Universal Systems Language: Lessons Learned from Apollo

Margaret H. Hamilton and William R. Hackler
Hamilton Technologies, Inc.

Based on a preventive, development-before-the-fact philosophy that does not allow errors in the first place, the Universal Systems Language has evolved over several decades, offering system engineers and software developers a language they can use to solve problems previously considered next to impossible to solve with traditional approaches.

An inordinate amount of money is spent in projects where system design and software development play a key role, huge portions of it wasted, and critical systems run the risk of failure, sometimes leading to a major catastrophe. This occurs in large part because of the "after the fact" paradigm on which the languages used to define systems are based.

The assumption made here is that system engineers and software developers can significantly reduce the well-known problems associated with doing business as usual by using a language based on a radically different approach, one that is preventive instead of curative. The Universal Systems Language is such a language.[1,2] Based on systems theory—to a great extent derived from lessons learned from the Apollo onboard flight software effort—USL has evolved over several decades and taken on multiple dimensions. Its purpose has been to solve problems considered next to impossible to solve with traditional approaches, at least in the foreseeable future.

According to users, USL eliminates any preconceived notions because it is a world unto itself—a completely new way to think about systems. Instead of object-oriented and model-driven systems, the designer thinks in terms of system-oriented objects (SOOs) and system-driven models. Much of what seems counter-intuitive with traditional approaches, which tend to be software centric, becomes intuitive with this systems-centric approach.

USL was created for designing systems with significantly increased reliability, higher productivity, and lower risk. We designed it with the following objectives in mind:

- reduce complexity and bring clarity into the thinking process;
- ensure correctness by inherent, universal, built-in language properties;
- ensure seamless integration from systems to software;
- develop unambiguous requirements, specifications, and design;
- ensure that there are no interface errors in a system design and its derivatives;
- maximize inherent reuse;
- ensure that every model captures real-time execution semantics (for example, asynchronous and distributed);
- establish automatic generation of much of design, reducing the need for designers' involvement in implementation details;
- establish automatic generation of 100 percent, fully production-ready code from system specifications for any kind or size of software application; and
- eliminate the need for a high percentage of testing without compromising reliability.

USL can address these objectives because of the universal systems theory that forms its foundations. The

Published by the IEEE Computer Society
0018-9162/08/$25.00 © 2008 IEEE

technology also takes roots from other sources—other real-world systems and formal linguistics, methods, and object technologies.

APOLLO BEGINNINGS

USL had as its origin our study of Apollo flight software development. Our primary questions were, "What could we do better for future systems?" and "What should we keep doing because we are doing it right?"[3-6] We analyzed almost every aspect of the flight software. Naturally, this study reaffirmed some earlier assumptions about systems and software, called into question others, and added new ones. Some of what was learned might seem obvious in today's world, but what might not be so obvious is what could be derived from these assumptions and later become part of USL's requirements.

Apollo was 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 "systems engineering." When there were no answers to be found, at times we just had to make it up, and we had to design things to work the first time. Many on the team were fearless 20-something-year-olds, and dedication and commitment were a given, but there was no time to be a beginner. Learning was by doing, and a dramatic event would often dictate change. Because software was a mystery, a black box, upper management gave us total freedom and trust. Mutual respect was across the board. We were the luckiest people in the world. There was no choice but to be pioneers. What would later become foundations for USL enabled the Apollo team to create the software for the trip to the moon.

USL had as its origin our study of Apollo flight software development.

Because system engineers threw requirements over the wall to software developers, engineers and developers necessarily became interchangeable, as did their life-cycle phases—suggesting that a system is a system, whether in the form of higher-level algorithms, software that implements the algorithms, or systems that execute them. From this perspective, system design issues became one and the same as software, reinforced by the fact that entire missions were tested by software simulations integrating hardware, software, the universe, and humanware (for example, astronauts).

Expect the unexpected

It quickly became clear that nothing and no one could be expected to be perfect. The team learned to plan accordingly. Take Apollo 11. Just before landing on the moon, onboard software discovered that the CPU was fast approaching overload and there would not be enough time to perform landing functions unless emergency steps were taken. With the software's global error detection and recovery mechanisms, nominal displays were interrupted with priority alarm displays. Every time the CPU approached overload, the software cleared out its entire queue of processes and restarted its functions, allowing only the highest priority processes to perform until the landing was completed.

