Universal Systems Language (USL) and its Automation, the 001 Tool Suite, for Designing and Building Systems and Software

Margaret H. Hamilton

September 27, 2012

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

Hamilton Technologies, Inc. (HTI)

Founded: 1986

Charter: provide the means to modernize system engineering and software development; maximize reliability and flexibility, minimize cost and risk and accelerate time to market

Vertical markets: real time, internet based, distributed and data base environments. Applications include battlefield management, communications, homeland security, aerospace, emergency management, manufacturing, banking, medical, energy, traffic, robotics and enterprise management systems; simulation and software tools

Development Platform: Unix, Linux

Deployment Platform: Unix, Linux...

Customer: system integrator, tool vendor, end user

What if there Was a Way to Design Systems and Build Software that Would Ensure:

- Seamless integration, including systems to software
- No interface errors in a system design and its derivatives
- Complete traceability and evolvability
- Maximum inherent reuse
- Automation of much of design
- Automatic generation of 100%, fully production ready code for any kind or size of software application
- Elimination of the need for a high percentage of testing without compromising reliability

Result:
Significantly increased reliability
Significantly lower risk
Significantly higher flexibility
Significantly higher productivity
Significantly lower cost

There is, but it Takes a Special Kind of Language

- It is possible today with the universal systems language, USL together with its automation, because of the technology that forms its foundations.
- Based on a theory; in large part derived and evolved from lessons learned from Apollo's on-board flight software effort*
- Also takes roots from—other real world systems, formal methods, formal linguistics and object technologies
- USL has evolved over several decades, offering solutions to problems previously considered next to impossible to solve with traditional approaches
- Always first when put to test (academic, government, commercial)
- Used in research and "trail blazer" organizations; now being positioned for more widespread use

A Radical Departure, Redefines what is Possible

- New to the marketplace 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 completely different way to think about systems

* M. Hamilton and W. R. Hackler, Universal Systems Language: Lessons Learned from Apollo, IEEE Computer, December 2 2008

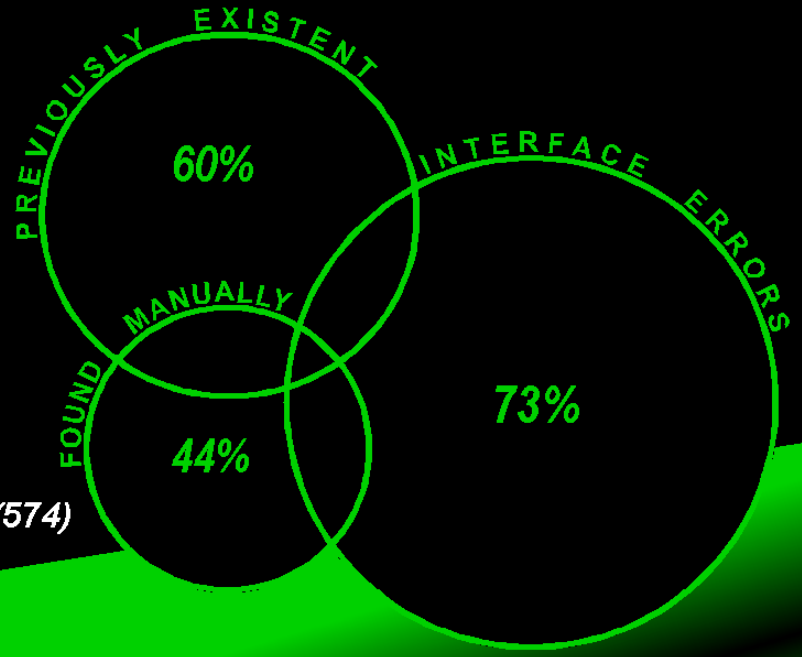
USL Began with

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

Note 1: no software errors known to occur during flight
Note 2: majority of 44% found by "Nortonizing"
Note 3: To this day we continue to discover new ways to prevent problems from happening; again, just by the way a system is defined and we continue to incorporate these findings into the evolving technology. Once solutions are made to solve problems, repeat the process over and over again…never assume anything or anyone is perfect.

Analysis Took on Multiple Dimensions, not Just for Space Missions but Systems in General. Lessons Learned from this Effort (and their Impact) Continue Today, e.g.,

- 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
- 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 is inherently a system of systems

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

Solution: Development Before the Fact (DBTF), Theory Captured by USL

Paradigm: each system defined with properties that "come along for the ride" and support its own development

- 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, USL is based on a preventive 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 USL a System is 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*

The Language is the Key: Every USL System Defined with DBTF Properties of Control

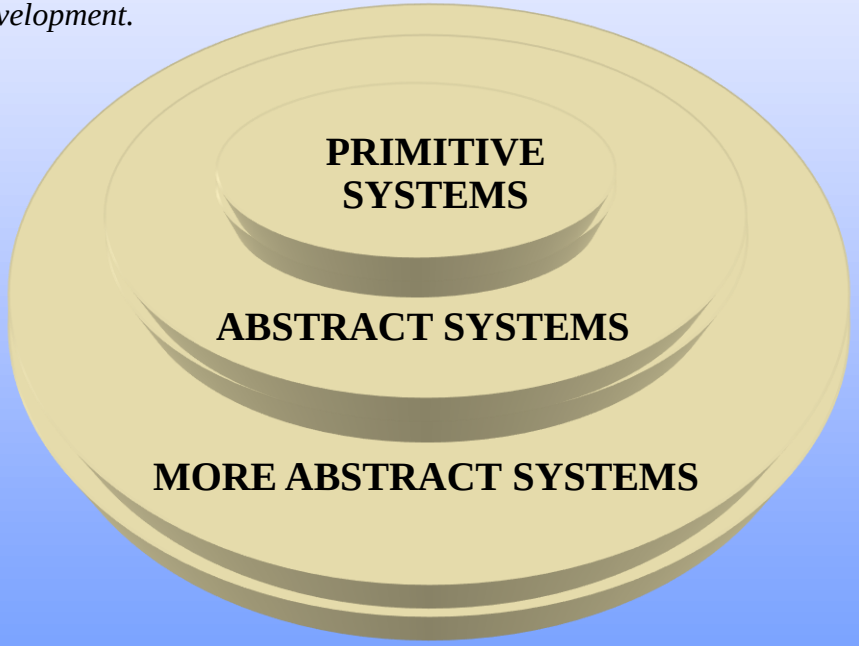
- A formalism for representing the mathematics of systems, USL is based on a set of axioms and formal rules for their application
- Same language used to define and integrate
  — All aspects of and about a system and its relationships and its evolutions
  — Functional, resource and allocation architectures, 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 USL System

- ***(Re)Define*** model with USL
- ***Analyze*** automatically the model to ensure it was defined properly
- ***Generate*** automatically much of the design and 100% of the code, production ready, for any kind or size of system
- ***Execute*** the model
- ***Deliver*** the real system

USL Philosophy: Reliable Systems are Defined in Terms of Reliable Systems

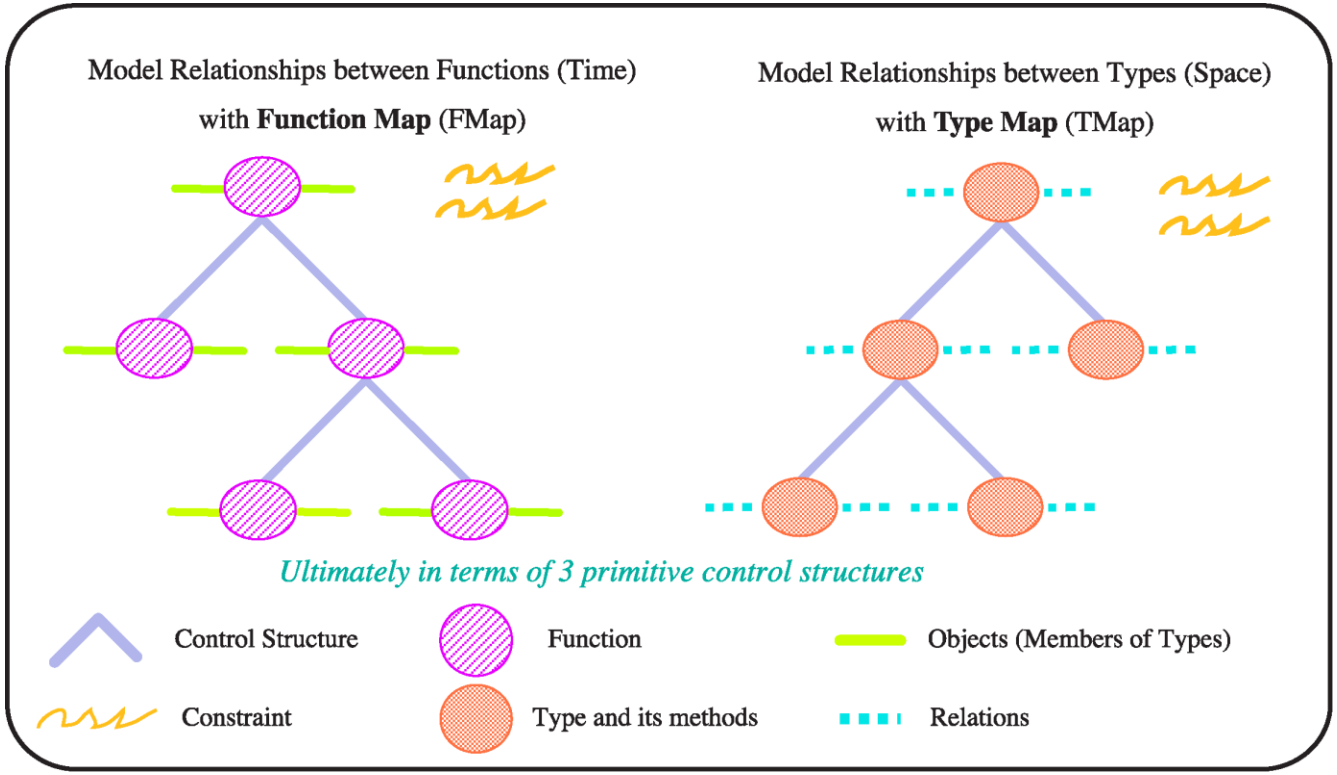
A large library of reusables has evolved over years of development.



- 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

A recursively reliable and reusable process

Every System Defined with Function Maps (FMaps) and Type Maps (TMaps), the Major Building Blocks of USL



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.

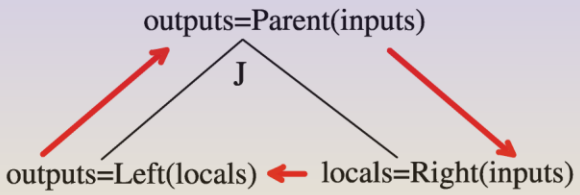
Primitive types reside at the bottom nodes of a TMap. Each type is defined by its own set of axioms. Inputs and outputs of each function are members of types in the TMap. Primitive functions in an FMap, each defined by a primitive operation of a type on the TMap, reside at the bottom nodes of an FMap. 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

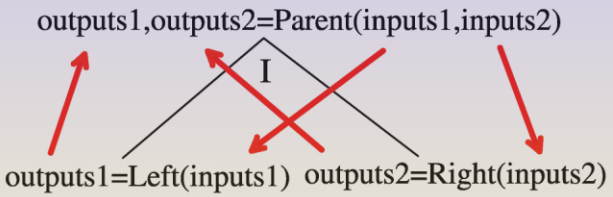
Dependent relationships



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).

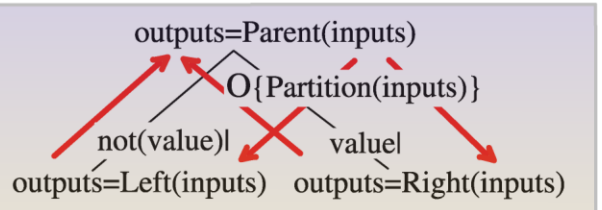
Independent relationships



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.

