conference on systems engineering research
Paper #40

Universal Systems Language for Preventative Systems Engineering

Margaret H. Hamilton and William R. Hackler
Hamilton Technologies, Inc.
17 Inman Street Cambridge, MA 02139
mhh@htius.com, ron@htius.com

Abstract

This paper is about a universal systems language based on a general systems theory, in large part, derived and evolved from lessons learned based on an empirical study of the Apollo on-board flight software effort. The language, 001AXES, was created for designing systems and building software to address problems considered next to impossible to solve, if not impossible, with traditional approaches, at least in the foreseeable future. It helps to suspend any and all preconceived notions when first introduced to it and the mathematical theory behind it, because it is a world unto itself—a complete new way to think about systems and software.

With this approach, instead of object oriented and model driven systems the designer thinks in terms of system oriented objects and system driven models. Much of what seems counter intuitive with traditional approaches, that tend to be software centric, becomes intuitive with this system centric approach. How to minimize errors and maximize integration of systems to software, reuse, open architectures, evolvable systems, and productivity in a system's development becomes better understood; this understanding can be used as a means to an end—designing better systems; building better software.

001AXES, its preventative paradigm, and how it is used to address today's pressing system engineering issues will be discussed. Examples are taken from research being done for military systems; not heretofore addressed by traditional approaches.

Introduction

The 001AXES universal systems language was created for modeling systems designed with significantly increased reliability, higher productivity and lower risk, including the following specific objectives: a) seamless integration: systems to software; requirements to specifications to design to code, and back again (using the same semantics for all systems, including software); b) reduce defect rates by a factor of 10; c) improve correctness by built-in language properties; d) unambiguous requirements, specifications, and design; e) guarantee of system integrity after implementation; f) enhance traceability and evolvability (application to application, architecture to architecture, technology to technology); g) increase in inherent reuse (within and between layers); h) full life cycle automation (e.g., automatic generation of production ready code for complete software systems of any kind or size of application from system specifications); i) automation of much of design—reduce need of designers to understand details of programming languages and operating systems; j) eliminate need for high percentage of testing; k) integration of design and development tools.

CSER 2007, Stevens Institute of Technology, ISBN - 0-9787122-1-8
PROCEEDINGS CSER 2007, March 14-16, Hoboken, NJ , USA

001AXES and its automation (Hamilton April, June 1994) are intended to address these objectives. In addition to lessons learned from Apollo, its technology takes roots from concepts older (e.g., mathematics) and newer than Apollo including other real world systems, systems theory, formal methods, formal linguistics and object technologies; keeping in mind the relevance of a technology is independent of its age.

The Apollo empirical study began with the question "what can we do better for future systems and what should stay the same because we are doing it right (Hamilton 1986, 2004)?" The search was for a means to build ultra-reliable systems. Early ideas for the technology surfaced as errors found during final testing were analyzed. Interface errors (data flow, priority and timing errors from the highest to the lowest levels of a system to the finest grain) accounted for ~75% of all errors found—finding ways to improve the integrity of integration and communication was of the highest priority. ~50% of the billions of dollars (by today's standard) spent on the life cycle was devoted to simulation, but 44% of the errors were found by manual means (eyeballing)—more automation was needed, especially static as opposed to dynamic. 60% of the errors had unwittingly existed in flights already flown—showing how subtle (and alarming) they were. Fortunately, no software errors surfaced during actual flights.

When the interface errors were analyzed in more detail, each error was placed into a category according to the means that could be taken to prevent it by the way a system is defined. During this process a mathematical theory was derived for defining systems such that the entire class of interface errors would be eliminated. Since these earlier beginnings we have continued to find ways to address other system issues just by the way a system is defined. Results of the analysis took on multiple dimensions, not just for space missions but for systems in general. Lessons learned from this effort continue today; e.g., systems are asynchronous in nature and this should be reflected inherently in the language used to define systems. This implies that a system's definition would characterize natural behavior in terms of real time execution semantics. Designers would no longer need to explicitly define schedules of when events were to occur. Events would instead occur when objects interact with other objects. By describing the interactions between objects the schedule of events is inherently defined. Combined with further research it became clear that the root problem with traditional approaches is that they support users in "fixing wrong things up" rather than in "doing things in the right way in the first place". A solution evolved—once understood, it became clear that the characteristics of good design can be reused by incorporating them into a language for defining systems.

001AXES Universal Systems Language

001AXES captures the theory based on the Apollo empirical studies. What had been created was a universal semantics for defining systems. What sets it apart from other languages is the systems paradigm upon which it is based, Development Before the Fact (DBTF) (Hamilton April 1994). Whereas the traditional approach is "after the fact", or curative, DBTF is preventative. Whereas a curative means to obtain quality is to continue testing until the errors are eliminated; a preventative means is not to allow the errors in, in the first place. Correctness is accomplished by how a system is defined, by "built-in" language properties (i.e., into the grammar). Whereas a curative means to accelerate design and development would be to add resources, a preventative approach would capitalize more on reuse or eliminate unnecessary parts of the process altogether.

A 001AXES definition not only "models" its application (e.g., as an avionics system) but also properties of control into its own life cycle that "come along for the ride" (ensuring, e.g., the inherent elimination of interface errors). Every object is a system oriented object (SOO), itself defined in terms of other SOOs. A SOO inherently integrates all aspects (e.g., function, object and timing oriented) of a system; every system is an object, every object is a system.

Unlike formal languages that are not friendly or practical, and friendly or practical languages that are not formal; 001AXES is considered by its users to be formal; yet practical and friendly (Krut 1993) (Ouwang 1995). Unlike a formal language that is mathematically based but limited in scope from a practical standpoint (e.g., with respect to size or kind of systems it can be used to define), 001AXES extends traditional mathematics with a unique concept of control, incorporating aspects such as time and space into its formalism; enabling it to support the definition of any kind or size of system.