The source of the error was later found to be the astronaut checklist document instructing the astronaut to place the rendezvous radar hardware switch in the wrong position, thus stealing valuable CPU time. The mechanisms the software used for this emergency were thought by many to have saved the Apollo 11 mission. Similarly, on Apollo 12, just prior to liftoff, lightning struck the spacecraft twice, each time causing a computer power failure. Again, the software restarted the mission functions in time for liftoff.

The flexibility required of these missions could not have been accomplished in real time without an asynchronous, multiprogramming operating system where higher priority processes interrupt lower priority processes. Assigning a unique priority to every function in the software was critical for ensuring that all events would take place in the correct order and at the right time—for example, turning the engine on or off or ensuring that the priority displays would interrupt normal mission sequences in an emergency.

To our surprise, changing from a synchronous OS used in unmanned missions to an asynchronous OS in manned missions supported asynchronous development of the flight software as well. In essence, the development process—a system in itself—inherited the same philosophy of "expect the unexpected" embodied in the system it developed. We also established that a system-wide "kill and start over again" recompute approach to error detection and recovery was far superior to a point-repair and "pick up from where you left off" approach.

Simplifying the software's operation simplified its development. Similarly, steps taken to create solutions within the multiprogramming environment later became solutions for multiprocessing environments. Although only one process is actively executing at a given time in a multiprogramming environment, other processes in the same system—sleeping or waiting—exist in parallel with the executing process. With this as a backdrop, we created the priority display mechanisms, essentially changing the man-machine interface between the astronauts and the onboard flight software from synchronous to asynchronous so that the mission could be reconfigured in real time.

Often, a problem that at first seemed impossible was eventually solved by changing its context. It seemed unthinkable to define and provide error detection and

recovery for every potential cause—for example, the two successive lightning strikes that shut down Apollo 12's computer systems prior to launch.

Our solution was to determine general ways in which hardware or software could be affected (for example, by a power outage triggered by one of many causes), reducing the problem to a small, finite number of predictable things to check for. Moreover, this approach provided new assurances that certain errors could be eliminated early in the life cycle—or even prevented—simply by adding rules used at definition time (for example, always assign a name directly to logic to be invoked, instead of referring to it relative to other logic; for example, refer to Sally instead of Fred + n). This eliminated the problem that would occur in the event that either logic's location relative to the other would later be changed and as a result the logic in question would be referred to incorrectly, possibly resulting in dire consequences. Although the need for this kind of rule is less likely today, its value at that time was the degree to which it suggested the importance of preventing errors "before the fact."

Of course, better can sometimes become the enemy of good. For example, lock mechanisms preventing human operators from entering an input error might also eliminate the possibility of fixing an unanticipated problem during a mission by going through the back door. On Apollo 14, for example, erroneous hardware signals were misleading the software, and it became necessary to manually intervene in real time to "fool" the software so that it would ignore the signals. The change, made at the eleventh hour by the developers working closely with the astronauts through Mission Control, would go against the software specification but would remain consistent with the original intent of the system requirements at large. After two attempts, the new change finally worked in simulations on the ground and was uploaded to the spacecraft, saving the mission with minutes to spare.

Clearly, we needed a way to "have our cake and eat it too"—built-in lock mechanisms that would not interfere in this kind of an emergency.

Fascination with errors

Because of the never-ending focus on making everything as perfect as possible, there was an ongoing fascination with errors: finding them, detecting and recovering from them, handling them, preventing them, learning from them, learning about systems from them—even defining what an "error" is (or isn't). We determined that we could not measure a system's reliability until we defined a formal, agreed-upon general concept of "error," along with all of its implications.[7]

We defined error in terms of system viewpoints (for example, requirements versus specification versus implementation), programs (lunar excursion module versus command module versus commonware); categories (system "glue" versus powered flight); weight (catastrophic versus FLTs, or "funny little things"); how to determine the source or cause of an error (for example, software versus hardware); kind of error (timing); and when an "error" is really an error or a "new feature" (or, for example, if two errors cancel each other, is there an error?). We developed a standard process for recording and relating to every error, including its history—for example, in what part of the life cycle it was created and found and, accordingly, what could be done to prevent it in the future.

Earlier ideas for a systems technology began to surface as we analyzed the kinds and causes of software problems found during verification and validation (V&V) testing of the Apollo onboard software. Because of Apollo's software design and development processes, at the outset we faced the likelihood of introducing almost any conceivable error—in hindsight, a blessing in disguise. This was due in part to size constraints in the hardware, which made it necessary for mission phases to share erasable memory. In addition, the flight software for each mission was developed concurrently with flight software for other missions, along with mission planning, hardware integration, simulators, and astronaut training—underscoring how much the software was part of a larger system. Finally, and most obviously, there were many unknowns given that we had not been to the moon before.