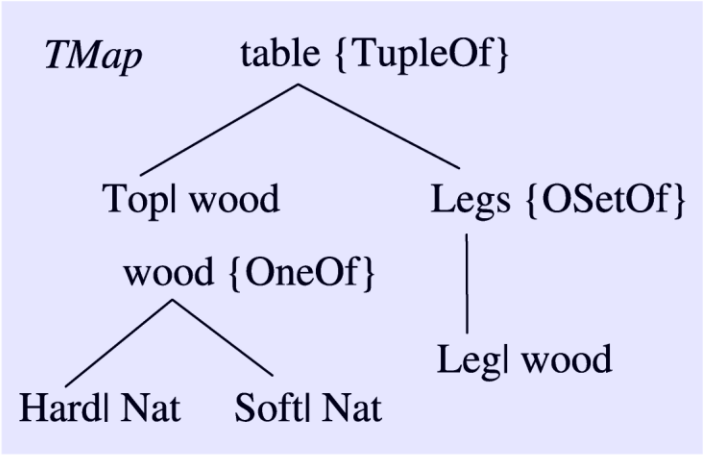
A USL system model defined in terms of the three primitive control structures will have all its (and its derivatives') interface errors (~75% to 90% of all errors) eliminated at the definition phase. These are typically found (if they are found) during testing in traditional development.

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.



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

FMap table=MakeATable(flat,round)Join

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

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

the parts, top and legs, are made independently

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

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

legs=make_legs(round)

different finish is applied depending on the type of wood

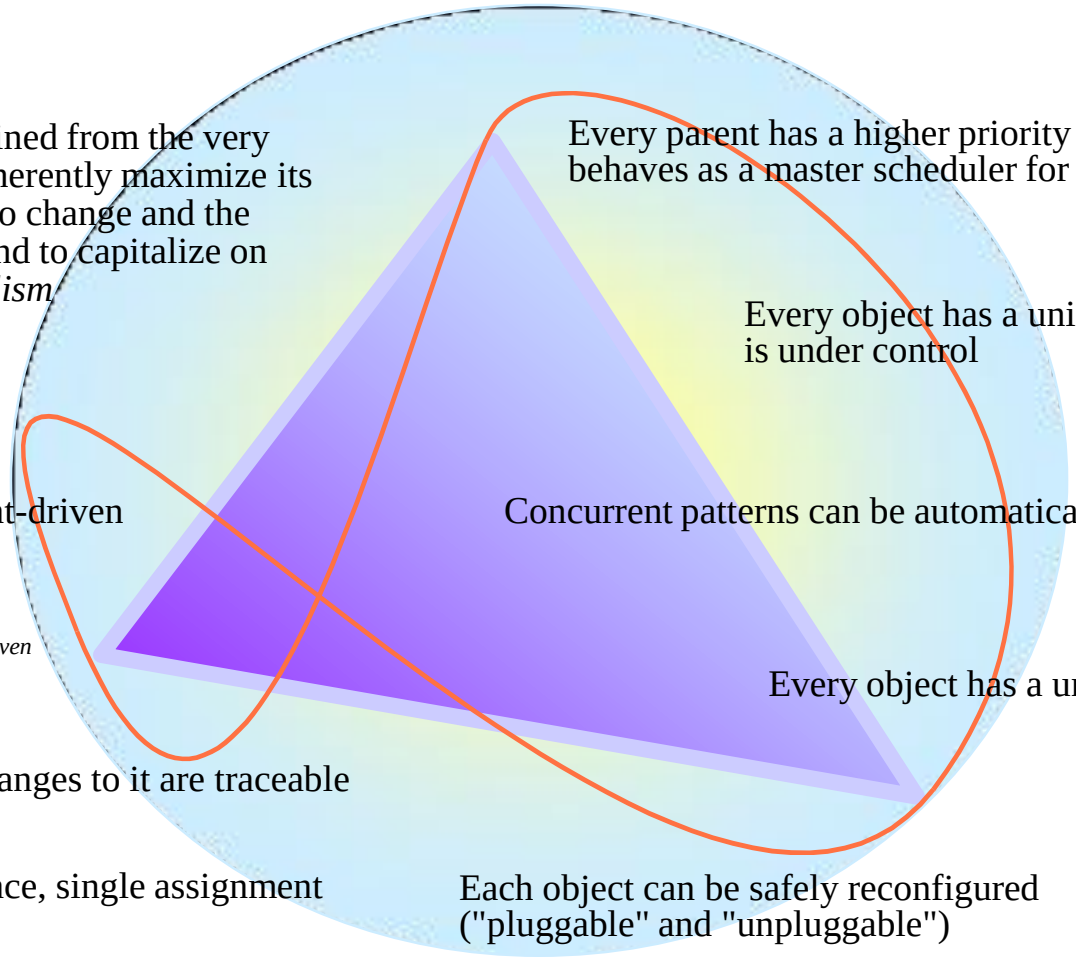
top=FinishSoftWood(flat)

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

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

Concurrent patterns can be automatically detected

Every object has a unique priority

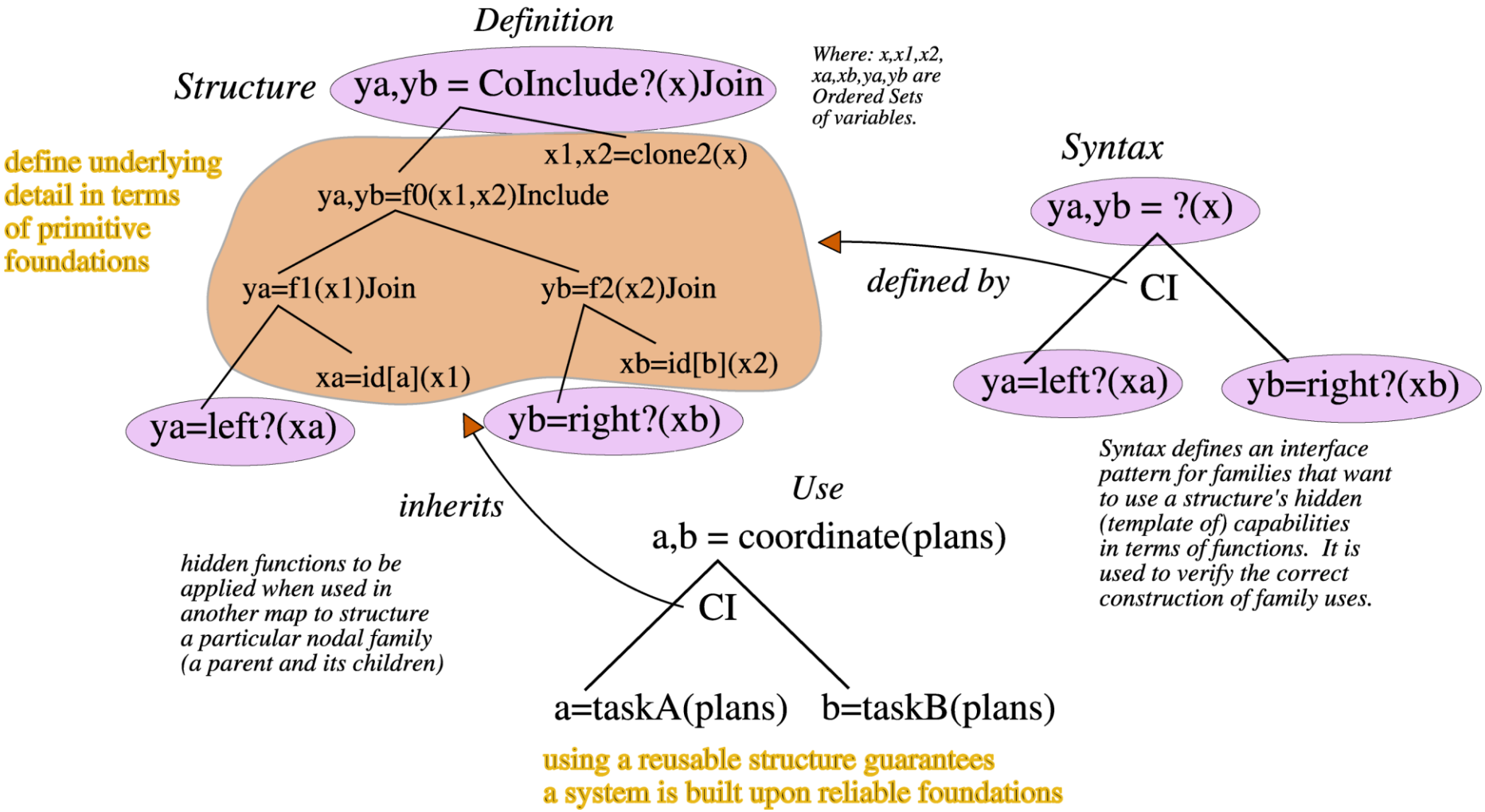
Each object and changes to it are traceable

Single reference, single assignment

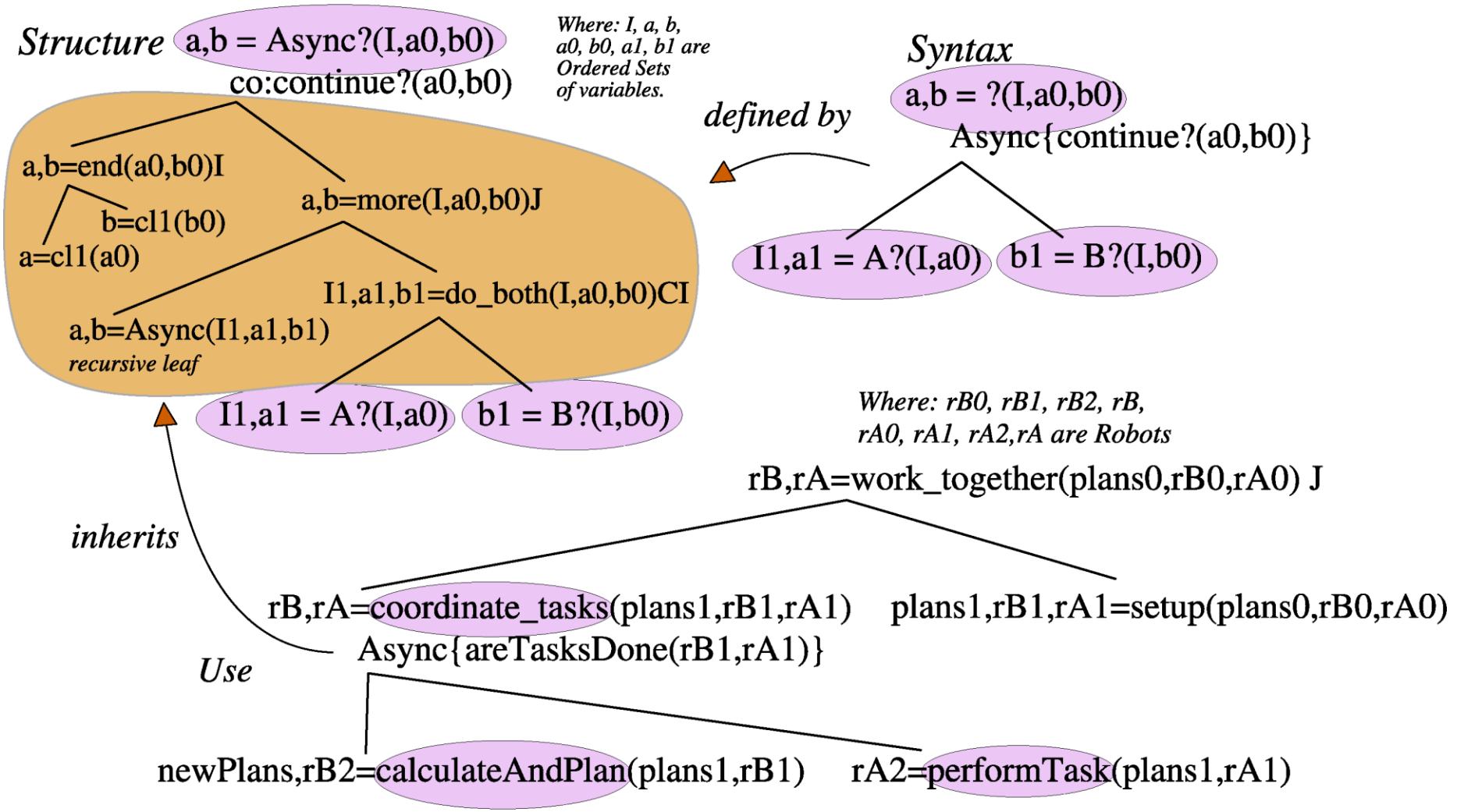
Each object can be safely reconfigured ("pluggable" and "unpluggable")