A formalism for representing the mathematics of systems, 001AXES is based on a set of control axioms and formal rules for their application. All representations of a system are defined in terms of a functional map (FMap) and a type map (TMap). FMaps and TMaps defined for a given system are inherently integrated. Three primitive structures (and non-primitive structures derived ultimately in terms of the primitive structures) are used to define each map. Primitive functions, corresponding to primitive operations on types defined in a TMap, reside at the bottom nodes of an FMap. Primitive types, each defined by its own set of axioms, reside at the bottom nodes of a TMap. Each primitive function (or type) can be realized as a top node of a map on a lower (more concrete) layer of the system.

001AXES has been used to define systems ranging from guidance, navigation and control (Hamilton and Hackler 1988, 1990, 1991, 2004) (Hamilton 2004)) to commercial applications (HTI 1997) (HOS 1980) (Keyes 2000a, 2000b) to system and software tools (HTI 1986-2007). It can be used to provide a formal semantics foundation for other languages such as UML2 and SysML (Friedenthal, S. et al. 2006) (OMG 2006). Diverse mappings (several automated) exist that go from a given syntax and semantics to 001AXES or from 001AXES to one of a possible set of syntactical forms (and semantics), e.g., (Krut 1993) (Hamilton and Hackler 2000) (Cushing 1978).

Integrated Modeling Environment

The 001AXES language—actually a meta-language—has mechanisms to define mechanisms for defining systems. Although the core language is generic, the user "language", a by-product of the definition of newer systems (and thus newer mechanisms), can be application specific, since the language that is semantics dependent is syntax independent. Yet, every syntax shares the same semantics. Also implementation and architecture independent, 001AXES adheres to the principle that everything is relative (one person's design is another's implementation). It can be used seamlessly throughout a system's life cycle to define and integrate all aspects and viewpoints (of and about the system and its evolution).

Providing a mathematical framework within which objects and their interactions and relationships with other objects may be captured, 001AXES's philosophy is: all objects are recursively reusable and reliable; reliable systems are defined in terms of reliable systems; only reliable systems are used as building blocks; and only reliable systems are used as mechanisms to integrate these building blocks. The new system along with more primitive ones can then be used to define (and build) more complex reliable

systems. If a system is reliable, all the objects in all its levels and layers are reliable.

Six Axioms of Control

At the base of the theory behind 001AXES that embodies every system is a set of six axioms—universally recognized truths—and the assumption of a universal set of objects (Hamilton and Zeldin 1976, 1979). Each axiom defines a relation of immediate domination of a parent object over its children objects. The union of these relations is control. The axioms establish the relationships of an object for invocation, input (domain) and output (codomain), input access rights, output access rights, error detection and recovery, and ordering during its developmental and operational states.

Overarching is that all aspects within a 001AXES universe are related to the real world and the language inherently captures this. In so doing it meets the challenge linguists describe of assuring consistency in meaning, of "fitting together the partially fixed semantic entities that we carry in our heads—to approximate the way reality is fitted together as it comes to us from moment to moment. The entities are the world reduced to its parts and secured in our minds; they are a purse of coins in our pocket with values to match whatever bargain or bill is likely to come our way." (Bolinger 1981)

It is important to be able to visualize a system definition both with respect to what it does (level by level) and how it does it (layer by layer). A hierarchical definition can run the risk of not being reliable, however, unless there are explicit rules that ensure that each decomposition is valid. The axioms of control provide the formal foundation for a 001AXES "hierarchy" (referred to as a map which is both a hierarchy of control and a network of interacting objects); explicit rules have been derived from these axioms for defining a map; where among other things structure, behavior and their integration are captured. An object is decomposed until the primitive objects it has ultimately been defined in terms of have been reached. Resident at every node on an FMap is a function; resident on every node of a TMap is a type. The object at each node plays multiple roles, e.g., it can serve as a parent (in control of its children) or a child (being controlled by its parent). What follows is a discussion of the six axioms of control and some derived theorems.

Axiom 1 states that a given parent controls the invocation of the set of children on its immediate, and only its immediate lower level. Take for example an FMap; the parent controls its children to perform its own mapping; that is, the parent's mapping is completely replaced by its children's mappings; no more, no less; yet the parent (as a controller) remains in control of its children. Note that a 001AXES function is a hybrid consisting of a traditional mathematical construct, i.e., an operation (mapping) and a linguistic construct, i.e., an assignment of particular variables to inputs and outputs. Some implications are that a parent can only invoke its immediate offspring; it cannot invoke itself, its parent, any of its descendants other than its immediate offspring, any other offspring of its own parent, another parent's offspring, or an offspring that invokes its parent; the children of each parent must collectively perform no more and no less than the parent's requirements; e.g., if a lower level function is removed and its ancestor still maintains its same mapping, the function is extraneous (extraneous functions proliferate test cases and complicate interfaces).

Axiom 2 states that a given parent controls the responsibility for elements of only its own output space (codomain). For an FMap this simply states that the role of the parent is to perform its own mapping. For any given element in the domain of the parent's function, the parent is responsible for producing the correct corresponding element in the range (codomain). While the parent can get "help"

from its offspring in the performance of this function, it cannot delegate this responsibility. For a given input, only the parent can ensure the "delivery" of the corresponding output. Some implications are a parent loses control (cannot ensure correct outputs) when any of its offspring stop before completion, go into an endless loop or do not return required information back to the parent; the decomposition stopping point can be determined and the bottom is reached when each function has been defined in terms of other functions on a defined type; the functions' behavior one level from the bottom can be defined by understanding the behavior of each function at the bottom level and how it relates to other functions on that level; one can define each next highest level function in the same manner until the top node is reached; the behavior of the top node is ultimately determined by the behavior of the collective set of bottom nodes; there may be more than one formulation for a particular function, it is only necessary that the mapping be identical.