We developed a standard process for recording and relating to every error, including its history.

What was accomplished (or not accomplished) provided a wealth of information from which to learn. Each error was recorded when it was discovered in the software released for formal testing, and over time we began to appreciate how important accuracy is in filling out such forms. Through these efforts, we learned that interface errors (dataflow, priority, and timing errors from the highest to the lowest levels of a system to the finest grain) accounted for approximately 75 percent of all errors—for example, ambiguous relationships, integration mismatches and conflicts, communication and coordination problems—a clear indication that finding ways to reduce errors in this category was of the highest priority.

Although half of the billions of dollars (by today's standard) spent on the life cycle was devoted to simulation, 44 percent of the errors were found by manual means, referred to on the project as the Augekugal method (eyeballing) or "Nortonizing" (named after the person who perfected this technique). More automation was needed, especially static as opposed to dynamic analysis. Alarmingly, 60 percent of the errors found during V&V had unwittingly existed in previous flights—showing how

subtle they were—though, fortunately, no software errors surfaced during actual flights.

The interface errors were analyzed in greater detail first because they not only accounted for the majority of errors, they also were often the most subtle and most difficult to find. Each interface error was placed into a category identifying the means to prevent it by way of system definition. This process led to a set of axioms forming the basis for a new mathematical theory for designing systems that would, among other things, eliminate the entire class of interface errors just by the way a system is defined.[1,2,5,8]

Given the ongoing evaluation of the Apollo effort, it became clear that a new kind of language was needed and that our mathematical theory could provide its core. Results of the analysis took on many dimensions, not just for space missions but for applications in general, and not just for software but for systems in general—the results of which were not readily apparent for many years to come.

Lessons learned from this effort continue today: Systems are asynchronous, distributed, and event-driven in nature, and this should be reflected inherently in the language used to define them and the tools used to build them. This implies that a system's definition should characterize natural behavior in terms of real-time execution semantics, and designers should no longer need to explicitly define schedules of when events are to occur. Instead, events should occur when objects interact with other objects so that by defining such interactions the schedule of events is inherently defined. Most important, it became clear that the root problem with traditional approaches is that they support users in "fixing up wrong things" rather than in "doing things the right way in the first place." Combined with further research, as this became more widely understood, it became clear that the characteristics of good design could be reused by incorporating them into a language for defining systems.

UNIVERSAL SYSTEMS LANGUAGE

USL captures the lessons learned from Apollo. What sets USL apart is the systems paradigm on which it is based.[1] Whereas the traditional software development approach is curative, testing for errors late into the life cycle, USL's development-before-the-fact philosophy is preventive, not allowing errors in the first place. Correctness is accomplished by the very way a system is defined, by built-in language properties inherent in the grammar. A USL definition models both its application (for example, an avionics or banking system) and properties of control into its own life cycle. Each SOO definition has built-in constraints that support the designer and developer, yet they do not take away flexibility in fulfilling requirements. A SOO inherently integrates all aspects of a system (for example, function-, object-, and timing-oriented). Every system is an object, every object a system.

Mathematical approaches are known to be difficult to understand and are limited in their use for nontrivial systems as well as for much of the system's life cycle. Unlike formal languages that are not friendly or practical, and friendly or practical languages that are not formal, its users consider USL to be not only formal but also practical and friendly.[9-11] Unlike other mathematically based formal methods, USL extends traditional mathematics with a unique concept of control: universal real-world properties internal to its grammar—such as those related to time and space—are inherent, enabling USL to support the definition and realization of any kind or size of system. The formalism along with its unfriendliness is "hidden" by language mechanisms derived in terms of that formalism.

What sets USL apart is the systems paradigm on which it is based.

General systems theory

A formalism for representing the mathematics of systems, USL is based on a set of axioms of a general systems theory and formal rules for their application. All representations of a system are defined in terms of a function map (FMap) and a type map (TMap). Every SOO is defined in terms of a set of FMaps and TMaps. Three primitive structures, derived from the set of axioms, and nonprimitive structures derived ultimately in terms of the primitive structures specify 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.

