The Language as a Software Engineer

Margaret H. Hamilton

mhh@htius.com

www.htius.com

May 31, 2018

When we ask developers who are looking for a better way to develop software and we ask them what their most pressing issues are:

- Integration too late if at all
- Lack of traceability, flexibility and evolvability
- Reuse methods ad hoc and error prone
- Software unreliable even with extensive testing
- Costs too much. Takes too long

Why still? Not unlike 50 years ago when the field was brand new*. What to do...

* M. Hamilton, “What the Errors Tell Us”, IEEE Software - Special issue "50 years of Software Engineering"

Most people would say impossible to do much—at least in the foreseeable future—software by its very nature is destined to have these kind of problems

Nevertheless, I believe it is not impossible, partly based on many of my own experiences together with others throughout the last 60 years or so:

- Some of the software projects
- What has and what has not changed
- Lessons we learned along the way
- What we have done (are doing) about it
- Implications for systems of the future

Observations on Earlier Experiences

- No field for software engineering...you were on your own...knowledge (or lack thereof) passed down from person to person
- A manager hired you if you "knew" the commands in *his* computer's native language. Like "knowing" a set of English words would mean you could write a novel
- Tricky programmers admired; organization vulnerable when one left
- If a system crashed, software the one blamed (THERAC-25)
- Terms undefined: leading to errors, misunderstandings and drama; e.g., "software", "error" and "computer systems" meant different things to different people
- Tribal. Software "types" could mix things up if involved with software areas unfamiliar to them; e.g., the operating system functionality with the target system functionality
- Some things quite different then; some not
- Life cycle not unlike today's traditional life cycle. Waterfall, Spiral, Agile... going from requirements to coding to endless testing and maintenance

First Project (LGP-30)

- Introduced to computers by Ed Lorenz at MIT in 1959
- Developed weather prediction applications in hexadecimal and binary
- Known for his ground breaking work in chaos theory, his love for software related experimentation was contagious
- Improving development/productivity techniques
  - Hardware/software relationships
  - Errors a nuisance. Each debug session took forever.
  - Paper tape solution
  - Stage computer runs and spend more time up front

Important not to make an error (the computer told everyone)!

When it crashed, we heard loud siren-like and fog horn-like sounds.

Since it belonged to the programmer standing in front of the console, it was no secret

Operators and programmers would come running to find out whose program it was

The crash site where the program halted could be found in a foot-long register on the console with its blinking and flashing lights

Another Early Project (SAGE)

Used assembly language on the first AN/FSQ-7 (XD-1), at MIT's Lincoln Labs, to look for enemy airplanes. HUGE machine; largest computer system ever built

- *Important not to make an error (the computer told everyone)!*
- *When it crashed, we heard loud siren-like and fog horn-like sounds. Operators and programmers would come running to find out whose program it was*
- *Since it belonged to the programmer standing in front of the console, it was no secret*
- *The crash site where the program halted could be found in a foot-long register on the console with its blinking and flashing lights*
- Next step: write the contents of the register on a piece of paper
- One challenge: keeping track of which program caused which crash
- My solution: take polaroid picture of each programmer posing next to his or her bug
- The pictures became more creative as time went on

SAGE: semi-automatic ground environment

Another Early Project (SAGE)...cont.

- One time, one computer operator called me at home at 4am. "Something terrible happened. Your program no longer sounds like a seashore".
- I got in the car and rushed to work. We had found a new way to debug, using sound
- Again, debugging was time consuming. No tools for finding errors
  - "Ernie challenge"...Latin and Greek
  - 24 hour turnaround encouraged careful thought and “playing computer” at the front end; multiple test cases created for each night's test runs
  - Followed by endless rounds of testing
- Fascination with errors: a never ending past time of mine was to look for more ways to understand what made a particular error(s) or class of errors happen and how to prevent it in the future (e.g., documenting code)
- Sage definitely came with drama, especially having to do with errors; but, this was only the beginning of what would come next: the Apollo onboard flight software project at MIT, under contract to NASA

Apollo On-Board Flight Software

- The challenge was unique: build man-rated software; meaning astronauts' lives were at stake. It had to WORK—the first time
- Not only did the software, itself, have to be ultra-reliable, but it would need to be able to detect an error and recover from it in real time
- Learning by "doing" and "being". Hardware engineers came with rules; we didn't. Problems had to be solved that had never been solved before. At times, we made it up
- Most developers were fearless and young; yet, dedication and commitment a given
- Managers (mostly from hardware backgrounds) for whom software was a mystery, gave us total freedom and trust

No Time to be a Beginner

- Unmanned missions
  - Unmanned missions flight software synchronous
  - AGC Block I assembly language
- Beginner's experience
  - "Augekugel" method
  - Lunar Landmark tables
  - "Forget it"?
- Manned missions came next
  - Manned missions flight software asynchronous
  - AGC Block II assembly language, interpreter language

On-Board Flight Software More Complex for Manned Missions

- Asynchronous (multi-programming environment): higher priority jobs interrupted lower priority jobs based on every job's priority relative to every other job's priority
- Developers manually assigned a unique priority to every process in the flight software to ensure all events would take place in the correct order and at the right time
- It quickly became clear; no one and nothing is perfect (software gurus, hardware gurus, *even* astronauts)
- Always searching for new ways to prepare for and recover from the unexpected: going from program specific to system-wide protection.
  - Lightning struck spacecraft at least twice
  - P01 (Apollo 8): daughter's "debug" session and reactions, program note resolution, real-time "debug" in flight, program change implemented for all later missions
- Houston meeting said it all

MIT SCAMA Room

P01 (Apollo 8): daughter's "debug" session and reactions, program note resolution, real-time "debug" in flight, program change implemented for all later missions.

Photograph taken by MIT-IL (now Draper). Courtesy of Draper.

"July 17, 1969 at MIT, during the Apollo 11 mission... checking the monitor console and analyzing data on information received while in direct communication with Houston's Mission Control."

Each Mission Exciting. Apollo 11 Special
We had Never Landed on the Moon Before

- Everything going perfectly until something totally unexpected happened; just as the astronauts were about to land on the moon, the flight computer became overloaded
- The software's Priority Displays (AKA Display Interface Routines) of 1201 and 1202 alarms interrupted the astronaut's normal mission displays to warn them there was an emergency
  - Allowing NASA's Mission Control to understand what was happening
  - Alerting the astronauts to place the rendezvous radar switch back in the right position
- It quickly became clear that the software not only informed everyone there was a hardware-related problem, but the software was compensating for it
- The Priority Displays gave the astronauts a go/no go decision (to land or not to land)
- With only minutes to spare, the decision was made to go for the landing
- The rest is history. The Apollo 11's crew became the first humans to *walk* on the moon; and, our software became the first software to r*un* on the moon

Software's Systems-Software Error Detection and Recovery Mechanisms Took Control

- System-wide “kill and recompute” from a "safe place" snapshot-rollback *restarts triggered by overload*; keeping only the highest priority jobs (Priority Displays highest priority)
- Steps earlier taken within the multi-programming environment became the basis for solutions within a multiprocessing environment
- With this as a backdrop, the Priority Displays were created, changing the interface between the astronauts and the flight software from synchronous to asynchronous (the software and astronauts becoming parallel processes in a system of systems)
- This would not have been possible without an integrated system of systems (and teams) approach and contributions made by other groups to support our flight software's systems-software team in making this become a reality
- The hardware team at MIT changed their hardware and the mission planning team in Houston changed their astronaut procedures; both working closely with us to accommodate the Priority Displays for both the CM and the LM; for any kind of emergency and throughout any mission
- Mission Control well-prepared to know what to do if the Priority Displays interrupted the astronauts

Story Behind the Story

- On-board flight software systems-software meeting with hardware people (65-66)
- Proposal for error detection and recovery in real time during an emergency
- Hardware concerns
- System concerns
- Solution...”count to 5”
- Resolution: all systems go!
- Steps taken towards implementation

Apollo Experiences Gave Insight to Understanding Software (and its Life Cycle) as a System

- Everything somehow related to everything else; for better or worse (e.g., systems software error's impact on everyone's runs and off-line versions)
- The very way one communicates can cause or prevent crashes, ACS daily memos. Define terms better (e.g., what is an "error", catastrophic vs FLT)
- Programmers and mission designers necessarily became interchangeable; as did life cycle phases
- Relationship of real-time and development (async)
- Everything a moving target. The only constant is change
- Never say never: more than one way to solve a problem (how affected, not what caused it)
- Once interface errors (data, timing, priority conflicts) resolved, related issues resolved: e.g., integration, traceability, flexibility, evolvability and reuse

Opportunity to Make Every Kind of Error Humanly Possible, Each With Hidden Secrets. On Hindsight, a Blessing in Disguise

- Task at hand: develop the CM and the LM software, including the systems-software shared between them and the structure of the software ("glue") that defined the relationships between and among the mission phases.
- Updates continuously submitted from hundreds of people (including "guests") over time and many releases for each and every mission (software for one mission worked on concurrently with software for other missions)
- Making sure everything would play together; that the software parts would successfully interface to and work together with each other as well as with other systems (hardware, peopleware and missionware)
- Handicapped by hardware time and space constraints: software "experts" prided themselves with tricky programming; carefully crafted, creative code admired more than number of lines of code a person wrote
- "Requirements thrown over the wall": assumptions by some "non-software experts" that all software programs would somehow interface together "magically"

Opportunity to Make Just About Every Kind of Error Humanly Possible...(cont.)

