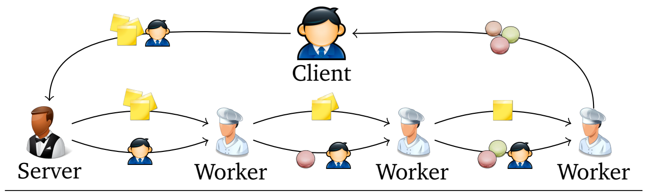


Figure 2. Daisy chain structure of the server system

```
interface Client { response(Value); }
interface Server { serve(Client, CompTask); }
interface Worker {
  do(CompTask);
  propagateResult(Value, Client);
}

actor class AServer implements Server {
   serve(Client c, CompTask t) { // taskSize(t) ≥ 1
      Worker w = new AWorker();
      w.do(t);
      w.propagateResult(null, c);
   }
}
```

Figure 3. Interfaces and the actor class `AServer`

client (cf. the interface `Client`). To enable concurrent execution of tasks, the server delegates the task to a dynamically created worker (interface `Worker`). If the task has more than one chunk (`taskSize(t) > 1`), the worker delegates the rest of the task to a newly created worker and works on the first chunk. By a series of `propagateResult` messages, initiated by the server, the results of the different chunks are collected, merged, and the final result is sent back to the client.

Reacting to messages. In AJ, a *message* consists of a method name and typed parameters. A message is produced when a statement of the form `r.m(p̄)` is executed. This statement, known also as a method call, sends the message `m(p̄)` to the receiver actor `r` where `m` is the method name with a list of parameters `p̄`. The parameters can be data values or actor references. Such a send-operation is non-blocking; execution directly continues with the next statement. Thus, in general, a message send leads to concurrent behavior. For each of its messages, an actor has a *body* that describes how it reacts to a message. For example, an actor of class `AServer` (see Fig. 3) reacts to a message `serve(c, t)` as follows: It creates a worker actor, sends first a `do`- and then a `propagateResult` message to the worker. We assume that the execution of message bodies must terminate.

Receiving and selecting messages. It remains to explain what happens on a message receive. We assume that actors have an unbounded input queue and are input enabled (cf. [21, p. 257]); i.e., actors can always accept new input. Messages are *selected* from the queue essentially in a FIFO manner, but if they have a guard that evaluates to **false**, their selection is postponed. Thus, an actor has control over the execution of incoming messages. Message selection is (weakly) fair for messages with true guards, meaning that a message whose guard is infinitely often evaluated to **true** will eventually be picked for processing. In Fig. 4, the actor class `AWorker` uses a guard to select a `propagateResult` message only if a result is available.

Further constructs. In addition to the actor-related aspects, AJ supports recursive data types and function definitions for handling data (as in functional programming languages). In

[1] Icons in the figure are taken from `http://www.iconarchive.com`

```
actor class AWorker implements Worker {
  Value myResult = null;
  Worker nextWorker = null;

  do(CompTask t) {
    if (taskSize(t) > 1) {
      nextWorker = new AWorker();
      nextWorker.do(restTask(t));
    } else {
      nextWorker = null;
    }
    myResult = compute(firstTask(t));
  }

  propagateResult(Value v, Client c)
      guard myResult != null {
    if (nextWorker == null) {
      c.response(merge(myResult, v));
    } else {
      nextWorker.propagateResult(merge(myResult, v), c);
    }
  }
}
```

Figure 4. Actor class `AWorker`

the running example, we assume appropriate definitions for the data types `CompTask` and `Value` and the total functions:

```
compute   :  CompTask ⟶ Value
taskSize  :  CompTask ⟶ Int
firstTask :  CompTask ⟶ CompTask
restTask  :  CompTask ⟶ CompTask
merge     :  Value × Value ⟶ Value
```

where `compute(t)` computes the result of `t`; `taskSize(t)` yields a number of chunks in which `t` could be partitioned; `firstTask(t)` returns the first chunk of `t`; `restTask(t)` returns the rest of `t`; and `merge` merges results. We assume the following properties:

```
taskSize(t) ≥ 1
taskSize(t) > 1 → compute(t) = merge(compute(firstTask(t)),
                                      compute(restTask(t)))
taskSize(t) = 1 → compute(t) = compute(firstTask(t))
merge(null, v) = v
```

A task consists of at least one chunk; computing a non-primitive task is the same as merging the result of computing the first task with the computation of the rest of the task; computing a single task chunk is the same as computing the first task of the chunk; and merging with **`null`** with some value `v` returns `v`.

Actor systems are started by creating actors and start their activities or connect them to activities in the environment, for example to user interfaces. In this perspective, we deal with open systems because of the interaction with their environments.

3. Semantics of Actors and Components

An execution of an actor system can be represented by a trace of observable events [9, 18]. The functional behavior of an actor system is represented by a trace set. Taking the work of Agha et al. [2] and Vasconcelos and Tokoro [30] as a guide, we consider the trace sets of actors and components in an open system setting. More precisely, we characterize the trace sets with respect to the *most general environment*, i.e., the environment that provides all admissible behaviors. For the sake of simplicity, we assume that each actor has a fair chance to do its computation

Table 1. Helper predicates and functions

Predicate/ Function	Description
$\mathit{Pref}(s)$	The set of all prefixes of sequence s.
$\mathit{class}(a)$	Returns the class of actor a.
$\mathit{isMtd}(m)$	Checks if message m is a method call.
$\mathit{callee}(e)$	Returns the callee of event e.
$\mathit{caller}(e)$	Returns the caller of event e.
$\mathit{msg}(e)$	Returns the message of event e.
$\mathit{acq}(t)$	Returns the accumulated actor names exposed in each method call event in t.
$\mathit{cr}(t)$	Returns the set of actors created in t.
$t{\downarrow_A}$	Projects t to non-external events of a set of actors A. The operator can also take *callee* or *caller* as an extra parameter.
$\mathit{exposedTo}(t,A)$	$\mathit{acq}(t{\downarrow_{A,\mathit{callee}}}) \cup \mathit{cr}(t{\downarrow_{A,\mathit{caller}}})$
$[\![C]\!]$	The set of traces $\mathit{Traces}(\{a\})$ where $\mathit{class}(a) = C$.
$[\![[C]]\!]$	The set of traces $\mathit{Traces}([L])$ where the initial actor of L is of class C.
$\mathit{ext}(T)$	Extracts the largest visible subset of local actors from a boxed trace set T.
$\mathit{split}(t,L)$	Splits t into input and output traces $(\mathit{ti},\mathit{to})$ based on local actors L.

and there is a type system handling the correctness of types of events and their content.

This section is divided into two subsections. The first subsection deals with the foundation of traces, namely what the trace events are, the basic operations one can perform on traces, what valid traces are, and what the trace set of the individual actors are. The second subsection defines what a component is, based on a composition of the traces of the actors contained in the component. To focus on the interaction with its environment, we define a boxing operator on a component to hide all internal interaction. However, this boxing operator is, for the proof system, too strong. In particular, we want to know how the *sub*components interact with each other. For this purpose, we provide a glass box view [9, p. 5], given the structure of the component. An informal summary of numerous helper predicates and functions is given in Table 1 as a quick reference.

3.1 Trace Foundation

Traces are represented using the finite sequence data structure $\mathit{Seq}\langle T\rangle$, with T denoting the type of the sequence elements. An empty sequence is denoted by $[]$ and $\cdot$ represents sequence concatenation. The function $\mathit{Pref}(s)$ yields the set of all prefixes of a sequence s.

The foundation of the trace semantics is the set of *actor names* $a, b \in \mathbf{A}$ and the set of *messages* $m \in \mathbf{M}$ that can be communicated between actors. We assign each actor with a specific behavior represented by a *class* $C \in \mathbf{CL}$. The function $\mathit{class}(a)$ gives the class of an actor a. A message m can either be an actor creation **`new`** C or a method call `mtd`$(\overline{p})$. `mtd` denotes some method name and $\overline{p}$ is a list of parameters. A parameter may be a data value $d \in \mathbf{D}$ or an actor name. The predicate $\mathit{isMtd}(m)$ checks whether the message m is a method call.

From this foundation, we build the set of *events* $\mathbf{E}$. An event $e \in \mathbf{E}$ represents the occurrence of a message $m = \mathit{msg}(e)$ being sent by the *caller* actor $a = \mathit{caller}(e)$ to the *callee* actor $b = \mathit{callee}(e)$. If m is a creation message, b will be the name of the newly created actor while a is its creator. Textually an event e is represented as $a \to b$:= **`new`** C or $a \to b$.`mtd`$(\overline{p})$ when the message is an actor creation or a method call, respectively.

The inclusion of the caller information allows us to distinguish between input and output events with respect to an actor or a group of actors. Eliminating the caller information from an event produces an *event content*. Taking $L \subseteq \mathbf{A}$ to be the set of (**l**ocal) actors we are considering and, by the open system setting, $F = \mathbf{A} - L$ as the set of all (**f**oreign) actors L is interacting with, events that happen in the trace can be categorized as follows. An event e is

- an *input* event if $a \in F$ and $b \in L$;
- an *output* event if $a \in L$ and $b \in F$;
- an *internal* event if $a, b \in L$;
- an *external* event if $a, b \in F$.

Only non-external events are of interest here as the environment's internal behavior is unknown. Method calls expose names to callee actors. As the exposure of actor names is important to decide when an actor can send a message to another actor or pass on names to other actors in the semantics, we define a function $acq(a \rightarrow b.\mathtt{mtd}(\overline{p}))$, short for acquaintance, to extract the finite set of actor names occurring in the parameter list of a method call event. The caller information is transparent to the callee, so it is not part of the acquaintance.

Example 3.1. Consider a server actor `s`, a client actor `c`, a worker actor `w`, some task `t` and $L = \{\mathtt{s}, \mathtt{w}\}$. The event `c` $\rightarrow$ `s.serve(c,t)` is an input event for `s` and an output event for `c`. The event `s` $\rightarrow$ `w` $:=$ **`new`** `Server` is an internal event of L.

A *trace* $t = e_1 \cdot e_2 \cdot \ldots \in Seq\langle \mathbf{E} \cup \{\surd\}\rangle$ is a finite sequence of events that represents a single execution of the entity L it represents. The $\surd$ symbol indicates that t is a *maximal* trace, that is when the environment of L represented by the trace stops sending more input to L, then L stops its activity.

The basic operator on a trace is the *projection* operator. t can be projected to a given a set of actor names $A \subseteq \mathbf{A}$, written $t{\downarrow_A}$, where all events, except $\surd$, whose neither caller nor callee is not in A are dropped. The function is refined by a caller (or callee) parameter $t{\downarrow_{A,caller}}$ ($t{\downarrow_{A,callee}}$) when the retained messages are those whose callers (callees) are in A. With respect to some local actor set L, a trace is called an *input* (*output*) trace when all its events are input (output) events.

Given a set T of traces, the projection $T{\downarrow_A}$ yields the set of traces T' where each trace t in T is projected to A. The set of acquaintance is lifted to the traces. It is straightforward to show that *acq* grows monotonically with respect to the length of the trace. The function $cr(t)$ produces the set of actors that are created in a trace t. Similar to *acq*, *cr* also grows monotonically. Because these functions are used in conjunction to know which actor has been exposed to a group of actors A in a trace t, we abbreviate $acq(t{\downarrow_{A,callee}}) \cup cr(t{\downarrow_{A,caller}})$ as $exposedTo(t, A)$.

The behavior of an actor system can be represented in terms of a set of *valid* traces. This notion of valid trace set ideally should be derived from the operational semantics of the actor systems, which lies outside of the scope of this paper. Intuitively, a valid trace is a trace that contains no external events, starts with the creation of some actor and allows the environment to make method calls to local actors when they are exposed. Taking into account the open environment setting, we require a valid trace set of a set of local actors to be a set of valid traces that is prefix-closed and allows foreign actors to make a method call to exposed local actors at any time. The prefix-closedness allows us to observe the behavior of a (group of) actor(s) at any point in time. In addition to this valid trace restriction, we assume a creation message always produces a fresh actor name (technically, this can be realized by using a hierarchical naming scheme). Thus, the creation of actors forms a tree indicating which actor is created (directly or indirectly) by which other actor.

Definition 3.1 (Valid trace set). Let $L \subseteq \mathbf{A}$ be the set of local actors, $F = \mathbf{A} - L$ and $C \in \mathbf{CL}$. Let also e be an event such that $caller(e) = a$, $callee(e) = b$, $msg(e) = m$. A set of traces $Traces(L)$ is *valid* if

1. $\forall t \cdot e \in Traces(L) \bullet t \in Traces(L)$ (prefix closed);
2. $\forall t \cdot e \in Traces(L) \bullet a \neq F \vee b \neq F$ (non-external events);
3. $\forall e \cdot t \in Traces(L) \bullet m = \mathbf{new}\ C \wedge a \in F \wedge b \in L$ (initial creation);
4. $\forall t \cdot e \in Traces(L) \bullet a \in F \implies$
   $\{b \mid isMtd(m)\} \cup acq(e) \subseteq exposedTo(t, F) \cup F$
   (proper local actor exposure to foreign actors);
5. $\forall t \in Traces(L), e \in \mathbf{E} \bullet a \in F \wedge b \in L \wedge isMtd(m) \wedge$
   $\{b\} \cup acq(e) \subseteq exposedTo(t, F) \cup F \implies t \cdot e \in Traces(L)$
   (open environment admissibility).

A trace that satisfies requirements 2, 3 and 4 is a *valid trace*.

Example 3.2. Let `s` be a server actor, `c` a client actor, `t` a task, and `w` a worker actor. The following trace t is a valid trace of a server actor `s`.
$t =$ `c` $\rightarrow$ `s` $:=$ **`new`** `Server` $\cdot$ `c` $\rightarrow$ `s.serve(c,t)` $\cdot$
`s` $\rightarrow$ `w` $:=$ **`new`** `Worker` $\cdot$ `s` $\rightarrow$ `w.do(t)` $\cdot$
`s` $\rightarrow$ `w.propagateResult(`**`null`**`,c)` $\cdot \surd$
Of particular interest is that `s` is exposed to `c` before `s` can pass `c` to `w`. In addition, $\surd$ indicates that the server receives no more input from the environment (which is the client) and it finishes processing both the server actor creation and the client's request.

The primitive building blocks of our systems are actors whose behavior is represented by some class. We assume that a self call does not appear in the trace of an actor. Changes that occur with a self call can be simulated allowing non-deterministic choices of trace continuation, because it does not affect the information the actor receives from its environment.

Definition 3.2 (Actor trace set). Given a class name $C \in \mathbf{CL}$ and an actor a where $class(a) = C$, the *trace set of the actor of class* C is a valid trace set (Def. 3.1) $Traces(\{a\})$ such that

- $\forall e \cdot t \in Traces(\{a\}) \bullet callee(e) = a \wedge msg(e) = \mathbf{new}\ C$
  (starts with a creation of a), and
- $\forall t \cdot e \in Traces(\{a\}) \bullet caller(e) = a \implies$
  $\{callee(e) \mid isMtd(msg(e))\} \cup acq(e) \subseteq acq(t{\downarrow_{F,callee}}) \cup \{a\}$
  (proper foreign actor exposure to a).

A valid trace which either callee or caller of its events is the actor a and satisfies the two properties above is an *actor trace*.

The definition above closes the exposure requirement from the environment side left open in the valid trace set definition. An actor trace of class C that ends with $\surd$ indicates that no other input events are sent to the actor and the actor finishes processing all input events. We denote $[\![C]\!] = Traces(\{a\})$ to represent the semantics of actors a of class C.

3.2 Actor-Based Components

The next step is to compose these primitive blocks to make a component. First, we define how we compose a group of actors. The basic operation is the *plain composition*, which takes a set of arbitrary actors. The interaction between these actors is taken as traces whose projection to each actor matches some trace of that actor. Because we deal with a group of actors, the scope of the environment shrinks. To be more precise, when an actor exposes some name to some other actor, it does not mean that the environment can directly use the exposed name, as the other actor may also be part of the group. For the validity to hold, the exposure clause in Def. 3.1 needs to be explicitly enforced.

Definition 3.3 (Plain trace set). Let $L \subseteq \mathbf{A}$ be a set of actors and $F = \mathbf{A} - L$. The *plain* trace set $Traces(L)$ is the largest possible set such that

- $\forall t \in Traces(L), a \in L \bullet t\downarrow_{\{a\}} \in Traces(\{a\})$, and
- $\forall e \cdot t \cdot e' \in Traces(L) \bullet caller(e') \in F \implies$
  $\{callee(e') \mid isMtd(msg(e))\} \cup acq(e') \subseteq exposedTo(e \cdot t, F)$.

It is straightforward to show that an actor trace set is also plain. Thus this composition does not violate the single actor behavior.

Lemma 3.1. *Let $L = \{a\}$. Then $Traces(L)$ is plain.*

Using plain composition, we can characterize a component as a set of actors that does not need to create actors outside of the set. This dynamic notion of a component is motivated by the hierarchical nature of actor creation, which makes the component independent of actors in the environment with respect to its internal behavior. Note that this does not prevent the component to interact with the environment. We restrict ourselves to components where only one actor is created by the component's environment. This actor is called the *initial actor* of the component. This restriction allows us to refer to a component by the class of the initial actor C. The notion of a component with an initial actor of class C is formalized as follows.

Definition 3.4 (Component). L is a *component* with an initial actor of class C if

- $\forall e \cdot t \in Traces(L) \bullet msg(e) = \textbf{new}\ C$
  (starts by creating some actor of class C), and
- $\forall e \cdot t \cdot e' \in Traces(L) \bullet \forall C' \in \mathbf{CL} \bullet$
  $msg(e') = \textbf{new}\ C' \implies callee(e') \in L$
  (all created actors afterwards are local).

By this definition, an actor that never creates another actor is a component. As with a single actor, when $\surd$ is present in a trace of a component, it indicates that the component receives no more input from the environment and finishes processing all input events.

Example 3.3. A server actor s is not a component because it may create worker actors, just as with a single worker actor. The set of worker actors $\mathsf{w}_1, \mathsf{w}_2, \ldots$ created for computing a task forms a component, because, by definition, all creations remain in this set. Combining s with the set of worker actors also produces a component.

Composing the traces of individual actors that form a component retains the validity property of the resulting trace set as shown in the following lemma. The main reason the validity holds is that we define the plain trace set to be the largest set of possible traces.

Lemma 3.2 (Component plain trace set validity). *Let $Traces(L)$ be a plain trace set of a component L. Then, $Traces(L)$ is valid.*

Proof (sketch). 1. Prefix-closedness holds because we consider the largest possible trace set. Thus, we also take the largest possible subset of the actor traces that build up the component. If we take the last event in some trace $t \in Traces(L)$ away, the projected trace to the affected actor(s) is also contained in the set of actor traces, because they are also prefix-closed.
2. Holds by the non-external property of each actor trace set.
3. Holds by the definition of component.
4. Holds from the same property of each actor trace set.
5. Holds by the definition of $Traces(L)$.