Axiom 3 states that a given parent controls the output access rights (*access rights* is the ability to obtain or alter the values of variables) to each set of variables whose values define the elements of the output space for each immediate, and only each immediate, lower level child. Axiom 3 is concerned with where the required range element (as produced by an offspring) is delivered as dictated by its parent. The parent can assign to its offspring the right to alter the values of the output variables of the function that the offspring replaces. Implications are: each range variable of the parent that an offspring replaces, must appear as a range variable of the function of at least one of its offspring; tracing of outputs can be traced for each and every performance pass (i.e., instance by instance); the parent's output variables are a subset of the output variables of the collective children.

Axiom 4 states that a given parent controls the input access rights to each set of variables whose values define the elements of the input space for each immediate, and only each immediate lower level child. It is concerned with the way the parent controls access to its domain elements; specifically the parent can grant its children the right to access its domain elements for reference only. Implications are: the parent does not have the ability to alter its domain elements; each domain variable of the parent must appear as a domain variable in at least one of its children; inputs can be traced for each and every performance pass.

Implications of both axioms 3 and 4 are: the variables of the output set of a function cannot be the variables of the input set of that same function. If $f(y, x) = y$ could exist, access to y would not be controlled by the parent at the next immediate higher level; the variables of the output set of one function can be the variables of the input set of another function only if they belong to functions on the same level. If $f1(x) = y$ and $f2(y) = g$, both functions exist at the same level.

Axiom 5 states that a given parent controls the rejection of invalid elements of its own, and only its own, input set (domain). It requires that the parent must ensure the rejection of inputs received that are not in the domain of the parent. A parent, in performing its corresponding function, is responsible for determining if such an element has been received; if so, it must ensure its rejection.

Axiom 6 states that a given parent controls the ordering of each tree for the immediate, and only the immediate, lower level. Axiom 6 requires the parent to control the order (including priority) based on e.g., time, events, importance, computational needs of the invocation of its children and their descendants. Implications are: total order relationships; if two processes are scheduled to execute concurrently, the priority of each process determines precedence at the time of execution; the priority of a process is higher

than the priority of any process on its most immediate lower level; if two processes have the same parent, all processes in the control tree of the process with the highest priority are of a higher priority than all the processes in the control tree with the lower priority; a process cannot interrupt itself, or its parent.

Other implications (derived theorems) of the axioms are: every object has a unique parent, is under control and has a unique priority; communication of children is controlled by the parent, and dependent functions exist at the same level; the priority of an object is always higher than its dependents and totally ordered with respect to other objects at its own level. Relative timing between objects (including functions) is therefore preserved; maximum completion or delay time for a process is related to a given interrupt structure. Absolute timing can therefore be established (i.e., it can be determined if there is enough time to do the job); the relationships of each variable are predetermined, instance by instance thus eliminating conflicts; each system has the property of single reference/single assignment. SOOs can therefore be defined independent of execution order; the nodal family (a parent and its children) does not know about (is independent of) its invokers or users; concurrent patterns can be automatically detected; every system is event driven (every input and every output is an event; every function is event driven), and can be used to define discrete or continuous phenomenon; each object, and changes to it, is traceable; each object can be safely reconfigured; every system can ultimately be defined in terms of three primitive control structures, each of which is derived from the six axioms—a universal semantics, therefore, exists for defining systems.

Universal Primitive Structures

A structure relates members of a nodal family according to a set of rules derived from the axioms of control. A primitive structure provides a relationship of the most primitive form of control between objects on a map. All maps are defined ultimately in terms of three primitive control structures, and therefore abide by the formal rules associated with each structure: a parent controls its children to have a dependent relationship (Join), independent relationship (Include), or a decision making relationship (Or). Figure 1 contains a description of the three primitive structures; used generically in both TMap or FMap definitions; Figure 2 contains a description of the rules as applied to an FMap. Figure 3 shows two independent syntactical forms that can be used to represent the semantics of the three primitive control structures.

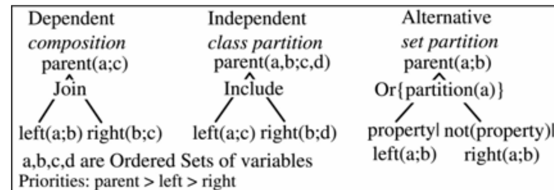

Figure 1: Primitive Control Structures

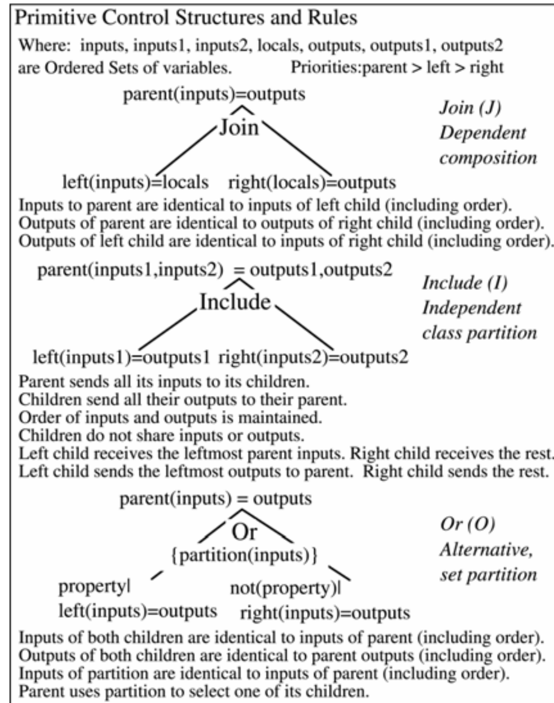

Figure 2: Rules for Primitive Structures

The structures ensure that all interface errors (75% to 90% normally found, if found at all, during testing in a traditional

development) are eliminated "before the fact" at the definition phase. Although a system defined in these structures has properties for systems in general, the properties have special significance for real time, distributed aspects of a system (that every system ultimately has): each system is event interrupt driven; each object is traceable, reconfigurable, and has a unique priority; independencies and dependencies can readily be detected (manually or automatically) and used to determine where parallel and distributed processing is most beneficial.