- Shared erasables (data) between mission phases
- With multi-programming, shared responsibilities and more interfaces within and between every mission phase (8 tasks based on timing, 7 jobs based on priority); man-in-the-loop multi-processing within the overall system of systems
- Push and pull on trade-offs: e.g., more error detection and recovery vs more accuracy in equations
- We evolved our “software engineering” rules with each new relevant discovery; while top management rules from NASA went from complete freedom to bureaucratic overkill
- Not possible (certainly not practical) to test the software by ”flying”a real mission: 6 levels of testing
- System of systems simulations: mix of hardware and digital simulations of every (and all aspects of an) Apollo mission which including man-in-the-loop simulations (real or simulated human interaction), making sure that a complete mission from start to finish would behave exactly as expected (of course, assuming the simulations themselves were ultra-reliable)
- *No on-board flight software errors ever known to occur during flight*

Having Been Through These Experiences One Could Not Help but Learn from Them

- We asked ourselves,"what can we do better for future systems? What should we keep doing because we are doing it right?"
- We were on a new "mission": create more advanced means for designing systems and building software
- Goal: address problems considered next to impossible to solve, if not impossible, with traditional approaches; at least in the foreseeable future
- With initial funding from NASA and DoD, we performed an empirical study of the Apollo effort

Analysis has Taken on Multiple Dimensions, not just for Space Missions but Systems in General: Lessons Learned from this Effort (and their Impact) Continue Today:

- Always ask "what if?". Always "expect the unexpected".
- Systems are asynchronous, distributed and event driven in nature. This should be reflected in the language to define them and the tools to build them, characterizing natural behavior in terms of real-time execution semantics
- Once having done so, no longer a need to explicitly define schedules of when events occur. By describing interactions between objects, the schedule of events is inherently defined
- The life cycle of a target system is a system with its own life cycle
- Every system inherently a system of systems

Most Interesting of All: Errors Found (During Pre-Flight Testing of the Software)

- The errors were full of surprises: *they told us what to do and where to go*
- We concentrated on:
  - Interface errors (data, timing and priority conflicts)
  - Errors found by manual means (Norton, Augekugel...)
  - Previously existing errors (most subtle and hardest to find)
- Categorizing the errors led to a systems theory of control* that has continued to evolve, based on lessons learned from Apollo and later projects
- Its axioms led to a set of allowable patterns that led to a language evolving together with its automation** and preventative development paradigm, development before the fact (DBTF)

* M. Hamilton and W. Hackler (2008), "Universal Systems Language: Lessons Learned from Apollo", IEEE Computer, Dec. 2008.
** http://htius.com/Examples/tax_example/documentation/do_all_taxes.op.collector-full

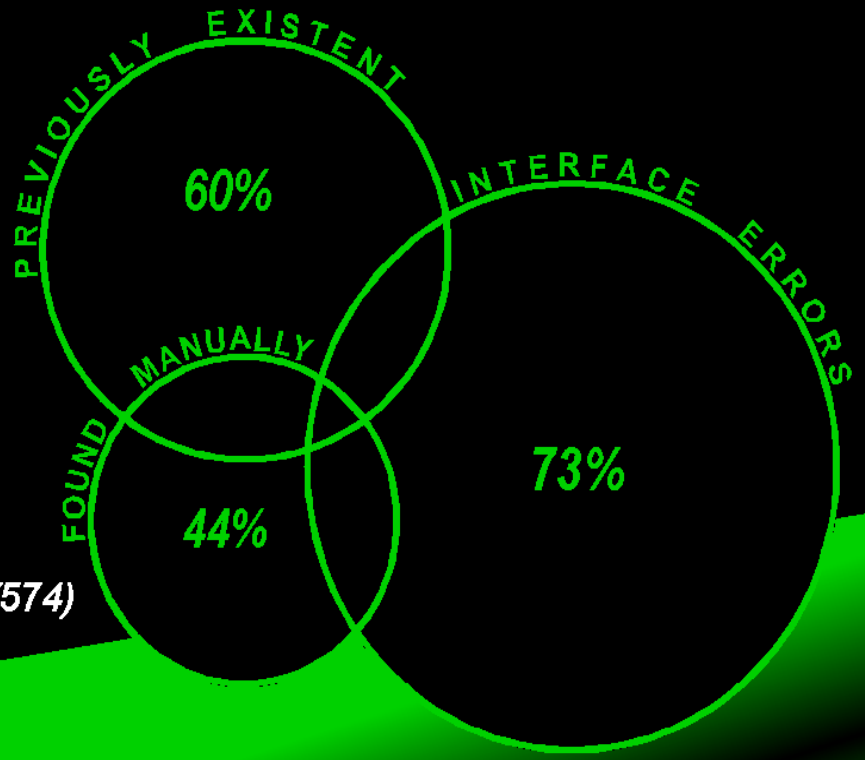
Began with

an Empirical Study

of Apollo and

Skylab Software

and its Development



Official Pre-flight Anomalies (574)

The First Results: a Formal Systems Theory Based on Six Axioms

Root problem: traditional system engineering and software development languages and their environments support users in "fixing wrong things up" ("after the fact") rather than in "doing things in the right way in the first place" ("before the fact").

A system defined with the language, the universal systems language (USL), that has the systems theory of control as its foundation, has properties that inherently support its own development, “before the fact”. With this language

- Every object a System Oriented Object (SOO), itself developed in terms of other SOOs. A SOO integrates all parts of a system including function, object and timing oriented. Every system an object; every object a system
- Instead of Object Oriented Systems, System Oriented Objects. Instead of model driven systems, system driven models
- Unlike traditional languages, it is based on a preventative philosophy: instead of finding more ways to test for errors, late into the life cycle, find ways not to allow them, in the first place; just by the way a system is defined

With the Language, Every System Defined from the Very Beginning to Inherently:

- Integrate all of its parts (e.g., types, functions, timing, structures)
- Maximize its own reliability
- Capitalize on its own parallelism
- Maximize the potential for its own
  - Reuse
  - Automation
  - Evolution

RESULT: a formal based system with built-in quality, and built-in productivity for its own development

Every System Defined with Properties of Control

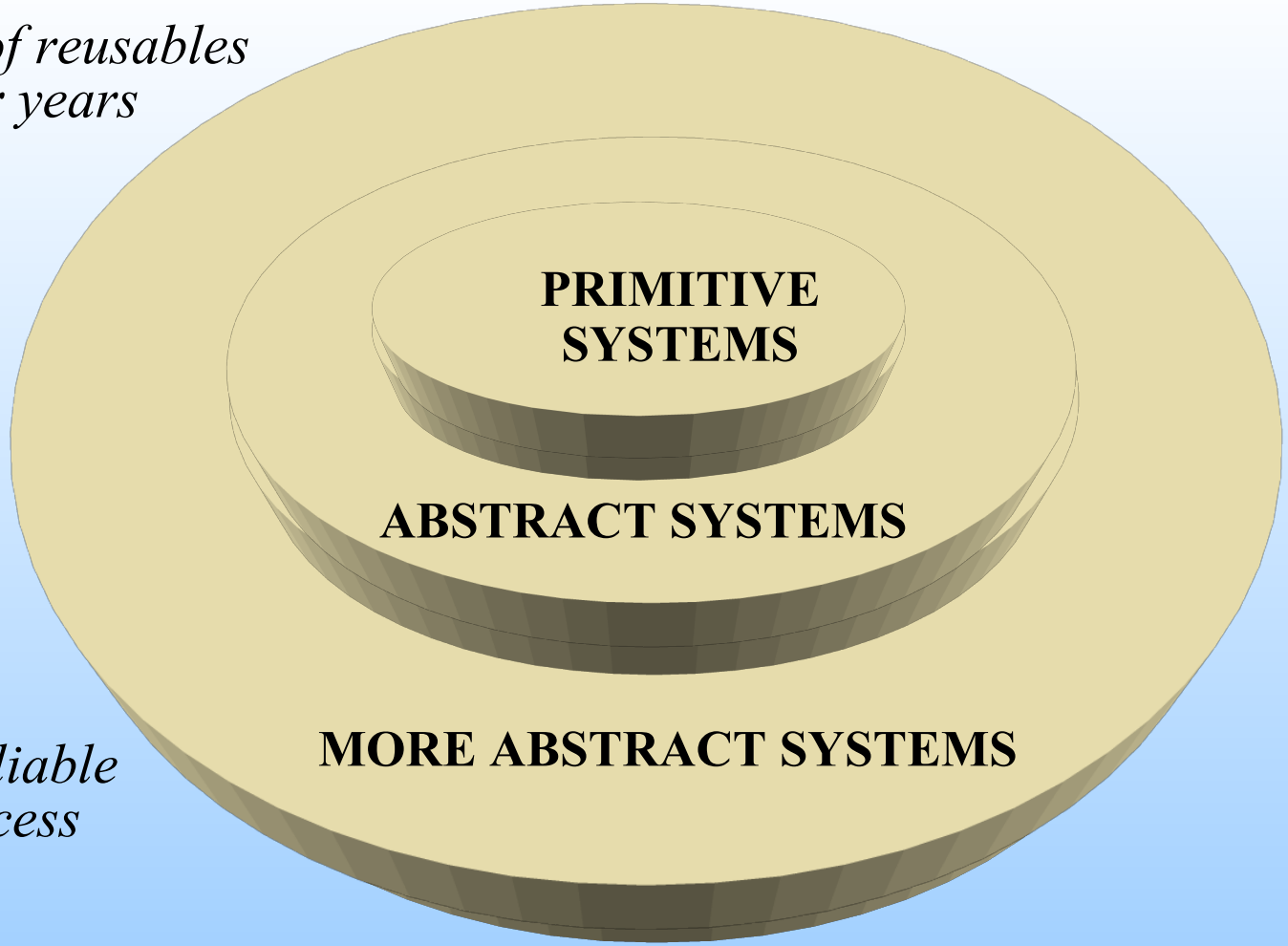
- A formalism for representing the mathematics of systems, USL is based on a set of axioms and formal rules for their application
- Same language used to define and integrate
  - All aspects of and about a system and its relationships and its evolutions
  - Functional, resource and allocation architectures, including hardware, software and peopleware
  - Sketching of ideas to complete system definitions
  - GUI with documentation…with application
  - All definitions
- Syntax, implementation, and architecture independent
- Unlike formal languages that are not friendly or practical, and friendly or practical languages that are not formal; USL is considered by its users to be not only formal, but friendly and practical as well
- Unlike a formal language that is mathematically based but limited in scope from a practical standpoint (e.g., kind or size of system), USL extends traditional mathematics with a unique concept of control enabling it to support the definition of any kind or size of system

Process of Building a System

- ***(Re)Define*** model with USL
- ***Analyze*** automatically the model to ensure it was defined properly
- ***Generate*** automatically much of the design and all of the code, production ready, for any kind or size of system
- ***Execute*** the model
- ***Deliver*** the real system

USL Philosophy:
Reliable Systems Defined in Terms of Reliable Systems

A large library of reusables has evolved over years of development.