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

A Derived FMap Structure

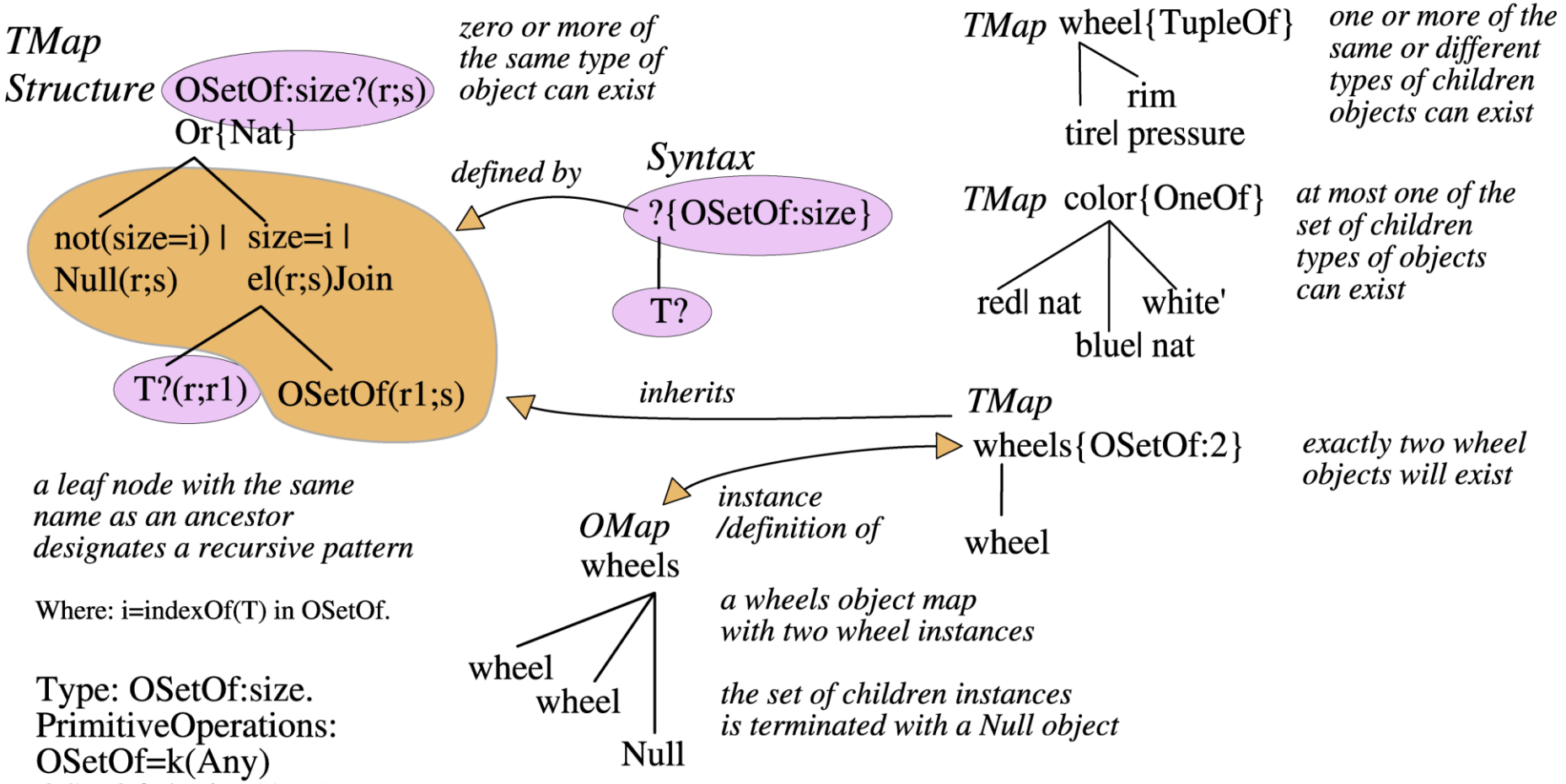


An Async Structure (that can be Distributed), with both Synchronous and Asynchronous Behavior, and its Use



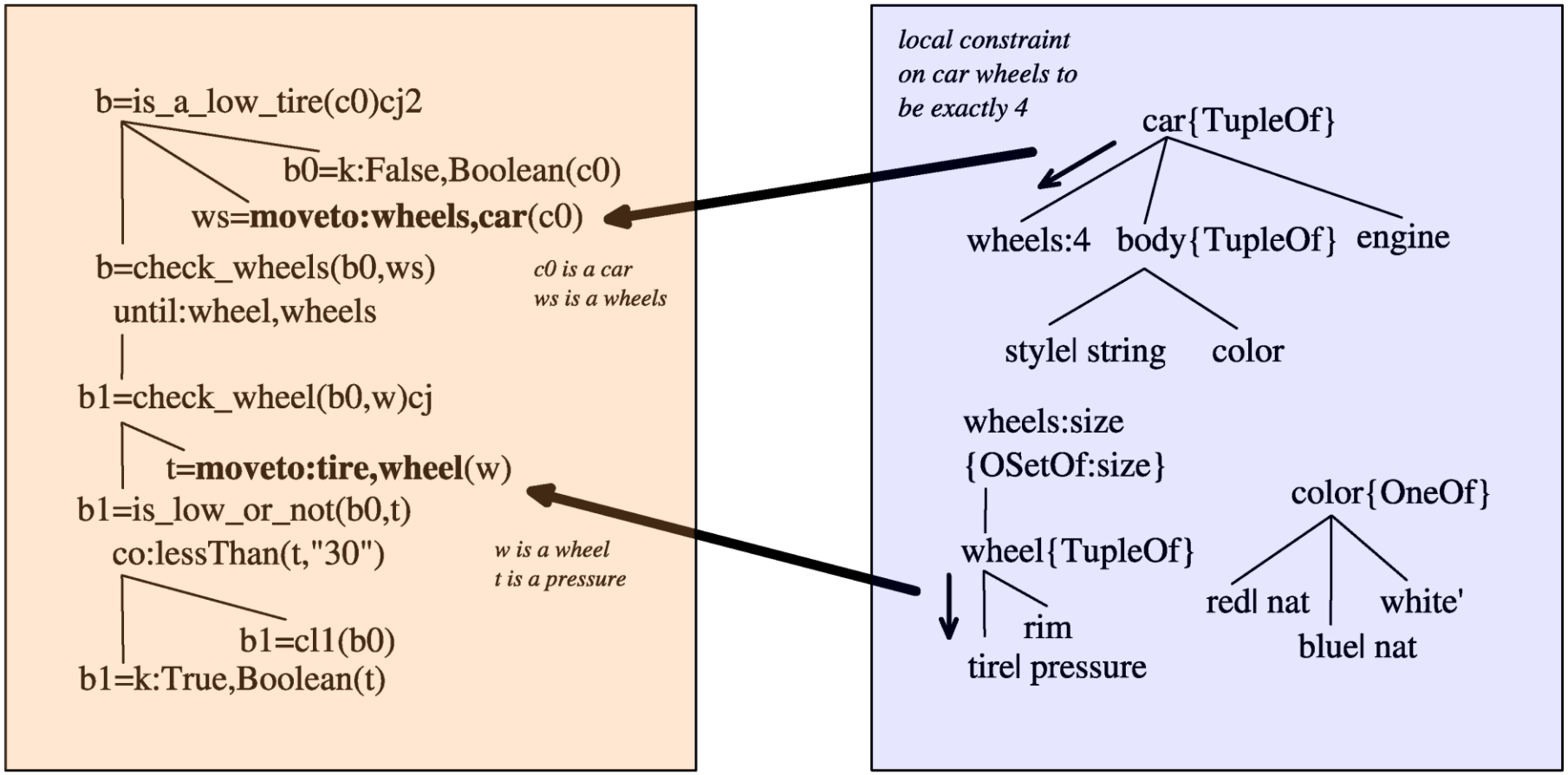
To understand Async's use at coordinate_tasks, go to Async's definition and to understand do_both go to CI's definition. Ultimately, coordinate_tasks is defined in terms of the Join, Include and Or primitive control structures.

Some Derived TMap Structures



A System: Integration of FMaps and TMaps

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



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.

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

each TMap node has a type and set of primitive operations

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

Operational Scenario

A view of the interaction of a user with a banking system via an ATM machine

TMap bank_system

user atm bank
name
card
password
money

FMap

customer requests money from atm

atm,cust=customer_atm_transaction(atm0,cust0)J*2

enter password to get authorization

atm1,cust1=authorize:ATM(atm0,cust0)

enter request for money

atm2,cust2=request:ATM(atm1,cust1)

on fully asynchronous distributed network communication/transaction infrastructure with dXecutor

atm,cust=result(atm2,cust2)cc

central,response=authorization(central0,request)

atm,money=response:ATM(atm2)

ATM outputs an error message or money

cust=cash_came_out_or_not(cust2,money)
co:is:Reject,money(money)

cust=clone(cust2)
customer leaves without money

cust=addto:cash(cust2,money)

customer adds the ATM money to any on hand

USL's distributed systems architecture is supported by the 001 distributed executor (dXecutor) runtime environment.

dXecutor running A on BANK

y,c=A(a)
b=B(a)
y,c=C(b)

for distributed communications, each variable is split into its control and data transfer aspects

asynchronous control negotiation

y,c=C(b)
c=G(b)

b=B(a)
b=F(x)

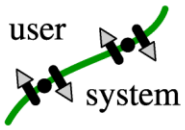
MONEY

direct data transport from F of B to G of C

dXecutor running C on USER

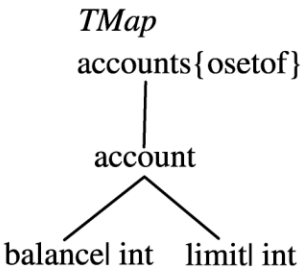
dXecutor running B on ATM

An operational model is aset of user/system interfaces



Constraints in Terms of TMaps and FMaps in One System Define Properties about Another USL System

Global constraints are defined as axioms for types in a TMap



this axiom (constraint) applies globally to all account objects in all FMaps

Account Axioms:
"True"=hasCredit(account)

the axiom statement states that an account must have a credit limit greater than its balance

FMap b=hasCredit(a:account)cj*2
limit=moveto:limit,account(a)
balance=moveto:balance,account(a)
b=greaterThan(limit,balance)

the hasCredit FMap defines the details of the constraint

Local constraints are defined as constraints for variables in an FMap

this constraint applies locally only to object, orderedAccounts, in FMap, order_accounts

Application

FMap

orderedAccounts=order_accounts(unOrderedAccounts)
sort_accounts
partiallyOrdered=order_two_accounts(i1,i2)

the accounts in orderedAccounts are ordered by credit with the accounts having less credit coming before accounts having more credit

Constraint:
"True"=orderedBy_credit(orderedAccounts)

FMap b=orderedBy_credit(accounts)
checkTwoByTwo

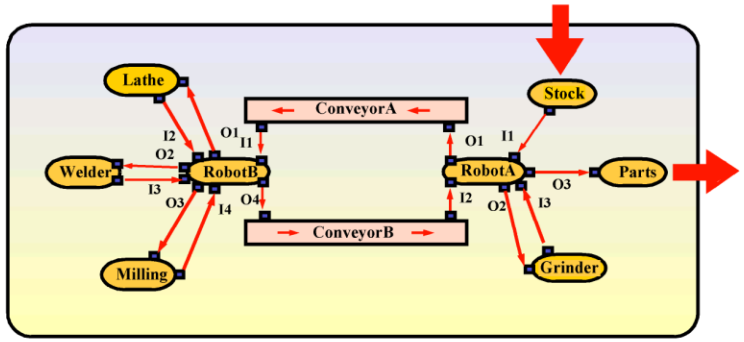
the orderedBy_credit FMap defines the details of the local constraint

ok=isOrdered(acnt1,acnt2)cj*2
credit1=credit_for(acnt1)
credit2=credit_for(acnt2)
ok=lessThan(credit1,credit2)

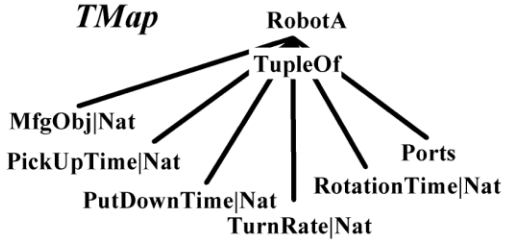
the values of both limit & balance abide by the hasCredit axiom

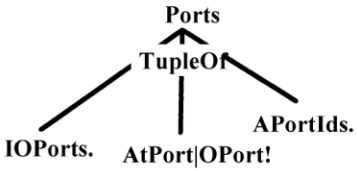
FMap credit=credit_for(acnt)cj*2
limit=moveto:limit,account(acnt)
balance=moveto:balance,account(acnt)
credit=subract(limit,balance)

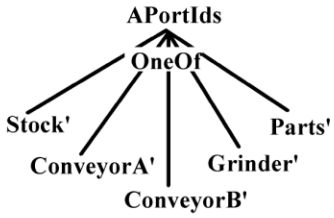
Operation: Rotate Robot Arm.