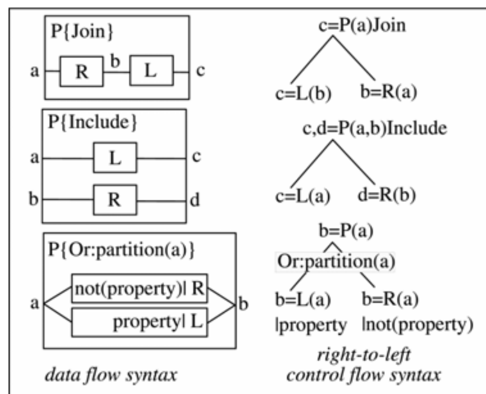

Figure 3: Syntax Independence

Definition and Execution Space

SOOs are defined in terms of FMaps and TMaps—FMaps to represent the dynamic (doing) world of action by capturing functional and time (including priority) characteristics and TMaps to represent the static (being) world of objects by capturing spatial characteristics (e.g., containment of one object by another). Maps guide a designer in thinking through concepts at all levels and layers of system design and the 001 Tool Suite (001), the automation of 001AXES, in automatically generating resource allocation designs and the system's software.

With a map, everything you need to know (no more, no less) is available. Inherent are features such as polymorphism, encapsulation and inheritance; the functional side is defined in terms of the type side and vice versa, providing the ability to automatically trace within and between levels and layers of a system (e.g., in an FMap, an output variable of a function is fully traceable to all other functions using that variable's object state).

FMaps are used for defining functions and their relationships to other functions using the types of objects in the TMap(s). Each function on an FMap has one or more objects as its input and one or more objects as its output. Each object resides in an object map (OMap) and is a member of a type from a TMap. TMaps are used for defining types and their relationships to other types. Every type on a TMap owns a set of inherited primitive operations for their allowed FMap functional relationships. FMaps are inherently integrated with TMaps, in fact recursively so, by using objects (members of the types in the TMap) and their primitive operations (e.g., if a type is changed on a TMap, all FMap areas impacted are traceable). FMaps are defined in terms of TMaps and TMaps are defined in terms of FMaps. FMaps are used to define, integrate, and control the transformations of objects from one state to another state.

A SOO is realized (has all its values instantiated for a particular performance pass) in terms of execution maps (EMaps), each an instantiation of an FMap, and OMaps, each an instantiation of a TMap. When an object state beginning event occurs, each function that depends on that object state is instantiated. This instantiation process always results in a totally ordered (in terms of priority) map of function instances; when a function instance becomes ready to execute it is always inherently correctly scheduled and allocated to the appropriate resource(s). Figure 4 shows an example of scheduling with a performance pass of a function, **F**.

Past, present and future related indicators are used to identify when a line of control or action (i.e., a function instance) occurs; or when an object state exists. Subfunctions, **A** and **B**, are concurrently executing in **interval 1**, with object states, **a** and **b**, respectively. In

interval 2, **A**(**a**) is still active and **F**'s control jumps ahead to include activating **E**(**b1**) concurrently. B is past, having produced **b1**, and *C(a1)* and *D(c)* will be future actions. In **interval 3**, E(b1) has completed, producing **e**, an output event in partial completion of **F**, while **F** continues with **C**(**a1**).

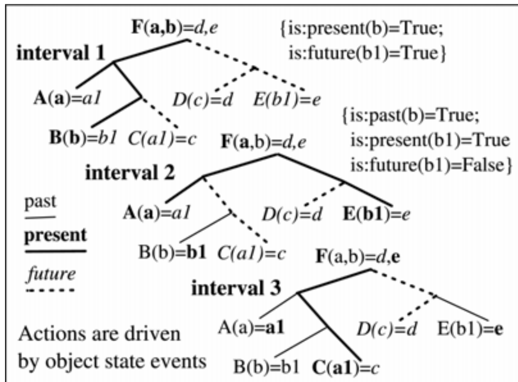

Figure 4: Real Time Event Scheduling

Using an integrated set of OMaps and EMaps a system is able to be understood in terms of its behavior (e.g., cost, risk, and real time characteristics) and structure. OMaps and EMaps are always under the control (thereby following the control axioms) of the FMaps and TMaps from which they were instantiated.

Typically, a team of designers begins by sketching a TMap(s); where they decide on the types of objects (and the relationships between them) in their system. Often a Road Map (RMap), that organizes all system objects including FMaps and TMaps, is sketched in parallel with the TMap. Once a TMap has been agreed upon, the FMaps begin almost to fall into place because of the natural functionality (or groups of functionality) resident in the TMap system. The TMap provides the structural criteria from which to evaluate the functional partitioning of the system (e.g., the shape of the structural organization of the FMaps is balanced against the structural organization of the shape of the potential objects as defined by the TMap). With FMaps and TMaps a system (and its viewpoints) is divided into components and groups of components which naturally work together.

User Defined Structures

Any system can be defined completely using only the primitive structures, but less primitive structures accelerate the process of defining and understanding a system. Since non-primitive structures are ultimately derived from the three primitives, they are governed by the control axioms. Defined structures for both FMaps and TMaps can be created for any kind of system. A powerful template form of reuse, the defined structure provides a mechanism to define a map without some of its elements being explicitly defined. An FMap structure has unknown functions; a TMap structure has unknown types. Async is an example of a real time, distributed, communicating FMap defined structure with both asynchronous and synchronous behavior. TreeOf is an example of a TMap defined structure (a collection of the same type of objects ordered using a tree indexing system). Each type structure assumes its own set of possible control relations for its parent and children types. Abstract types decomposed with the same type structure on a TMap inherit (or reuse) the same primitive operations and therefore the same behavior (each of which is available to FMaps that have access to members of each of its types). With the use of FMaps, TMaps and user defined structures as well as other forms of 001AXES reuse, a system is defined from the very beginning to inherently maximize the potential for its own reuse.