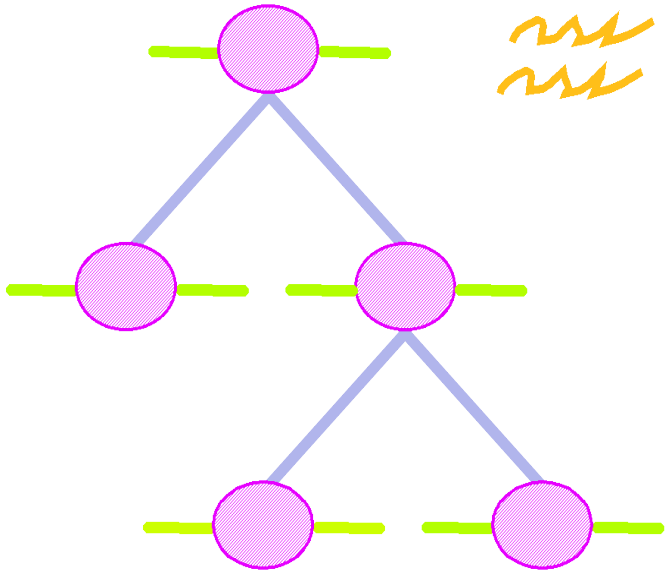
A recuarsively reliable and reusable process

- Use only reliable systems
- Integrate these systems using reliable systems
- The result is a system(s) which is reliable
- Use resulting reliable system(s) along with more primitive ones to build new and larger reliable systems

Every System Defined with Function Maps (FMaps) and Type Maps (TMaps), the Major Building Blocks of USL. Every FMap and Every TMap is Defined in Terms of 3 Primitive Control Structures

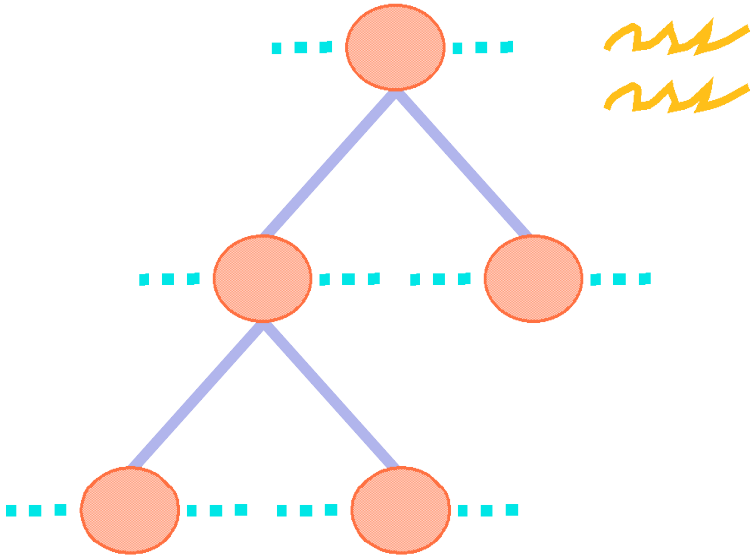
Model Relationships between Functions (Time)

with **Function Map** (FMap)

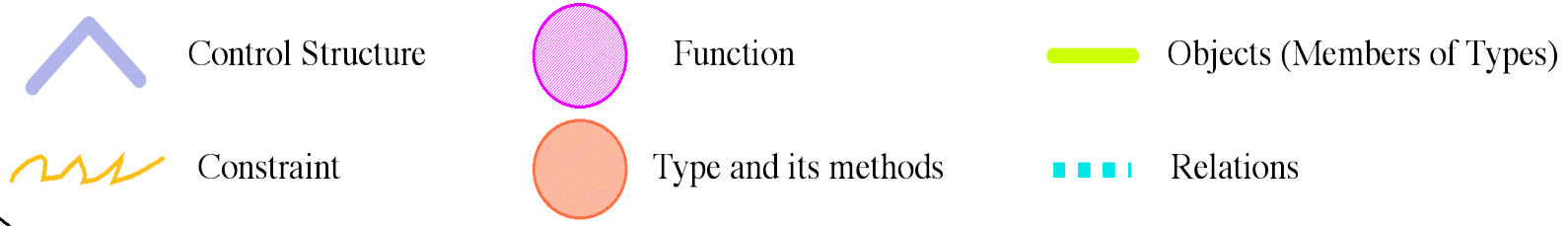


Model Relationships between Types (Space)

with **Type Map** (TMap)



Ultimately in terms of 3 primitive control structures



All model viewpoints can be obtained from FMaps and TMaps. FMaps of functions are by their very nature integrated with TMaps of types*.

TMap properties ensure the proper use of objects in an FMap. Types TMap and Object Map (OMap, an instance of a TMap), facilitate the ability of a system to understand itself better and manipulate all objects the same way.

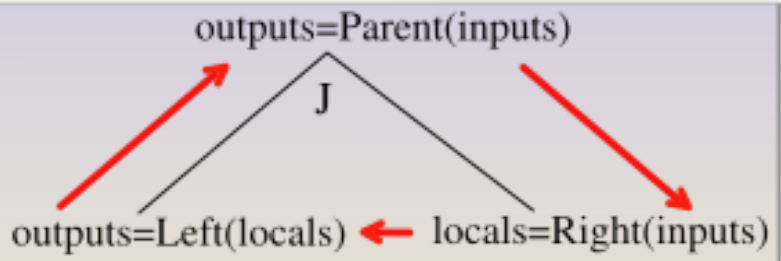
Primitive types reside at the bottom nodes of a TMap. Each type is defined by its own set of axioms. Inputs and outputs of each function are members of types in the TMap. Primitive functions in an FMap, each defined by a primitive operation of a type on the TMap, reside at the bottom nodes of an FMap. Each primitive function (or type) can be realized on a top node of a map on a lower (more concrete) layer of the system.

A system is defined from the very beginning to inherently *integrate* and make *understandable* its own real world definition.

*Map: tree of control spanning networks of relations between objects

The Three Primitive Control Structures

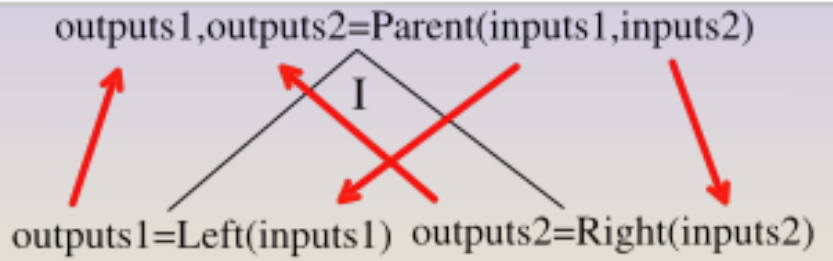
Dependent relationships



Rules Governing Join (J)

Inputs to parent are identical to inputs of right offspring (including order).
Outputs of parent are identical to outputs of left offspring (including order).
Outputs of right child are identical to inputs of left offspring (including order).

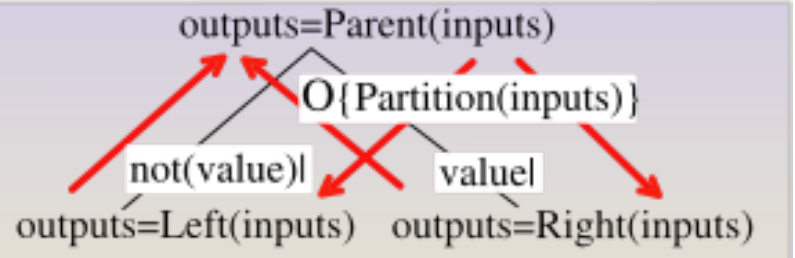
Independent relationships



Rules Governing Include (I)

A parent sends all its inputs to its children.
Children send all their outputs to their parent.
Order of inputs and outputs is maintained.
Children do not share inputs or outputs.
Left Child receives the first parent inputs. Right Child receives the rest.
Left Child sends the first outputs to parent. Right Child sends the rest.

Alternative relationships



Rules Governing Or (O)

Inputs of both offspring are identical to inputs of parent (including order).
Outputs of both offspring are identical to outputs of parent (including order).
Inputs of partition function are identical to inputs of parent (including order).

Where: inputs, locals, outputs, inputs1, inputs2, outputs1 and outputs2 are Ordered Sets of variables. In the Include structure, the Left child is a higher priority than the Right child; and the leftmost output variable is the highest priority variable.

A USL system model defined in terms of the three primitive control structures will have all its (and its derivatives') interface errors (~75% to 90% of all errors) eliminated at the definition phase. These are typically found (if they are found) during testing in traditional development.

Each of the 3 primitive control structures has a set of rules that follow the 6 axioms.

A system is defined from the very beginning to inherently maximize its own reliability and predictability

Definition for Making a Table

Requirements: Build a system for making a table. The legs are round and the top is flat; both made of hard or soft wood.

Determine:
- relevant parts (objects)
- tasks needed (actions) and their relationships for making a table using available parts

Where flat and round are each a wood type of object.