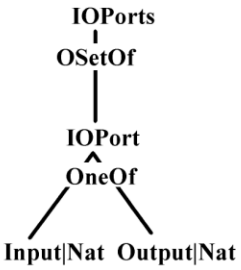


TMap

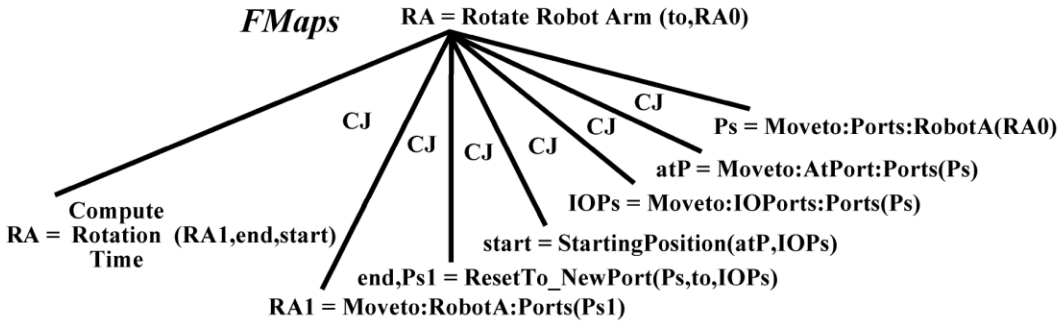


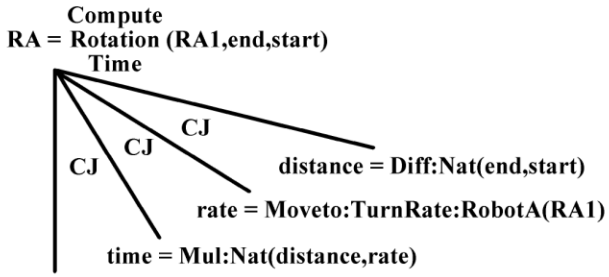


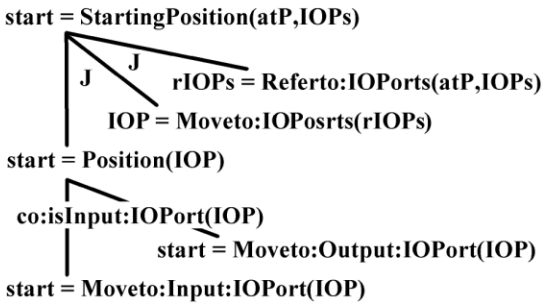


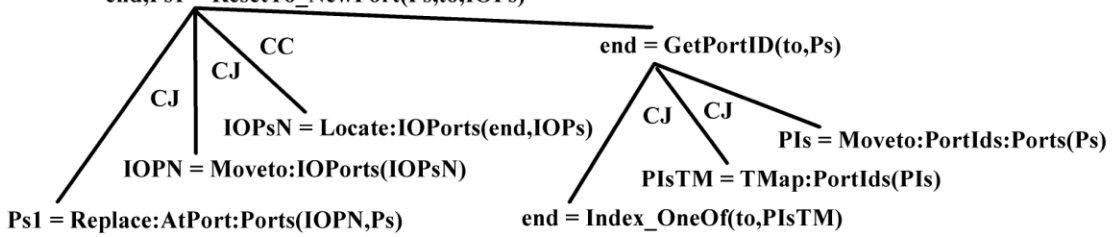


FMaps



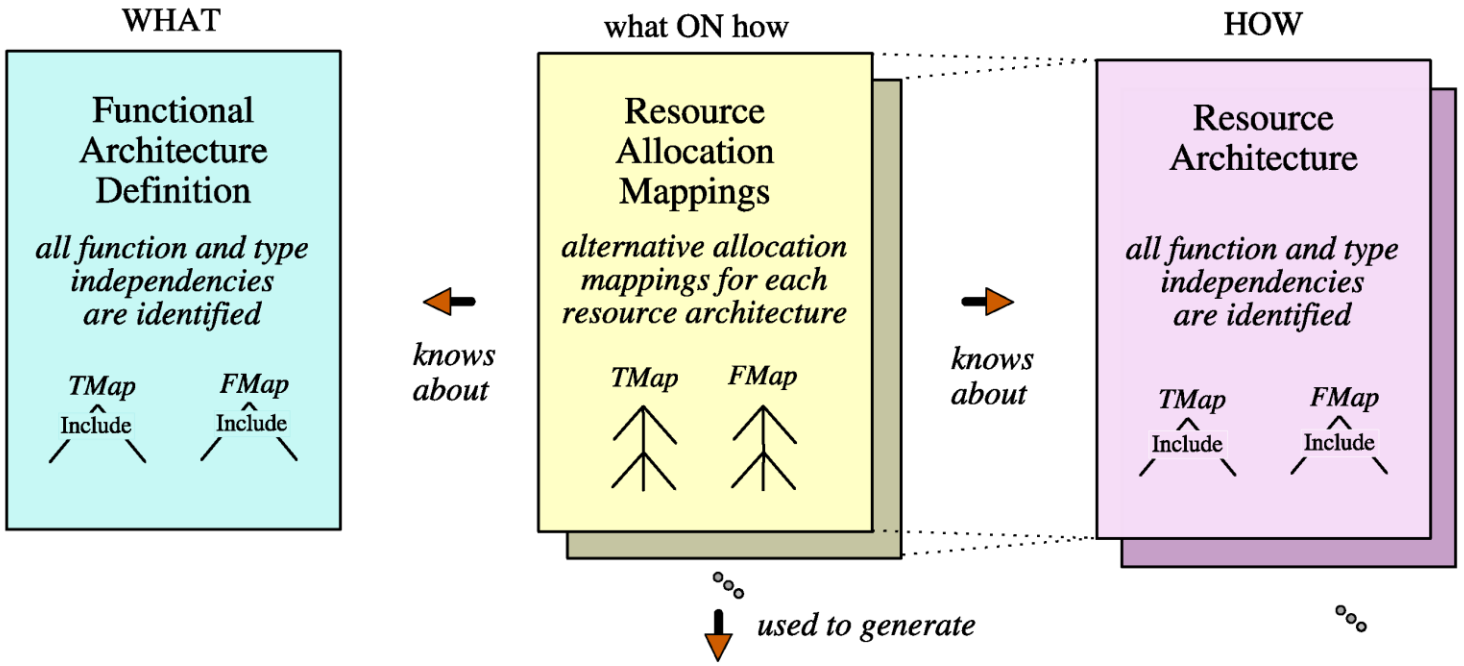






FMa

System Architecture: Integration of Functional, Allocation and Resource Architectures



USL layering separates the functionality of a system from its implementation, allowing one to determine how best to allocate available resources in a resulting system architecture.

This approach leads to automated scheduling of:
- resources
- best cases of parallelism
- asynchronous communications between distributed resources

sense state of Real World (e.g., market conditions) → respond to state of Real World (e.g., market trends)

Defining a USL System Architecture Using the Approach of Layering Architectural Independence while Integrating Functionality and Resources

ship two boxes

Functional Architecture (FA)

boxB1,boxA1=ship_boxes(boxB,boxA)I

boxB1=B| ship(boxB) boxA1=A| ship(boxA)

ship them by plane and car

Allocation Architecture (AA)

car1,plane1=ship_boxes(car,plane)I

car1=B| ship(car) [allocateTo: sendBy:car]

plane1=A| ship(plane) [allocateTo: sendBy:plane]

shipping a box is done by
- loading it
- transporting it to its destination
- then, unloading it

Resource Architecture (RA)

veh,box=sendBy:vehicle(veh0,box0)J

veh1=load:vehicle(veh0,box0)

veh,box=deliver(veh1)J

veh2=transportBy:vehicle(veh1)

veh,box=unload:vehicle(veh2)

concept: defining parallelism in real world systems
example: shipping two boxes: one by car and the other by plane

WHAT is to be DONE

MAPPING of WHAT is to be DONE onto HOW it is to be DONE

HOW it is to be DONE

System Architecture (SA)

car1,boxB1,plane1,boxA1=ship_boxes(car,boxB,plane,boxA)I

plane1,boxA1=A| sendBy:plane(plane,boxA)

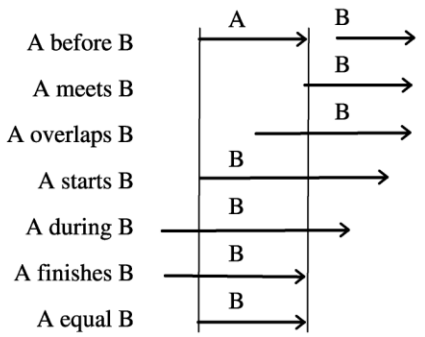
car1,boxB1=B| sendBy:car(car,boxB)

this integration results in parallelism (car and plane)