□

Plain composition is not the ideal semantical representation for components because it reveals all internal events. To abstract away from all these internal events, we *box* the components. This means that all internal events become hidden. This characterization allows using the component without having to care about the internal details and simply focus on what happens on its boundary. In other words, only the interface of the component is of importance. The hiding is done by projecting away events that do not involve foreign actors.

Definition 3.5 (Boxed component). Let L be a component and $F = \mathbf{A} - L$. The *boxed* component of L, denoted by $[L]$, is the trace set $Traces([L])$ where

$$Traces([L]) = \{t\downarrow_F \mid t \in Traces(L)\}\ .$$

We refer to a trace in $Traces([L])$ as a boxed component trace. Given that the component L has an initial actor of class C, then $[\![[C]]\!]$ denotes trace set $Traces([L])$. Boxing a component does not affect its validity as shown by the following lemma.

Lemma 3.3 (Boxed component trace set validity). *Let L be a component. The trace set $Traces([L])$ is valid.*

Proof (sketch). 1. Projection does not affect prefix-closedness property.
2. Projection does not add new events into the trace set.
3. The initial creation property remains after projection because the caller is a foreign actor.
4. Internal events does not provide exposure to the environment, thus projecting them away does not affect local actor exposure to foreign actors.
5. Projection does not remove any input events from the trace.

□

Boxing a single actor component does not change the trace set because all events appearing in the trace set are not internal.

Lemma 3.4. *Let $L = \{a\}$ be a component. Then $Traces(L) = Traces([L])$.*

Proof (sketch). Because self call events are ruled out, all events in some trace $t \in Traces(L)$ are either input or output. Therefore, it will not disappear after projection. □

If we know that a trace set T is a trace set of a boxed component L, then we can derive $L' \subseteq L$ based on the name transfer that happens within the traces. This derivation, denoted by $ext(T)$, short for name extraction, is made by collecting the names that are exposed through actor creation, method call parameters and callee of a method call event. The name extraction becomes useful when we want to split the trace of a boxed component into input and output traces.

Definition 3.6 (Name extraction). Let T be a (valid) trace set of some boxed component. $L' = ext(T)$ is the subset of actors in the component, where $ext(T) = \bigcup_{t \in T} ext(t)$, $ext([]) = \emptyset$, and

$$ext(t \cdot e) = \begin{cases} ext(t) \cup \{callee(e)\}\ , \text{ if } msg(e) = \textbf{new}\ C \\ ext(t) \cup \{caller(e)\} \cup acq(e) - (acq(t) - ext(t))\ , \\ \qquad\qquad \text{if } isMtd(msg(e)) \wedge callee(e) \notin ext(t) \end{cases}$$

The local actor name extraction of T is done by examining each trace t in T and combining the result of each examination. If t ends with a creation event e, then the callee is part of the local name. By definition of the boxed component, there is exactly one creation event visible in any trace of T which is the creation of the initial actor of the component. If t ends with a method call and it is directed to some foreign actor, then the caller of this event and all non-foreign actor names in the method call arguments are included.

Hiding all internal events of a component trace is at times too strict, especially when we want to know the interaction between the component's subcomponents. If we know how the component is structured, we may allow internal events between these entities to appear in the trace set of the component. This way of composing subcomponents and actors into a component is called glass box composition [9, p. 5].

Definition 3.7 (Glass box composition).
Let $L = L_1 \cup \ldots \cup L_n \cup \{a_1, \ldots, a_m\}$ be a component such that $L_1, \ldots, L_n$ are components and $a_1, \ldots, a_m$ are actor names where $L_1, \ldots, L_n, \{a_1\}, \ldots, \{a_m\}$ are pairwise disjoint. Let $F = \mathbf{A} - L$, $a \in \{a_1, \ldots, a_m\}$ and $C = class(a)$. The *glass box composition* of L, denoted $\langle L_1|\ldots|L_n|a_1|\ldots|a_m\rangle$, is the largest trace set $T = Traces(\langle L_1|\ldots|L_n|a_1|\ldots|a_m\rangle)$ where

- $\forall t \in T \bullet (\forall L_i \bullet t{\downarrow_{L_i}} \in Traces([L_i]) \wedge \forall a_i \bullet t{\downarrow_{\{a_i\}}} \in Traces(\{a_i\}))$ (all traces can be projected to all elements of L),
- $\forall e \cdot t \in T \bullet callee(e) = a \wedge caller(e) \in F \wedge msg(e) = \texttt{new}\ C$ (a is the initial actor),
- $\forall e \cdot t \cdot e' \in T \bullet msg(e') = \texttt{new}\ C \implies caller(e') \in L \wedge callee(e') \in L$ (all other creation messages create local actors), and
- $\forall e \cdot t \cdot e' \in T \bullet caller(e') \in F \wedge isMtd(msg(e')) \implies \{callee(e')\} \cup acq(e') \subseteq acq(t{\downarrow_{F,callee}}) \cup cr(e)$ (the environment uses only exposed local actors).

Restrictions in terms of creation are applied to the actors and components that are composed, because the resulting composition should be a component. Each actor or component must be created by some local actor except for the initial actor a_m. As with the plain trace set definition, traces where name exposure property is not preserved must be excluded in order to maintain validity. Composing components and actors in a glass box manner produces a valid trace set.

Lemma 3.5 (Glass box trace set validity).
Let $L_1, \ldots, L_n, a_1, \ldots, a_m$ fulfill Def. 3.7.
Then, $Traces(\langle L_1|\ldots|L_n|a_1|\ldots|a_m\rangle)$ is valid.

Proof. Similar to the proof for Lemmas 3.2 and 3.3. □

Deriving the black box semantics of a glass box composition is done by projecting the trace set to the foreign actors.

Lemma 3.6 (Boxing glass box component). *Let component $L = L_1 \cup \ldots \cup L_n \cup \{a_1, \ldots, a_m\}$ and $F = \mathbf{A} - L$ fulfill Def. 3.7. Then, $Traces(\langle L_1|\ldots|L_n|a_1|\ldots|a_m\rangle){\downarrow_F} = Traces([L])$.*

Proof. Follows directly from Def. 3.5. □

4. Specification Technique

Hoare advocates that to specify the functional behavior of a program is to specify the connection between the input and output of the program [17]. This approach is extended by Broy [8] to deal with components by letting the input and output be streams of events. Here we adopt their approaches to specify the functional behavior of actor components by letting the specification be a set of triples (similar to Hoare triple) where the pre- and postconditions are described using assertions on traces. As in Hoare logic, triples have the form

$$\{p\}\ D\ \{q\}$$

where p and q are input and output trace assertions, respectively, and D is either C or $[C]$. We call $\{p\}\ C\ \{q\}$ an *actor triple*, where as $\{p\}\ [C]\ \{q\}$ is called a *component triple*.

The trace assertions are first-order logic formulas that can use a special constant \$ called the *trace constant*. A trace assertion is checked against some variable assignment and input/output trace, and every occurrence of \$ is replaced by the trace.[2] The input and output traces are obtained from a valid actor trace by filtering the input and output events. Different to Hoare logic, where the second part of the triple is usually the program or the implementation of some sort, here we only have the name of the entity we represent. Nevertheless, knowing whether the entity is a boxed component or an actor class allows us to link the specification with the correct kind of trace semantics.

Before going into more details about the syntax and semantics, we motivate the specification technique by specifying our server example. To distinguish the logical variables appearing in the specification from the program variables, the logical variables will be underlined.

4.1 Specifying the Running Example

In this subsection, we illustrate how the server and worker actor and components described in Sect. 2 behave when a single request comes from a client. To save space, the following abbreviations are employed. The `Server` class is abbreviated as `Srv`, `Worker` as `Wrk`, `serve` as `sv`, `propagateResult` as `pR`, `response` as `resp`, `taskSize` as `sz`, `firstTask` as `fst`, `restTask` as `rst`, `compute` as `cmp` and `merge` as `mrg`.

The following specification of the server class states that when a server is created and a request comes, the server creates a new worker and passes the worker the task and tells it to start propagating the result.

$\{\ \$ = (\underline{\texttt{this}} := \texttt{new Srv} \cdot \underline{\texttt{this}}.\texttt{sv}(\underline{\texttt{c}},\underline{\texttt{t}}) \cdot \surd)\ \}$
$\quad\texttt{Srv}$
$\{\ \exists \underline{\texttt{w}} \bullet \$ = (\underline{\texttt{w}} := \texttt{new Wrk} \cdot \underline{\texttt{w}}.\texttt{do}(\underline{\texttt{t}}) \cdot \underline{\texttt{w}}.\texttt{pR}(\texttt{null},\underline{\texttt{c}}) \cdot \surd)\ \}$

For the server case above, the input trace assertion restricts the behavior to cases in which the environment creates the server and sends a single `sv` message. The server actor processes the input by creating a new worker, passing the task to the newly created worker and starting result propagation before stopping. When the input trace assertion is not satisfied by the input trace, the behavior of the server is unspecified. For example, we do not know what the server does when it receives more than one `sv` request. As can be seen in the specification, the trace constant is used by comparing it to the sequence of the content of the events, that is, the pairs of callee and message. We are not concerned with the caller part of the event because the trace is an actor trace. By using this comparison technique, we specify exactly what the server does when it receives the exact input that is stated in the input trace assertion. As standard in Hoare logic, any logical variable that appears only in the output trace assertion needs to be existentially quantified.

When we consider the server component as a whole, we would like to see that a request from the client is replied by a response to the client with the computed result. This requirement is represented using the specification below.

$\{\ \$ = (\underline{\texttt{this}} := \texttt{new Srv} \cdot \underline{\texttt{this}}.\texttt{sv}(\underline{\texttt{c}},\underline{\texttt{t}}) \cdot \surd)\}$
$\quad[\texttt{Srv}]$
$\{\ \$ = (\underline{\texttt{c}}.\texttt{resp}(\texttt{cmp}(\underline{\texttt{t}})) \cdot \surd)\}$

The name `[Srv]` indicates that the triple deals with a boxed

[2] To be exact, \$ is replaced by the event content sequence of the trace, where an event content is an event without the caller information. This is done because each actor/component should be oblivious to who the caller of an event is.

server component. That is, the assertions are checked against a boxed component trace with an initial actor of class $\mathtt{Srv}$.

Specifying the worker class motivates why we may have more than one triple that represents the functional behavior of a worker. More precisely, the worker class is described using two specification triples, each handling the base and inductive cases, respectively. The first specification triple handles the case when the task has exactly one subtask. In this particular case, the worker sends back to the client the result of merging the propagated result value with the computation of the subtask.

$\{\ \$ = (\underline{\mathtt{this}} := \mathbf{new}\ \mathtt{Wrk} \cdot \underline{\mathtt{this}}.\mathtt{do}(\underline{\mathtt{t}}) \cdot \underline{\mathtt{this}}.\mathtt{pR}(\underline{\mathtt{v}}, \underline{\mathtt{c}}) \cdot \surd) \wedge \mathtt{sz}(\underline{\mathtt{t}}) = 1\ \}$
$\quad\mathtt{Wrk}$
$\{\ \$ = (\underline{\mathtt{c}}.\mathtt{resp}(\mathtt{mrg}(\underline{\mathtt{v}}, \mathtt{cmp}(\underline{\mathtt{t}}))) \cdot \surd)\ \}$

In the second case, the task consists of multiple subtasks. In this case, the worker creates another worker to pass on the rest of the task while processing the current subtask. When the computation of the current subtask is finished, the worker merges the computation result with the previous result it receives and propagates the merged result to the worker it created.

$\{\ \$ = (\underline{\mathtt{this}} := \mathbf{new}\ \mathtt{Wrk} \cdot \underline{\mathtt{this}}.\mathtt{do}(\underline{\mathtt{t}}) \cdot \underline{\mathtt{this}}.\mathtt{pR}(\underline{\mathtt{v}}, \underline{\mathtt{c}}) \cdot \surd) \wedge$
$\qquad \mathtt{sz}(\underline{\mathtt{t}}) = \underline{\mathtt{n}} \wedge \underline{\mathtt{n}} > 1\}$
$\quad\mathtt{Wrk}$
$\{\ \exists \underline{\mathtt{w}} \bullet \$ = (\underline{\mathtt{w}} := \mathbf{new}\ \mathtt{Wrk} \cdot \underline{\mathtt{w}}.\mathtt{do}(\mathtt{rst}(\underline{\mathtt{t}})) \cdot$
$\qquad\qquad \underline{\mathtt{w}}.\mathtt{pR}(\mathtt{mrg}(\underline{\mathtt{v}}, \mathtt{cmp}(\mathtt{fst}(\underline{\mathtt{t}}))), \underline{\mathtt{c}}) \cdot \surd)\}$

This concludes the worker class specification for our example.

If we box the worker class, then we obtain a component whose members are exactly the set of workers needed to process a task. Its specification is exactly as that of the worker class, but instead of splitting the tasks into subtasks, the worker group evaluates the whole task in a chunk and returns the computation result merged with the previous result to the client.

$\{\ \$ = (\underline{\mathtt{this}} := \mathbf{new}\ \mathtt{Wrk} \cdot \underline{\mathtt{this}}.\mathtt{do}(\underline{\mathtt{t}}) \cdot \underline{\mathtt{this}}.\mathtt{pR}(\underline{\mathtt{v}}, \underline{\mathtt{c}}) \cdot \surd)\}$
$\quad[\mathtt{Wrk}]$
$\{\ \$ = (\underline{\mathtt{c}}.\mathtt{resp}(\mathtt{mrg}(\underline{\mathtt{v}}, \mathtt{cmp}(\underline{\mathtt{t}}))) \cdot \surd)\}$

4.2 Syntax and Semantics

Triples use *trace assertions* to formulate input and output conditions. A trace assertion is a first-order logic formula in which the special trace constant $\$$ can be used. In the input (output) condition, $\$$ represents the event content sequence of the input (output) trace. We assume that there are functions and predicates over traces that can be used in trace assertions. For the purposes of this paper, we only need an equality comparison operation, written $\$ = ec$, that compares the result with a sequence of event contents ec, as seen in the previous subsection. Event contents are used instead of events because from an actor's point of view, the origin of the events is not known unless it is the actor who is initiating them. Thus, the caller of the event does not play a role when we want to specify the behavior of a component. The main idea of the equality comparison is that given an (input or output) trace, there is a mapping of the variables in ec to data and actor names such that stripping this trace of the caller information yields a match to the mapped ec. We require that any event content comparison must end with $\surd$ to ensure only maximal traces are compared. When a non-maximal trace is compared, the information whether the actor or component has finished with the tasks is lacking. It is possible that the actor or component responses with less or more events.

Definition 4.1 (**Trace assertions**)**.** Let $\$$ be a trace constant representing a trace. *Trace assertions* p, q are defined inductively by the following first-order logic clauses:

- Boolean expressions, where $\$$ may be present, are assertions.
- If p, q are assertions and $\underline{x}$ is a variable, then $\neg p, p \wedge q, \exists \underline{x} : p$ are also assertions.

The other logical operators, such as $\vee$, $\implies$ and $\forall$, are derived in the usual way. Given a trace assertion p, the function $free(p)$ extracts the set of all free variables appearing in p.

To define the semantics of a trace assertion, the variables must be assigned to some values. Let $\mathbf{V}$ be the set of all variables. A *variable assignment* $\sigma : \mathbf{V} \to \mathbf{A} \cup \mathbf{D}$ is a function that maps (some) variables to values.

The semantics of a trace assertion p with respect to a variable assignment σ and a trace t is a mapping

$$[\![p]\!] : (\mathbf{V} \to \mathbf{A} \cup \mathbf{D}) \times Seq\langle E \cup \{\surd\}\rangle \to \{\mathbf{true}, \mathbf{false}\}\ .$$

We write $p(\sigma, t)$ if $[\![p]\!](\sigma, t) = \mathbf{true}$. This mapping of a trace t is similar to the standard first-order logic interpretation based on states (see, e.g., Apt, de Boer and Olderog [4]). Occurrences of $\$$ are replaced directly by the event content sequence of t. The equality comparison operator $\$ = ec$ can be formulated in first-order logic by using σ to replace all the variables in ec before comparing it with the stripped t. Because of the assumption that creation events in the trace always yields a fresh name, this freshness aspect does not need to be handled explicitly by the semantics of the trace assertion.

Substitution of all free occurrences of a variable $\underline{x}$ by some expression or assertions r in a trace assertion p is denoted by $p[\underline{x}/r]$. We assume that all variables and all substitutions are correctly typed.

As seen in the example, a *specification triple* $\{p\}\ D\ \{q\}$ consists of the trace assertions p and q and some entity name D, which is either some actor class name C or a boxed component with an initial actor of class C.[3] We call p and q input and output trace assertions, respectively. All variables that only appear in q (possibly due to an explicit creation of another actor or an implicit exposure of locally created actors) must be existentially quantified. As convention, the initial actor created is referred to by the variable $\underline{\text{this}}$. The specification triple $\{p\}\ D\ \{q\}$ partially characterizes the trace semantics of the entity represented by D. Partial means that for each trace $t \in [\![D]\!]$ whose input part satisfies p, its output part satisfies also q. This specification technique does not give information about the rest of the traces that do not satisfy p. Despite the underspecification, the specification triple eliminates traces which satisfy p and do not satisfy q.

An actor triple of class C enforces that an actor of class C is created and the environment can only call methods of this actor. The restriction given in the definition above is not enough to ensure that indeed only a single actor is considered local. Consider a trace assertion $q \equiv \mathbf{true}$. A group of actors whose initial actor is the only exposed actor of the group can produce traces that matches the specification. By comparing the specification with a real trace semantics of the actor whose characteristics are described in Def. 3.2, we avoid this problem. To define the semantics of the actor triple, a trace t needs to be split into input and output traces. The function $split(t, L) = (t{\downarrow_{F,caller}}{\downarrow_{L,callee}}, t{\downarrow_{L,caller}}{\downarrow_{F,callee}})$ does exactly so, where $F = \mathbf{A} - L$.

Definition 4.2 (**Actor triple semantics**)**.** Let $[\![C]\!]$ be a trace set satisfying Def. 3.2 and representing the trace semantics of some actor a of class C. $[\![C]\!]$ satisfies $\{p\}\ C\ \{q\}$, written $\vDash \{p\}\ C\ \{q\}$, if for all trace $t \in [\![C]\!]$ with $split(t, \{a\}) = (ti, to)$, the following holds:

$$\forall \sigma \bullet p(\sigma, ti) \implies q(\sigma, to)\ .$$

[3] In the classical Hoare logic [17], p and q are said to be the *precondition* and *postcondition* of the triple, respectively. However, the specification in this paper deals with input and output traces, which may include passing of a (new) actor name in an output event that later appears as the callee of an input event. In this sense, p and q are not pre- and postconditions.