FMap table=MakeATable(flat,round)Join

table=assemble(top,legs) ← top,legs=make_parts(flat,round)Include

table assembly depends on parts, top and legs, being made

the parts, top and legs, are made independently

top=MakeTop(flat) Or:is:soft,wood(flat)

legs=make_legs(round)

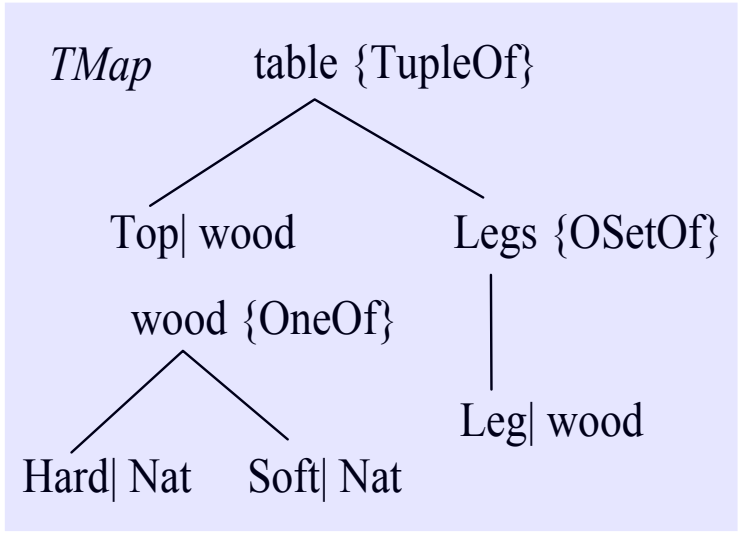
different finish is applied depending on the type of wood

top=FinishSoftWood(flat)

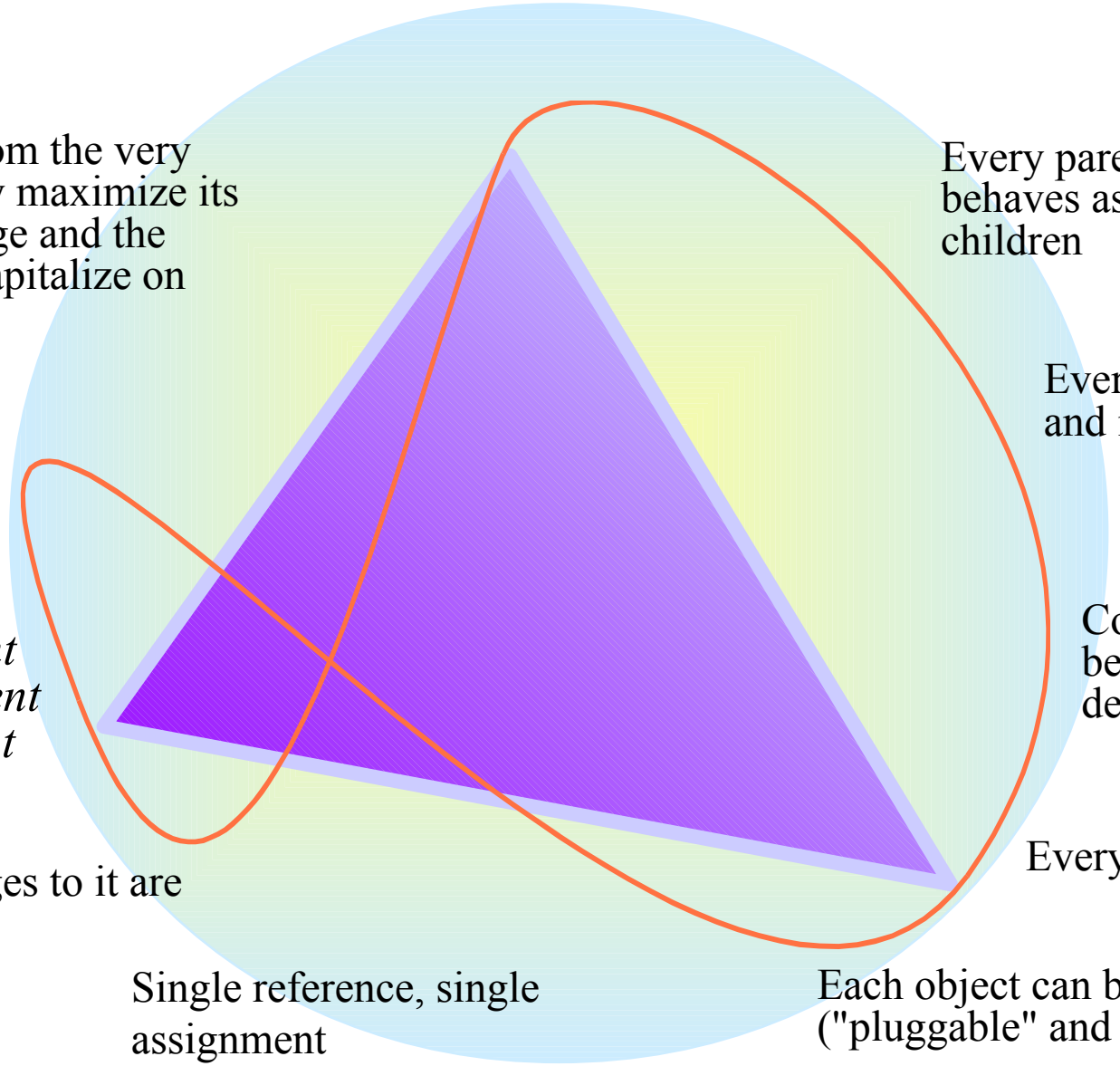
True: when flat wood is Soft

top=FinishHardWood(flat)

False: when flat wood is Hard



Systems Defined in Terms of the Primitive Control Structures Result in Properties for Real Time Distributed Environments



A system is defined from the very beginning to inherently maximize its own *flexibility* to change and the unpredictable and to capitalize on its own *parallelism*

Every parent has a higher priority and behaves as a master scheduler for its children

Every object has a unique parent and is under control

Every system is event-driven

Concurrent patterns can be automatically detected

Every input is an event
Every output is an event
Every function is event driven

Every object has a unique priority

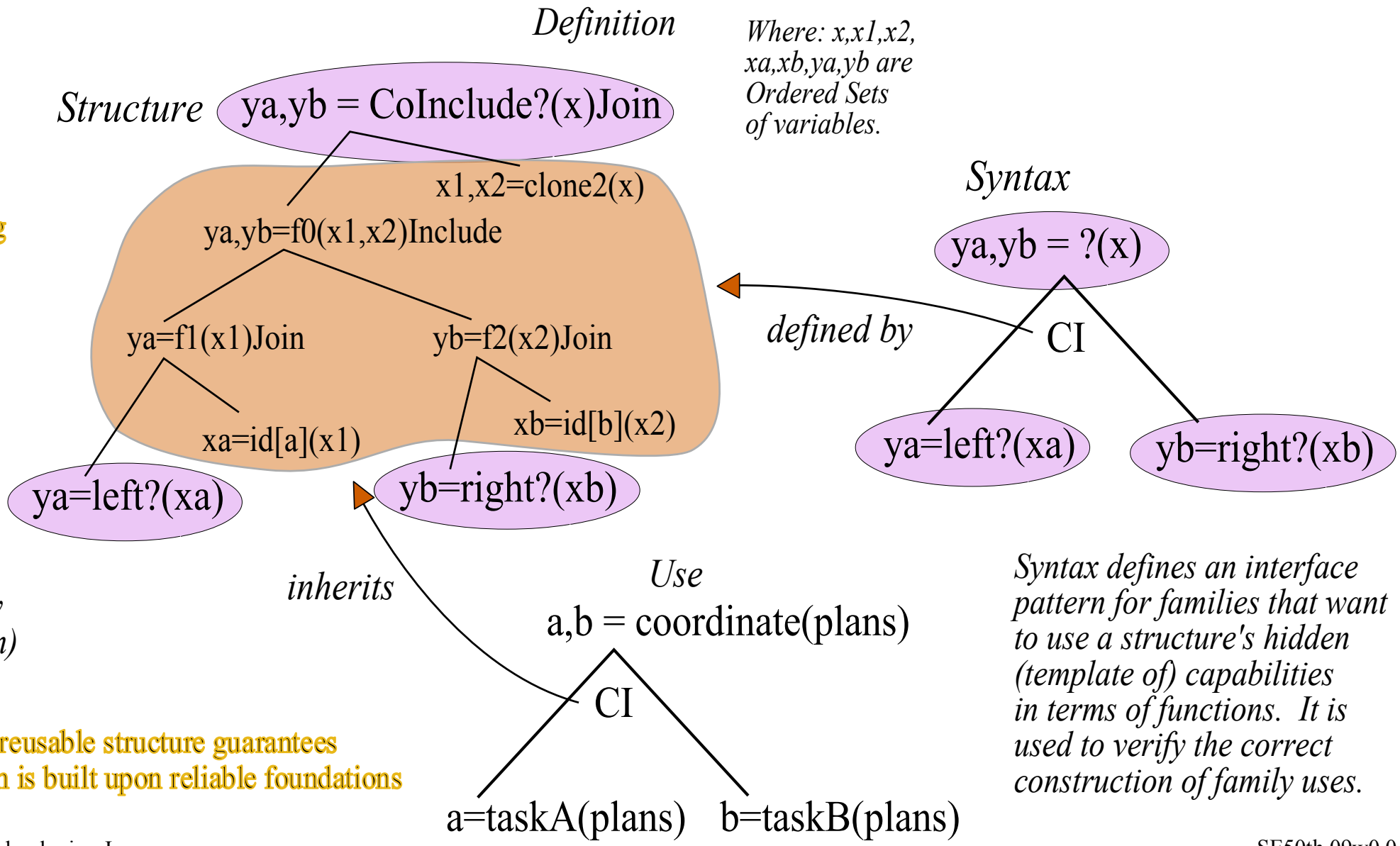
Each object and changes to it are traceable

Single reference, single assignment

Each object can be safely reconfigured ("pluggable" and "unpluggable")

A System is Defined from the Very Beginning to Inherently Maximize the Potential for its Own Reuse