WHAT is to be DONE is INTEGRATED with HOW it is to be DONE

scheduling is completely inherently defined

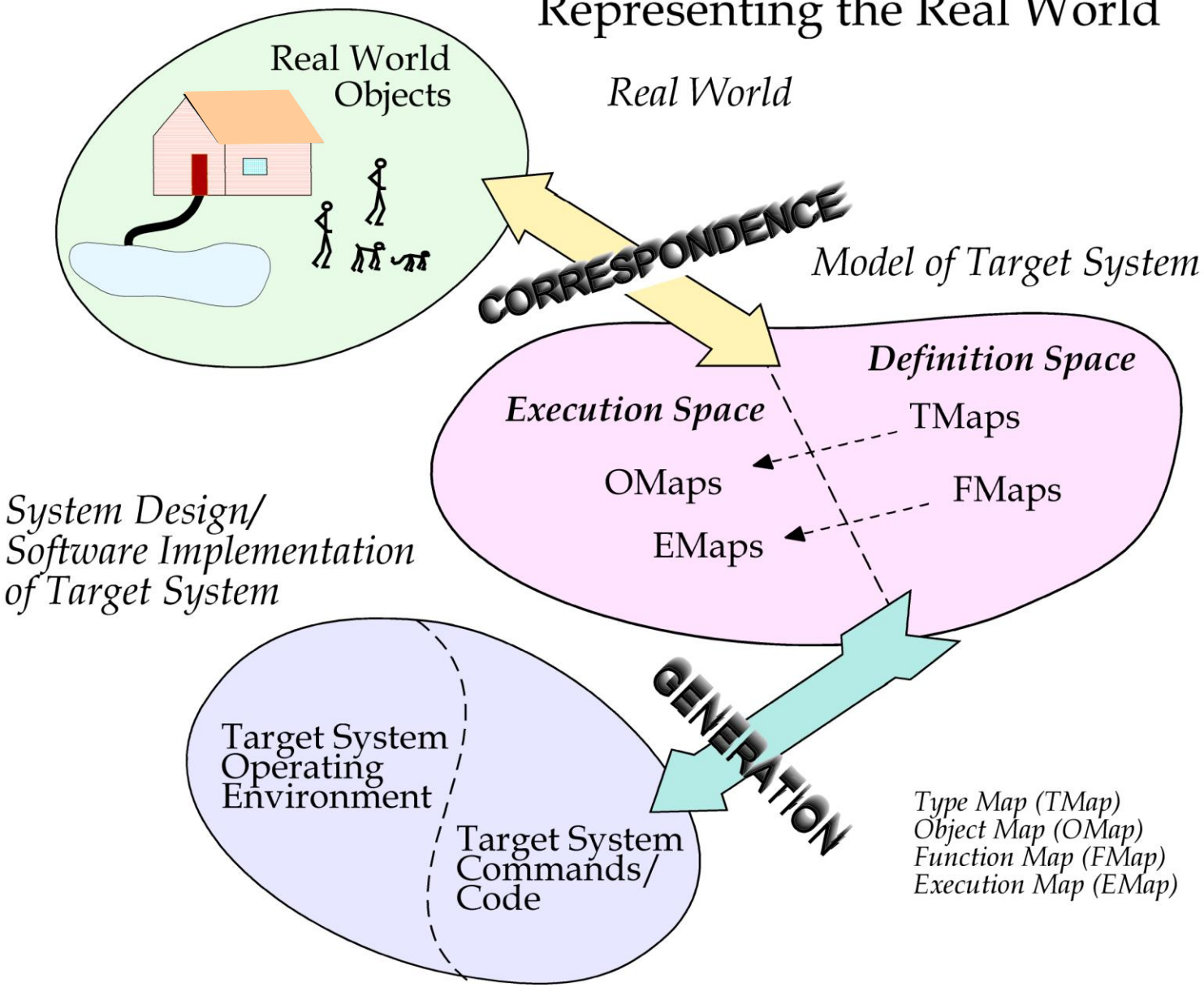
SA Schedule Implications



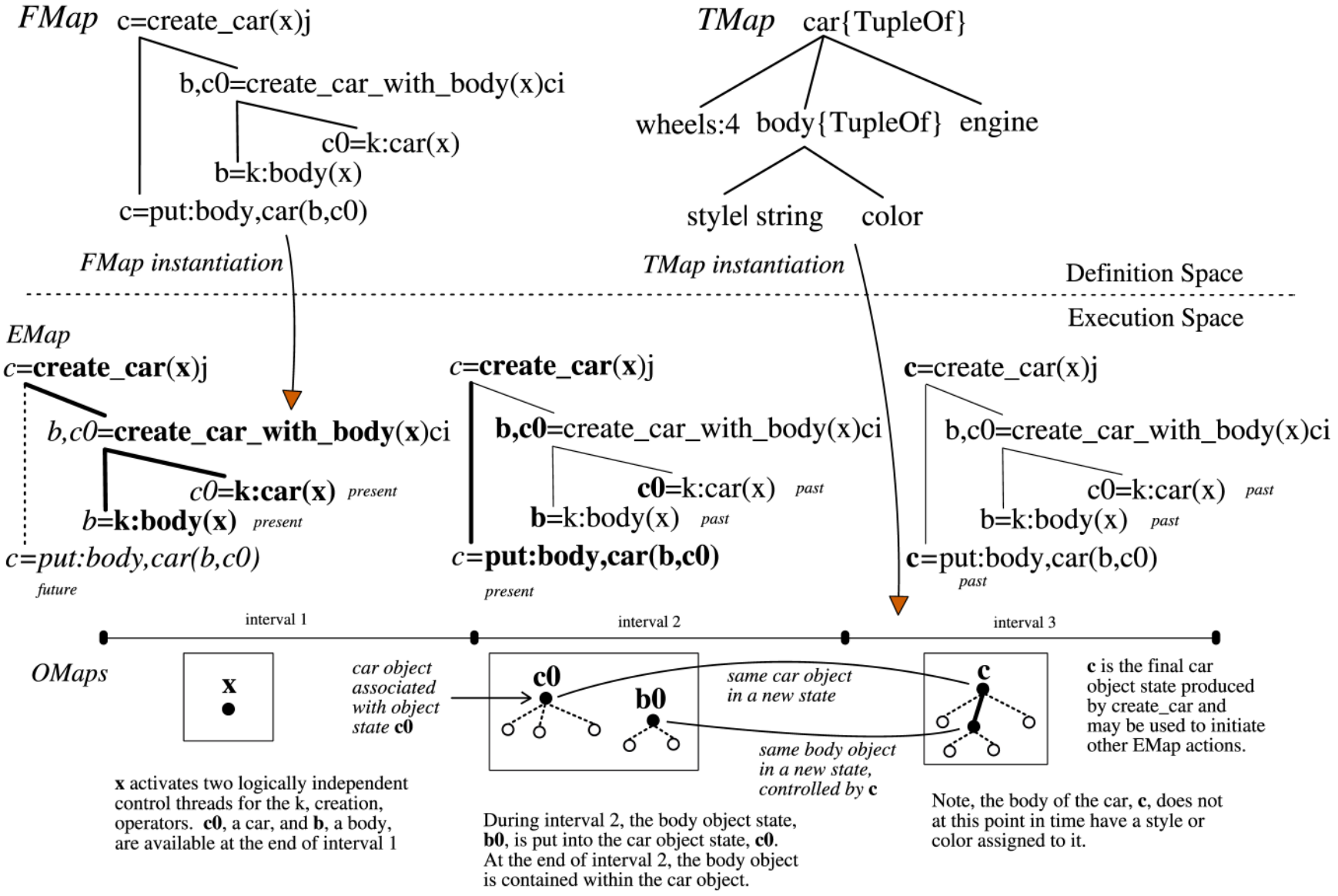
Include implies 13 possible time intervals pairs.

Only six additional timing relations exist when A and B are reversed in the time relationship predicates,since the inverse of equal is itself.

Representing the Real World



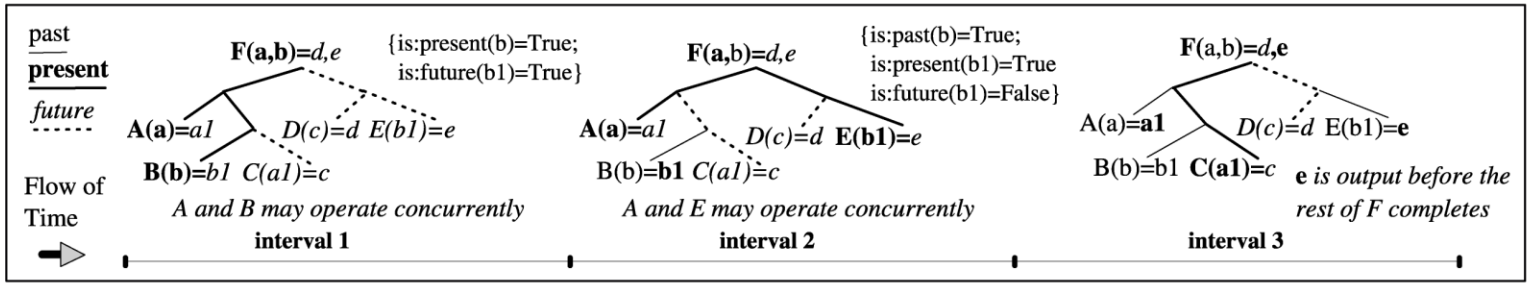
Definition and Execution Spaces



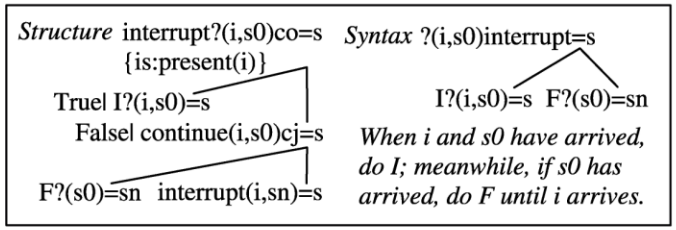
A Real-Time Event Driven Multi-threaded Control System

An EMap, an instance of an FMap, shows past, present and future events. Actions are driven by object states and selected for execution based upon their priority in terms of control.

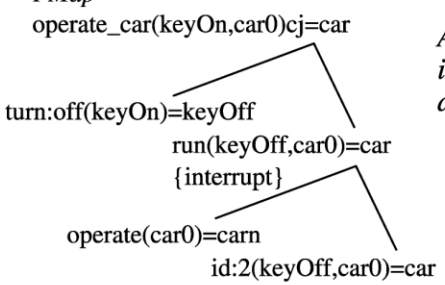
concept: map invocation process



Interrupt Structure



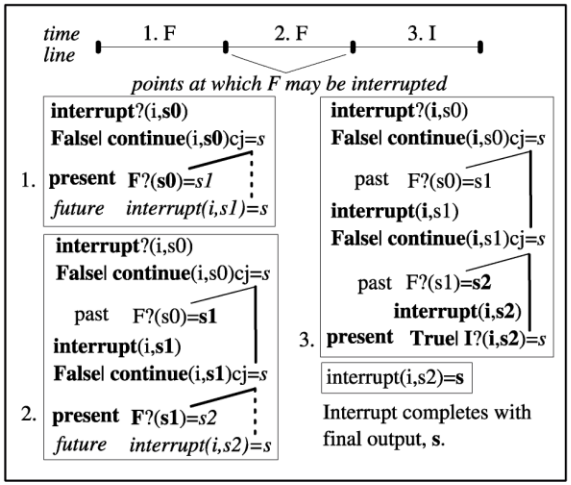
Interrupt Application



An example application of the interrupt structure used to run a car until the key is turned off.

EMap snapshots: an FMap instantiation of the interrupt structure

Interrupt Execution



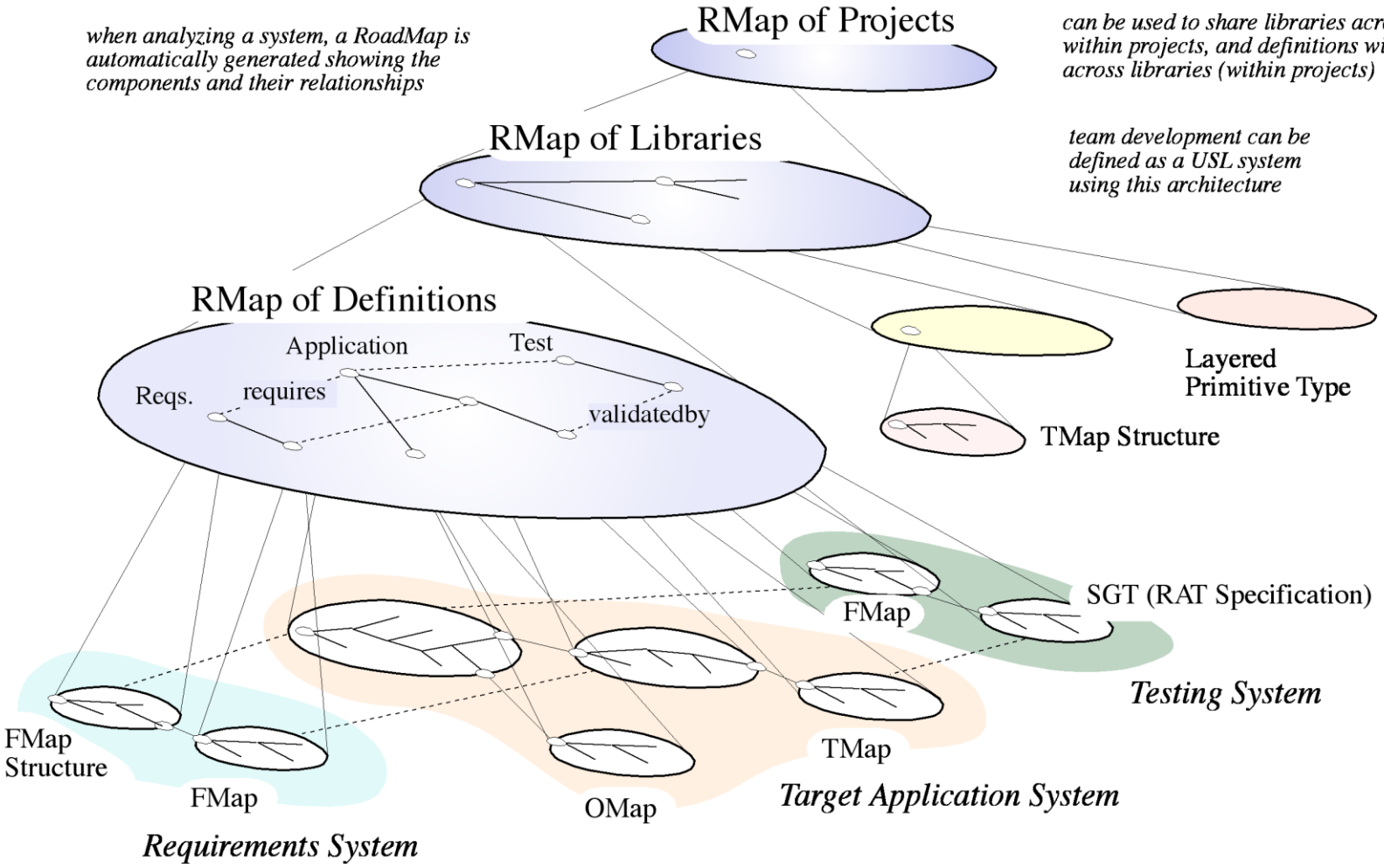
The is:present function returns True when its input has a value (i.e., it exists).