A component triple $\{p\}\ [C]\ \{q\}$ states how the component with an initial actor of class C replies to a given sequence of input events. The semantics of a component triple is compared to the appropriate boxed component trace set and has the same form as for the actor triple. Thus the traces need to be split into input and output traces. This splitting can be done using the help of the function $ext(t)$ from Def. 3.6. The following definition covers the semantics of boxed component triples.

Definition 4.3 (Component triple semantics). Let $[\![[C]]\!]$ be a trace set satisfying Def. 3.5 that represents the trace semantics of component with an initial actor of class C. $[\![[C]]\!]$ satisfies $\{p\}\ [C]\ \{q\}$, written $\vDash \{p\}\ [C]\ \{q\}$, if for all trace $t \in [\![[C]]\!]$ with $split(t, ext(t)) = (ti, to)$, the following holds:

$$\forall \sigma \bullet p(\sigma, ti) \implies q(\sigma, to)\ .$$

Given triples as defined above, we can state the specification of an actor class or a boxed component as a set of such triples. A proper trace set representing the actual behavior of the class or the component must satisfy each of the specification triples.

Definition 4.4 (Specification). Let D be an actor class C or a boxed class representing a boxed component $[C]$. A specification for D is a set of specification triples

$$S = \{\{p_1\}\ D\ \{q_1\}, \dots, \{p_n\}\ D\ \{q_n\}\}\ .$$

$[\![D]\!]$ satisfies S, written $\vDash S$, if $\forall(\{p_i\}\ D\ \{q_i\}) \in S \bullet \vDash \{p_i\}\ D\ \{q_i\}$.

5. Proof System

The specification technique described in the previous section allows us to focus on interesting functional properties of actor-based components and systems. Unlike the standard Hoare logic, where a primitive program statement (i.e., the second element of the triple) holds the basis how the assertions can evolve, we only have the information of an actor class name and its boxed status. In the proof system, we use the actor class specifications as axioms assuming that they are satisfied by the implementation. This assumption allows us to focus on proving the component specifications. To be able to prove the component specifications from the actor class specifications, we need to come up with proof rules based on the trace semantics given in Sect. 3. In this paper, we provide a proof system called Relational Proof System for Actors (*RPSA*) that can handle daisy chain composition. By daisy chain composition, we mean that an actor that creates another actor, sends messages to this newly created actor, never exposes its own name to the newly created actor nor the name of the newly created actor to other actors forming a chain of one way interaction.

RPSA presented in Fig. 5 (with the helper predicates stated in Fig. 6) contains an axiom and a number of rules. A *rule* consists of a number of *premises* and a specification triple as *conclusion*. The premises are trace assertions or specification triples. A rule allows to prove the conclusion from the premises. A rule with no premise is called an *axiom* (that is, the conclusion is assumed to be true). To ensure that we obtain the correct conclusions, each rule must be proven to be *sound*. Sound means that when the premises are assumed, then using the semantics of the assertions and the specifications we can argue for the conclusion.

In context of a proof system, a *proof* of a component triple is a sequence of proof rule applications. This sequence of applications are represented using a proof tree, where each node contains the specification triples and assertions that hold and the edge is labeled with the proof rules that is used.

RPSA also includes standard auxiliary rules similar to what is done by Apt, de Boer, and Olderog [4] and Poetzsch-Heffter [24]. The auxiliary rules we need for our example are given in

ActorTripleAxiom

$$\{p\}\ C\ \{q\}$$

Consequence

$$\frac{p \implies p_1 \qquad \{p_1\}\ D\ \{q_1\} \qquad q_1 \implies q}{\{p\}\ D\ \{q\}}$$

Boxing

$$\frac{\{p\}\ C\ \{q \wedge nonCr(\$)\}}{\{p\}\ [C]\ \{q\}}$$

BoxedComposition

$$\frac{\{p \wedge \underline{i} = \$\}\ C\ \{q \wedge noSelfExp(\underline{i}, \$)\} \qquad \{q'\}\ D\ \{r \wedge nonCr(\$)\} \qquad match(q, q', D)}{\{p\}\ [C]\ \{r\}}$$
where $\underline{i} \notin free(p) \cup free(q)$

Induction

$$\frac{\{p \wedge m = 0\}\ [C]\ \{q\} \qquad match(p', p, C) \qquad \{p \wedge m = \underline{z} \wedge m > 0 \wedge \underline{i} = \$\}\ C\ \{p' \wedge m < \underline{z} \wedge noSelfExp(\underline{i}, \$)\}}{\{p\}\ [C]\ \{q\}}$$
where $\underline{i} \notin free(p) \cup free(q)$

Figure 5. Proof rules for *RPSA*

$$noSelfExp(i, \$) \overset{\text{def}}{=} \forall c \cdot ec \in Pref(i) \bullet callee(c) \notin acq(\$)$$

$$nonCr(\$) \overset{\text{def}}{=} \forall ec' \cdot e \in Pref(\$), C' \in \mathbf{CL} \bullet msg(e) \neq \texttt{new}\ C'$$

$$match(q, q', D) \overset{\text{def}}{=} q \implies \exists a \in \mathbf{A} \bullet firstCreated(a, \$) \wedge classOf(a, D) \wedge q'$$

Figure 6. Helper predicates for *RPSA*

Invariance

$$\frac{\{p\}\ D\ \{q\}}{\{p \wedge r\}\ D\ \{q \wedge r\}}$$
where $consFree(r)$

Substitution

$$\frac{\{p\}\ D\ \{q\}}{\{p[\underline{x}/r]\}\ D\ \{q[\underline{x}/r]\}}$$
where $x \in free(p) \cup free(q)$ and $consFree(r)$

Figure 7. Auxiliary rules for *RPSA*

Fig. 7. Invariance allows to add conjuncts that do not refer to the trace constant. The predicate $consFree(p)$ checks that there are no occurrences of \$ in p. Substitution allows substitutions of free variables x to some assertion or expression r that does not contain the trace constant. Except for the standard Consequence rule and ActorTripleAxiom (which should be checked against the implementation), all rules in *RPSA* are explained in more details along side their application in the running example.

5.1 Verifying Example Component Specifications

In Sect. 4, specifications for server and worker actor classes are given and we assume that they are correct. The specifications for server and worker components are also stated, but they are left as proof obligations. The server component is built by composing the worker component with the server actor. Thus BoxedComposition is used to verify the server component specification. The worker component is built by composing as many worker actors as needed to complete the task that is given to the worker component. To achieve the verification of this component, Induction is utilized.

To focus more on the proof rules and their usage, the event content equality assertions are abbreviated as follows.

- $\mathsf{InSrv} \overset{\text{def}}{=} \$ = (\underline{\mathsf{this}} := \texttt{new Srv} \cdot \underline{\mathsf{this}}.\mathsf{sv}(\underline{\mathsf{c}}, \underline{\mathsf{t}}) \cdot \surd)$

- $\mathsf{OutSrv} \overset{\text{def}}{=} \$ = (\underline{w} := \mathbf{new}\ \mathtt{Wrk} \cdot \underline{w}.\mathtt{do}(\underline{t}) \cdot \underline{w}.\mathtt{pR}(\mathbf{null}, \underline{c}) \cdot \surd)$
- $\mathsf{InWrk} \overset{\text{def}}{=} \$ = (\underline{\text{this}} := \mathbf{new}\ \mathtt{Wrk} \cdot \underline{\text{this}}.\mathtt{do}(\underline{t}) \cdot \underline{\text{this}}.\mathtt{pR}(\underline{v}, \underline{c}) \cdot \surd)$
- $\mathsf{OutWrkP} \overset{\text{def}}{=} \$ = (\underline{c}.\mathtt{resp}(\mathtt{mrg}(\underline{v}, \mathtt{cmp}(\underline{t}))) \cdot \surd)$
- $\mathsf{OutWrkI} \overset{\text{def}}{=} \$ = (\underline{w} := \mathbf{new}\ \mathtt{Wrk} \cdot \underline{w}.\mathtt{do}(\mathtt{rst}(\underline{t})) \cdot$
  $\underline{w}.\mathtt{pR}(\mathtt{mrg}(\underline{v}, \mathtt{cmp}(\mathtt{fst}(\underline{t}))), \underline{c}) \cdot \surd)$
- $\mathsf{InSrvC} \overset{\text{def}}{=} \$ = (\underline{\text{this}} := \mathbf{new}\ \mathtt{Srv} \cdot \underline{\text{this}}.\mathtt{sv}(\underline{c}, \underline{t}) \cdot \surd)$
- $\mathsf{OutSrvC} \overset{\text{def}}{=} \$ = (\underline{c}.\mathtt{resp}(\mathtt{cmp}(\underline{t})) \cdot \surd)$

The name are picked such that InSrv, for example, represents the input event content equality of the server actor triple, where as OutWrkI represents the output event content euality of the worker actor triple in the inductive case. The C suffix indicates the assertion is used in a component triple. Note that the event content equality assertions in both worker actor triples and the worker component triple are the same, i.e., InWrk. In addition, the event content equality assertions for the worker actor triple of the primitive case and the worker component triple is the same, i.e., OutWrkP. Let *cse*, short for *c*ontent *s*equence *e*xtractor, be a function that extracts the event content sequences from these abbreviations.

Server Component. We start with proving the server component triple. The intention is to compose the server actor with the worker component. The rule that accomodates this composition is BoxedComposition.

The BoxedComposition rule defines how an actor of class C can be combined with another actor or boxed component D to create boxed component $[C]$. For this rule to be applicable, three premises must hold. First, the actor triple C must guarantee that the actor's name will not be exposed.

$$noSelfExp(i, \$) \overset{\text{def}}{=} \forall c \cdot ec \in Pref(i) \bullet callee(c) \notin acq(\$)$$

The predicate *noSelfExp*, short for *no self exposure*, takes variable i and the trace constant \$ representing the input and output traces, respectively. It guarantees a one way interaction, or in other words, ensures daisy chaining because the current actor is not exposed. To ensure no exposure of an actor is made, the acquaintance of the output trace must not contain that actor. Second, the triple of D must ensure that no foreign actor is created by the instance of D. The predicate *nonCr* does exactly that.

$$nonCr(\$) \overset{\text{def}}{=} \forall ec' \cdot e \in Pref(\$), C' \in \mathbf{CL} \bullet msg(e) \neq \mathbf{new}\ C'$$

Third, the output produced by the actor of class C must match the input of the instance of D. In other words, the actor of class C exclusively feeds the instance of D in this particular case. This matching is handled by predicate *match*.

$$match(q, q', D) \overset{\text{def}}{=} q \implies \exists a \in \mathbf{A} \bullet firstCreated(a, \$) \wedge classOf(a, D) \wedge q'$$

The predicate *firstCreated* checks if the first event is an actor creation and a represents the created actor. The predicate *classOf* checks if the created actor is of class D. The *match* predicate relies on the valid trace restriction (see Def. 3.1) where a trace starts with an actor creation. This restriction applies because the evaluation of q' is done against an input trace, which always starts with an actor creation. For *match* to hold, the free variables of q and q' should coincide. Note that because *match* is only used to link output trace assertion q to input trace assertion q', there is no need to explicitly check that q represents an output trace assertion.

Neither the server actor triple nor the worker component triple is in the form needed to apply BoxedComposition. Therefore, we need to transform these triples using the Consequence, Invariance and Substitution rules.

$$\textsc{Cns}\ \dfrac{\textsc{Inv}\ \dfrac{\textsc{Axiom}\ \dfrac{}{\{\mathsf{InSrv}\}\ \mathtt{Srv}\ \{\exists \underline{w} \bullet \mathsf{OutSrv}\}}}{\{\mathsf{InSrv} \wedge \underline{i} = cse(\mathsf{InSrv})\}\ \mathtt{Srv}\ \{\exists \underline{w} \bullet \mathsf{OutSrv} \wedge \underline{i} = cse(\mathsf{InSrv})\}} \quad \begin{array}{c}\mathsf{InSrv} \wedge \underline{i} = \$ \implies \mathsf{InSrv} \wedge \underline{i} = cse(\mathsf{InSrv}) \\ \exists \underline{w} \bullet \mathsf{OutSrv} \wedge \underline{i} = cse(\mathsf{InSrv}) \implies \exists \underline{w} \bullet \mathsf{OutSrv} \wedge noSelfExp(\underline{i}, \$)\end{array}}{\{\mathsf{InSrv} \wedge \underline{i} = \$\}\ \mathtt{Srv}\ \{\exists \underline{w} \bullet \mathsf{OutSrv} \wedge noSelfExp(\underline{i}, \$)\}}$$

By Invariance, a logical variable $\underline{i}$ is introduced to store the input trace. Because $\underline{i}$ cannot directly refer to \$, we first extract the event content sequence from the event content comparison InSrv. By Consequence, the input trace assertion is strengthened by having $\underline{i}$ to indeed be the input trace. At the same time, the output trace assertion is enriched with *noSelfExp*.

$$\textsc{Cns}\ \dfrac{\textsc{Sub}\ \dfrac{\{\mathsf{InWrk}\}\ [\mathtt{Wrk}]\ \{\mathsf{OutWrkP}\}}{\{\mathsf{InWrk}[\underline{v}/\mathbf{null}]\}\ [\mathtt{Wrk}]\ \{\mathsf{OutWrkP}[\underline{v}/\mathbf{null}]\}} \quad \mathsf{OutWrkP}[\underline{v}/\mathbf{null}] \implies \mathsf{OutSrvC} \wedge nonCr(\$)}{\{\mathsf{InWrk}[\underline{v}/\mathbf{null}]\}\ [\mathtt{Wrk}]\ \{\mathsf{OutSrvC} \wedge nonCr(\$)\}}$$

The worker component triple needs to be adjusted so it is ready to receive the input from the server actor. By substituting the variable $\underline{v}$ with $\mathbf{null}$ ensures that we can match the output of the server actor to the input of the worker component. From the assumption in Sect. 2, we know that $\mathtt{mrg}(\mathbf{null}, \mathtt{cmp}(\underline{t})) = \mathtt{cmp}(\underline{t})$. Therefore, we can infer OutSrvC from $\mathsf{OutWrkP}[\underline{v}/\mathbf{null}]$. The output trace assertion is enhanced by *nonCr* to state that the output of the worker component creates no other actors, which holds from the definition of components (Def. 3.4). Now we are ready to apply BoxedComposition to obtain that the server component specification holds.

$$\textsc{Cmp}\ \dfrac{\begin{array}{c}\{\mathsf{InSrv} \wedge \underline{i} = \$\}\ \mathtt{Srv}\ \{\exists \underline{w} \bullet \mathsf{OutSrv} \wedge noSelfExp(\underline{i}, \$)\} \\ \{\mathsf{InWrk}[\underline{v}/\mathbf{null}]\}\ [\mathtt{Wrk}]\ \{\mathsf{OutWrkP}[\underline{v}/\mathbf{null}] \wedge nonCr(\$)\} \\ match(\exists \underline{w} \bullet \mathsf{OutSrv}, \mathsf{InWrk}[\underline{v}/\mathbf{null}], \mathtt{Wrk})\end{array}}{\{\mathsf{InSrvC}\}\ [\mathtt{Srv}]\ \{\mathsf{OutSrvC}\}}$$

When a server actor receives a single request, we can match the output of the server actor to the input of the worker component. Thus the last premise of BoxedComposition is handled and we can derive the server component specification.

Worker Component. To prove the worker component triple, we can apply the Induction rule. Similar to the server component case, the worker actor triples need to be massaged before the Induction rule can be applied.

The Induction rule deals if a component that creates as many instances of itself as needed to solve the task it has to do. This rule relies on having a measure expression m depending on the event content sequence. The base and inductive cases are represented by the measure comparison $m = 0$ and $m > 0$, respectively, as mentioned.

If the measure yields zero, the actor on its own must represent the behavior of a component. That is, it does not create any other actor. For the inductive case, we see how the initial actor behaves. If from the actor specification it creates another actor of the same class and passes on a similar input to this new actor with the measure being reduced, this means we could apply the same specification again and again until we end in the base case. The reduction in the measure is captured by $\underline{z}$, a variable that does not appear in other parts of the corresponding triple. The *match* predicate enforces this behavior. As in BoxedComposition, *noSelfExp* ensures that no self exposure is done.

When these premises are fulfilled, then the component triple holds.

For the worker component, let the function gt get the task parameter from an event content sequence (including \$); in other words, gt refers to the task parameter of the $\mathtt{do}$ method. Thus, we can define the measure m as $\mathtt{sz}(gt(\$)) - 1$.We apply directly arithmetic manipulation to any boolean expressions to improve the presentation.