Figure 5 shows an interrupt structure that performs the functions, I? and/or F?, that are to be defined when interrupt is used in another FMap. The key to understanding interrupt is the primitive operation, is:present(i). is:present is evaluated asynchronously when the value of s0 is available. If the value of i is available, then I? is performed; otherwise, F? is performed and this process is repeated. Figure 6 shows a set of execution snapshots

(EMaps) depicting the performance of interrupt.

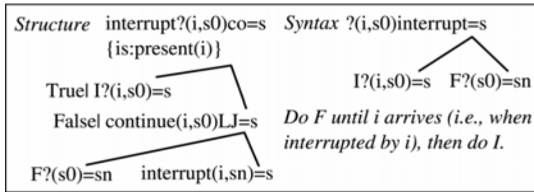

Figure 5: Interrupt Structure

In the first snapshot, since **s0** is available and *i* is not, is:present returns **False** and **continue** invokes **F?(s0)**. After the first F completes, the interrupt leaf function checks the state of *i* again (still not available), selects continue, invokes another F, **F?(s1)** with a revised state of s0. This process continues until **i** becomes available. Interrupt granularity is based on the time it takes to complete F.

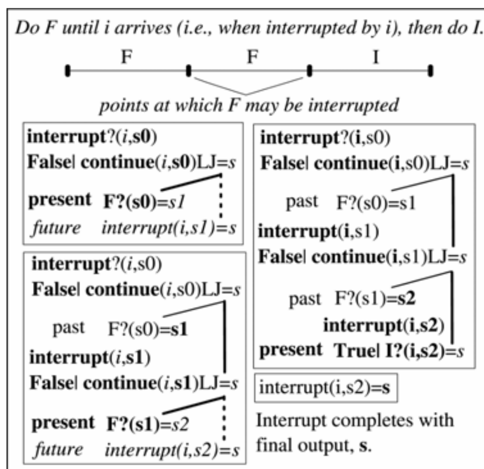

Figure 6: Interrupt Execution Snapshots

Figure 7 shows a real time structure designed for the repeated scheduling of a function within a period of time. The result of the repeated application of F is returned when run:timer(et)=end completes, setting is:present,any(end) to True. The parent of db0 is returned by into(db0)=P. Since the invocation of functions are asynchronous and event driven, the function, run:timer, does not block continued processing of RUN. is:present is asynchronously evaluated only when the object states for db0 and/or st are available. F?(db0) and run:timer(st) are able to execute simultaneously. When both complete, when:all,present synchronizes st1 and db1 as st2 and db2 respectively; only then will the RUN EMap leaf node be invoked. In the use of this structure, F? should perform within the schedule time interval so that other lower priority functions (i.e., those having lower priority than the parent user of this structure) have an opportunity to execute. The period time of a schedulePeriod TMap structure is assumed to be larger than the schedule time (and could be defined explicitly with constraints).

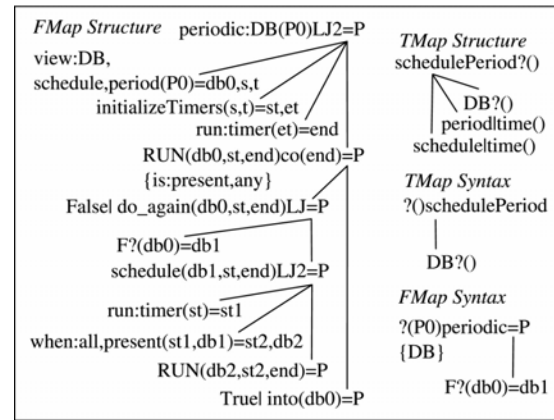

Figure 7: Periodic Structure

Figure 8 shows a guidance, navigation and control application of the periodic structure that could be used within a larger (e.g., vehicle) system. The nesting of the functions, NGC, GC and C, inherit the real time schedule characteristics defined by the periodic structure. The priority of a function is determined by its location within its FMap. The ci structure derived from the Include structure defines C as a higher priority than guidance and GC as a higher priority than navigation. Therefore, control may interrupt guidance which in turn may interrupt navigation. If only one processor is available, then control always executes first. Any leftover processing time is then given to guidance and then to navigation. Any number of processors can be added without rewriting the application because scheduling is built-in.

The completion of a nested period of time (e.g., C within GC) corresponds to when finer grained information of a faster cyclic period is

available to a slower cyclic period of time. The support structure, getPut, provides access and security between databases (e.g., control, guidance and navigation databases). In an OMap, get:c removes a child from its parent, p0, and put:c puts the child back, under, its parent, p1. Once put:c has been applied by the getPut structure, guidance has access to (and may change) control's period and schedule times; but, control can not change the schedule or period times of guidance or navigation, providing a form of security.

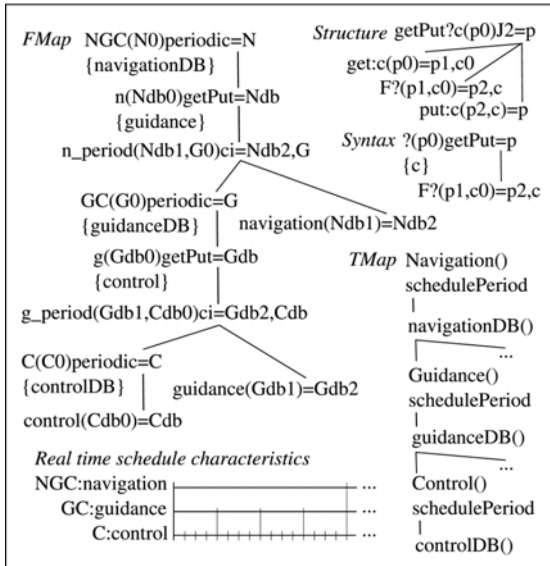

Figure 8: GN&C Application

A user defined structure that can be used generically in both TMaps and FMaps is called a universal structure. This universality derives from the fact that each map node has a mapping and each universal primitive mapping may be used at a leaf node with an interpretation that is dependent on whether it is used in a TMap or an FMap. When interpreted within a TMap a primitive mapping makes a static correspondence between its domain and codomain. Because of the static nature of a TMap mapping, either its left or right set of relations may become the domain of an FMap function; the other set then becomes the corresponding codomain of the FMap function. In an FMap, a primitive mapping makes a dynamic correspondence from its domain to its codomain.