A Derived FMap Structure



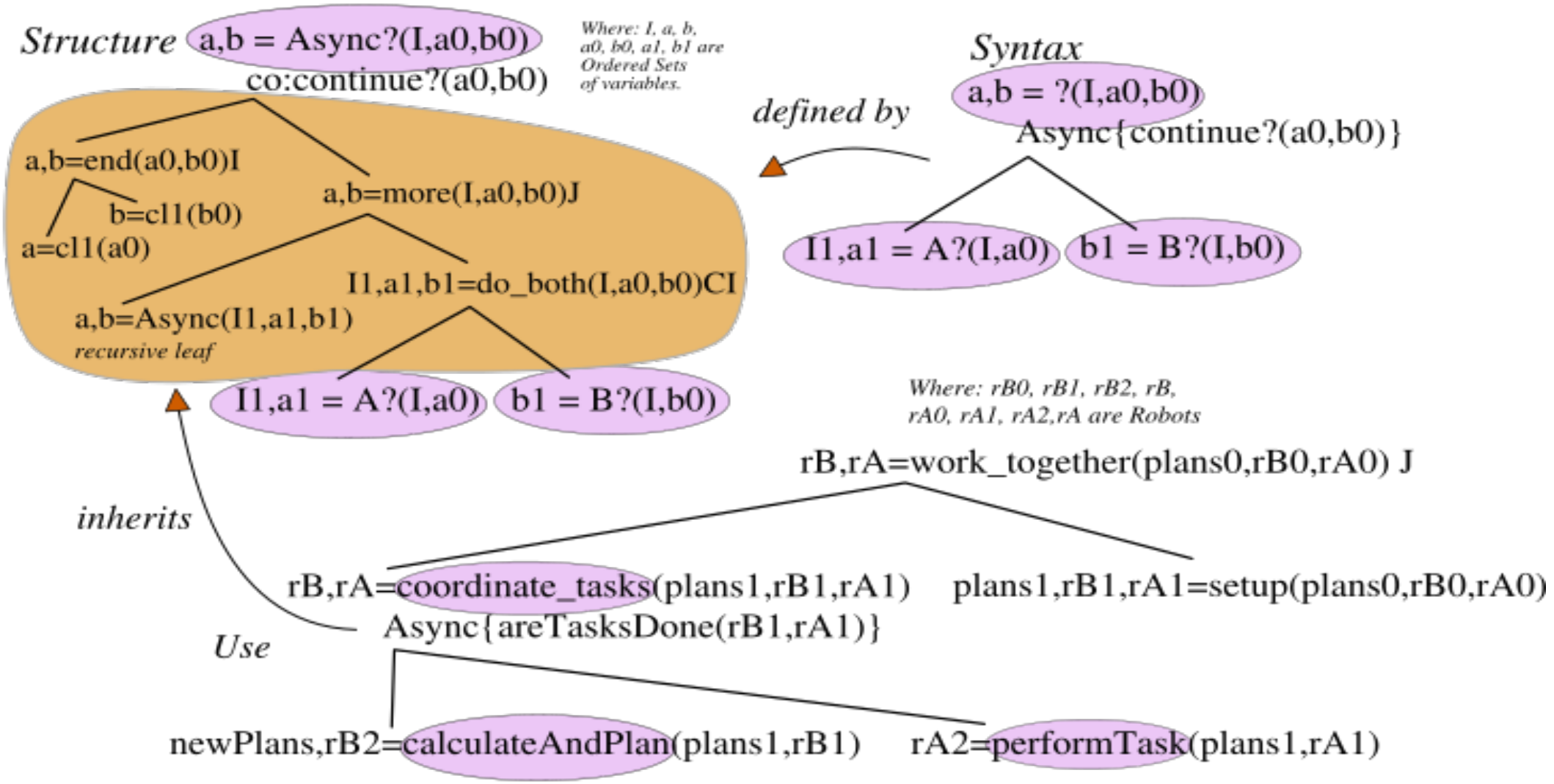
define underlying detail in terms of primitive foundations

hidden functions to be applied when used in another map to structure a particular nodal family (a parent and its children)

using a reusable structure guarantees a system is built upon reliable foundations

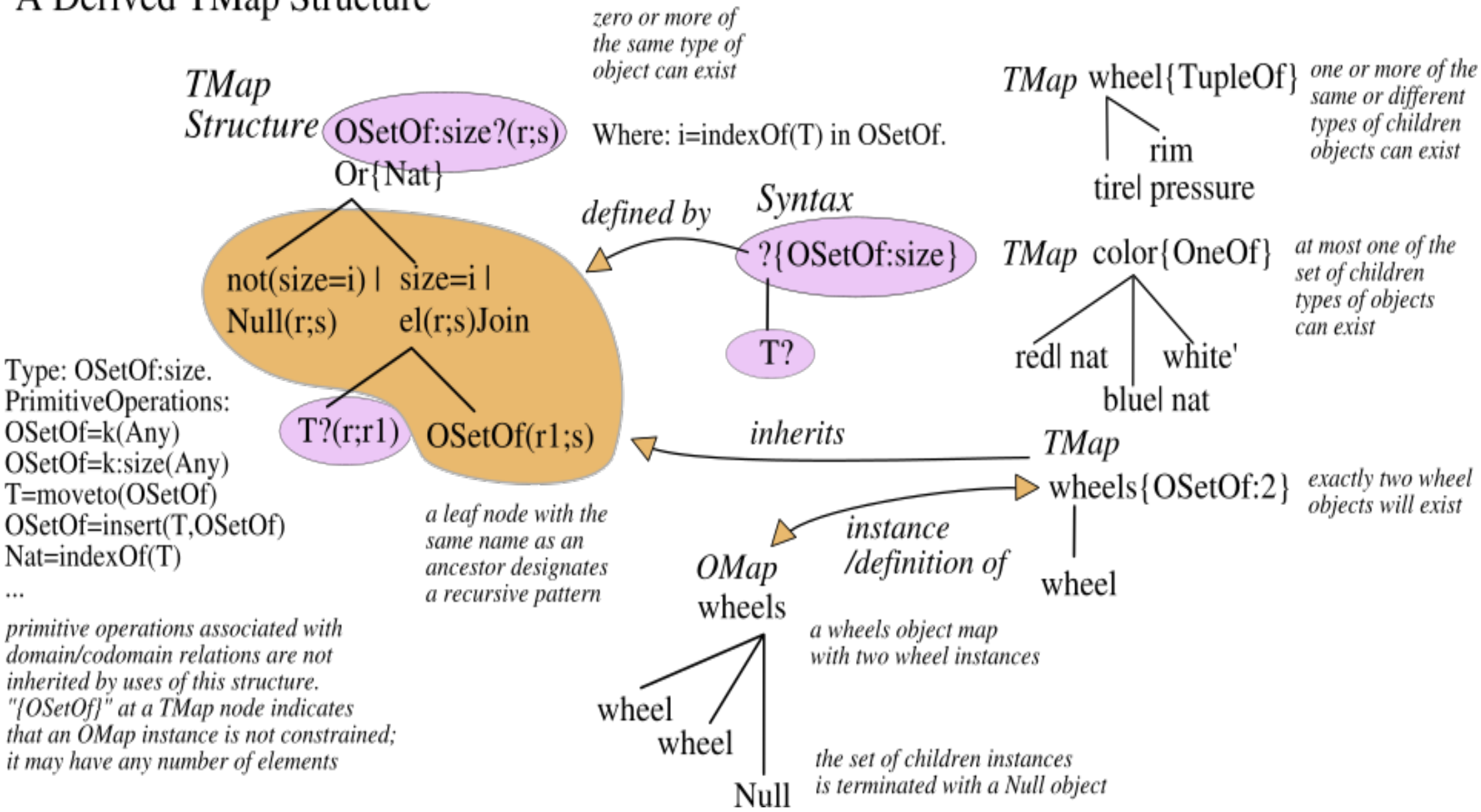
Syntax defines an interface pattern for families that want to use a structure's hidden (template of) capabilities in terms of functions. It is used to verify the correct construction of family uses.

An Async Structure (that can be Distributed) and its Use with both Synchronous and Asynchronous Behavior

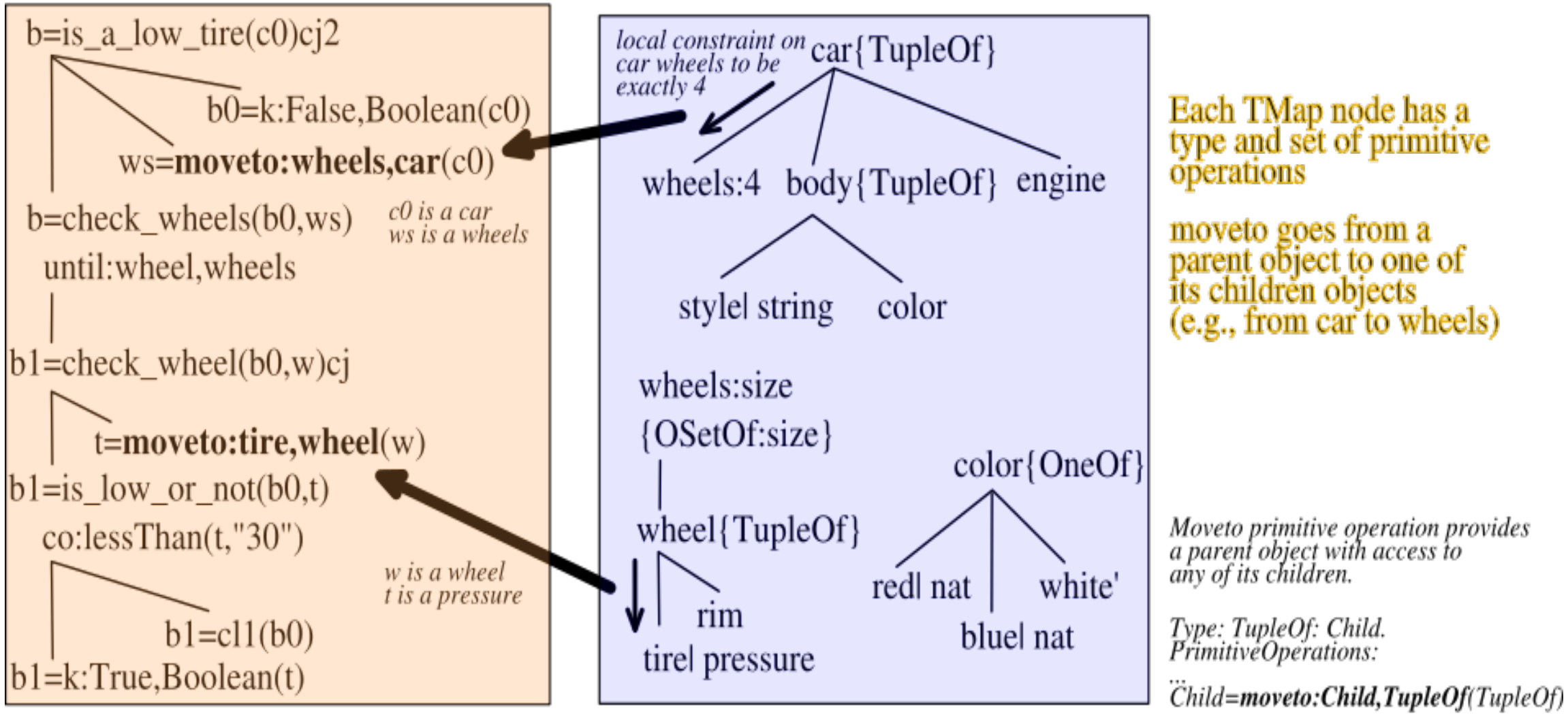


To understand Async's use at coordinate_tasks, go to Async's definition and to understand do_both go to CI's definition. Ultimately, coordinate_tasks is defined in terms of the Join, Include and Or 5primitive control structures.