When the object state, s0, has a value and the object state, i, does not have a value; then, F? happens. When both i and s0 have values; then, I? happens.

NOTE:
Map representations on this page are read left-to-right with the following form:

function(input)structure=output

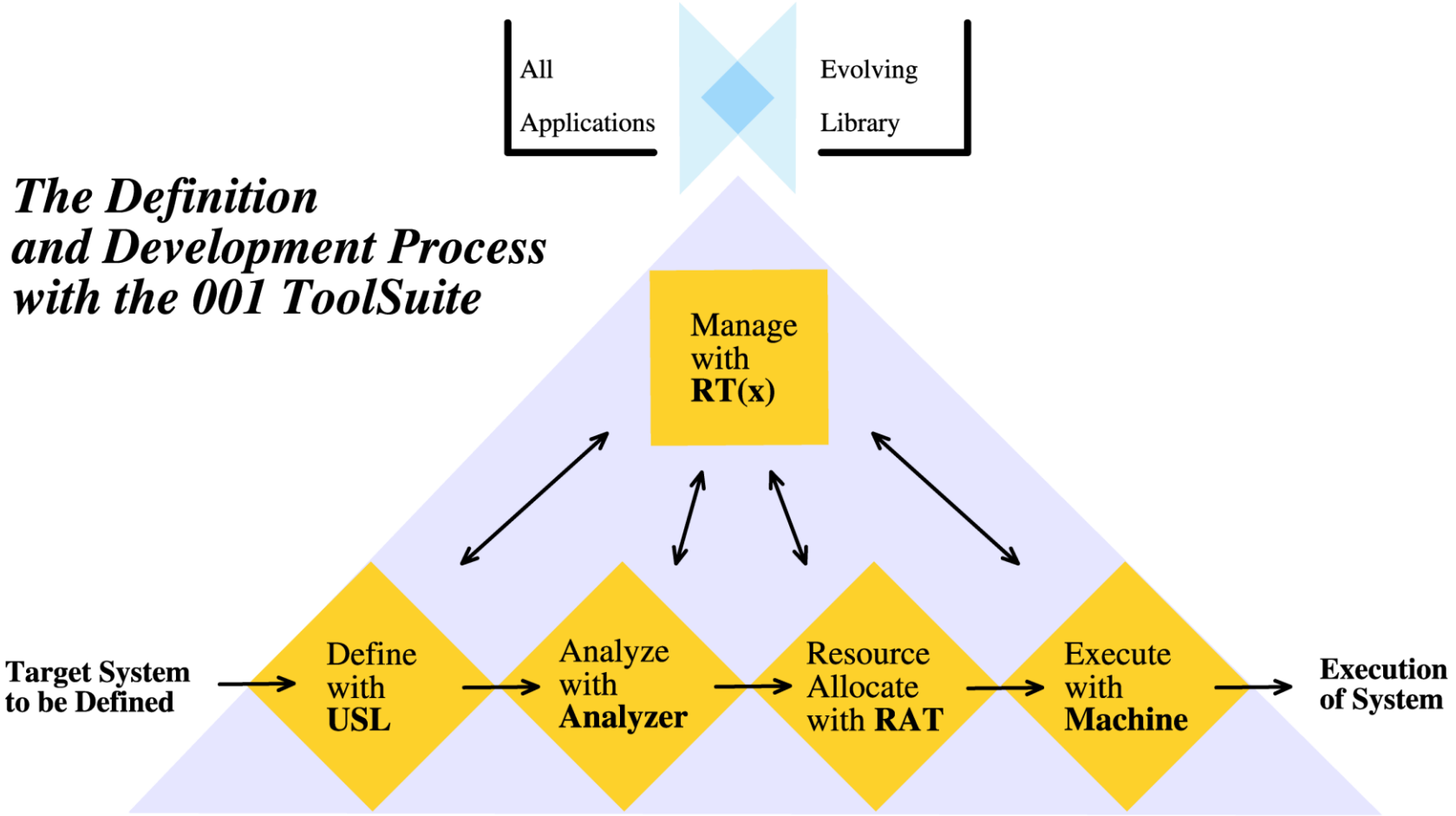
Road Map (RMap): Shows the Integration of (and Connections Between) all the Different Components of a System—e.g., How Requirements Relate to Testing System, User System (Operational Scenarios), Target System



*001** is used
to make System Oriented Objects, each of which is based upon a unique concept of ***control***

- Every system defined with USL, including 001 itself, is defined in terms of System Oriented Objects
- 001 was used to completely define–and completely and automatically (re)generate–itself

*001 Tool Suite (automation of USL)



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

- A USL model is independent of language, architecture, technology, communications protocol ... i.e., not locked in
- The 001 Analyzer generates the road map (RMap) for the target system
- The 001 RAT can be used to weave aspects about the system into the generated code (e.g., dynamic constraint testing)

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

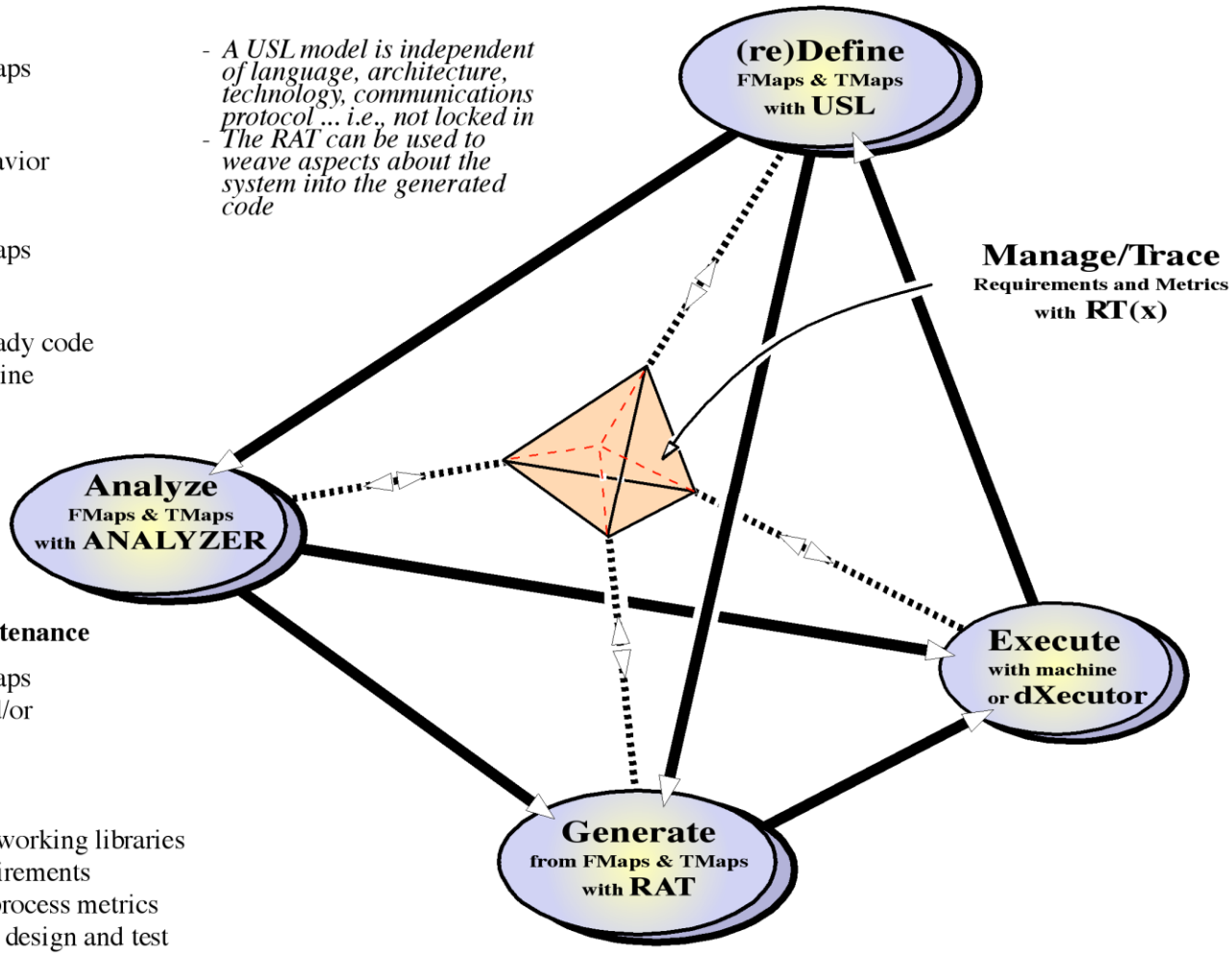
Design Changes and Maintenance

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

Management

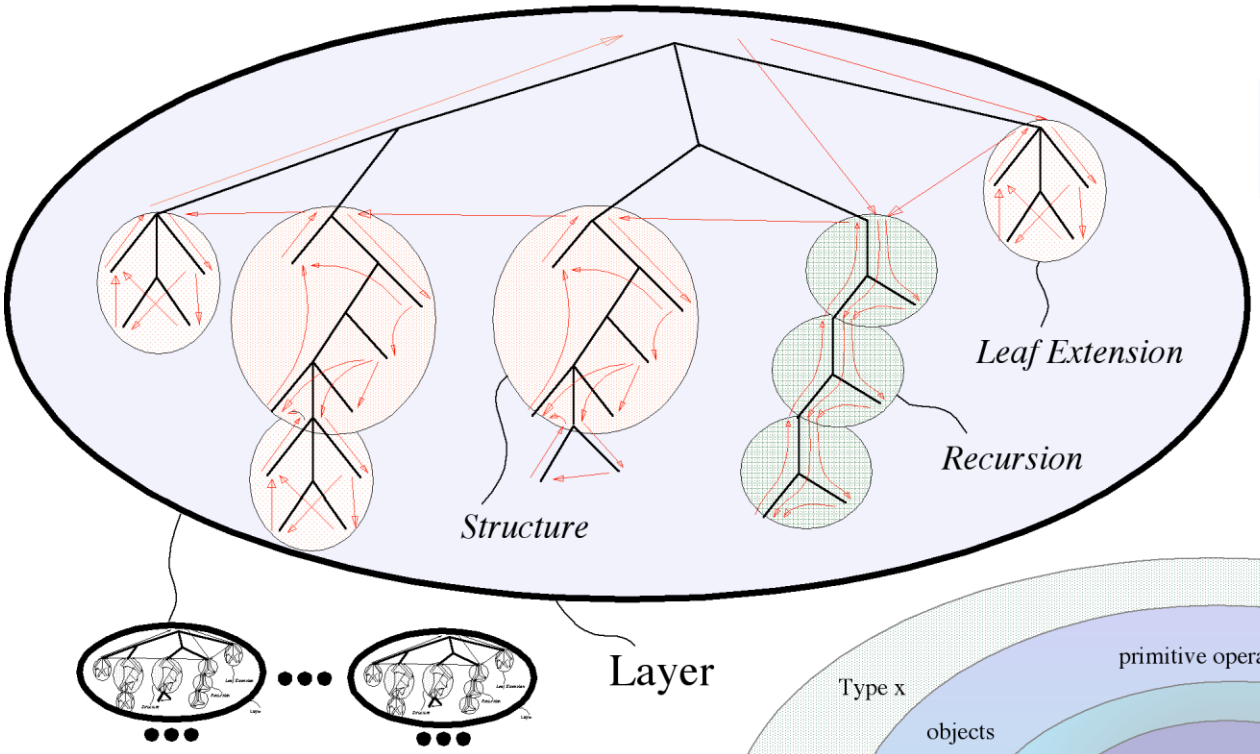
- Organize projects into working libraries
- Manage and trace requirements
- Generate product and process metrics
- Generate specification, design and test documentation