Jset is a user defined universal structure (see Figure 9). Jset is recursive (because of its reuse leaf node, Jset). Object instances of this structure result in co-dependent patterns of some type, T?, with zero, one or two other T elements. This structure can easily define cars on a road, people standing in a line or a repeated set of dependent functions. Dependencies between T elements are identified by the relations, r and r1. r and r1 are inverse relations. When a T is r1 related to a second T, the second T is r related to the first T. In a TMap r and r1 are interpreted as relations between different object states of T; in an FMap, r and r1 are interpreted as different object states.

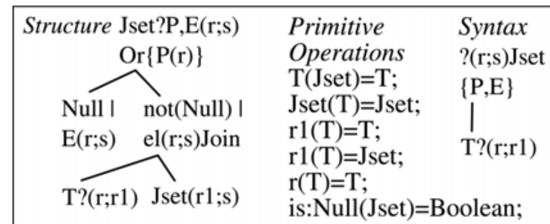

Figure 9: The Jset Structure

Within Jset interface elements, identified by the Syntax statement, are unknowns that need to be satisfied by a particular use. An interface element resolved in some context may again be unknown due to its participation in its context map's interface. Ultimately, unknown elements are statically or dynamically resolved, completing their related definitions so that they can be instantiated (or executed). In addition, if a structure has primitive operations, their unknown elements will be resolved by its use in a TMap (e.g., the parent, cars, using Jset will inherit Jset's primitive operations with appropriate resolutions such as cars replacing Jset, see Figure 9 and Figure 10).

Jset is used, in Figure 10, to decompose both a TMap type, cars, and an FMap function, largestCar. Unknowns resolved in the TMap context are types: cars, car and Any; and the value Null. Unknowns resolved in the FMap context are functions: largestCar, check_car, is:Null and Id:2. Since Jset is used

to structure the set of cars, each car may have a before and next relation to two other cars. A car object depends on other car objects with these relations. The use of these relations in an FMap are supported by the primitive operations, before(car)=car and next(car)=car, respectively. When a cars OMap is created (instantiated), manipulated or examined a set of universal map operators may be used. These are inherited by any map type of object (e.g., a concrete object like car or a more generic object like an OMap, a TMap, an EMap or an FMap); and resolved just as with Jset (e.g., k:cars(Any)=cars inherited from type TMap).

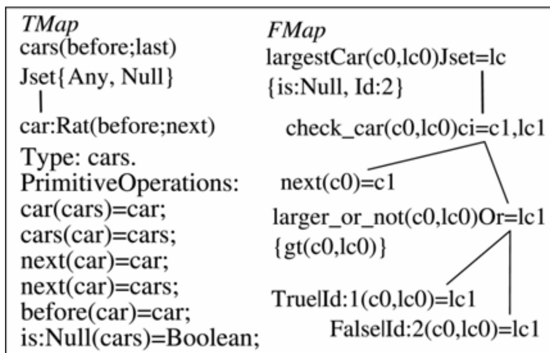

Figure 10: An Application of Jset

The largestCar function uses Jset to recursively process all cars and return the largest car (a Rational number, Rat). check_car evaluates a car with larger_or_not and advances from the current car, c0, to the car, c1, at the other end of the next relation using the next(c0)=c1. The decision function, larger_or_not, determines if car, c0, is greater than the currently known largest car, lc0, using partition function, gt(c0,lc0), where gt is an infix operator meaning greater than. Its implicit output (a Boolean) is used to select c0 (when True) as the newest largest car, lc1; or select (when False) the currently known one, lc0, to remain being the largest one as lc1. All cars are evaluated when is:Null which replaces P in Jset's Syntax being used as a partition function implicitly returns True, selecting the Id:2 function as the final step in largestCars's recursive processing. The function Id:2 (that replaces E in Jset's Syntax) is used to select the second element of the ordered set (i.e., c0,lc0) that replaced r in this use of Jset. The final outcome will be lc, the latest value of lc1. This is because r1 in the recursive Jset leaf corresponds to r (by position) in its ancestor; and when r1 is replaced by "c1,lc1" in largetCar, r becomes "c1,lc1". Id:2 then selects lc1 to be lc (which replaced s in Jset).

Universal Primitive Operations

The TMap provides universal primitive operations, for controlling objects and object states, inherited by all types. They create, destroy, copy, reference, move, access a value, detect and recover from errors, access the type of an object and access instances of a type; providing an easy way to manipulate and think about different types of objects. With the universal primitive operations, building systems can be accomplished in a more uniform manner. TMap and OMap are available as types to facilitate the ability of a system to understand itself better and manipulate all objects the same way. TMap properties ensure the proper use of objects in an FMap. A TMap has a corresponding set of control properties for controlling spatial relationships between objects (e.g., two objects can not exist in the same place at the same time. Thus one cannot put a leg on a table where a leg already exists; conversely, one cannot remove a leg from the table where there is no leg; a reference to the state of an object cannot be modified if there are other references to that state in the future). Reject values exist in all types, allowing the FMap user to recover from failures if encountered.

As experience is gained with different types of applications, new reusables emerge. For example, a set of mechanisms was derived for defining interruptable, asynchronous, communicating, distributed controllers. This is essentially a second order control system (with rules that parallel the primary control system of the primitive structures) defined

with the formal logic of user defined structures. In such a system, each distributed region is cooperatively working with other distributed regions and each parent controller may interrupt the children under its control. These reusables can also be used to manage other types of processes (e.g., those used to manage a development environment).