A Derived TMap Structure



A System: Integration of FMaps and TMaps



Each TMap node has a type and set of primitive operations

moveto goes from a parent object to one of its children objects (e.g., from car to wheels)

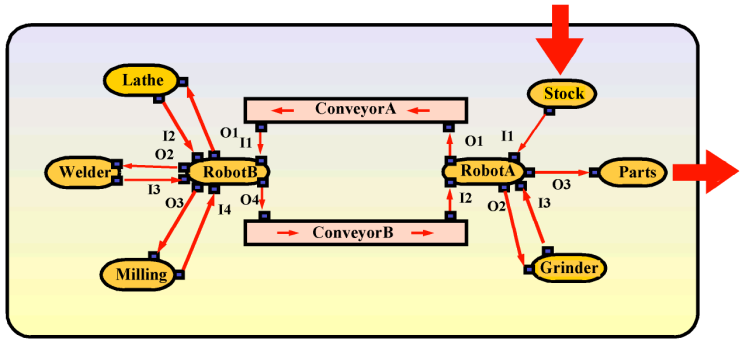
Moveto primitive operation provides a parent object with access to any of its children.

Type: TupleOf: Child.
PrimitiveOperations:
...
*Child=**moveto:Child,TupleOf**(TupleOf)*

Each abstract type (a parent) in a TMap inherits primitive operations from its type structure.
Both car and wheel inherit their moveto primitive operations from the TupleOf type structure as:
*wheels=**moveto:wheels,car**(car), and pressure=**moveto:tire,wheel**(wheel)*

These primitive operations may then be applied as primitive functions in an FMap.

Operation: Rotate Robot Arm.



TMap

RobotA
TupleOf
MfgObj|Nat
PickUpTime|Nat
PutDownTime|Nat
TurnRate|Nat
RotationTime|Nat
Ports

Ports
TupleOf
IOPorts.
AtPort|OPort!
APortIds.

APortIds
OneOf
Stock'
ConveyorA'
ConveyorB'
Grinder'
Parts'

IOPorts
OSetOf
IOPort
OneOf
Input|Nat
Output|Nat

FMaps

RA = Rotate Robot Arm (to,RA0)
CJ
Compute RA = Rotation Time (RA1,end,start)
RA1 = Moveto:RobotA:Ports(Ps1)
end,Ps1 = ResetTo_NewPort(Ps,to,IOPs)
start = StartingPosition(atP,IOPs)
IOPs = Moveto:IOPorts:Ports(Ps)
atP = Moveto:AtPort:Ports(Ps)
Ps = Moveto:Ports:RobotA(RA0)

Compute RA = Rotation (RA1,end,start) Time
CJ
distance = Diff:Nat(end,start)
rate = Moveto:TurnRate:RobotA(RA1)
time = Mul:Nat(distance,rate)
RA = Replace:RotationTime:RobotA(time,RA1)

start = StartingPosition(atP,IOPs)
J
rIOPs = Referto:IOPorts(atP,IOPs)
IOP = Moveto:IOPosrts(rIOPs)
start = Position(IOP)
co:isInput:IOPort(IOP)
start = Moveto:Output:IOPort(IOP)
start = Moveto:Input:IOPort(IOP)

end,Ps1 = ResetTo_NewPort(Ps,to,IOPs)
CC
IOPsN = Locate:IOPorts(end,IOPs)
IOPN = Moveto:IOPorts(IOPsN)
Ps1 = Replace:AtPort:Ports(IOPN,Ps)

end = GetPortID(to,Ps)
PIs = Moveto:PortIds:Ports(Ps)
PIsTM = TMap:PortIds(PIs)
end = Index_OneOf(to,PIsTM)

Robot Exploration System*

Robot senses its environment, ev (via ENV), Reacts based on event memories and uses efferent impulses to activate executive motor actions to effect (via ENV) its environment. If no event history is found, a new dIset memory relation is added. Executive selects a potential motor action response based on afferent impulse patterns in its 'map'.

FMap ***Syntax*: Function(Domain)=coDomain**

RunRobot(R0,E0)CJ=R,E

R,...Rn are Robots
E,...En are Environments
d,ev,effect are Stimuli

Clone2(E0)=E1,E2
Exploration(E1,E2,R0)CJ=R,E

SeeDanger(E1)=E3,d
DangerOrExplore(E3,d,R0,E2)=R,E
{Interrupt}

Danger(E3,d,R0,E2)CJ2=R,E

Avoid(R0,d)=Rn
Clone2~(E3,E2)=En
RejectOrNot(Rn,En)O=R,E
{is:Reject,Robot(Rn)}

True|**Exit**(Rn,En)I=R,E
Clone1(Rn)=R
Clone1(En)=E
not(True)|
RunRobot(Rn,En)=R,E

Explore
(R0,E2)CJ,CC=R1,E5
ENV
(E2)=E4,ev
React
(R0,ev)=R1,effect
ENV(E4,effect)=E5

Universal Mechanisms for FMaps and TMaps

Primitive Structures

?(r;s)J
1st?(r;z) 2nd?(z;s)

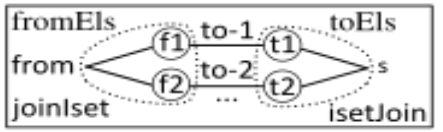
?(r1,r2;s1,s2)I
1st?(r1;s1) 2nd?(r2;s2)

?(r;s)O{P(r;s)}
value|1st?(r;s) 2nd?(r;s)
not(value)|

Type: Any	Constant: Reject
is:present(a;b)	If a has a value, b=True
Clone1(r;r1)	r=r1
Clone2(r;r1,r2)	r=r1=r2
Clone2~(r1,r2;r)	r=r1=r2
...	*~ means inverse*

dIset provides a reactive sensorymotor memory-map of Robot's history of interactions with its environment.
memory-map: reproduction of relations between events in the outside world.

Syntax ?(from;s)dIset
fromEl?(from;to) toEl?(to;s)



TMap ***Syntax*: Type(Domain;coDomain)**

Robot(sense;effect)J
motor(efforent;effect)
memory(sense;efforent)dIset
executive(to;efforent)
receptor(sense;to)O{Event(sense)}

danger|**Thunder**(sense;to)
not(danger)|TreeOrNot(sense;to)
O{Event(sense)}

tree|**Tree**(sense;to)
not(tree)|HouseOrNot(sense;to)
...

Do F until i arrives then do I.

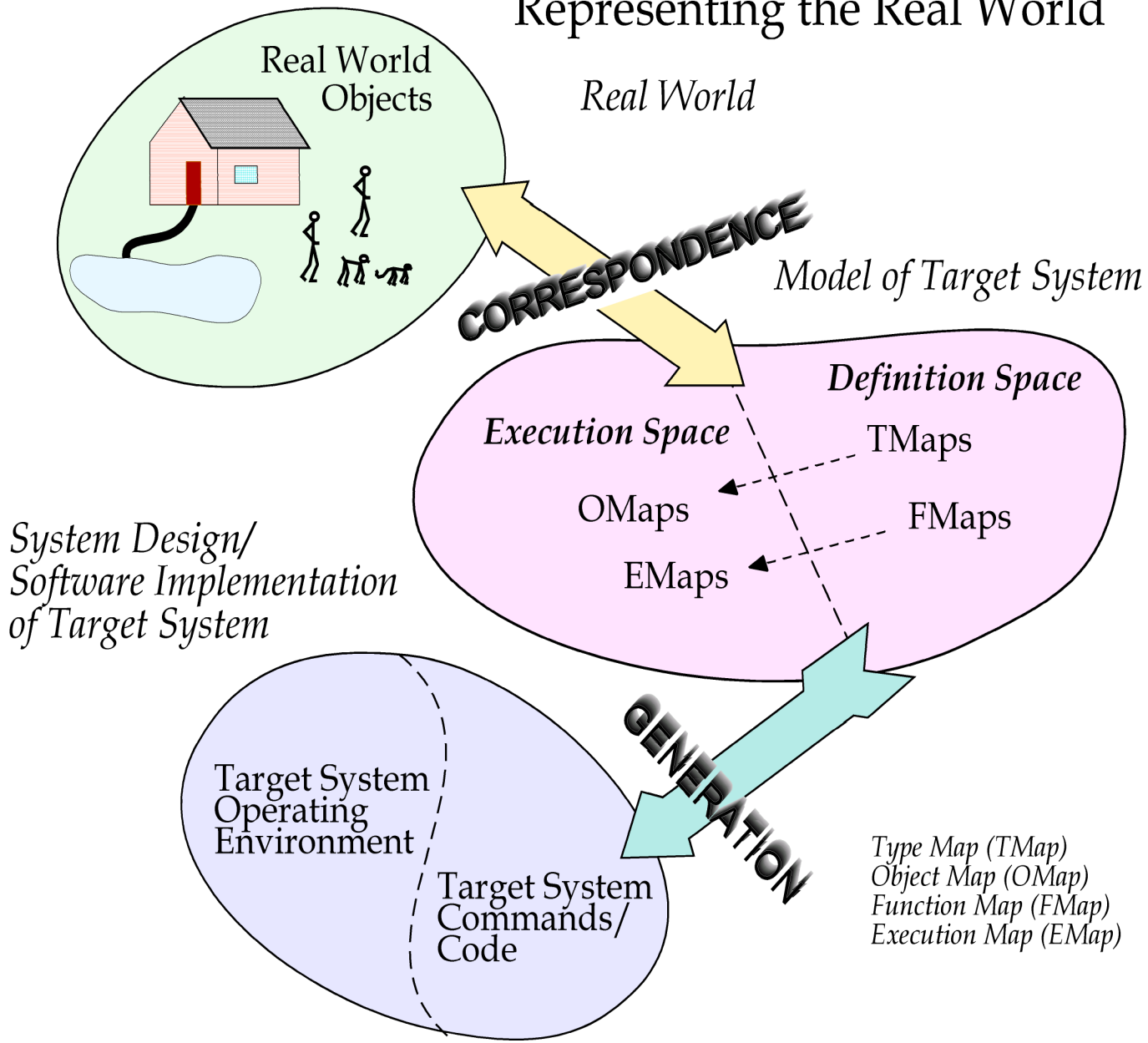
Syntax
?(i,s0)=s
{interrupt}
I?(i,s0)=s
F?(s0)=sn