- *A 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*



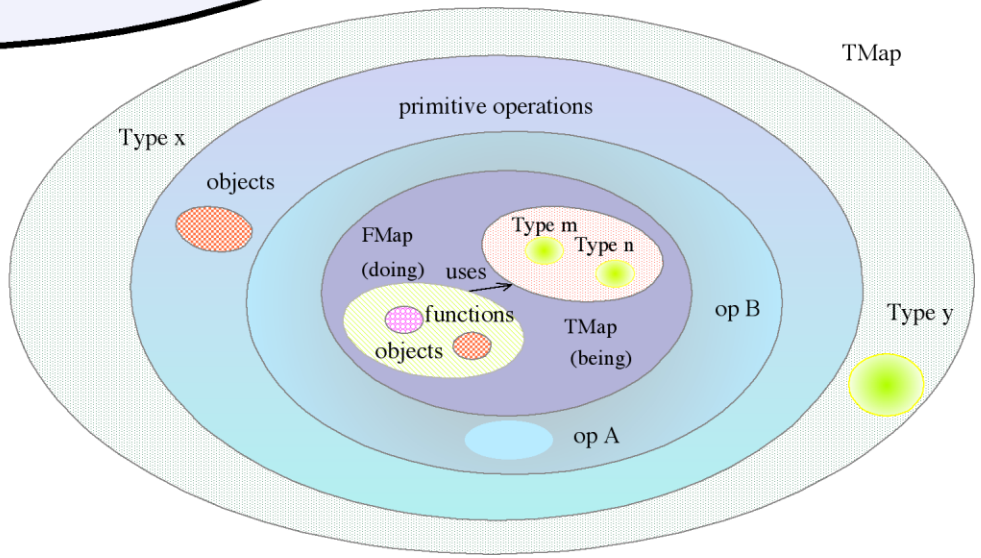
A system is defined from the very beginning to inherently maximize the potential for its own *automation* and *evolution*

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



*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, 100% of code and much of the design automatically generated by 001 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

Some USL Applications

- Enterprise Modeling (FEMA, the US Army Acquisition Process)
- Integrated Computer Aided Manufacturing (Rexham
- Manufacturing, ICAM for the US Air Force, Boeing, Rockwell)
- Flight Management (NASA)
- Ballistic Missile Defense (SDIO)
- Battle Management (Los Alamos National Laboratories (LANL), SDIO, Army)
- Space Shuttle Flight Software (NASA, Lockheed Martin)
- Banking including Cash Management, World Wide Funds Transfer and Brokering (Citibank, EBS)
- Embedded Avionics Systems (Honeywell, NASA)
- Nuclear Engineering (MIT)
- Software Development Tools (001 itself, SDIO, AT, Ingres, SAT, IBM)
- Distributed Real Time, Real World, Simulation (DoD, LANL, SDIO)
- Web Based Development (State of MA, Canadian Government, AT, Cadeon, CRG, EBS)
- Global Planning (SDIO)
- Communications (Army, BDM, AT&T, US Navy)
- Closed Combat Training Systems (US Army, Lockheed Martin)
- Theater Control (SDIO, IBCS, DI-GNU)
- Web Based Trading Systems (AT, EBS)
- Mission Planning/Troop Movement (DoD, SEI)
- Process Modeling (Mass. Police Dept., CRG, Citizens Bank, US Army)
- Guidance, Navigation and Control (NASA, US Navy)
- Domain Analysis (SEI Carnegie Mellon, US Army)
- Vehicle Control System Simulation (Lockheed Martin)
- Factory Management, Control of Job Shop, Process Control (Scott Paper, Rexham)
- Library Management (Cadeon)
- Radar Scheduling (Mitre, DoD, Army)
- Traffic Engineering (Mass. Highway Dept.)
- Financial Systems (AT, EBS, CitiBank)
- End to End Testing (US Army)
- Customer Support Systems (BDM)
- Missile Tracking and Interception (McDonnell Douglas)
- Simulation of Military Vehicles for Training (e. g., M1 tank for Lockheed Martin, U. of Florida)
- War Games (Lockheed Martin, CSC, WARSIM)
- Large Systems Environment Simulation (SDIO, LANL)
- Toll Booth Systems (MFS)
- Fisheries (Alaskan Fisheries)
- Groceries Systems Management (Toldark)
- Object Tracking and Designation (Mitre, SDIO)
- Medical Systems (SAT)
- Educational Systems (Compaq)
- Classified Systems (NSA, MRJ, SDIO, LANL)

The technology is used today to address some of today's most pressing issues including for military and space related systems to solve problems not heretofore addressed in this arena.

001™ RECEIVES OUTSTANDING MARKS AT THE NATIONAL TEST BED

Area	Estimate	Teamwork	**HTI 001**	RDD/DCDS
Software Requirements	200	151	**86**	87+10 Derived
Documentation		SRS, IRS, SDDD	**SRS, IRS, SDDD**	SRS, IRS, SEN, SDDD
SLOC			**21,000 C 24,000 Ada**	
Staff Days	838	140	**140**	120
Amount of System Completed		Detailed design complete, some Ada skeletons produced	**Code generated and executed; not fully**	Detailed design partially completed
% Complete		75%	**90%**	50%

NOTE: Lockheed Martin was the prime contractor for the HTI/001 team.

Source: Software Engineering Tools Experiment - Final Report, Vol. 1 Experiment Summary. Department of Defense, Strategic Defense Initiative Organization, Washington, D.C.

AN INTEGRATED FORMAL APPROACH FOR DEVELOPING HIGH QUALITY SOFTWARE OF SAFETY-CRITICAL SYSTEM

By
M. Ouyang
M. W. Golay

Table 2.2 Formal Methods Comparison Matrix

Meeting Requirements? (Numbers) / Method (& tool) / Features	DBTF (001)	Gypsy	HDM (STP)	HOS (USE.IT)	IBM Cleanroom	SCR	VDM	Z
Mathematically Based (f)	1, 7, 8**	1, 7	none	1, 7, 8	none	7	7	7
Model-Based Specification (f)	7	7	7	7	7	7	7	7
Specification Self-Checking (f)	3, 4	none	none	3	none	none	none	none
Validation/ Prototyping (f)	4	4	3	4	none	4	4	4
User-Friendly in Application (e)	2	none	2	2	2	none	none	none
Specification Auditable/ Traceable (e)	5	5	5	5	none	5	5	5
Specification Easy to Modify (e)	6	6	6	6	6	6	6	6
Specification Easy to Maintain	6, 8	none	none	6,8	none	none	none	none
Automatic Code Generation (e)	3	none	none	3	none	none	none	none
Supporting Tool Availability (e)	2, 3	none	2, 3	none	none	none	none	none
Development Productivity (*)	highest	lower	lower	higher	higher	lower	lower	lower
Improvement On Software Reliability (*)	highest	average	average	higher	average	average	average	average

Legend: (e) Engineering Feature

(f) Formality Feature

(*) Subjective Assessment

none: The feature does not exist for the method

** the number (N) in the cells shows the satisfaction of attributes N of Table 2.1

Market share of all of these tools is negligible compared to the overall market. StP probly has
The largest market share in this category if one includes only the core products.

N/m – Not meaningful
N/a – Not Available
N/l – Not Listed

1	Company		Aonix	Rational Corp	Sun Micro-systems	IBM Corp	HTI	Objectime	Objects In'tl
2	Product Name		StP, Validator	Several	Java Studio	Visual Age	OO1, RT(X)	Objectime	Together/J and C++
3	Foundation Date		1996	1997	1983	1890	1993	1997	1993
4	Customers		20,000	50	n/a	n/a	n/a	1000	n/a
5	Users		500,000	n/a	n/a	n/a	n/a	n/a	n/a
6	WW Revenues		42.2MM	310.5MM	1,117.7MM	12,844MM	n/l	n/l	n/l
7	SentryTech Ranking		167	41	9	1	n/l	n/l	n/l
8	Methodology	Weight (%)	Extended Waterfall	Extended Waterfall	Prototype	Component	Spiral	Component	Extended Waterfall
9	Meth. Rank	100%	5	5	3	5	9	5	5
10	Testable Architecture	100%	0	0	9	0	9	5	0
11	Traceability to code	100%	8	0	9	1	9	5	5
12	Trace to Test	100%	9	0	2	2	9	0	7
13	Integration	75%	7	2	7	7	9	5	9
14	Scaleability	100%	5	2	1	2	9	5	6
15	Ease of Use	75%	2	1	9	2	8	5	2
16	Pattern Reuse	100%	2	2	2	5	9	0	5
17	UML	50%	5	5	5	5	5	5	5
18	Content Searchable	100%	9	2	0	0	9	0	5
19	Automation	75%	2	2	2	7	9	7	7
20	Code generation	100%	5	2	9	2	9	7	5
21	Performance	100%	5	5	1	5	9	2	5
22	Tool integration	20%	8	5	2	2	8	3	1
23	Repository/CM	50%	0	9	0	9	9	5	9
24	Strong Typing	50%	0	0	0	0	9	0	0
25	Low Risk	100%	0	0	0	0	7	0	0
26	Maturity	75%	9	5	0	5	9	5	1
27	Productivity	100%	5	2	5	3	9	7	5
28	Quality of code	100%	3	1	9	1	9	3	3
29	Rules	50%	0	0	9	0	9	0	3
30	Documentation	20%	5	1	0	0	9	1	3
31	Cost	20%	0	3	6	4	2	1	5
	Weighted Average		3.31	1.62	3.13	2.17	6.61	2.67	3.28
	Rank		2	7	4	6	1	5	3

Results to date – Study

- Study compared 001 with current environment.
- Current Environment for Developing Embedded Systems
  - Requirements management - Rational RequisitePro
  - Design modeling - Rational ROSE
  - Design documentation - Rational ROSE & custom macro
  - Structural coverage - Eastern Systems LDRA
  - Debugger - Borland
  - Testing & regression testing - custom script builder

Results to date – Study

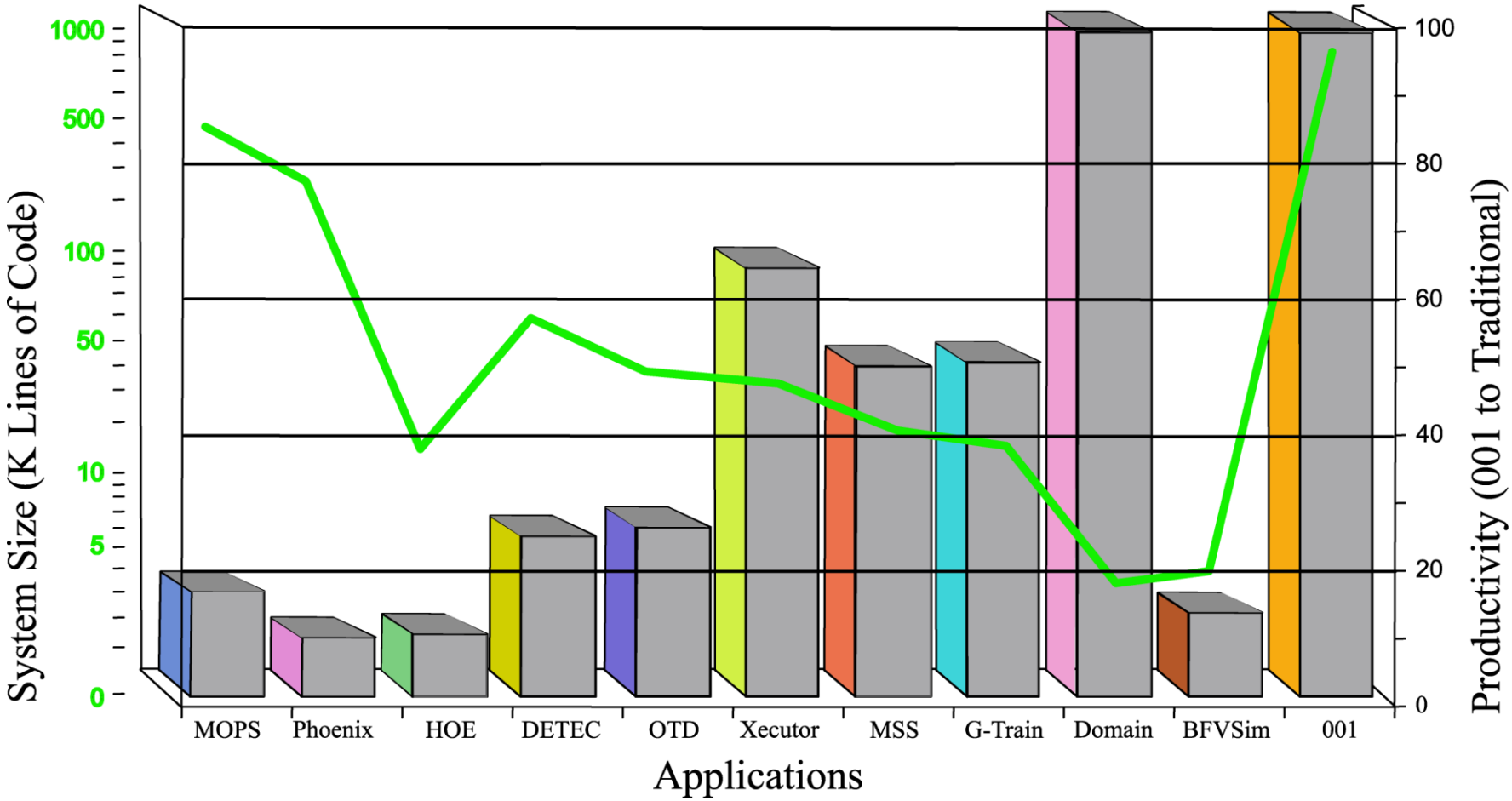
- Findings
  - 50% to 75% improvement in requirements management
  - Equal structural coverage capability
  - 400% improvement in design modeling
  - 500% improvement in quality and completeness of auto-generated code
  - 100% improvement in quality and completeness of auto-generated design documentation
  - 100% improvement in design change / iteration process

Results to date – Study

- Findings
  - Equal training required
  - 100% improvement in user interface
  - Equal regression testing capability
  - 1000% improvement in reuse!