We begin with the worker actor that receives only a primitive task $\underline{\mathsf{t}}$ (i.e., $\mathtt{sz}(\underline{\mathsf{t}}) = 1$). In this particular case the goal is to box the worker actor for creating no other actors. Boxing captures exactly this intention.

$$\textsc{Box}\ \dfrac{\textsc{Cns}\ \dfrac{\begin{array}{c}\textsc{Axiom}\ \dfrac{}{\{\mathsf{InWrk} \wedge \mathtt{sz}(gt(\$)) = 1\}\ \mathtt{Wrk}\ \{\mathsf{OutWrkP}\}} \\ \mathsf{OutWrkP} \implies \mathsf{OutWrkP} \wedge nonCr(\$)\end{array}}{\{\mathsf{InWrk} \wedge \mathtt{sz}(gt(\$)) = 1\}\ \mathtt{Wrk}\ \{\mathsf{OutWrkP} \wedge nonCr(\$)\}}}{\{\mathsf{InWrk} \wedge \mathtt{sz}(gt(\$)) = 1\}\ [\mathtt{Wrk}]\ \{\mathsf{OutWrkP}\}}$$

When the worker deals with a non-primitive task, the worker follows the inductive case of the worker actor triple. To achieve the second premise of Induction, the output trace assertion must include the information that the measure is reduced. In our case, we know that $\mathtt{sz}(\mathtt{rst}(\underline{\mathsf{t}})) < \mathtt{sz}(\underline{\mathsf{t}})$, but we cannot extract the right task $\underline{\mathsf{t}}$ information from the trace constant of the output trace assertion. To get around this problem, we utilize the same approach to capture the event content sequence contained in the input trace assertion into a variable. By introducing $\mathtt{sz}(gt(cse(\mathsf{InWrk}))) = \underline{\mathsf{z}}$ to the output trace assertion using Invariance, we can weaken the output trace assertion to include the needed information. To include the no self exposure information in the output trace assertion, we follow the same approach to transform the worker component triple. In the proof tree below, we abbreviate $\mathtt{sz}(gt(\$)) = \underline{\mathsf{z}} \wedge \mathtt{sz}(gt(\$)) > 1$ as *ind*.

$$\textsc{Cns}\ \dfrac{\begin{array}{c}\textsc{Inv}\ \dfrac{\textsc{Cns}\ \dfrac{\begin{array}{c}\textsc{Inv}\ \dfrac{\textsc{Axiom}\ \dfrac{}{\{\mathsf{InWrk} \wedge ind\}\ \mathtt{Wrk}\ \{\exists \underline{\mathsf{w}} \bullet \mathsf{OutWrkI}\}}}{\begin{array}{c}\{\mathsf{InWrk} \wedge ind \wedge \underline{\mathsf{i}} = cse(\mathsf{InWrk})\}\ \mathtt{Wrk} \\ \{\exists \underline{\mathsf{w}} \bullet \mathsf{OutWrkI} \wedge \underline{\mathsf{i}} = cse(\mathsf{InWrk})\}\end{array}} \\ \mathsf{InWrk} \wedge \mathtt{sz}(gt(\underline{\mathsf{i}})) = \underline{\mathsf{z}} \wedge ind \wedge \underline{\mathsf{i}} = \$ \implies \\ \mathsf{InWrk} \wedge ind \wedge \underline{\mathsf{i}} = cse(\mathsf{InWrk}) \\ \exists \underline{\mathsf{w}} \bullet \mathsf{OutWrkI} \wedge \underline{\mathsf{i}} = cse(\mathsf{InWrk}) \implies \\ \exists \underline{\mathsf{w}} \bullet \mathsf{OutWrkI} \wedge \underline{\mathsf{i}} = cse(\mathsf{InWrk}) \wedge noSelfExp(\underline{\mathsf{i}}, \$)\end{array}}{\begin{array}{c}\{\mathsf{InWrk} \wedge \mathtt{sz}(gt(\underline{\mathsf{i}})) = \underline{\mathsf{z}} \wedge ind \wedge \underline{\mathsf{i}} = \$\}\ \mathtt{Wrk} \\ \{\exists \underline{\mathsf{w}} \bullet \mathsf{OutWrkI} \wedge \underline{\mathsf{i}} = cse(\mathsf{InWrk}) \wedge noSelfExp(\underline{\mathsf{i}}, \$)\}\end{array}}}{\begin{array}{c}\{\mathsf{InWrk} \wedge \mathtt{sz}(gt(\underline{\mathsf{i}})) = \underline{\mathsf{z}} \wedge ind \wedge \underline{\mathsf{i}} = \$\}\ \mathtt{Wrk} \\ \{\exists \underline{\mathsf{w}} \bullet \mathsf{OutWrkI} \wedge \underline{\mathsf{i}} = cse(\mathsf{InWrk}) \wedge noSelfExp(\underline{\mathsf{i}}, \$) \wedge \mathtt{sz}(gt(\underline{\mathsf{i}})) = \underline{\mathsf{z}}\}\end{array}} \\ \mathsf{InWrk} \wedge ind \wedge \underline{\mathsf{i}} = \$ \implies \mathsf{InWrk} \wedge \mathtt{sz}(gt(\underline{\mathsf{i}})) = \underline{\mathsf{z}} \wedge ind \wedge \underline{\mathsf{i}} = \$ \\ \exists \underline{\mathsf{w}} \bullet \mathsf{OutWrkI} \wedge \underline{\mathsf{i}} = cse(\mathsf{InWrk}) \wedge noSelfExp(\underline{\mathsf{i}}, \$) \wedge \mathtt{sz}(gt(\underline{\mathsf{i}})) = \underline{\mathsf{z}} \\ \implies \exists \underline{\mathsf{w}} \bullet \mathsf{OutWrkI} \wedge \mathtt{sz}(gt(\$)) < \underline{\mathsf{z}} \wedge noSelfExp(\underline{\mathsf{i}}, \$)\end{array}}{\begin{array}{c}\{\mathsf{InWrk} \wedge ind \wedge \underline{\mathsf{i}} = \$\}\ \mathtt{Wrk} \\ \{\exists \underline{\mathsf{w}} \bullet \mathsf{OutWrkI} \wedge \mathtt{sz}(gt(\$)) < \underline{\mathsf{z}} \wedge noSelfExp(\underline{\mathsf{i}}, \$)\}\end{array}}$$

Now that the base and inductive cases are covered, we can apply the Induction rule. Thus, the worker component triple holds.

$$\textsc{Ind}\ \dfrac{\begin{array}{c}\{\mathsf{InWrk} \wedge \mathtt{sz}(gt(\$)) = 1\}\ [\mathtt{Wrk}]\ \{\mathsf{OutWrkP}\} \\ \{\mathsf{InWrk} \wedge \mathtt{sz}(gt(\$)) = \underline{\mathsf{z}} \wedge \mathtt{sz}(gt(\$)) > 1 \wedge \underline{\mathsf{i}} = \$\}\ \mathtt{Wrk} \\ \{\exists \underline{\mathsf{w}} \bullet \mathsf{OutWrkI} \wedge \mathtt{sz}(gt(\$)) < \underline{\mathsf{z}} \wedge noSelfExp(\underline{\mathsf{i}}, \$)\} \\ match(\exists \underline{\mathsf{w}} \bullet \mathsf{OutWrkI}, \mathsf{InWrk}, \mathtt{Wrk})\end{array}}{\{\mathsf{InWrk}\}\ [\mathtt{Wrk}]\ \{\mathsf{OutWrkP}\}}$$

As the worker component specification is proved, the server component specification holds. The proof above depends on the *RPSA* being sound, which is addressed in the next subsection.

5.2 Soundness of *RPSA*

The soundness of *RPSA* is proven by induction on the depth of the proof trees. This means that each rule applied within the proof trees must be sound and the axiom is applied only when the actor triple is proved in the underlying system. The following lemmas show for all rules that the derived component's specifications are valid if the premises are valid.

Lemma 5.1 (Soundness of Consequence). *Let D be either an actor class C or a component with initial actor of class C. Suppose $\vDash \{p_1\}\ D\ \{q_1\}$, $p \implies p_1$ and $q_1 \implies q$. Then $\vDash \{p\}\ D\ \{q\}$.*

Proof. Follows from the first-order logic semantics. □

The soundness of Consequence follows from a straightforward manipulation of first-order logic.

Lemma 5.2 (Soundness of Boxing). *Let C be an actor class, p and q be trace assertions. Suppose $\vDash \{p\}\ C\ \{q \wedge nonCr(\$)\}$ holds. Then, $\vDash \{p\}\ [C]\ \{q\}$.*

Proof. The created actor is acting as a component (i.e., it follows Def. 3.4). □

The soundness of the Boxing rule comes from the actor fulfilling the component definition (Def. 3.4) for input traces that fulfills the input trace assertion p. This rule immediately becomes unsound when we remove the no actor creation restriction because it falsifies the component definition.

Lemma 5.3 (Soundness of BoxedComposition). *Let the following assumptions hold.*

A1. $\vDash \{p \wedge \underline{\mathsf{i}} = \$\}\ C\ \{q \wedge noSelfExp(\underline{\mathsf{i}}, \$)\}$
A2. $\vDash \{q'\}\ D\ \{r \wedge nonCr(\$)\}$
A3. $match(q, q', D)$

Then, $\vDash \{p\}\ [C]\ \{r\}$.

Proof. By cases. Here we consider D to be an actor class C'. The proof for $D = [C']$ follows a similar outline.

Suppose a is an actor of class C and b an actor of class C', t is a glass box trace that involves actors a and b. Let $split(t, \{a\}) = (ti_a, to_a)$, $split(t, \{b\}) = (ti_b, to_b)$, $F = \mathbf{A} - \{a, b\}$ and σ be a variable assignment such that $p(\sigma, ti_a)$. The goal is that if all assumptions hold, $t{\downarrow_F} \in Traces([\{a, b\}])$, where in this case $Traces([\{a, b\}])$ represents $[\![[C]]\!]$.

- If $\neg(q \wedge noSelfExp(\underline{\mathsf{i}}, \$))(\sigma, to_a)$, then $t{\downarrow_{\{a\}}} \notin Traces(\{a\})$ by A1 and Def. 4.2. By Def. 3.7, $t{\downarrow_F} \notin Traces([\{a, b\}])$.
- If $(q \wedge noSelfExp(\underline{\mathsf{i}}, \$))(\sigma, to_a)$, then $t{\downarrow_{\{a\}}} \in Traces(\{a\})$ by A1 and Def. 4.2. Because of A3, we have $q'(\sigma, ti_b)$.
  - If $\neg(r \wedge nonCr(\$))(\sigma, to_b)$, then either $t{\downarrow_{\{b\}}} \notin Traces(\{b\})$ by A2 and Def. 4.2 or $\neg nonCr(\$)$. For the former case, by Def. 3.7, $t \notin Traces([\{a, b\}])$. For the latter, A2 is not fulfilled.
  - If $(r \wedge nonCr(\$))(\sigma, to_b)$, then $t{\downarrow_{\{b\}}} \in Traces(\{b\})$ by A2 and Def. 4.2. This means $t \in Traces(\langle a|b\rangle)$ and by Lemma 3.6 $t{\downarrow_F} \in Traces([\{a, b\}])$.

By Def. 4.3, $\vDash \{p\}\ [C]\ \{r\}$. □

The essense of the proof above is that whenever the initial actor is given some input trace that satisfies the input trace assertion, the actor will produce an output trace to the other subcomponent of $[C]$ such that this subcomponent produces the output trace that is required by the output trace assertion. The matching, no self exposure and non-creational predicates restrict the case such that no other output event is leaked out except for the expected ones.

Lemma 5.4 (**Soundness of Induction**). *Suppose the following assumptions hold.*

B1. $\vDash \{p \wedge m = 0\}\ [C]\ \{q\}$
B2. $\vDash \{p \wedge m = \underline{z} \wedge m > 0 \wedge \underline{i} = \$\}\ C\ \{p' \wedge m < \underline{z} \wedge noSelfExp(\underline{i}, \$)\}$
B3. $match(p', p, C)$

Then, $\vDash \{p\}\ [C]\ \{q\}$.

Proof. By induction. Suppose we clone the class C into unbounded number of classes named $C_0, C_1, \ldots$. Without loss of generality, we reformulate assumptions B2 and B3 as follows:

B2'. $\forall k \in \mathbb{N} \bullet \vDash \{p_k \wedge m = \underline{z} \wedge m > 0 \wedge \underline{i} = \$\}\ C_k$
$\{p'_k \wedge m < \underline{z} \wedge noSelfExp(\underline{i}, \$)\}$
B3'. $\forall k \in \mathbb{N} \bullet match(p'_{k+1}, p_k, C_k)$

where p_k and p'_k are the same as p and p', respectively, except for the created actor in the input and output traces of C_k being of class C_k and C_{k-1}, respectively. $\mathbb{N}$ represents the set of natural numbers. Moreover, k is always picked exactly to handle the measure m.

Suppose $L = \{a_0, a_1, \ldots\}$ are actors of classes $C_0, C_1, \ldots$ respectively (i.e., $a_0, a_1, \ldots$ are all of class C) and $F = \mathbf{A} - L$. We proceed with the proof by induction on the number of actors that are created. Let t be a valid trace such that there are $k+1$ actors $a_0, \ldots, a_k$ created with a_k to be the initial actor. The goal is to prove for each case that if the input trace assertion p is fulfilled, then if output trace assertion q is fulfilled, $t{\downarrow_F} \in [\![[C]]\!]$.
Base case: $k = 0$. Because a_0 is the only actor created in t, t is already an actor trace of a_0. By Lemma 3.4, t is also a boxed trace. Let $split(t, \{a_0\}) = (ti, to)$. Let σ be a variable assignment such that $p(\sigma, ti)$ holds. Because k is always picked exactly to handle the measure m, B1 is applicable. This means that when $q(\sigma, to)$, $t{\downarrow_F} \in [\![[C]]\!]$, otherwise $t{\downarrow_F} \notin [\![[C]]\!]$.
Inductive step: $k = n + 1$. Let $L' = \{a_0, \ldots, a_{n-1}\}$ and t' be a boxed trace of L' with the initial actor a_{n-1} and $split(t', L') = (ti', to')$. The induction hypothesis is that L' is a component and if t' satisfies $p_{n-1}(\sigma, ti')$ for some variable assignment σ, then $q(\sigma, to')$.

Let $split(t{\downarrow_{\{a_n\}}}, \{a_n\}) = (ti_n, to_n)$ and $split(t{\downarrow_{L'}}, L') = (ti_{L'}, to_{L'})$. Let σ be a variable assignment such that $p_n(\sigma, ti_n)$.

If $\neg p'_n(\sigma, to_n)$, then B2' is violated. Thus, $p'_n(\sigma, to_n)$ must hold. Furthermore, a_n does not expose itself to actors it creates. From B3' we obtain that $p_{n-1}(\sigma, ti_{L'})$. By the induction hypothesis, $q(\sigma, to_{L'})$ must hold. Because a_n is not exposed, $to_{L'}$ is also the output trace of L (that is, the output trace of L' does not contain output events that are directed to a_n). Note that a_n is the only actor that can be created by the environment, because L' is boxed and the initial actor a_{n-1} is created by a_n. By Def. 3.4, L is a component. Moreover, as p_n is essentially the same as p, the input trace of L is the same as ti_n. Therefore, $t{\downarrow_F} \in [\![[C]]\!]$.

By induction principle and Def. 4.3, $\vDash \{p\}\ [C]\ \{r\}$. □

The soundness of Induction is proven by induction on the number of actors of class C that are created. Because all actors carry the same characteristics, never expose one actor to the next one, and produce output that is passed as a whole to the next actor (except for the last actor that produces output exclusively to the environment), we can hierarchically box the actors from the innermost to the outermost layer by layer. By employing the boxed second outermost layer as induction hypothesis, the proof of the inductive case is carried out in a similar way to the proof for the soundness of the BoxedComposition rule.

Based on the lemmas above, we conclude that *RPSA* is sound.

Theorem 5.5. *The proof system RPSA is sound.*

6. Related Work

We consider related work in the areas of semantics, component specification, and logic.

There are plenty of semantics proposed for actor-based systems (e.g., [2, 7, 10, 28, 30]). The trace semantics used in this paper is inspired by Vasconcelos and Tokoro's trace semantics [30] and Talcott's interaction path [28]. Rather than having the behavior of actors evolve depending on what input the actors receive, classes are used to provide more structure to the behavior of the actors. Instead of employing an independence relation or a partial-order relation between events to mimic the buffered message passing communication [2], we use method interaction and the caller information is introduced into the events to allow projection-based composition. This approach avoids the need to come up with and maintain these relations. The traces can be extracted from actor-based programming languages using the guess and merge approach [3].

π-calculus [22] can be used to specify actor systems, but it needs some restrictions on the syntax and introduction of actor identities. Verifying a component specification from the actor class specifications involves bisimulation.

Specification Diagram [26] provides a very detailed way to specify how an actor system behaves. To check whether a component specification produces the same behavior as the composition of the specification of its subcomponents one has to perform a non-trivial interaction simulation on the level of the state-based operational semantics. By extending π-calculus, a may testing ([14]) characterization of Specification Diagram can be obtained [29].

Our specification technique is strongly related to Focus [9]. The main difference is that Focus provides no support for messages, name transfer and dynamic creation — necessary features for actor systems.

Several logics have been developed to reason about the functional behavior of actor-based implementations. For the verification process, most of them are based on a state-based semantics and rely on having the actual implementations. For example, Darlington and Guo [12] provides a linear logic characterization of the state-based semantics of an actor system. Arts and Dam [6], the source of our running example, used extended first-order μ-calculus [11] to verify Erlang programs ([5]) by evaluating the state of the program. The use of temporal logic have been considered by Duarte [16] and Schacht [25].

De Boer [13] presented a Hoare logic for concurrent processes that communicate by message passing through FIFO channels (similar to actors). He described a similar bilayered architecture, where the assertions are based on local and global rules. The local rules deal the local state of a process, whereas the global rules deal with the message passing and creation of new processes. However, they only work for closed systems.

An example of a logic that is based on trace semantics is the work by Soundarajan [27]. Soundarajan proposed a specification technique more general than ours and a proof system than can handle a fixed finite number of processes. A specification of an object of a single class may be represented using Soundarajan's specification technique as follows.

$$p[\$/t_{input}] \implies q[\$/t_{output}]$$

Nevertheless, our Hoare-like triple is convenient as the interleaving of input and output need not be specified.

Ahrendt and Dylla [3] and Din et al. [15] have extended Soundarajan's work to deal with actor systems. They consider only finite prefix-closed traces, justifying their consideration of having only finite number of actors to consider in the verification process. Din et al. [15], in particular, verified whether an implementation of an actor class satisfies its actor triples by transforming the implementation in a simpler sequential language, applying the transformational method proposed by Olderog and Apt [23]. The main difference between this work and the aforementioned work on trace semantics is on the notion of component that hides a group of actors into a single entity. This avoids starting from the class specifications of each actor belonging to a component when verifying a property of the component.

7. Conclusion

In this paper, we have presented a logic that supports compositional specification and verification of open concurrent component-based systems in terms of an actor model. The semantics of actors is represented using traces of events, from which we derived the notion of dynamic, hierarchical components. As each event reveals either an action of sending a message from an actor to another or creation of a new actor, our trace semantics provides all information about the observable behavior of the actors and components. The Hoare-like specification triples are designed to state the precise, albeit partial, response of an actor or a component to the input it receives. We then proposed a sound and compositional axiomatic proof system that handles components that form a daisy chain with only one-way interaction between their subcomponents. By assuming the actor specifications, the proof system can focus on proving component and system-wide functional properties. An illustration on how the specification technique and the proof system can be applied is given by means of a client-server example.

Future work. Several directions we are taking include deriving the trace validity definition from a simple operational semantics of actor systems, incorporating a more general specification technique by adapting the Soundarajan's approach and allowing abstract state information, and an extension of the proof system to cover more flexible composition schemes. In addition, to have a closer connection to actor programming/modeling languages, such as ABS, we would like to support more complex communication constructs such as futures.

Acknowledgments

We thank Mads Dam for several fruitful discussions and for his suggestion to use the problem described in [6] as an example. Yannick Welsch gave feedback to improve the presentation of the paper. This work is partially supported by the EU project *FP7-231620 HATS: Highly Adaptable and Trustworthy Software using Formal Models*.

References

[1] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, USA, 1986. ISBN 0-262-01092-5.

[2] G. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott. A foundation for actor computation. *J. Funct. Program.*, 7(1):1–72, 1997.

[3] W. Ahrendt and M. Dylla. A system for compositional verification of asynchronous objects. *Science of Computer Programming*, 2011.

[4] K. R. Apt, F. S. de Boer, and E.-R. Olderog. *Verification of Sequential and Concurrent Programs, 3rd Edition*. Texts in Computer Science. Springer-Verlag, 2009. 502 pp, ISBN 978-1-84882-744-8.