Constraints

When designing a system, it is important to understand the performance constraints of a functional architecture and to have the ability to rapidly change configurations. A system is flexible to changing resource requirements if the functional architecture definition is separated from its resource definitions. To have the necessary built-in controls, the same language, 001AXES, is used to define functional, resource and allocation architectures.

The meta-language properties of the language can be used to define global and local constraints for both FMaps and TMaps; constraints themselves defined in maps. If we place a constraint on the definition of a function (e.g., where sendBy:vehicle takes 2 hours), it influences all other functions that use this definition. Such a constraint is global with respect to the uses of the original function.

Global constraints may be further constrained or overridden by local constraints placed in the context of the definition that uses the original function definition (e.g., where sendBy:car takes 4 hours, overriding the default). The validity of constraints and their interaction with other constraints can be analyzed by static or dynamic means with 001. The property of being able to trace an object throughout a definition supports this type of analysis; it provides the ability to collect information on an object as it transitions from function to function. As a result, one can determine direct and indirect causal effects of constraints.

Automation

Because of a SOO's inherent support of automation; more automation is possible (e.g., much of a system's design can be automatically generated from SOOs). Given a set of FMaps and TMaps, 001 has the means to perform requirements analysis and to simulate and observe the behavior of a system as it evolves and executes in terms of OMaps and EMaps; if it is software the same FMaps and TMaps can be used to automatically generate a complete software system of any kind or size resulting in production ready code and documentation; in fact, 001 is defined with itself and automatically generates itself. That used to build other systems builds itself.

One might ask "how can one build a more reliable system and at the same time increase the productivity in building it"? Take for example, testing. Unlike a traditional approach with a "test to death" philosophy where the more reliable the system the less the productivity, with 001 the more reliable the system the higher the productivity—less testing is needed with each new before the fact capability. Before the fact "testing" is inherently part of every design and development step. Errors are prevented because of that which is inherent or automated. Correct use of 001AXES eliminates interface errors; the 001Analyzer statically hunts down errors in case the language was not used correctly. Testing for integration errors is minimized, since SOOs are inherently integrated. Automation removes the need for most other testing (e.g., since the 001 Resource Allocation Tool (RAT) automatically generates all the code, no manual coding errors will be made). And, since the RAT can be configured to generate to an architecture of choice, no manual errors result from conversion to a new architecture. Other test cases are not necessary to develop because they are automatically generated as part of the RAT generation process.

The 001DXecutor component of 001 is a distributed runtime execution engine. 001DXecutors form a hierarchy, each managing its own resources (e.g., different CPUs) and communicating (e.g., using TCPIP) to other 001DXecutors. They form a substrate upon which a 001AXES system can be executed with asynchronous event driven behavior. This takes advantage of the inherent asynchronous and priority properties built into the grammar of 001AXES to automatically coordinate and schedule, providing enhanced reliability and eliminating unnecessary design tasks (e.g., it is estimated that ~80% of the UML2 specification standard could be eliminated with a 001AXES 001DXecutor active object substrate).

Take also reuse. The more the inherent reuse; the higher the reliability and productivity. Not only does a SOO have properties to support the designer in finding, creating and using commonalty from the very beginning of a life cycle; commonalty is ensured simply by having used 001AXES to define it; providing the opportunity for many parts of the life cycle to become no longer needed. Every object is a candidate reusable—and integratable—within the same system, other systems and their evolution.

Conclusion

Unlike having first created a language with a syntax first, syntax dependent approach with informal semantics; with 001AXES a formal systems theory was derived from an empirical study of real world systems; a universal systems language was then derived for defining (and developing) system oriented objects based on the generic system semantics of the systems theory (a semantics first, syntax independent approach). Unlike additional languages, language mechanisms, rules and tools being added, ad hoc and "after the fact", as more is learned about a class of systems; with 001AXES, additional language mechanisms and tools are derived ultimately in terms of the core set of the universal language's primitive mechanisms.

With 001AXES semantics, system designers have the potential to eliminate well known problems; because of the properties that in essence "come with the territory": more reliable systems, integration of systems to software, reduction of testing without compromising a system's integrity and having the ability to maximize reuse are all well within reach. It is not magic. No more than many things we now take for granted, that were themselves once thought of as magic. What at first appears to be magic, because it is not yet familiar, transitions to common sense once understood—a duality of control and flexibility in the process of organizing one's thoughts and recording them—so automation can take over and finish the job. Collective experience strongly confirms that quality and productivity both increase with the increased use of properties of preventative systems. Compared to traditional techniques, the productivity of 001AXES systems has been shown to be significantly greater (DoD 1992) (Krut 1993) (Ouyang 1995) (Keyes 2000a) (Schindler 1990) (SPC 1998) (htius.com) (icb.nasa.gov/001). It was also discovered that the productivity was greater the larger and more complex the system—the opposite of what one finds with traditional approaches. This is, in part, because of the high degree of 001AXES's formal and inherent support of reuse. The larger a 001AXES system, the more it has the opportunity to capitalize on reuse. As more reuse is employed, productivity increases. Measuring it becomes a process of relativity—relative to the latest system developed.

By inheriting the preventative philosophy of 001AXES, users have the potential to solve a given problem as early as possible, which means finding a problem statically is better than finding it dynamically. Preventing it by the way a system is defined is even better. Better yet, is not having to define (and build)

it at all. The ultimate reusable is in the application of the technology to both systems and software unifying their understanding by a formal means with a commonly held set of system semantics.

001AXES's formal systems theory began with Apollo, the ideal environment for jump starting a "never in the box" technology. There was no school to attend or field to learn what today is known as software engineering (or computer based "systems engineering" as it has become known today). One had to “learn” a field(s) that did not yet exist. When there were no answers, problems had to be solved that no one had ever solved before. Things had to work, and work the first time.