Productivity Results

All of these system were defined with USL
and developed with 001, including 001 itself.



USL/001 PRODUCTIVITY

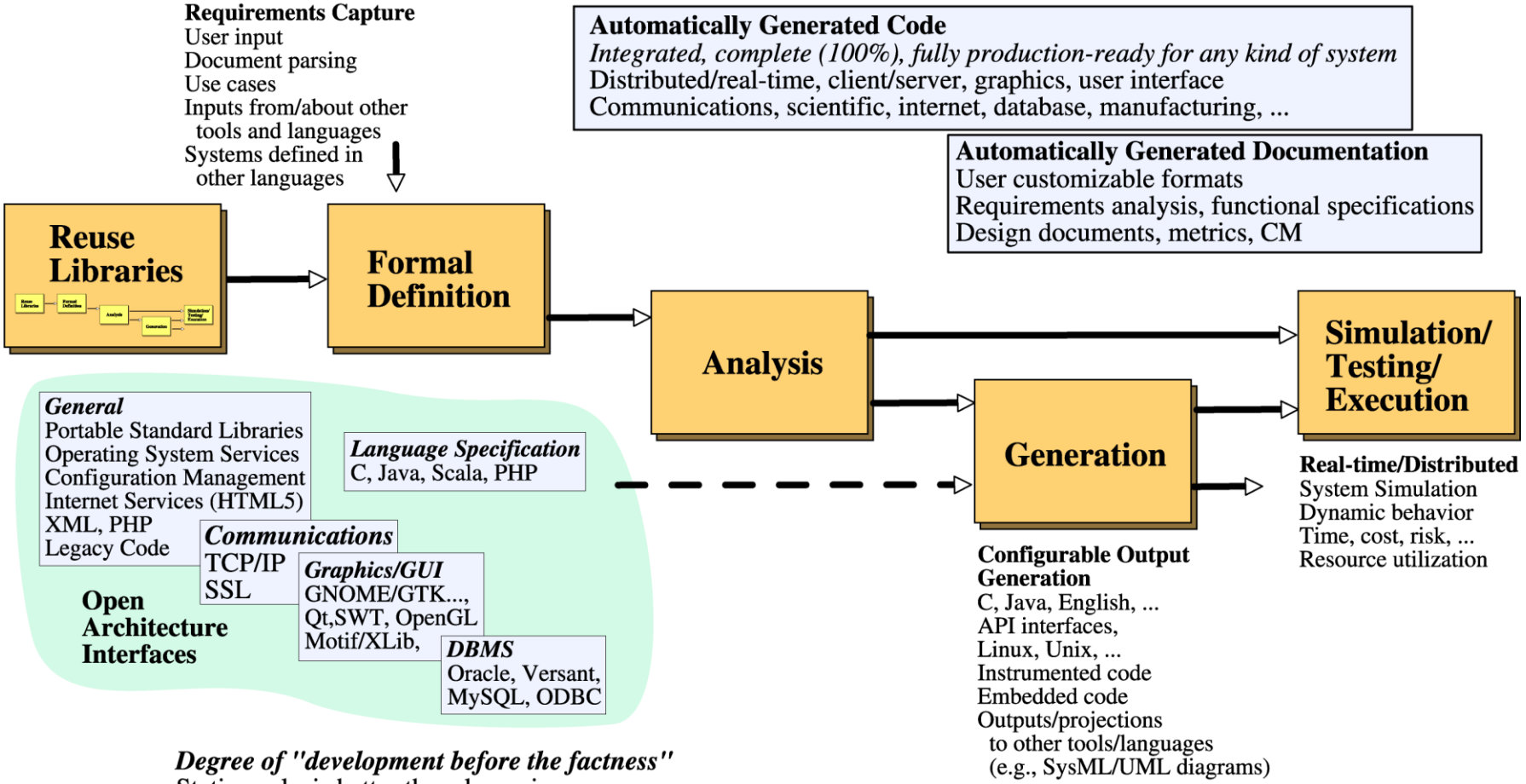
Productivity measured from initial establishment of system functional requirements through operational, validated code. System must insure ultra high level productivity,reliability, reusability, and documentation. Cost of a line of code includes:

- Definition of system in terms of its functional requirements
- Implementing the code
- Testing, validation and verification of the code
- Testing the system as defined
- Full documentation

A large number of comparative productivity tests have been held over the past 15 years in which USL/001 has been used to develop a particular software application in parallel with other software development approaches used to develop the same identical application in order to determine the relative productivity of these approaches. These comparisons have been conducted by the US Government, industry, and academic organizations and refereed by competent third party observers or by the agency sponsoring the competition. The results of these comparative tests:

- USL/001 systems were in every case No. 1 in productivity
- USL/001 systems exhibited an even greater productivity increase when applied to larger and more complex systems
- USL/001 systems productivity factors were always greater than 2

Integrated, Seamless, Configurable Environment for Systems Engineering and Software Development with USL and the 001 Tool Suite



Degree of "development before the factness"

Static analysis better than dynamic
Inherent (by way system is defined) better than static
Better yet, not having to define it at all
Ultimate reusable: application of USL to both systems and software unifying their understanding by a formal means with a commonly held set of system semantics

Some Differences

Traditional	USL (Before the Fact)
Integration ad hoc ~Life cycle not seamless ~System not integrated with software ~Function oriented or object oriented systems ~Model driven systems ~Simulation not integrated with software code	*Integration inherent* ~Seamless life cycle ~System integrated with software ~System oriented objects ~System driven models ~Simulation integrated with software code
Behaviour uncertainties until after delivery	*Correctness by built-in language properties*
Interface errors ~Most of those found are found after implementation ~Some found manually ~Some found by dynamic testing ~Some never found	*No interface errors in a model and its derivatives* ~All found before implementation ~Some prevented inherently ~Some found by automatic static analysis ~Always found
Ambiguous requirements, specifications, designs… *introduce chaos, confusion and complexity* ~Informal or semi-formal language ~Language allows for unsecure systems ~Different phases; different languages and tools ~Different language for software than for other systems	*Unambiguous requirements, specifications, designs…* *remove chaos, confusion and complexity* ~Formal, but friendly and practical language ~Language properties promote security ~All phases; same language and tools ~Same language for software and any other system
No guarantee of function integrity after software implementation	*Guarantee of function integrity after software implementation*

Some Differences

Traditional	USL (Before the Fact)
Inflexible: Systems not traceable or evolvable ~Locked in bugs, requirements, products (e.g., operating systems, programming languages), architectures ~A need to understand details of programming languages and operating systems ~Painful transition from legacy ~Maintenance performed at code level	*Flexible: Systems traceable and evolvable* ~Open architecture ~No longer a need to understand details of programming languages and operating systems ~Smooth transition from legacy ~Maintenance performed at spec level
Need for inherent reuse (e.g., inherent priority/scheduling) ~Customization and reuse mutually exclusive	*Inherent reuse* ~Every object a candidate for reuse ~Customization increases reuse pool
Automation supports manual process instead of doing real work ~Design process is manual ~Manual/semi-automatic: documentation, programming, test generation, traceability, integration	*Automation does real work* ~Automates much of the design ~Automatic programming, documentation, test generation, traceability, integration ~100%, fully production ready code automatically generated for any kind or size of software application
Trapped in "test to death" philosophy	*Less testing becomes necessary with each new before the fact capability*
Tool set not integrated, not defined with and not generated by itself	*001 integrated, defined with and completely generated by itself*
The more reliable the system, the lower the productivity in its development	*The more reliable the system, the higher the productivity in its development*

With USL, the Potential Exists for Reaching the Goal of High Quality, “More for Less” Systems and Software

- Advantage can be taken of properties that "come with the territory", e.g., the potential to obtain the highest form of reuse, i.e., built into the language itself; much of what was needed before to develop an application no longer necessary.
- Unlike what has been in large part a manual process or automation to support the manual process, with USL, automation does the real work; much of the design is automated as well as all of the software
- Much of what seems counter intuitive with traditional approaches becomes intuitive with USL. The more reliable the system, the higher the productivity in its development...less testing becomes necessary with the use of each new before the fact capability
- This is because of the formal foundation upon which USL/001 is based.

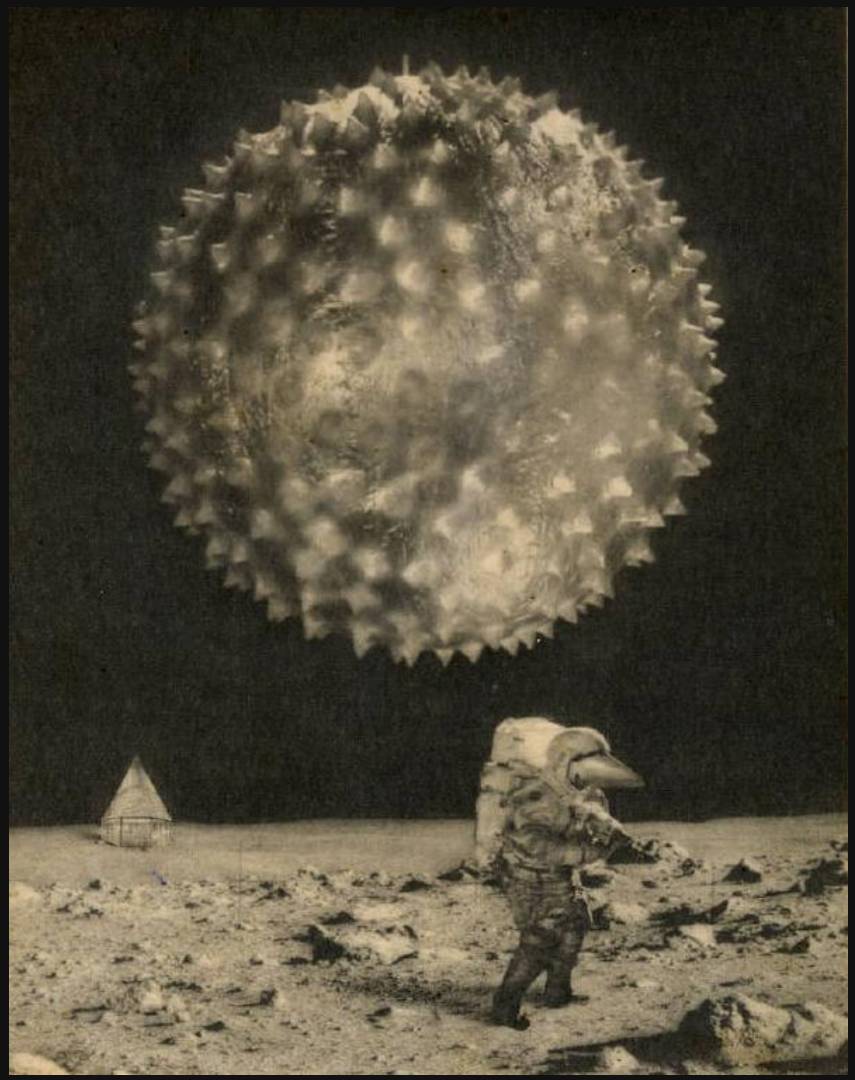
Universal Systems Language (USL) and its Automation, the 001 Tool Suite, for Designing and Building Systems and Software

Lockheed Martin/IEEE Computer Society
Webinar Series
Margaret H. Hamilton
Hamilton Technologies, Inc.

September 27, 2012

Images on Slide 1 and this Slide are from The Apollo Prophecies
Copyright © Nicholas Kahn & Richard Selesnick

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



48