[5] J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent programming in ERLANG (2nd ed.)*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, 1996. ISBN 0-13-508301-X.

[6] T. Arts and M. Dam. Verifying a distributed database lookup manager written in Erlang. In *World Congress on Formal Methods*, pages 682–700, 1999.

[7] H. C. Baker, Jr. and C. Hewitt. The incremental garbage collection of processes. *SIGART Bull.*, pages 55–59, August 1977.

[8] M. Broy. A logical basis for component-oriented software and systems engineering. *Comput. J.*, 53(10):1758–1782, 2010.

[9] M. Broy and K. Stølen. *Specification and Development of Interactive Systems: Focus on Streams, Interfaces, and Refinement*. Springer-Verlag New York, Secaucus, NJ, USA, 2001. ISBN 0-387-95073-7.

[10] W. D. Clinger. *Foundations of Actor Semantics*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, 1981.

[11] M. Dam, L.-Å. Fredlund, and D. Gurov. Toward parametric verification of open distributed systems. In *COMPOS*, pages 150–185, 1997.

[12] J. Darlington and Y. Guo. Formalising actors in linear logic. In *OOIS*, pages 37–53, 1994.

[13] F. S. de Boer. A Hoare logic for dynamic networks of asynchronously communicating deterministic processes. *Theor. Comput. Sci.*, 274(1–2):3–41, 2002.

[14] R. De Nicola and M. Hennessy. Testing equivalences for processes. *Theor. Comput. Sci.*, 34:83–133, 1984.

[15] C. C. Din, J. Dovland, E. B. Johnsen, and O. Owe. Observable behavior of distributed systems: Component reasoning for concurrent objects. *J. Log. Algebr. Program.*, 81(3):227–256, 2012.

[16] C. H. C. Duarte. Proof-theoretic foundations for the design of actor systems. *Mathematical Structures in Computer Science*, 9(3):227–252, 1999.

[17] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, Oct. 1969.

[18] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, Aug. 1978.

[19] E. B. Johnsen, O. Owe, and I. C. Yu. Creol: A type-safe object-oriented model for distributed concurrent systems. *Theor. Comput. Sci.*, 365(1-2):23–66, 2006.

[20] E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A core language for abstract behavioral specification. In *FMCO 2010*, LNCS, pages 142–164. Springer, 2011.

[21] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996. ISBN 1-55860-348-4.

[22] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. *Inf. Comput.*, 100(1):1–77, 1992.

[23] E.-R. Olderog and K. R. Apt. Fairness in parallel programs: the transformational approach. *ACM Trans. Program. Lang. Syst.*, 10 (3):420–455, July 1988.

[24] A. Poetzsch-Heffter. *Specification and Verification of Object-Oriented Programs*. PhD thesis, Habilitation thesis, Technical University of Munich, Jan. 1997.

[25] S. Schacht. Formal reasoning about actor programs using temporal logic. In *Concurrent Object-Oriented Programming and Petri Nets*, pages 445–460, 2001.

[26] S. F. Smith and C. L. Talcott. Specification diagrams for actor systems. *Higher-Order and Symbolic Computation*, 15(4):301–348, 2002.

[27] N. Soundarajan. Axiomatic semantics of Communicating Sequential Processes. *ACM TOPLAS*, 6(4):647–662, Oct. 1984.

[28] C. L. Talcott. Composable semantic models for actor theories. *Higher-Order and Symbolic Computation*, 11(3):281–343, 1998.

[29] P. Thati, C. L. Talcott, and G. Agha. Techniques for executing and reasoning about specification diagrams. In *AMAST*, pages 521–536, 2004.

[30] V. T. Vasconcelos and M. Tokoro. Traces semantics for actor systems. In *Object-Based Concurrent Computing*, volume 612 of *LNCS*, pages 141–162. Springer, 1991.

Timed-Rebeca Schedulability and Deadlock-Freedom Analysis Using Floating-Time Transition System

Ehsan Khamespanah
School of Electrical and Computer Engineering, University of Tehran
e.khamespanah@ece.ut.ac.ir

Zeynab Sabahi Kaviani
School of Electrical and Computer Engineering, University of Tehran
z.sabahi@ece.ut.ac.ir

Ramtin Khosravi
School of Electrical and Computer Engineering, University of Tehran
r.khosravi@ut.ac.ir

Marjan Sirjani
School of Electrical and Computer Engineering, University of Tehran / School of Computer Science, Reykjavik University
s.sirjani@ece.ut.ac.ir / marjan@ru.is

Mohammad-Javad Izadi
School of Electrical and Computer Engineering, University of Tehran
m.j.izadi@ece.ut.ac.ir

Abstract

"Timed-Rebeca" is an actor-based modeling language for modeling real-time reactive systems. Its high-level constructs make it more suitable for using it by software practitioners compared to timed-automata based alternatives. Currently, the verification of Timed-Rebeca models is done by converting into timed-automata and using UPPAAL toolset to analyze the model. However, state space explosion and time consumption are the major limitations of using the back-end timed automata model for verification. In this paper, we propose a new approach for direct schedulability checking and deadlock freedom verification of Timed-Rebeca models. The new approach exploits the key feature of Timed-Rebeca, which is encapsulation of concurrent elements. In the proposed method, each state stores the local time of each actor separately, avoiding the need for a global time in the state. This significantly decreases the size of the state space. We prove the bisimilarity of the generated transition system (called floating-time transition system) and the state space generated based on Timed-Rebeca semantics. Also, we provide experimental results showing that the new approach mitigates the state space explosion problem of the former method and allows model-checking of larger problems.

Keywords Actor model, Timed-Rebeca, Verification, Realtime systems, Schedulability, Deadlock

1. Introduction

In recent years, the modeling and verification of schedulability and deadlock freedom of component-based and distributed real-time systems has become very important [16]. Distributed and component-based systems consist of multiple cooperating components where the components are typically encapsulated subsystems or objects spread over a network, interacting using asynchronous communication. Providing quality of service guarantees—despite the ever-increasing complexity of distributed systems—has been and remains a grand challenge. Using formal methods, in general, and model-checking [8], in particular, has been advocated as a response to this challenge. Model checking tools exhaustively explore the state space of a system to make sure that a given property holds in all possible executions of the system. The properties preserve both functional and timing correctness of realtime distributed systems.

A well-established paradigm for modeling distributed and asynchronous component-based systems is the *Actor* model. This model was originally introduced by Hewitt [10] as an agent-based language and a mathematical model of concurrent computation. It treats *actors* as the universal primitives of concurrent computation [4, 12]. The asynchronous and nonblocking message-based communication and encapsulation of state and behavior make the actor model suitable for modeling of component-based software [11]. Each actor provides a certain number of services which can be requested by other actors by sending messages to the provider. Messages are put in the message buffer of the actor, then the actor takes the message and executes the requested service, possibly responding to some other components. There are some extension on the actor model for realtime systems like RT-synchronizer [20], real-time Creol [9], and Timed-Rebeca[3].

Reactive Objects Language, *Rebeca* [23], is an operational interpretation of the actor model with formal semantics, supported by model-checking tools. *Rebeca* is designed to bridge the gap between formal methods and software engineering. The formal semantics of *Rebeca* is a solid basis for its formal verification. Compositional and modular verification, abstraction, symmetry and partial-order reduction have been investigated for verifying Rebeca models. The theory underlying these verification methods is already established and is embodied in verification tools [15, 21–23]. With its simple, message-driven and object-based computational model, Java-like syntax, and a set of verification tools, *Rebeca* is an interesting and easy-to-learn model for practitioners.

Timed-Rebeca [3] has been proposed as an extension of *Rebeca* for modeling actor-based distributed and realtime systems. One of the first approaches in verifying Timed-Rebeca models, is suggested in [13]. In this method, the Timed-Rebeca model is translated into a set of timed automata, that was claimed to be collectively bisimilar to the original Timed-Rebeca model. The generated timed automata model, can then be verified using timed automata tools such as UPPAAL [7]. The drawback is that the resulting timed automata model generates a huge state space, which tends to grow exponentially as the model becomes larger. This is a result of the difference in the nature of the two models: while Timed-Rebeca uses asynchronous message passing , UPPAAL uses transition synchronisation as the only form of communication between the automata. Therefore, the mapping needs to use extra automata locations and transitions to simulate asynchrony. In addition, the translation algorithm uses a set of clocks to convert the absolute time model of Timed-Rebeca into a relative time model. The number of clocks used, which is dependent on the number of concurrent messages, highly affects the size of the state space and the model-checking time consumption.

There is the same drawback using *Realtime Maude* [19] and *Timed I/O Automata* [18] as back-end model checkers of Timed-Rebeca models. Realtime Maude is a timed logic language which has the ability to define a tick rule for time progress. The tick rule specifies the amount of the global time lapse for each step in model progress. So the synchronous tick progression causes unnecessary interleaving of independent behavior of components. Timed I/O automata is a basic mathematical framework to support description and analysis of timed computing systems [17]. The timed I/O automata formalism supports decomposing timed system descriptions to a number of automata. In particular, the framework includes a notion of external behavior for a timed I/O automaton, which captures its discrete interactions with its environment. A composition operation for timed I/O automata assigns an execution fragment to each action implying that a timed I/O automata does not block the passage of time. Such a fine-grained time progress causes unnecessary interleaving of concurrent independent actions which increases the size of the state space.

In this work, we introduce the notion of *floating-time transition system* for tackling the state space explosion problem for Timed-Rebeca. Contrary to timed automata and timed I/O automata where the modeler resets the clocks explicitly to avoid unbounded time progress, the progress of time in Timed-Rebeca models will automatically become bounded by using floating-time transition system. Our technique is detecting the recurrent patterns of behaviors while building the transition system of the model, and allowing different local times for each actor in the same state. This is achieved by having relative time specifiers in Timed-Rebeca models.

In all other described methods, each state has a consistent time interval specifier. Therefore, at each state the global time specification of the state is defined. In contrast, the new approach is developed based on the key ingredient of Timed-Rebeca models, which is encapsulation of concurrent elements. Each state stores the local times of its concurrent elements. Therefore, there is no need for global time in the state for deadlock freedom and schedulability analysis. The local times of different actors in a state can be different because each element has its own local message and time management. This property significantly decreases the size of the state space as will be shown in the experimental results. In this paper, we define the formal definition of the floating-timed transition system and a bisimulation relation between this transition system and the transition system derived form SOS (Structural Operational Semantics) of Timed-Rebeca[3]. Based on this bisimilarity, it can be concluded that deadlock freedom and schedulability analysis of SOS and floating-time transition systems have similar results.

Motivation and Contribution. The contributions of this paper can be summarized as follows:

– Introducing the notion of floating-time transition system as an abstract way for state space generation of Timed-Rebeca models

– Proving the bisimilarity of the floating-time transition system and the SOS transition system of Timed-Rebeca models by introducing a behavioral equivalency of timed systems states as time-shift equivalency relation

– Implementing a tool for schedulability and deadlock-freedom analysis based on the proposed techniques

– Providing experimental results and measuring the efficiency of our technique by means of a number of case studies

Roadmap. The rest of this paper is structured as follows. The next section gives some background about the Timed-Rebeca modeling language, its operational semantics, and its translator to the timed automata. Section 3 defines the concept of floating-time transition system. Section 4 defines the schedulability and deadlock analysis of the floating-time transition system, proving the existence of bisimulation relation between SOS rules and the floating-time transition system. In Sections 5 and 6, we present the implementation issues of state space generation algorithm and report the experimental results respectively. The concluding remarks are presented in Section 7.

2. Preliminaries

2.1 Timed-Rebeca

Timed-Rebeca [3] has been proposed as an extension of *Rebeca* [23, 24] for modeling actor-based distributed and realtime systems. A Timed-Rebeca model consists of the definition of *reactive classes* and the instantiation part which is called *main*. The main part defines instances of reactive classes, called *rebec*s. The reactive class comprises three parts: *known rebecs*, *state variables*, and *message server* definitions.

The known rebecs of a reactive class are the destination rebecs of the messages which may be sent by the instances of the reactive class. The internal state of a reactive class is represented by the valuation of its state variables. Because of the encapsulation of *actor*s, the state variables of an actor cannot be directly accessed by other actors. The behavior of the instances of a reactive class is determined by the definitions of its message servers. In Rebeca, communication takes place by asynchronous message passing, which is non-blocking for both sender and receiver. The sender rebec sends a message to the receiver rebec and continues its work. The message is immediately put in the message bag of the receiver until its release time. It is then taken from the bag and the execution of the corresponding message server is started. Execution of a message server body takes place non-preemptively. Each message server has a name, a (possibly empty) list of parameters, and the message server body which includes a number of statements. The statements may be assignments, sending of messages, selections, and delays. We do not consider dynamic rebec creation or reference passing (dynamic topology). Note that there is no explicit receive statement in Rebeca. Time progress is modeled by delays and the time quantifiers of messages (Message release time and deadline). We illustrate this with an example. Figure 1 shows the Timed-Rebeca model of a ticket service system. The model consists of three reactive classes: *TicketService*, *Agent*, and *Customer*. *Customer* sends the “ticket issue” message to *Agent* and *Agent* forwards the issue to *TicketService*. *TicketService* rebec replies to *Agent* by sending a “ticket issued” message and *Agent* responds to *Customer* by sending the issued ticket identifier. As shown in line 12 of the model, issuing the ticket takes three time units (based on the configuration parameters, the issueDelay initial value equals to three). In addition, line 24 shows that *Agent* waits for five time units for *Ticket-Service* to take the requested message and starts serving it.

```
1  reactiveclass TicketService {
2    knownrebecs {
3      Agent a;
4    }
5    statevars {
6      int issueDelay;
7    }
8    msgsrv initial(int myDelay) {
9      issueDelay = myDelay;
10   }
11   msgsrv requestTicket() {
12     delay(issueDelay);
13     a.ticketIssued(1);
14   }
15 }
16
17 reactiveclass Agent {
18   knownrebecs {
19     TicketService ts;
20     Customer c;
21   }
22   msgsrv requestTicket() {
23     ts.requestTicket()
24         deadline(5);
25   }
26   msgsrv ticketIssued(byte id) {
27     c.ticketIssued(id);
28   }
29 }
30
31 reactiveclass Customer {
32   knownrebecs {
33     Agent a;
34   }
35   msgsrv initial() {
36     self.try();
37   }
38   msgsrv try() {
39     a.requestTicket();
40   }
41   msgsrv ticketIssued(byte id) {
42     self.try() after(30);
43   }
44 }
45
46 main {
47   Agent a(ts, c):();
48   TicketService ts(a):(3);
49   Customer c(a):();
50 }
```

Figure 1. The Timed-Rebeca model of ticket service system.

The behavior of a Rebeca model is defined as the parallel execution of the released messages of the rebecs. At the initialization state, a number of rebecs are created statically, and an "*initial*" message is implicitly put in their bags. The release times of the initial messages are zero. The execution of the model continues as rebecs change the values of their state variables and send messages to each other.

2.1.1 Timed-Rebeca Structural Operational Semantics

In this section we provide an overview of the SOS semantics of Timed-Rebeca which has been proposed in [3].

Timed-Rebeca states are pairs (Env, B), where Env is a finite set of environments and B is a bag of messages. For each rebec r of the model there exists $\sigma_r \in Env$ which stores information about the actor, including the values of its state variables and local time, as well as structural characteristics like the body of the message servers. The bag B contains an unordered collection of all the messages of all rebecs. Each message is a tuple of the form $(r_i, m(v), r_j, TT, DL)$. Intuitively, such a tuple says that at time TT (time tag), the sender r_j sent the message to the rebec r_i requesting it to execute its message server m with actual parameters v. Moreover this message expires at time DL [3]. Note that DL specifies the time that the message has to be released, i.e., the messages server has to start its execution..

The system transition relation $\rightarrow$ is defined by the rule *scheduler* of Figure 2 where the condition C is defined as follows: σ_{r_i} is not contained in Env, and $(r_i, m(\overline{v}), r_j, TT, DL) \notin B$, and $\sigma_{r_i}(rtime) \leq DL$, and $TT \leq min(B)$. The scheduler rule allows the system to progress by picking up messages from the bag and executing the corresponding methods. The third side condition of the rule, namely $\sigma_{r_i}(rtime) \leq DL$, checks whether the selected message carries an expired deadline, in which case the condition is not satisfied and the message cannot be picked. The last side condition is the predicate $TT \leq min(B)$, which shows that the time tag TT of the selected message is the smallest time tag of all the messages (for all the rebecs r_i) in the bag B [3].

The τ transition shows the execution of the message server of the rebec r_i and its effects formally defined in [3]. Intuitively, execution of a message server means carrying out its body statements atomically. The execution may change the environment of the rebec r_i by assigning new values to its state variables, modifying the *now* value of r_i, or changing the bag by sending messages to other rebecs. The new message is put in the bag with its time tag and deadline. Time tag is its relative receive time which is computed by the value of *after* statement. The $\sigma_{r_i}(now)$ (current time of rebec) may be modified if the body of message contains the timing statement *delay*. The current time of the rebec increases by the value of the *delay* statement.

2.2 Model Checking Timed-Rebeca using UPPAAL

Mapping the Timed-Rebeca model to timed automata [5] and model-checking the resulting timed automata is an approach which has been suggested in [13]. In this approach, the author generates three timed automata for each rebec. These three automata model the behavior of the message servers, the timed-bag, and the "after" usage in the Timed-Rebeca model. Such a mapping does not seem very straightforward mainly because in Timed-Rebeca message passing is asynchronous, while timed automata models have a synchronous messaging mechanism. There were also some other obstacles, which will be explained in the following lines, together with the suggested solutions.

The author considers a single timed automaton per rebec, called rebec timed automaton. This timed automaton has (as its internal variable) an array that models the message bag. To implement sending messages, the underlying timed automata synchronization mechanism over channels is used. This requires the receiving timed automaton to always be ready to accept messages (as specified in the semantics of Timed-Rebeca: messages are instantaneously received in the message bag of the receiver). For this, it is needed to have transitions on every location in the rebec timed automata, which accepts the message synchronously and puts it in the message bag of the receiving rebec.

Another solution would be to consider another timed automaton per rebec to model its message bag. This timed automaton, called the rebec's Inbox timed automaton, would then always accept messages asynchronously, regardless of the state of the corre-

$$\frac{(\sigma_{r_i}(m), \sigma_{r_i}[rtime = max(TT, \sigma_{r_i}(now)), [\overline{arg} = \overline{v}], sender = r_j], Env, B) \xrightarrow{\tau} (\sigma'_{r_i}, Env', B')}{(\{\sigma_{r_i}\} \cup Env, \{(r_i, m(\overline{v}), r_j, TT, DL)\} \cup B) \rightarrow (\{\sigma'_{r_i}\} \cup Env', B')} C$$

Figure 2. Timed-Rebeca system transition relation scheduler sos rule

sponding rebec timed automaton, and then deliver it, upon the rebec timed automaton's request. The Inbox rebec is responsible to handle message activation time and deadlines. Because of its simplicity and better readability the author has chosen the latter approach. Early comparisons between these approaches did not show significant performance difference. Figure 3 shows the timed automaton of rebec's Inbox. As depicted in Figure 3, rebec's Inbox automaton inserts the incoming messages of the owner rebec, discards the messages with passed deadlines, and extracts the messages from Inbox and delivers them.