Structure
interrupt?(i,s0)co=s
{is:present(i)}
True| I?(i,s0)=s
False| continue(i,s0)CJ=s
F?(s0)=sn
interrupt(i,sn)=s

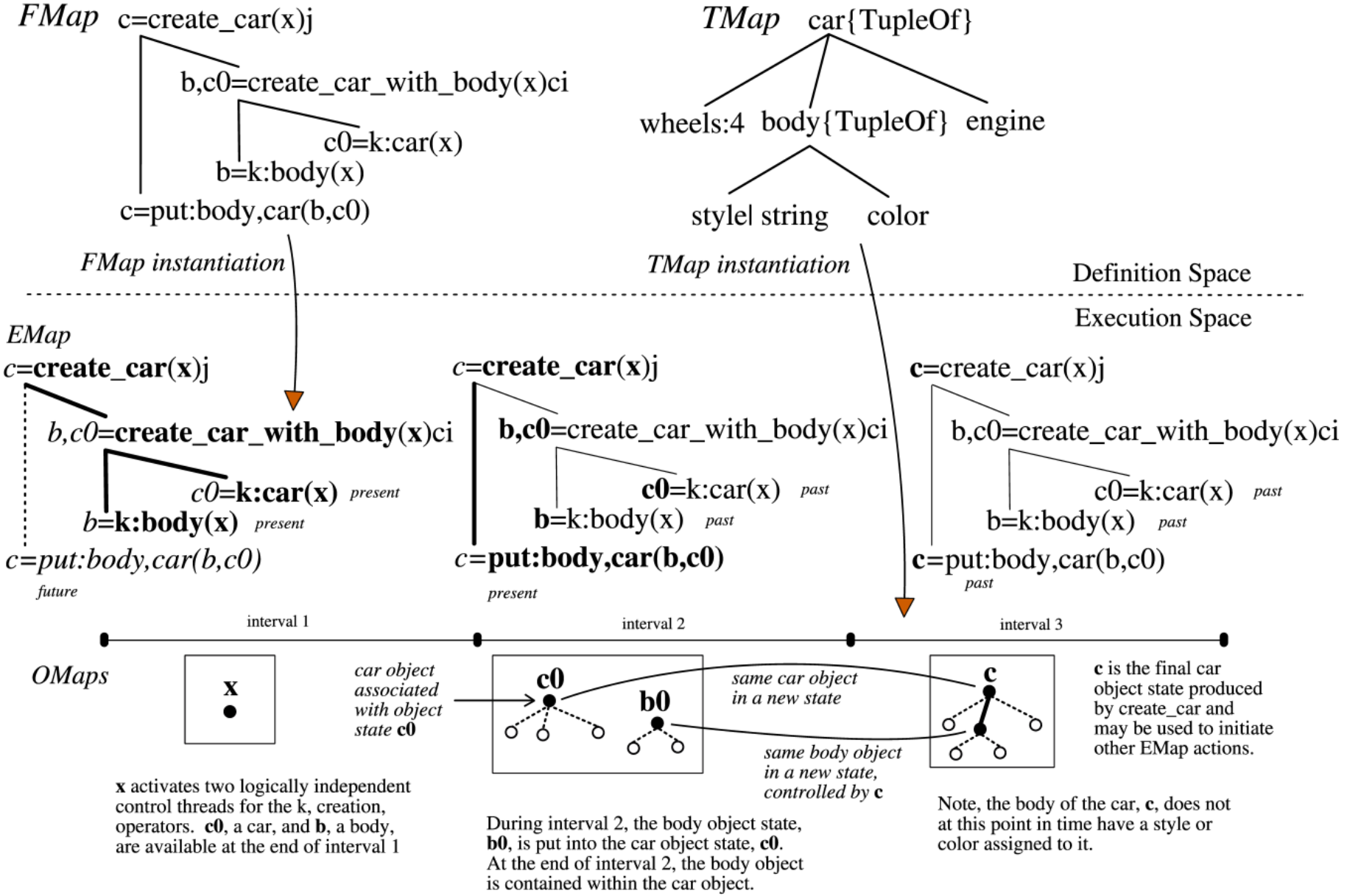
SeeDanger interrupts Robot's ongoing exploration with d. If Robot, Rn, can Avoid d, RunRobot is recursively restarted; or, Robot is terminated since it is a Reject.

* M. Hamilton, "What the Errors Tell Us", IEEE Software-Special Issue "50 Years of Software Engineering"

Representing the Real World



Definition and Execution Spaces



System Engineering Seamlessly Integrated with Software Development

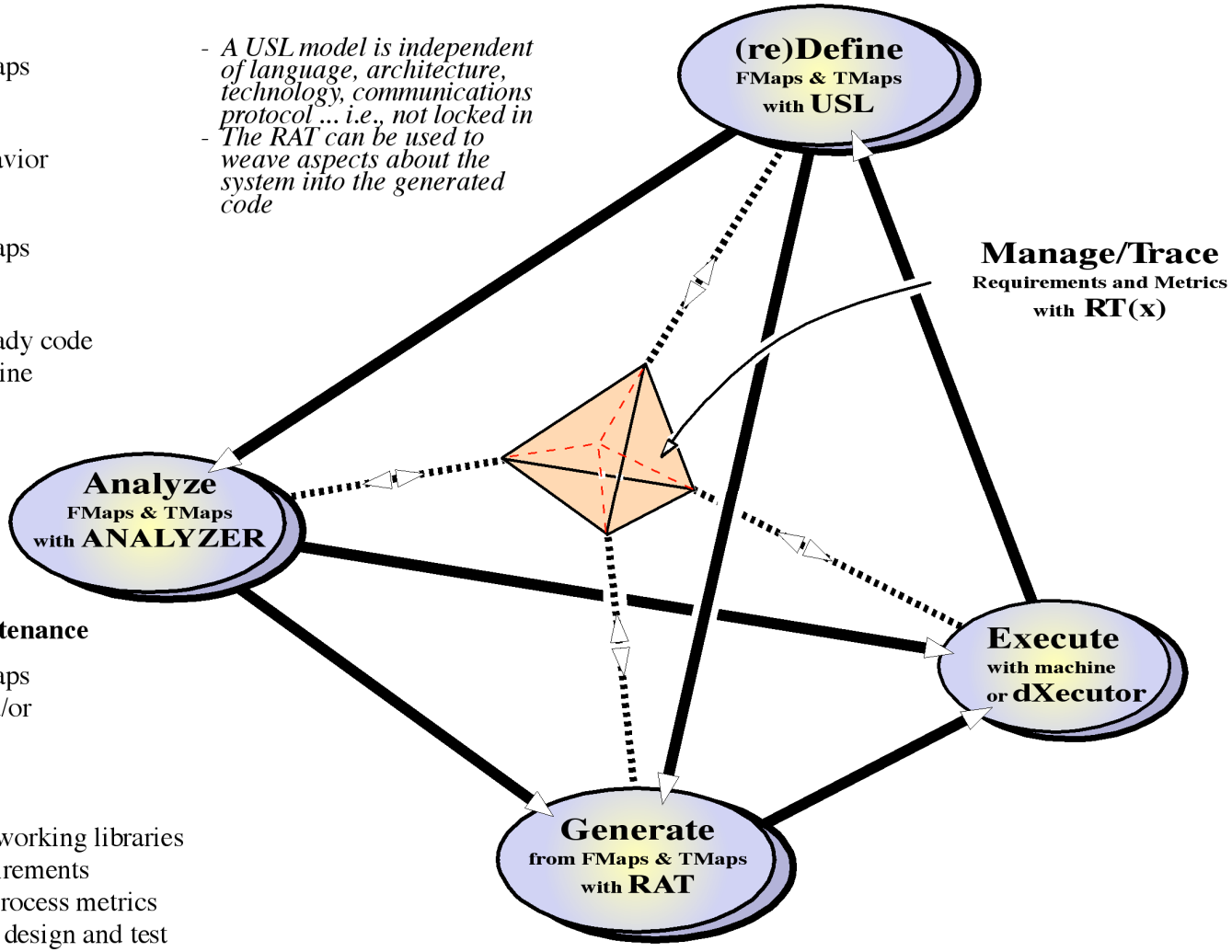
System Engineering

- Define FMaps and TMaps for system architecture
- Analyze
- Simulate real-time behavior