001AXES's creation together with its automation and experiences in its application is research and development in progress. Its technology was created to address problems considered difficult, at best, to solve (not the least of which was that of responding to the actions resulting from lessons learned). Analysis of lessons learned using 001AXES and its automation continues in a manner not unlike the empirical studies of Apollo's systems. Again and again we learn from experience (that of 001AXES users, including our own experience as 001AXES users) and evolve accordingly; maximizing the degree of preventiveness, i.e., that which is inherent and that which becomes no longer necessary.

References

Bolinger, Dwight and Sears, Donald A., Aspects of Language. New York: Harcourt Brace Jovanovich, Inc., 1981, p. 109.

Cushing, S., A Note on Arrows and Control Structures: Category Theory and HOS, Candidate BMD Data and Axioms, Contract #DASG60-77-C-0155, HOS, Prepared for Ballistic Missile Defense, Advanced Technology Center, June 1978.

Department of Defense. 1992. Software engineering tools experiment-Final report, Vol. 1, Experiment Summary, Table 1, p. 9. Strategic Defense Initiative, Wash., DC.

Friedenthal, S., Moore, A., Steiner, A., OMG Systems Modeling Language (OMG SysML) Tutorial, INCOSE 2006, Orlando, Florida, July 11 2006

Hamilton, M. 1986. Zero-defect software: The elusive goal. *IEEE Spectrum* 23(3):48-53, March.

Hamilton, M., “Inside Development Before the Fact”, Electronic Design, April, 1994, ES

Hamilton, M., “Development Before the Fact in Action”, Electronic Design, June, 1994, ES.

Hamilton, M., The Heart and Soul of Apollo: Doing it Right the First Time, 7th International MAPLD Conference, Wash. D.C., Sept. 2004

Hamilton, M, Hackler, R: "Prototyping: An Inherent Part of the Realization of Ultra-Reliable Systems", Final Report to University of California Los Alamos National Laboratory Contract No. 4-X28-8698F-1, DETEC Conceptual Model, 1988.

Hamilton, M, Hackler, R. 1990, 001: A rapid development approach for rapid prototyping based on a system that supports its own life cycle. IEEE Proceedings, First International Workshop on Rapid System Prototyping (Research Triangle Park, NC) pp. 46-62, June 4.

Hamilton, M., Hackler, W.R., Final Report: AIOS Xecutor Demonstration, Los Alamos National Laboratory, Los Alamos, NM, No. 9-XG1-K9937-1, Nov. 1991

Hamilton, M., Hackler, W. R., Towards Cost Effective and Timely End-to-End Testing, HTI, prepared for Army Research Laboratory, Contract No. DAKF11-99-P-1236, July 17, 2000.

Hamilton, M, Hackler, W.R.., Deeply Integrated Guidance Navigation Unit (DI-GNU) Common Software Architecture Principles (revised dec-29-04),

DAAAE30-02-D-1020 and DAAB07-98-D-H502/0180, Picatinny Arsenal, NJ .

Hamilton, M., Zeldin, S, "Higher Order Software—A Methodology for Defining Software," IEEE Transactions on Software Engineering, vol. SE-2, no. 1, Mar. 1976.

Hamilton, M., Zeldin, S., The Relationship Between Design and Verification", *The Journal of Systems and Software,* vol. 1, no. 1, pp. 20-56, Elsevier North Holland, Inc. 1979.

HOS, Application of a Formal Systems Methodology to Civil Defense, Prepared for Defense Civil Preparedness Agency, Wash., D.C. 20301, Mar. 1980.

HOS, USE.IT Reference Manual, Cambridge, MA, 1980-1985

HTI, Accident Record System Interim Solution (ARS-IS) Requirements Design Specification, Prepared for Massachusetts Highway Department Traffic Operations, Feb. 1997

HTI, 001 Tool Suite Reference Manual, Cambridge, MA, 1986-2007

http://icb.nasa.gov/001

http://www.htius.com, What Others Say

Keyes, J., Internet Management,. chapters 30-33, on 001-developed systems for the Internet, Auerbach, 2000a.

Keyes, J. Financial Services Information Systems, chapter 18, Systems that Build Themselves, Auerbach, 2000b.

Krut, Jr., B., "Integrating 001 Tool Support in the Feature-Oriented Domain Analysis Methodology" (CMU/SEI-93-TR-11, ESC-TR-93-188), Pittsburgh, SEI, Carnegie Mellon University, 1993.

Object Management Group (OMG), 2006, Systems Modeling Language version 1.0, Available from http://www.omgsysml.org.

Ouyang, M., Golay, M.W. 1995, An Integrated Formal Approach for Prototyping High Quality Software of Safety-Critical Systems, Massachusetts Institute of Technology, Cambridge, MA, Report No. MIT-ANP-TR-035

Schindler, Max, Computer Aided Software Design, John Wiley & Sons, 1990.

Software Productivity Consortium, (SPC), Object-Oriented Methods and Tools Survey, Herndon, VA.SPC-98022-MC, Version 02.00.02, December 1998.

Biography

Margaret H. Hamilton is the founder and CEO of HTI. She is responsible for the DBTF paradigm (along with its formal universal systems language, 001AXES. and its automation. 001. Hamilton was the founder and CEO of Higher Order Software, Inc. (HOS), where she was responsible for the first CASE product line, USE.IT (HOS 1980-1985), in the industry and the theory behind it (HOS). She was responsible for the Apollo (and Skylab) on board flight software effort while Director of the Software Engineering Division at Charles Stark Draper Laboratory. She created the mathematical theory based on Apollo that formed the earlier beginnings of DBTF.

William R. Hackler is the lead engineer for 001's development. He has been responsible for many 001AXES advanced language concepts and applications including a simulator, missile tracking simulation (which led to HTI's being nominated for SBA Subcontractor of the Year), asynchronous real time distributed applications for SDI, factory models and Internet related applications including an accident record system and a financial trading system. Hackler was Director of Advanced Concepts at HOS where he developed technologies based on the HOS theory. He was responsible for many applications including battle management and aerospace manufacturing, applying HOS and USE.IT.