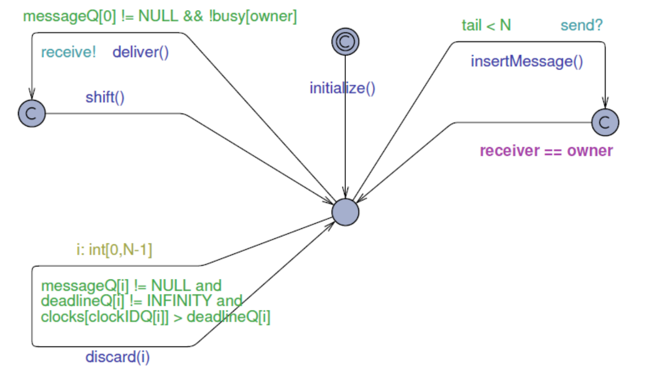
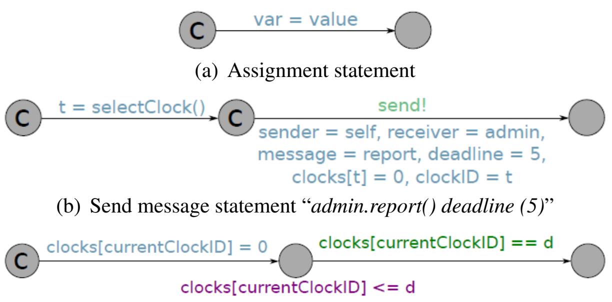


Figure 3. Rebec's Inbox automaton

The rebec timed automaton, will then need to only model the behavior of the rebec itself, as specified by its message servers and state variables' valuations. State variables are converted to rebec timed automaton variables. For the message servers, first consider a timed automaton which simulates the exact behavior of the message server. For simple statements like conditionals, loops, assignments, etc, the mapping is straightforward. Delays are also trivially converted, by the use of one clock, and addition of a location and transition guards to the timed automaton. Figure 4 shows implementations of these statements in timed automata. The mapping for message sending statements will be described shortly.

After deriving the timed automaton equivalent to each of the message servers, all these timed automata are integrated into a single timed automaton and the constructs needed for receiving the next message from the Inbox timed automaton are added. Then the timed automaton will find out which message it has received, and direct the execution to the corresponding message server. Upon completion of the execution of the message server, the rebec timed automaton will requests for the next message, and responds to that message. This completes the reactive behavior of the rebec.

To implement sending of messages, first the exact time constructs of the timed automaton should be specified. In Timed-Rebeca, each rebec has an internal clock, which shows the time elapsed since the creation of the rebec. This specifies an absolute model of time, which cannot be implemented in timed automata, because it makes clock values to grow unboundedly. Therefore, this absolute time model should be converted to relative time. The only time-related constructs in Timed-Rebeca are delay statements and message sending statements. The case for delays is studied before. For message sending, instead of giving timestamps to messages, the author attached one clock to the message. The clock is extracted


var = value
t = selectClock()
send!
sender = self, receiver = admin,
message = report, deadline = 5,
clocks[t] = 0, clockID = t
clocks[currentClockID] = 0
clocks[currentClockID] == d
clocks[currentClockID] <= d

(a) Assignment statement

(b) Send message statement "*admin.report() deadline (5)*"

(c) Delay statement

Figure 4. Implementation of three different Timed-Rebeca statements in timed automata

from a pool of clocks. This clock is used when checking activation times and deadlines and is returned to the pool, when the message is delivered to the rebec timed automaton for execution.

In addition to the formal specification of the mapping algorithm, the author has proved the existance of an equivalence between the resulting timed automata and the structural operational semantics of Timed-Rebeca and has developed a tool for automatic mapping of the Timed-Rebeca models to timed automata. The parallel composition of the resulting timed automata and the schedulability analysis of the model is done using the UPPAAL toolset[7].

A major limiting factor in using UPPAAL and applying the proposed method to practical systems is the generation of huge state space. Generating the state space of large-scale practical timed systems undoubtedly results in state spaces that cannot fit in the memory of a computer. In addition to memory limitation, the model-checking time consumption increases rapidly and makes the model-checking impossible. We will come back to these limitations in Section 6.

3. Floating-Time Transition System of Timed-Rebeca

In this section, we describe the floating-time transition system of Timed-Rebeca models which is used for schedulability and deadlock-freedom analysis.

DEFINITION 1 (Rebecs of a Timed-Rebeca Model). *For a Timed-Rebeca model Reb, the function $O(Reb)$ returns all rebec instances in the model.*

The state of a rebec in model Reb consists of the values of its state variables, its local time, and its message bag. For a Timed-Rebeca model Reb the collection of the states of all the rebecs of $O(Reb)$ is called a Timed-Rebeca state.

DEFINITION 2 (Timed-Rebeca State). *A state of a Timed-Rebeca model is a tuple $s = \prod_{r_i \in O(Reb)} \mathrm{state}(r_i)$, where $\mathrm{state}(r_i)$ is the current state of rebec r_i. The functions $\mathrm{statevars}(s, r_i)$, $\mathrm{bag}(s, r_i)$, and $\mathrm{now}(s, r_i)$ return the state variables valuation function, the message bag content, and the current time of rebec r_i in state s, respectively.*

	S_0
a	State vars:
	Message Bag: $[(initial, 0, \infty)]$
	Now: 0
ts	State vars: issueDelay=?
	Message Bag: $[(initial, 0, \infty)]$
	Now: 0
c	State vars:
	Message Bag: $[(initial, 0, \infty)]$
	Now: 0

I The initial state

	S_{15}
a	State vars:
	Message Bag: [ ]
	Now: 6
ts	State vars: issueDelay=3
	Message Bag: [ ]
	Now: 6
c	State vars:
	Message Bag: $[(ts.ticketissued, 6, \infty)]$
	Now: 0

II An intermediate state

Figure 5. The initial state and one of other states of the model of ticket service system which has been depicted in Figure 1. The receiver of each message is shown in the left-most column (as a, ts, c). Each message is shown as: (sender.message-server-name (list of actual parameters), arrival-time, deadline)

Two different states of the Timed-Rebeca model of Figure 1 are depicted in Figure 5. The state in Figure 5I is an initial state with its rebecs' "now" time set to zero and the "initial" message put in their message bags. The sender of the initial messages are omitted in Figure 5I because the initial messages have no specific sender. Figure 5II depicts one of the intermediate states of the model. In Figure 5II rebec c has a message from rebec ts for time 6 which has no argument. As shown in Figure 5II, there is no guarantee on the equality of the local times of rebecs of a state, so we call it "Floating Time State (FTS)". To ease the reading of the paper, we use "state" instead of FTS in the following.

As shown in definition 2 the state contains the rebecs' message bags. The message bag of rebec is an unordered collection of messages which is structured as defined in the following definition.

DEFINITION 3 (Rebec Message Bag). *A message tuple* $\mathrm{tmsg} = (\mathrm{msgsig}, \mathrm{arrival}, \mathrm{deadline})$ *is a message where* msgsig *is the message content,* arrival *is the arrival time of the message (which is computed based on the value of "after" of send message statement in a Timed-Rebeca model), and* deadline *is the deadline of the message based on the rebec local time. The message msgsig consists of message name, the sender, the receiver, and its actual parameters. For* $\mathrm{tmsg} \in \mathrm{bag}(s, r_i)$ *the functions* $\mathrm{msgsig}(\mathrm{tmsg})$, $\mathrm{arrival}(\mathrm{tmsg})$, *and* $\mathrm{deadline}(\mathrm{tmsg})$ *return the msgsig, arrival, and deadline of the message* tmsg.

The *release time* of a message is the earliest time in which it can start its execution. It depends on the arrival time of the message and the current time of the rebec. The release time of a received message can be later than its arrival time, because upon arrival the receiving rebec may still be busy executing another message server. This happens because message servers execute non-preemptively. Therefore, the execution of a message may delay the execution of another enabled message of a rebec.

DEFINITION 4 (Message Release Time). *The message release time of* $\mathrm{tmsg} \in \mathrm{bag}(s, r_i)$ *is defined as* $\max\{\mathrm{now}(s, r_i), \mathrm{arrival}(\mathrm{tmsg})\}$, *denoted by* $\mathrm{releasetime}(\mathrm{tmsg})$.

The next enabled messages of each rebec is defined based on the Earliest-Release-Time-First policy of Timed-Rebeca using the messages' release times. The set of enabled messages are called *enabled messages* which is a set of messages which should be executed before other messages of r_i.

DEFINITION 5 (Rebec Enabled Messages).
$\mathrm{enabledmessages}(s, r_i) = \{\mathrm{tmsg} \in \mathrm{bag}(s, r_i) | \forall \mathrm{tmsg}' \in \mathrm{bag}(s, r_i) \cdot \mathrm{releasetime}(\mathrm{tmsg}) \leq \mathrm{releasetime}(\mathrm{tmsg}')\}$.

DEFINITION 6 (Rebec Next Message Release Time). *The release time of the currently enabled messages of rebecs* r_i *in state* s *is defined as* $\mathrm{NMRT}(s, r_i) = \mathrm{releasetime}(\mathrm{enabledmessages}(s, r_i))$.

Based on definitions 1 to 6, a new equivalence relation between states is defined. The suggested equivalence relation helps to have a bounded state space and avoid infinite state space which occurs because of the time progress of the model. For an informal and simplified description, note that the time expressions used in Timed-Rebeca delay and message-sending statements are relative times. Therefore, for two states s and s' which only differ in the local time of rebecs, shifting the local time and messages time qualifiers of all the rebecs of state s' backwards to make it equal to s, does not affect its behavior. It only affects the arrival and deadline times of the sent messages, so the behavior of the model after state s' is the same as s and there is no need for state generation after reaching s'. We illustrate this with an example. Consider Figure 6 presenting three different states of the Timed-Rebeca model of the ticket service system of Figure 1.

Assume that the model is in the state shown in Figure 6I. Based on the bag of the rebecs *ts*, *c*, and *a*, $\mathrm{enabledmessages}(S_{20}, a) = \{\}, \mathrm{enabledmessages}(S_{20}, ts) = \{\}, \text{and enabledmessages}$ $(S_{20}, c) = \{[\mathrm{ticketIssued}, 36, \infty]\}$. Therefore, only rebec c has an enabled message whose execution results in S_{21}, shown in Figure 6II.
Here, shifting the time of S_{21} by 33 units, makes it equal to S_{16}, so S_{21} and S_{16} are shift equivalent by shifting 33 units, denoted by $S_{21} \cong_{33} S_{16}$.

DEFINITION 7 (Rebec State Shift Equivalence). *Two states* s *and* s' *are shift equivalent with respect to rebec* r_i, *if* $\mathrm{statevars}(s, r_i) = \mathrm{statevars}(s', r_i)$ *and one of the following conditions hold:*

- $\mathrm{now}(s', r_i) = \mathrm{now}(s, r_i)$ *and* $\mathrm{bag}(s', r_i) = \mathrm{bag}(s, r_i)$. *In this case* s' *and* s *are zero-time shift equivalent with respect to* r_i, *denoted by* $s' \cong_{i,0} s$.
- *There exists an integer number* t *such that* $\mathrm{now}(s', r_i) = \mathrm{now}(s, r_i) + t$ *and there is a bijective relation* $\leftrightarrow$ *between* $\mathrm{bag}(s', r_i)$ *and* $\mathrm{bag}(s, r_i)$ *where for* $\mathrm{tmsg}' \in \mathrm{bag}(s', r_i)$ *and* $\mathrm{tmsg} \in \mathrm{bag}(s, r_i)$ $\mathrm{tmsg}' \leftrightarrow \mathrm{tmsg}$ iff $\mathrm{msgsig}(\mathrm{tmsg}') = \mathrm{msgsig}(\mathrm{tmsg}) \wedge \mathrm{arrival}(\mathrm{tmsg}') = \mathrm{arrival}(\mathrm{tmsg}) + t \wedge \mathrm{deadline}(\mathrm{tmsg}') = \mathrm{deadline}(\mathrm{tmsg}) + t$. *In this case, the two states are "by t unit(s)" shift equivalent with respect to* r_i, *denoted by* $s' \cong_{i,t} s$. *Note that the time specifiers of Timed-Rebeca are integer numbers, hence* t *is a member of natural numbers.*

For two states s *and* s' *shift equivalence is defined as* $\exists t \in \mathbb{N} \cdot \forall r_i \in O(Reb) \cdot s' \cong_{i,t} s$, *denoted by* $s' \cong_t s$. *Note that the time shift preserves the relative difference of the rebec now and its bag's message specifiers.*

Now the Timed-Rebeca floating-time transition system can be defined based on the notion of shift equivalence. The floating-time transition system is a transition system similar to Figure 7

S_{20}		
a	State vars:	
	Message Bag:	[ ]
	Now:	36
ts	State vars:	issueDelay=3
	Message Bag:	[ ]
	Now:	36
c	State vars:	
	Message Bag:	$[(c.ticketIssued, 36, \infty)]$
	Now:	36

I State number 20

S_{21}		
a	State vars:	
	Message Bag:	[ ]
	Now:	36
ts	State vars:	issueDelay=3
	Message Bag:	[ ]
	Now:	36
c	State vars:	
	Message Bag:	$[(c.try, 66, \infty)]$
	Now:	36

II State number 21

S_{16}		
a	State vars:	
	Message Bag:	[ ]
	Now:	3
ts	State vars:	issueDelay=3
	Message Bag:	[ ]
	Now:	3
c	State vars:	
	Message Bag:	$[(c.try, 33, \infty)]$
	Now:	3

III State number 16

Figure 6. Three different states of the Timed-Rebeca model of ticket service system

which depicts a floating-time transition system of the ticket-service model. To make the transition system easy to understand, transition labels from state S_0 to state S_{15} are omitted. As shown in Figure 7, a transition label is a pair consisting of the executed message and the time shift. The time shifts of all the transitions of Figure 7 are zero except the transition from S_{20} to S_{16}, because of the equivalence relation defined in Definition 7.

DEFINITION 8 (Timed-Rebeca Floating-Time Transition System). *A* Timed-Rebeca Floating-Time Transition System (FTTS) *is a labeled transition system* $\mathrm{FTTS}(Reb) = (S, s_0, Act, \hookrightarrow, AP, L)$, *where:*

- *S is a set of states.*
- *$s_0 \in S$ is an initial state.*
- *Act is a set of actions. Each action is a pair of message and time shift.*
- *$\hookrightarrow \subseteq S \times \mathrm{Act} \times S$ is a set of transition relations. $\forall s, s' \in S, (s, (tmsg, t), s') \in \hookrightarrow$ iff there exists $r_i \in O(Reb)$, $\mathrm{tmsg} \in \mathrm{enabledmessages}(s, r_i)$ such that the execution of tmsg results in a state s'' where $s'' \cong_t s'$ and $\forall r_j \in O(Reb) \cdot \mathrm{NMRT}(s, r_i) \leq \mathrm{NMRT}(s, r_j)$. The execution of a message conforms to the Timed-Rebeca semantic SOS rule which modifies the rebec's state variables, sends some messages to other rebecs, and change the local time of rebec as described in Section 2.1.1. There is a non-deterministic choice in those states with more than one enabled message.*
- *AP=$\{\mathrm{Deadlock}, \mathrm{DeadlineMissed}\}$ is a set of atomic propositions.*
- *$L : S \to 2^{AP}$ is a labeling function defined over the set of states as follows:*
  - *Deadlock* $\in L(s)$ iff $\nexists s', act \cdot (s, act, s') \in \hookrightarrow$.
  - *DeadlineMissed $\in L(s)$ iff there exists a state s and a rebec $r_i \in O(Reb)$ such that $\mathrm{NMRT}(s, r_i) > \mathrm{now}(s, r_i)$.*

4. Transition System Schedulability and Deadlock-Freedom Analysis

A Timed-Rebeca model is schedulable if the deadline of no message is missed while executing the model. In other words, there is no reachable state such that the value of "now" of a rebec is greater than the deadline of any message in its bag. Because of the progress of time in Timed-Rebeca semantics, applying the SOS rules on a model results in an unbounded state space which makes the schedulability analysis impossible. However, the bisimilarity of SOS-based transition system and the floating-time state space of a model should be proved. Since the elements of SOS-based transition system and floating-time transition system are syntactically different, in the first step, the following functions should be defined to extract the elements of a state of the SOS-based transition system.

DEFINITION 9 (Functions on SOS Rules Based Transition System). *Consider an unbounded transition system derived from applying the Timed-Rebeca SOS rules [3] on Reb such as* $\mathrm{SOSTS} = (S, s_0, Act, \to, AP, L)$, *a state $s = (Env, B) \in S$, $r_i \in O(Reb)$, and $\sigma_{r_i} \in Env$.*

- *the function* $\mathrm{bag}(s, r_i)$ *returns* $\{(r_t, m(\overline{v}), r_s, TT, DL) \in B | r_s = r_i\}$ *,i.e., the bag of messages of rebec r_i in state s.*
- *the function* $\mathrm{now}(s, r_i)$ *returns $\sigma_{r_i}(now)$, i.e., the "now" of rebec r_i in state s.*
- *the function* $\mathrm{statevars}(s, r_i)$ *returns the valuation function of the state variables of rebec r_i in state s which is extracted from σ_{r_i}.*

Here, $AP = \{\mathrm{Deadlock}, \mathrm{DeadlineMissed}\}$ and L for a state $s = (Env, B) \in S$ is defined as follows:

- *Deadlock* $\in L(s)$ iff $B = \emptyset$.
- *DeadlineMissed* $\in L(s)$ iff for some $(r_i, m(v), r_j, TT, DL) \in B$ condition $max\{\sigma_{r_i}(now), TT\} > DL$ holds.