Software Development

- Define FMaps and TMaps for application
- Analyze
- Generate production ready code
- Execute on target machine

- *A USL model is independent of language, architecture, technology, communications protocol ... i.e., not locked in*
- *The RAT can be used to weave aspects about the system into the generated code*



Design Changes and Maintenance

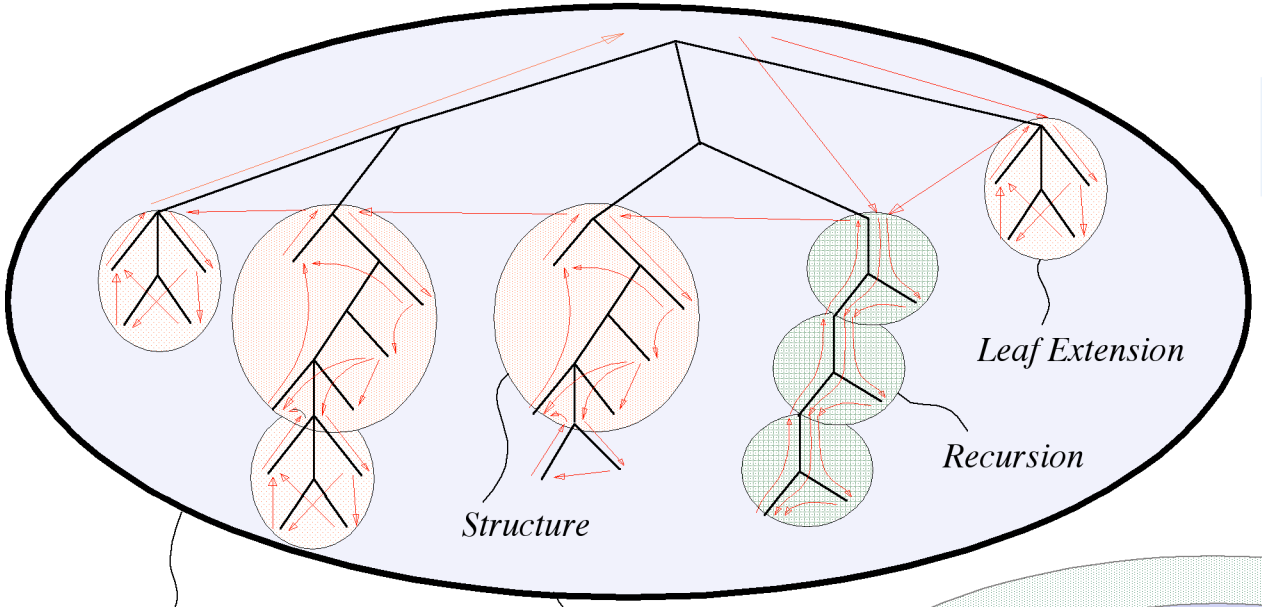
- Revise FMaps and TMaps
- Repeat engineering and/or development process

Management

- Organize projects into working libraries
- Manage and trace requirements
- Generate product and process metrics
- Generate specification, design and test documentation

A system is defined from the very beginning to inherently maximize the potential for its own *automation* and *evolution*

One of the Most Powerful Aspects: the Degree to which Reuse Inherently Takes Place

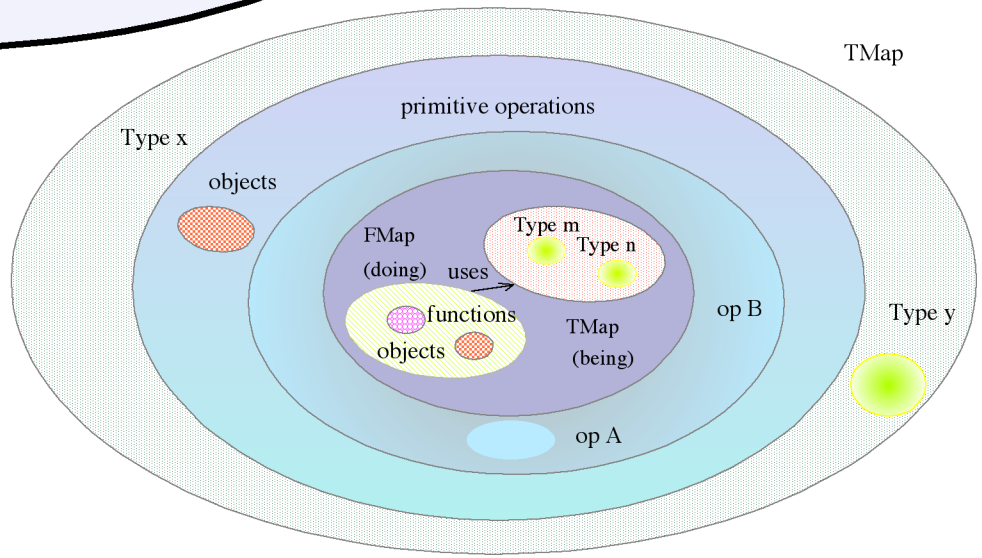


Alternative Layer Implementations

*Inherent and explicit patterns of reuse provided with FMaps and TMaps**

Recursive, Inherent, Reuse: Types are Defined in Terms of Functions; Functions are Defined in Terms of Types

A layer of primitive Types isolates the system being specified (the WHAT) from its possible implementations (the HOW)



* Reuse also provided with RMaps, OMaps, EMaps, RAT (reusable architecture configurations), OMap Editor, etc.

One Might Ask "How Can One Build a More Reliable System and at the Same Time Increase the Productivity in Building it?" Unlike the Traditional "Test to Death" Philosophy, Less Testing Needed with the Use of Each New DBTF Capability

- Correct use of USL eliminates majority of errors including all interface errors and their derivatives
- 001Analyzer hunts down the errors resulting from incorrect use of USL
- Inherent reuse and, if software, automation removes need for most other testing: e.g., built in aspects, inherent integration, all of code and much of the design automatically generated by 001 Resource Allocation Tool (RAT) with same integrity and consistent with the system definition
- 001 RAT generates 1) embedded test cases into the code for finding incorrect object use during execution; 2) test harnesses with OMap editor for testing each object and its relationships
- Maintenance shares same benefits as development
  - developer doesn't ever need to change the code
  - application changes made to the specification-not the code
  - architecture changes made to the configuration-not the code
  - only the changed part of the system is regenerated and integrated with the rest of the application (again, all automatically). The system is automatically analyzed, generated, compiled, linked and executed without manual intervention.

The need for most kinds of testing used in a traditional environment is removed. Most errors are prevented because of that which is inherent or automated (i.e., reused)

Since RT(x) automates the process of going from requirements to design to tests to use cases to other requirements and back again the need to ensure the implementation satisfies the design and the design satisfies the requirements is minimized

We Continue to Discover New Properties in USL Systems (and their Derivatives). Just by the very way a system is defined:

- No interface errors
- Seamless integration
- Traceability, flexibilty and evolvability
- Maximum reuse (derivable and inherent)
- Much inherent design

A system's definition provides input to and serves as the basis for USL's automation to inherit and pass on the definition's properties of control to the system's derivatives (e.g., the code)

Properties of Control Built into the Language "Come Along for the Ride"

- With the language to define it, software can be developed with "built-in" reliability and "built-in" productivity throughout its life cycle
- Much of what seems counter intuitive with traditional approaches becomes intuitive...less testing becomes necessary with each new before the fact capability
- A language without preconceived notions: the more reliable the system, the higher the productivity in its development
- Unlike what has been in large part a manual process or automation to support the manual process; automation does the real work
- The language's own automation, a large system (millions of lines of code) in its own right, is completely defined with and generates itself

This is possible because of USL's mathematical foundation

- Roots from Apollo and later systems
- Also takes roots from—other real world systems, formal methods, formal linguistics and object technologies
- Evolved over several decades
- Always stood its own when put to test (academic, government, commercial)
- Used in research and "trail blazer" organizations; positioned for more widespread use
- *A Radical Departure, Redefines what is Possible*
- New to the world at large, it would be natural to make assumptions about what is possible and impossible based on its superficial resemblance to other languages—like traditional object oriented languages
- It helps to suspend any and all preconceived notions when first introduced to this language because it is a world unto itself—a different way to think about systems

Along the way it becomes clear that many aspects of the pressing issues can be addressed with a preventative approach; ultimately eliminated altogether

- ✓ *Integration too late if at all*
- ✓ *Lack of traceability, flexibility and evolvability*
- ✓ *Reuse methods ad hoc and error prone*
- ✓ *Software unreliable even with extensive testing*
- ✓ *Costs too much. Takes too long*

Several software engineers' tasks become no longer needed

What we do about what we have learned, however, depends on how ready and how open developers (managers, technical people) are to a change in how we build software

- Research and "early user" applications in diverse environments including comparisons, studies, experiments, contests, "shootouts" e.g,
  - Academic Research: MIT, Carnegie Mellon, Stevens Institute of Technology
  - Commercial: CitiBank, Scott Paper
  - Aerospace: Honeywell, Lockheed
  - Software Productivity Consortium
  - DoD: Star Wars National Test Bed, Navy, Air Force, DOE, Army, NSA
- Unique criteria for each study: comparisons conducted by US government agencies, industry and academic organizations; refereed by third party observers or by the agency sponsoring the competition
- From initial establishment of system functional requirements through operational validated code
- USL did not disappoint when put to the test
- The larger and the more complex the system, the better the results

The errors showed us how we can do without them. They led us to a language where each of its definitions:

- *Inherently replaces* what used to be aspects of the system's own life cycle, or
- *Serves as the input* for the language's automation of what used to be manual processes in the system's own life cycle, or
- Results in *many parts of the system's own life cycle no longer needed*

A language that by the very way a system is defined can serve as a software engineer in its own right

A language with a preventative paradigm that leads "before the fact" to the future

The Language as a Software Engineer

Margaret H. Hamilton
Hamilton Technologies, Inc.

May 31, 2018

Images on Slide 1 and this Slide are from The Apollo Prophecies

mhh@htius.com *www.htius.com*