Before presenting the bisimilarity theorem, it should be remarked that shifting the time of a state preserves its enabled messages. Consider two states $s_1, s_2 \in S'$ where $s_1 \cong_t s_2$ for some $t \in \mathbb{N}$ and a rebec $r_i \in O(Reb)$. Based on Definition 7, $\forall \mathrm{tmsg}_1 \in \mathrm{bag}(s_1, r_i)$ there exists $\mathrm{tmsg}_2 \in \mathrm{bag}(s_2, r_i)$ such that $\mathrm{releasetime}(\mathrm{tmsg}_1) + t = \mathrm{releasetime}(\mathrm{tmsg}_2) \wedge \mathrm{deadline}(\mathrm{tmsg}_1) + t = \mathrm{deadline}(\mathrm{tmsg}_2) \wedge \mathrm{msgsig}(\mathrm{tmsg}_1) = \mathrm{msgsig}(\mathrm{tmsg}_2)$. So in case of $\mathrm{tmsg}_1 \in \mathrm{enabledmessages}(s_1, r_i)$ and $\mathrm{tmsg}_2 \notin \mathrm{enabledmessages}(s_2, r_i)$ there exists $\mathrm{tmsg}_2' \in \mathrm{enabledmessages}(s_2, r_i)$ such that the $\mathrm{releasetime}(\mathrm{tmsg}_2') < \mathrm{releasetime}(\mathrm{tmsg}_2)$. Here, based on shift equivalence relation, there exists $\mathrm{tmsg}_1' \in \mathrm{bag}(s_1, r_i)$ such that $\mathrm{releasetime}(\mathrm{tmsg}_1') + t = \mathrm{releasetime}(\mathrm{tmsg}_2') \wedge \mathrm{deadline}(\mathrm{tmsg}_1') + t = \mathrm{deadline}$ $(\mathrm{tmsg}_2') \wedge \mathrm{msgsig}(\mathrm{tmsg}_1') = \mathrm{msgsig}(\mathrm{tmsg}_2')$. So there exists uniquality $\mathrm{releasetime}(\mathrm{tmsg}_1') < \mathrm{releasetime}(\mathrm{tmsg}_1)$ which contradicts the assumption of $\mathrm{tmsg}_1 \in \mathrm{enabledmessages}(s_1, r_i)$.

THEOREM 1. *For a Timed-Rebeca model Reb, an* $\mathrm{SOSTS} = (S, s_0, act, \to, AP, L)$ *which is an unbounded transition system*

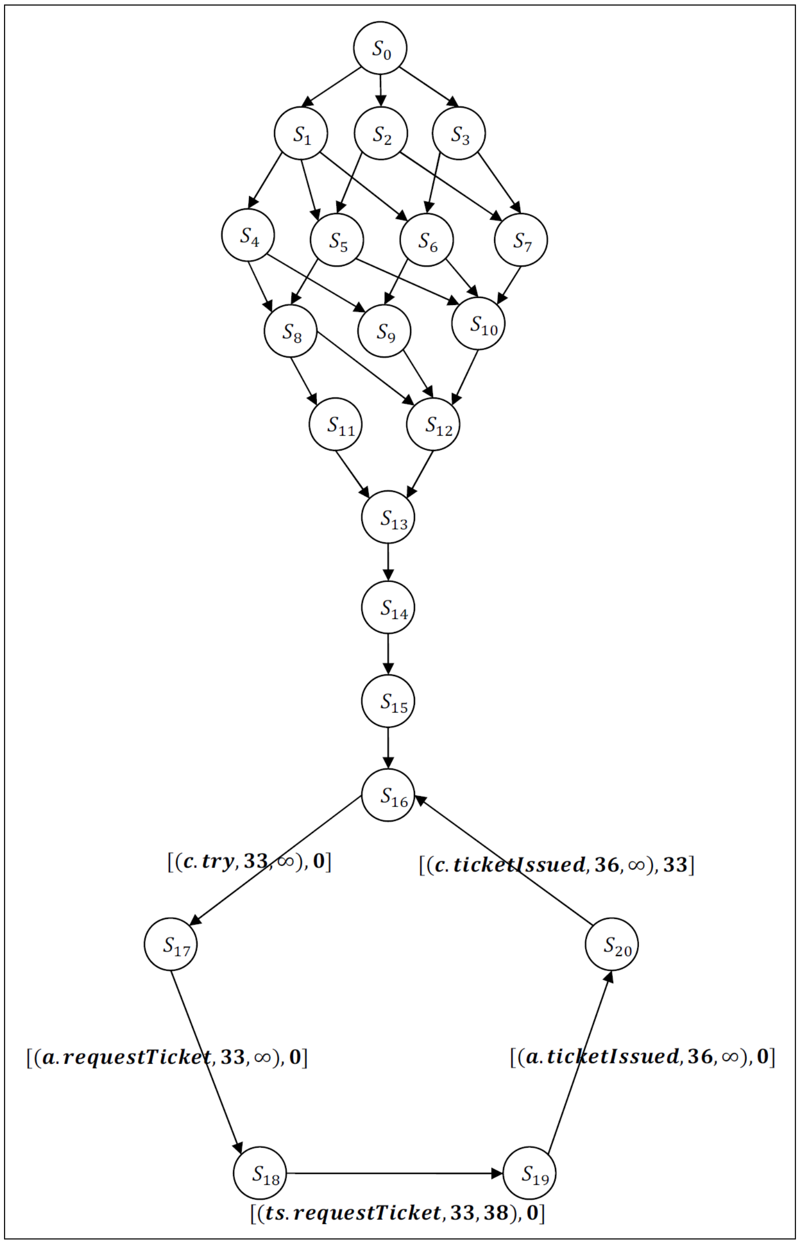

S_0
S_1
S_2
S_3
S_4
S_5
S_6
S_7
S_8
S_9
S_{10}
S_{11}
S_{12}
S_{13}
S_{14}
S_{15}
S_{16}
$[(c.try, 33, \infty), 0]$
$[(c.ticketIssued, 36, \infty), 33]$
S_{17}
S_{20}
$[(a.requestTicket, 33, \infty), 0]$
$[(a.ticketIssued, 36, \infty), 0]$
S_{18}
S_{19}
$[(ts.requestTicket, 33, 38), 0]$

Figure 7. Floating-Time transition system of the Timed-Rebeca model of Figure 1.

derived from applying the Timed-Rebeca SOS rules [3] on Reb, *is bisimilar to the Timed-Rebeca floating-time transition system* $\mathrm{FTTS} = (S', s'_0, act', \hookrightarrow, AP', L')$ *of model Reb.*

PROOF 1. *Based on the renaming function and the Timed-Rebeca semantics, a bisimulation binary relation* $\mathcal{R} \subseteq S \times S'$ *is defined such that for two states* $s \in S$ *and* $s' \in S'$, $(s, s') \in \mathcal{R}$ *iff* $\exists s'' \in S', \exists t \in \mathbb{N} \cup \{0\} \cdot s'' \cong_t s'$ *and* $\forall r_i \in O(Reb) \cdot \mathrm{bag}(s, r_i) = \mathrm{bag}(s'', r_i) \wedge \mathrm{statevars}(s, r_i) = \mathrm{statevars}(s'', r_i) \wedge \mathrm{now}(s, r_i) = \mathrm{now}(s'', r_i)$.

Now it must be shown that for two states $s \in S$ *and* $s' \in S'$, *and* $(s, s') \in \mathcal{R}$, *the following bisimulation transfer conditions are satisfied [6].*

I. If $q \in \mathrm{Post}(s)$ *then there exists* $q' \in \mathrm{Post}(s')$ *with* $(q, q') \in \mathcal{R}$

II. If $q' \in \mathrm{Post}(s')$ *then there exists* $q \in \mathrm{Post}(s)$ *with* $(q, q') \in \mathcal{R}$

III. $L(s) = L'(s')$

In the above conditions Post(s) denotes all successor states of a state s according to the transition relation (for both transition systems). As mentioned in Definition 8, the execution of a message conforms to the Timed-Rebeca SOS rules, so the execution of a message has the same effect on both SOS-based transition system and floating-time transition system. Therefore, conditions I and II hold because as $(s, s') \in \mathcal{R}$ *they have the same enabled messages and the execution of enabled messages in both transition systems have the same effect on the values of state variables and the bags of rebecs.*

The third condition holds for s and s' *because* $AP = AP' = \{\mathrm{Deadlock}, \mathrm{DeadlineMissed}\}$ *and two following conditions hold:*

- *if* Deadlock $\in L(s)$ *then* $B = \emptyset$. *As* $(s, s') \in \mathcal{R}$, $\forall r_i \in O(reb) \cdot \text{bag}(s', r_i) = \emptyset$. *Therefore,* Deadlock $\in L'(s')$*, and vice versa.*
- *if* DeadlineMissed $\in L(s)$ *then* $(r_i, m(v), r_j, TT, DL) \in B$ *exists such that* $max\{\sigma_{r_i}(now), TT\} > DL$. *As the bag of the rebecs in* s *and* s' *are same (because* $(s, s') \in \mathcal{R}$*), the corresponding message* tmsg *in the bag of* r_i *in* s' *should exist such that by* $t \in \mathbb{N} \cup \{0\}$ *units time shift, its time quantifiers are the same as* TT *and* DL *and* $\text{now}(s, r_i) = \text{now}(s', r_i) + t$. *Therefore,* $\text{releasetime}(tmsg) > \text{deadline}(tmsg)$*, so* DeadlineMissed $\in L'(s')$*, and vice versa.*

COROLLARY 1 (Timed-Rebeca Model Analysis). *For the Timed-Rebeca model Reb both schedulability and deadlock-freedom can be examined using its floating-time state space.* SOSTS(Reb) *has a deadline missed message iff there is a state* s *in* FTTS *labeled DeadlineMissed and* SOSTS(Reb) *has a deadlock state iff there is a state* s *in* FTTS *labeled Deadlock.*

5. Implementation

Rebeca comes equipped with an on-the-fly explicit-state LTL model-checking engine called Modere [14]. Modere uses both the nested DFS and BFS search algorithms to explore the state space. To generate state space based on semantics of floating-time transition system and its required analyzer, Modere's BFS search algorithm has been extended to support Timed-Rebeca models, incorporating our novel shift equivalent states detection.

5.1 Rebeca BFS State Space Generation Algorithm

The BFS exploration algorithm, creates and explores the transition system in a level-by-level fashion. In the first step of the BFS algorithm, the initial state of the Rebeca state space is stored and marked as visited. Then, for each level, the successors of the states are generated by applying the successor function to the states; when there are no unexplored states in the next level, the algorithm terminates. For specification of the successor function and its formal semantics refer to [23].

The BFS state space generation algorithm can be implemented using two queues to manage states of each level. The first queue stores the current level states (CLQ) and the second one stores the successors of the CLQ states. The latter queue is called the next level queue (NLQ). In each iteration, the unexplored states of the CLQ are dequeued and their unvisited successors are generated. When all states of the CLQ are dequeued, the content of the NLQ is moved to the CLQ and the algorithm continues until NLQ is empty, i.e., all successors of the states in the CLQ are visited before. Figure 8 shows a pseudo code of the algorithm.

```
BFS-STATE-SPACE-GENERATOR ()
   CLQ ← $\emptyset$
   NLQ ← $\emptyset$
   Visited ← $\emptyset$
   ENQUEUE (CLQ, initState)
   while CLQ $\neq \emptyset$ do
      for each state $S \in$ CLQ do
         NewStates ← SUCCESSOR(S)
         for each State $N \in$ NewStates do
            if $N \notin$ Visited
               then PUT(Visited, N)
                    PUT(LocalHashTable, N)
                    ENQUEUE(NLQ, N)
            fi
         od
      od
      swap(CLQ, NLQ)
      NLQ ← $\emptyset$
   od
```

Figure 8. Modere BFS state space generation pseudo code.

5.2 Timed-Rebeca BFS State Space Generation Algorithm

From the implementation view, the major differences between Rebeca and Timed-Rebeca transition system generation algorithms are in the state structure. A list of time bundles is attached to each state to store the time specification of states – for each rebec its *now*, and for all the messages in the queues of each rebec the *deadline* and *after* specifiers. Therefore, state S with for example two time bundles in its time bundles list represents two different states which their rebecs state variables valuations and message queue content are the same, however the time specifiers are different. Therefore, the approach is to split state information into time-invariant (the valuation of variables and message queues) and time-dependent parts. Multiple time-dependent parts, which we call time bundles, associate with one time-invariant part. In this way, checking for time-shift equivalence of states can be done efficiently. Based on this structure, states shift equivalence checking has been reduced

```
1   BFS-STATE-SPACE-GENERATOR ()
2      CLQ ← $\emptyset$
3      NLQ ← $\emptyset$
4      Visited ← $\emptyset$
5      ENQUEUE (CLQ, initState)
6      while CLQ $\neq \emptyset$ do
7         for each state $S \in$ CLQ do
8            NewStates ← SUCCESSOR(S)
9            for each State $N \in$ NewStates do
10              if $N \notin$ Visited
11                 then PUT(Visited, N)
12                      PUT(LocalHashTable, N)
13                      ENQUEUE(NLQ, N)
14                      CHECK-FOR-DEADLINE-MISSED(N)
15                      CHECK-FOR-DEADLOCK(N)
16                 else
17                   SL ← GET(LocalHashTable, N)
18                   for each bundle ∈ TIME-BUNDLES(SL) do
19                      tb = TIME-BUNDLE(N)
20                      if tb = bundle
21                         then
22                            // N $\cong_0$ SL, so discard visited state
23                         elseif tb - t = bundle
24                            // N $\cong_t$ SL, so discard visited state
25                         else
26                            ADD-BUNDLE(N, tb)
27                            ENQUEUE(NLQ, N)
28                      fi
29                   od
30              fi
31           od
32        od
33        swap(CLQ, NLQ)
34        NLQ ← $\emptyset$
35     od
```

Figure 9. Timed-Rebeca BFS state space generation pseudo code.

to the $O(|time\ bundles|)$ (the size of time bundles) search algorithm which has been shown in Figure 9 lines 17 to 29.

6. Experimental Results

Model-checking execution time and state space size of Timed-Rebeca models based on timed automata transition system and floating-time transition system are compared using three different examples. The examples are *Distributed Sensor Network*, simplified version of *Slotted ALOHA Protocol*, and *Ticket Service*. The test platform is a HP DL-386 G7 server with 4 CPUs of 2.80GHz Intel Xeon and 16GB of RAM storage running Ubuntu 12.04 operating system.

Problem	Size	Using FTTS		Using Timed Automata	
		#states	time	#states	time
Ticket Service	**1 customer**	8	<1(sec)	801	<1(sec)
	2 customers	56	<1(sec)	19263811	≈ 5(hours)
	3 customers	20617	3(secs)	-	-
Sensor Net.	**1 sensor**	52	<1(sec)	-	-
	2 sensors	592	<1(sec)	-	-
	3 sensors	8154	1	-	-
	4 sensors	122900	18	-	-
Slotted ALOHA Protocol	**1 interface**	159	<1(sec)	-	-
	2 interfaces	2030	1(sec)	-	-
	3 interfaces	17253	5(secs)	-	-
	4 interfaces	132200	52(secs)	-	-
	5 interfaces	966147	490(secs)	-	-

Table 1. The model-checking time and state space size, using two different approaches

Sensor Network The distributed sensor network model is a model of a set of sensors which measure the toxic gas level of the environment. Upon sensing a dangerous level of gas, the sensors alarm the scientist who is working there to escape, or alternatively send a rescue team to save him.
There are four different reactive classes *Sensor*, *Admin*, *Scientist*, and *Rescue* in the model. *Sensor* rebecs send the measured gas level value to *Admin* rebec over the network. If *Admin* receives a report of dangerous gas levels, it notifies *Scientist* immediately and waits for *Scientist* acknowledge. If *Scientist* does not respond, *Admin* requests *Rescue* to reach and save *Scientist*. The main property to be checked is saving *Scientist* before the rescue deadline missed. We have varied the number of the sensors to produce state spaces of different sizes.

Slotted ALOHA Protocol The Slotted ALOHA protocol [2] is a network random access protocol which controls the data link medium access. Slotted ALOHA divide the time in to a number of slots and each network interface sends its data at the beginning of a time slot. We have modeled the Slotted ALOHA using four different reactive classes *User*, *Interface*, *Medium*, and *Controller*. *Controller* rebec is a police of medium (*Medium* rebec). *Interface*s of the model are waiting for *Contoller* signal and send the data via *Medium* afterwards. To make the model more realistic, we assigned a *User* to each *Interface* which provides *Interface* with requested data. We generated different size of models by varying the number of *User*s and *Interface*s.

Ticket Service. The detailed description of the *Ticket-Service* has been explained at Section 2.1. Hitting the deadline of issuing the ticket is the desired property of this model. Varying the number of customers helped us to generate models of different sizes.

The Rebeca code of each case study can be found at the Rebeca homepage [1]. Table 1 shows the results of model checking these examples using the two different approaches. The model-checking time is limited to one day for each model. There is "-" as the results of model-checking for the models which take more than a day to model-check.

7. Conclusion

In this paper we introduced the floating-time transition system for schedulability and deadlock freedom analysis of Timed-Rebeca models. Floating-Time transition system exploits the key features of Timed-Rebeca. In summary, having no shared variables, no blocking send or receive, single-threaded actors, and non-preemptive execution of each message server give us an isolated message server execution, meaning that execution of a message server of a rebec will not interfere with execution of a message server of another rebec. Moreover, for checking schedulability and deadlock freedom we can focus only on events. In FTTS each transition shows releasing an event , or in other words execution of a message server of a rebec. Hence, in each state in FTTS rebecs may have different local times, but the transitions still gives us a correct order of release times of events of a specific rebec. We have proved a bisimulation relation between the SOS-based transition system and the floating-time transition system of Timed-Rebeca models. Our proposed approach is implemented as a part of Afra toolset [1]. Experimental evidence supports that direct model-checking of Timed-Rebeca models using floating-time transition system decreases both model-checking state space size and time consumption in comparison with translating to secondary models such as timed automata. Therefore, we can efficiently model-check more complex models.

In addition, our technique is based on the actor model of computation where the interaction is solely based on asynchronous message passing between the components. So, the proposed transition system and analysis techniques are general enough to be applied to similar computation models where they have message-driven communication and autonomous objects as units of concurrency such as agent-based systems.

References

[1] *Rebeca Home Page*. http://www.rebeca-lang.org.

[2] Norman Abramson. THE ALOHA SYSTEM: another alternative for computer communications. In *AFIPS '70 (Fall): Proceedings of the November 17-19, 1970, fall joint computer conference*, pages 281–285, New York, NY, USA, 1970. ACM.

[3] Luca Aceto, Matteo Cimini, Anna Ingólfsdóttir, Arni Hermann Reynisson, Steinar Hugi Sigurdarson, and Marjan Sirjani. Modelling and simulation of asynchronous real-time systems using timed rebeca. In Mohammad Reza Mousavi and António Ravara, editors, *FOCLASA*, volume 58 of *EPTCS*, pages 1–19, 2011.

[4] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, USA, 1990.

[5] Rajeev Alur and David L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235, 1994.

[6] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, 2008.

[7] Johan Bengtsson, Kim Guldstrand Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. Uppaal - a tool suite for automatic verification of real-time systems. In Rajeev Alur, Thomas A. Henzinger, and Eduardo D. Sontag, editors, *Hybrid Systems*, volume 1066 of *Lecture Notes in Computer Science*, pages 232–243. Springer, 1995.

[8] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transaction on Programming Languages and Systems*, 8(2):244–263, 1986.

[9] Frank S. de Boer, Tom Chothia, and Mohammad Mahdi Jaghoori. Modular schedulability analysis of concurrent objects in creol. In Farhad Arbab and Marjan Sirjani, editors, *FSEN*, volume 5961 of *Lecture Notes in Computer Science*, pages 212–227. Springer, 2009.

[10] C. Hewitt. Description and theoretical analysis (using schemata) of PLANNER: A language for proving theorems and manipulating models in a robot. MIT Artificial Intelligence Technical Report 258, Department of Computer Science, MIT, April 1972.

[11] Carl Hewitt. What is commitment? physical, organizational, and social (revised). In *Proceedings of Coordination, Organizations, Institutions, and Norms in Agent Systems II*, Lecture Notes in Computer Science, pages 293–307. Springer, 2007.

[12] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In Nils J. Nilsson, editor, *IJCAI*, pages 235–245. William Kaufmann, 1973.

[13] Mohammad-Javad Izadi. An actor based model for modeling and verification of real-time systems. Master's thesis, University of Tehran, School of Electrical and Computer Engineering, Iran, 2010.

[14] Mohammad Mahdi Jaghoori, Ali Movaghar, and Marjan Sirjani. Modere: The model-checking engine of Rebeca. In Hisham Haddad, editor, *SAC*, pages 1810–1815. ACM, 2006.

[15] Mohammad Mahdi Jaghoori, Marjan Sirjani, Mohammad Reza Mousavi, Ehsan Khamespanah, and Ali Movaghar. Symmetry and partial order reduction techniques in model checking Rebeca. *Acta Informatica*, 47(1):33–66, 2010.

[16] M. M. Jaghouri. *Time At Your Service: Schedulability Analysis Of Real-Time And Distributed Services*. PhD thesis, LIACS, December 2010.

[17] Dilsun Kirli Kaynar, Nancy A. Lynch, Roberto Segala, and Frits W. Vaandrager. Timed i/o automata: A mathematical framework for modeling and analyzing real-time systems. In *RTSS*, pages 166–177. IEEE Computer Society, 2003.

[18] Dilsun Kirli Kaynar, Nancy A. Lynch, Roberto Segala, and Frits W. Vaandrager. *The Theory of Timed I/O Automata, Second Edition*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2010.

[19] Peter Csaba Ölveczky and José Meseguer. Specification of real-time and hybrid systems in rewriting logic. *Theor. Comput. Sci.*, 285(2):359–405, 2002.

[20] Shangping Ren and Gul Agha. Rtsynchronizer: Language support for real-time specifications in distributed systems. In Richard Gerber and Thomas J. Marlowe, editors, *Workshop on Languages, Compilers, & Tools for Real-Time Systems*, pages 50–59. ACM, 1995.

[21] Hamideh Sabouri and Marjan Sirjani. Slicing-based reductions for Rebeca. In *Proceedings of FACS 2008*. ENTCS, 2008.

[22] Marjan Sirjani, Frank S. de Boer, and Ali Movaghar-Rahimabadi. Modular verification of a component-based actor language. *J. UCS*, 11(10):1695–1717, 2005.

[23] Marjan Sirjani, Ali Movaghar, Amin Shali, and Frank S. de Boer. Modeling and verification of reactive systems using Rebeca. *Fundam. Inform.*, 63(4):385–410, 2004.

[24] Marjan Sirjani, Amin Shali, Mohammad Mahdi Jaghoori, Hamed Iravanchi, and Ali Movaghar. A front-end tool for automated abstraction and modular verification of actor-based models. In *ACSD*, pages 145–150. IEEE Computer Society, 2004.

AGERE! @ SPLASH 2012

Actor Idioms

Dale Schumacher

Independent

dale.schumacher@gmail.com

Abstract

Actor systems are driven by asynchronous message reception events. Taking full advantage of the Actor Model requires recognizing relevant patterns of actor interaction. We describe several idioms here, in hopes of beginning to build a catalog of useful interactions. Some idioms simply implement already-familiar mechanisms, in terms of actors. Others illustrate novel strategies that can only be realized with asynchronous actor messaging. All provide new perspective on the process of computation.

Categories and Subject Descriptors D.2.11 [*Software Architectures*]

General Terms Design, Algorithms

Keywords Actors, Actor Model, Interaction Patterns, Idioms, Asynchronous Messaging

1. Introduction

Programming in the Actor Model focuses on patterns of asynchronous message-passing [3]. Concurrent activity is the default. Sequencing must be specifically arranged. The model itself is very general, going beyond even the Lambda Calculus [4]. As with any model of computation, experience with Actor programming yields a variety of idiomatic expressions. We explore some of these idioms here.

All Actor computation is driven by message reception events. In response to receiving a message, an actor may:

- Send messages to other actors
- Create new actors
- Designate how the next message will be processed

From these three primitives, we can construct an endless variety of activities and interactions. Some of these interactions occur repeatedly and in several contexts. They seem to form the basis for a catalog of useful interaction idioms.

We will briefly describe nearly twenty idioms in this paper. The goal is to survey a collection of useful idioms. Figure 1 summarizes the idioms and the important relationships among them. The collection is by no means complete, and only the essential characteristics of each idiom will be described.

We have chosen to present example code for selected idioms using the Erlang programming language [1]. Erlang seems to be the most widely understood language in which Actor concepts can be directly expressed. Note, however, that our example code is *not* idiomatic Erlang, and Erlang is not a pure Actor language.

2. Basic Plumbing

The idioms in the section provide the basic plumbing used to establish relationships among actors.

2.1 Service

The most common actor interaction idiom is the Service. It is so common that it is often built-in and practically invisible in many programming environments. In fact, conventional object-oriented systems are restricted to just this idiom.

A Service is an actor whose protocol involves providing a Customer with every message (Request). A service interaction involves two asynchronous messages, a Request (which includes a Customer) and a Reply (sent to the Customer). This protocol is the basic sequencing mechanism in actor interactions.

2.2 Customer

When we say that a Service receives a request-message *and produces a reply-message in return*, the computational context to which the reply-message is sent is the Customer. We can think of the Customer as the Return-Address or Continuation. Making the Customer explicit provides significant flexibility in designing interaction and synchronization protocols.

A Subroutine can be described as sending a request to a service where the Customer is "the rest of the computation" in the caller. Tail-Call Optimization is evident when we simply pass *our* Customer on to a subroutine rather than create a new Customer. A Pipeline can be described as a chain of actors where each actor is the Customer for its predecessor. Many idioms involve managing the Customer in various ways.

2.3 Sink

A Sink actor simply throws away all messages that it receives. If we make a Request, but don't care about the Reply, we use a Sink as the Customer.

```
sink_beh() ->
    receive
        _ -> sink_beh()
    end.
sink() ->
    spawn(fun sink_beh/0).
```

2.4 Forward

The Forward idiom is also quite simple. We can think of a forwarding actor as an Alias or Proxy. Messages sent to a forwarding actor are passed on to another actor (the Subject).

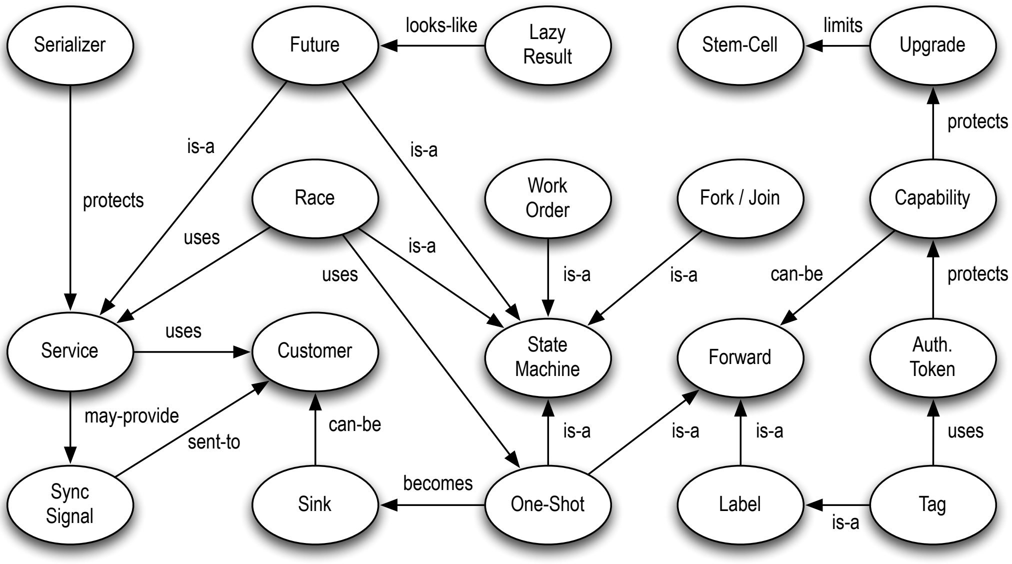


Figure 1. Idiom Relationships

```
forward_beh(Subject) ->
    receive
        Msg ->
            Subject ! Msg,
            forward_beh(Subject)
    end.
```

2.5 Label

A Label is a Forward actor that adds some fixed information to each message. It acts like a Decorator for messages [2]. Sometimes it plays the role of an Adaptor between actors, structuring messages to meet the expectations of the subject.

```
label_beh(Cust, Label) ->
    receive
        Msg ->
            Cust ! [Label | Msg],
            label_beh(Cust, Label)
    end.
```

2.6 Tag

A Tag is a special kind of Label. It labels each message with a reference to itself. A Tag actor is often used as a Customer for a Request when we want to identify a specific Reply.

```
tag_beh(Cust) ->
    receive
        Msg ->
            Cust ! [self() | Msg],
            tag_beh(Cust)
    end.
```

2.7 Sync-Signal

A Sync-Signal is used to indicate the completion of an activity. It is a message sent to a customer where the content is not important, only the fact that an event occurred. If a Service offers a Sync-Signal, and the caller doesn't need it, a Sink actor can be passed as the Customer.

3. State

The idioms in this section represent various ways to manage state in an actor configuration.

3.1 State-Machine

Every actor can be viewed as a State-Machine. When an actor *becomes* a new behavior, we can think of that as a state-transition. The *event* that triggers the state-transition is the message received by the actor. In Erlang, *become* is represented by a tail-call to a new behavior function.

3.2 One-Shot

A One-Shot actor is a Forward actor that will only forward a single message. Once it has forwarded a message, it changes state, becoming a Sink. Subsequent messages will be quietly ignored.

```
one_shot_beh(Cust) ->
    receive
        Msg ->
            Cust ! Msg,
            sink_beh()
    end.
```

3.3 Race

Sometimes we have more than one method available for generating a desired result. If we cannot determine which method will be fastest in a particular case, we would like to try all methods in parallel and use the result generated first.

A Race actor replicates a request, and sends it to multiple Services. The Customer for each service-call is a One-Shot shared by all the calls. The One-Shot is configured to forward the first result on to the original Customer.

```
race_beh(List) ->
    receive
        [Cust | Req] ->
            One_Shot = spawn(fun() ->
                one_shot_beh(Cust) end),
            send_to_all([One_Shot | Req], List),
            race_beh(List)
    end.

send_to_all(Msg, []) -> Msg;
send_to_all(Msg, [First | Rest]) ->
    First ! Msg,
    send_to_all(Msg, Rest).
```

3.4 Work-Order

It often takes several steps to fulfill a particular service request. A Work-Order is an actor that represents the state-transitions involved in processing a service request. An actor representing the Service interface receives the initial request and creates a Work-Order to handle subsequent interactions. When a final result is produced, the Work-Order sends it directly to the original customer, without involving the Service.

4. Coordination

The idioms in this section are used to coordinate activity among actors.

4.1 Capability

You can only send messages to an actor if you know the actor's address (an unforgeable secret). Conversely, if you know an actor's address you can send it any message you like. This means that each actor represents a Capability, and the reference graph represents authority [5].

When we want to restrict authority, we can interpose a capability attenuator actor. This is basically a Forwarding actor that selectively forwards only messages that conform to whatever policy we wish to enforce. Restricted access is therefore securely granted without exposing the original actor's address.

In addition, a Capability can support additional features. It may allow modification, or even revocation, of the capability. The capability to perform these actions may be granted through a separate Capability.

4.2 Authorization-Token

An actor's identity can be the Shared-Secret used to validate the authority to make a request. The request is tagged with an actor address representing an Authorization-Token. This is commonly used to validate modification or revocation actions on a Capability. A Tag actor uses the actor's identity as an Authorization-Token. This works because the actor that created the Tag is the one granting authority.

4.3 Future

A Future represents the result of a computation that executes concurrently with Customers that will eventually use the computed value. A Future has three states. Initially, the future has no value and no Customers waiting for the value. If Customers arrive to read the value, they are queued until the value is available. When the value is available, any waiting Customers are notified, and any subsequent Customers are given the value immediately. Once set, the value never changes.

```
future(Computation) ->
    V = spawn(fun future_beh/0),
    spawn(fun() ->
        Value = Computation(),
        V ! [sink(), write, Value] end),
    V.
```

```
future_beh() ->
    receive
        [Cust, write, Value] ->
            Cust ! self(),  % sync-signal
            value_beh(Value);
        [Cust, read] ->
            wait_beh([Cust]);
        _ ->
            future_beh()
    end.

value_beh(Value) ->
    receive
        [Cust, read] ->
            Cust ! Value,
            value_beh(Value);
        _ ->
            value_beh(Value)
    end.

wait_beh(Waiting) ->
    receive
        [Cust, write, Value] ->
            send_to_all(Value, Waiting),
            Cust ! self(),  % sync-signal
            value_beh(Value);
        [Cust, read] ->
            wait_beh([Cust | Waiting]);
        _ ->
            wait_beh(Waiting)
    end.
```

4.4 Lazy-Result

A Lazy-Result is the opposite of a Future, from a timing perspective, although their protocols are the same. Whereas a Future eagerly computes a result, and may determine the value before any Customer asks for it. A Lazy-Result does not begin computation until at least one Customer asks for the value. This strategy can avoid unnecessary computation, and even non-termination in some cases.

```
lazy(Computation) ->
    spawn(fun() ->
        lazy_beh(Computation) end).

lazy_beh(Computation) ->
    receive
        [Cust, read] ->
            V = self(),
            spawn(fun() ->
                Value = Computation(),
                V ! [sink(), write, Value] end),
            wait_beh([Cust]);
        _ ->
            lazy_beh(Computation)
    end.
```

4.5 Fork-Join

The Fork-Join idiom describes a group of parallel computations that synchronize on completion of all the computations. If the computations produce values, their values are collected into a single combined result. If computations do not produce values, they must at least provide a Sync-Signal to indicate their completion.

The basic Fork-Join idiom coordinates two independent computations. Multiple Forks can be composed, extending the idiom to an arbitrary number of parallel computations. The results are ordered in the same way the computations were composed.

```
fork_beh(Cust, H, T) ->
    receive
        [H_Req | T_Req] ->
            S = self(),
            H_Tag = spawn(fun() ->
                tag_beh(S) end),
            T_Tag = spawn(fun() ->
                tag_beh(S) end),
            H ! [H_Tag | H_Req],
            T ! [T_Tag | T_Req],
            join_beh(Cust, H_Tag, T_Tag)
    end.

join_beh(Cust, H_Tag, T_Tag) ->
    receive
        [H_Tag | Head] ->
            receive
                [T_Tag | Tail] ->
                    Cust ! [Head | Tail]
            end;
        [T_Tag | Tail] ->
            receive
                [H_Tag | Head] ->
                    Cust ! [Head | Tail]
            end
    end.
```

4.6 Serializer

The Serializer is the fundamental mechanism for mutual-exclusion among potentially interfering computations. A Serializer ensures that a Service completes computation of a result for one request before accepting any subsequent requests. The coordination performed by a serializer is built on the Request-Reply protocol between a Service and a Customer.

```
serializer_beh(Service) ->
    receive
        [Cust | Req] ->
            S = self(),
            Tag = spawn(fun() ->
                tag_beh(S) end),
            Service ! [Tag | Req],
            Q = queue:new(),
            busy_beh(Service, Cust, Tag, Q)
    end.

busy_beh(Service, Cust, Tag, Waiting) ->
    receive
        [Tag | Reply] ->
            Cust ! Reply,
            case queue:is_empty(Waiting) of
                true -> serializer_beh(Service);
                false ->
                    [C | R] = queue:head(Waiting),
                    Q = queue:tail(Waiting),
                    S = self(),
                    T = spawn(fun() ->
                        tag_beh(S) end),
                    Service ! [T | R],
                    busy_beh(Service, C, T, Q)
            end;
        [C | R] ->
            Q = queue:in([C | R], Waiting),
            busy_beh(Service, Cust, Tag, Q)
    end.
```

5. Configuration

The idioms in this section describe mechanisms for dynamically altering the run-time configuration of an actor system.

5.1 Stem-Cell

The most flexible (and consequently most dangerous) mechanism is the Stem-Cell. A Stem-Cell simply takes on the behavior it receives in a message. In this way, it can literally become *anything*. The behavior of a Stem-Cell is limited only by the information available to the sender of the message.

```
stem_cell_beh() ->
    receive
        Behavior -> Behavior()
    end.
```

5.2 Upgrade

A more limited kind of configuration control is offered by the Upgrade. Instead of replacing the entire behavior of an actor (like the Stem-Cell), an Upgrade replaces, or parameterizes, a controlled portion of an actor behavior.

There is considerable flexibility in specifying relevant kinds of Upgrades. Authority to perform these Upgrades is often granted by Capabilities.

6. Conclusion

We have only scratched the surface of the rich variety of idioms that occur in Actor programming. It is my hope that this catalog will be greatly extended. The idioms we have explored can be described in much more detail. We should examine the contexts in which they operate, the forces involved, the trade-offs among alternatives, and the idioms' relationships to each other.

We should keep in mind that effective use of the Actor model requires careful consideration of message protocols and interactions. We should be more concerned about behavior, and less concerned about data.

Acknowledgments

This paper was greatly improved by ongoing conversations and collaboration with Tristan Slominski and Carl Hewitt, as well as by comments from reviewers. Any remaining errors are mine alone.

References

[1] J. Armstrong (2003) *Making reliable distributed systems in the presence of software errors*. Ph.D. Dissertation. The Royal Institute of Technology, Stockholm, Sweden.

[2] E. Gamma, R. Helm, R. Johnson, J. Vlissides (1995) *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA.

[3] C. Hewitt (1976) *Viewing Control Structures as Patterns of Passing Messages*. AI Memo 410, MIT. Journal of Artificial Intelligence, June 1977.

[4] C. Hewitt (2011) *Actor Model of Computation: Scalable Robust Information Systems*. arXiv:1008.1459

[5] M. S. Miller (2006). *Robust Composition: Towards a Unied Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, Baltimore, MD.