Providing a mathematical framework within which objects, their interactions, and their relationships can be captured, USL—a metalanguage—has "metamechanisms" 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 USL is semantics-dependent but syntax-independent, yet every syntax shares the same semantics.

Implementation- and architecture-independent, USL adheres to the principle that everything is relative (one person's design is another's implementation); the same language can be used seamlessly throughout a system's life cycle to define and integrate all aspects of, and viewpoints about, the system and its evolution. The overarching principle is that all aspects of a USL universe are related to the real world and that the language itself inherently captures this relationship.

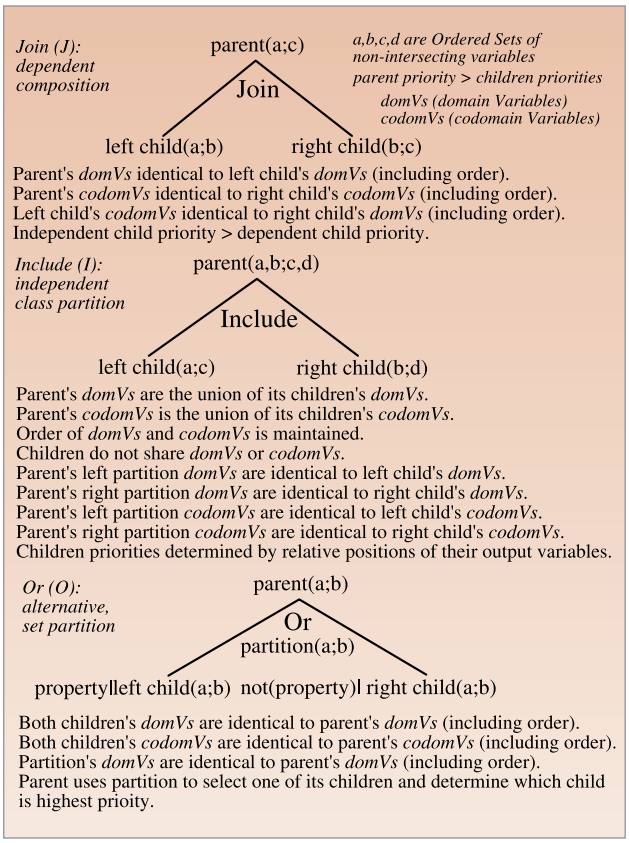

Figure 1. The three primitive control structures and their rules form a universal foundation for constructing maps in the domains of time and space as FMaps and TMaps.

Developers have used USL to define systems and develop software ranging from mission-critical systems[4,12] to commercial applications[13] to the development of system and software tools.[14,15] In so doing, it meets the challenge linguists describe as assuring consistency in meaning—fitting together the partially fixed semantic entities that we carry in our heads—to approximate the way reality fits together as it comes to us from moment to moment. The entities are the world (or perceptions of the world) reduced to its parts and secured in our minds.[16]

USL's philosophy is that 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 to form a new system. Designers can then use the new system, along with more primitive ones, to define (and build) more comprehensive reliable systems. If a system is reliable, all the objects in all its levels and layers are reliable.

Six axioms of control

We must visualize a system definition both by what it does (level by level, for example, a parent node in a hierarchy is on a higher level than its children nodes) and how it does it (layer by layer, for example, a specification is on a higher layer than its implementation). However, a hierarchical definition runs the risk of not being reliable unless there are explicit rules that ensure each decomposition is valid; for example, the behavior of a successive lower level (or layer) completely replaces the behavior of that which it replaces. A SOO can be defined from its most general state to its most detailed states. Objects, related properly, can replace other objects. An object is decomposed until the primitive objects by which it has been defined have been reached.

At the base of every USL system is a set of six axioms—universally recognized truths—and the assumption of a universal set of objects.[2,5,8] The axioms provide the formal foundation for a USL "hierarchy"—referred to as a map, which is a tree of control that spans networks of relations between objects. Explicit rules for defining a map have been derived from the axioms, where—among other things—structure, behavior, and their integration are captured.

Resident at every node on a map is the same kind of object (for example, a function on every node of an FMap and a type on a TMap). The object at each node plays multiple roles; for example, the object can serve as a parent (in control of its children) or a child (being controlled by its parent). Whereas each function on an FMap has a *mapping* from its input to output (domain to codomain), each type on a TMap has a *relation* between its domain and codomain.

Each axiom defines a relation of immediate domination of a parent over its children. The union of these relations is control. Among other things, the axioms establish the relationships of an object for invocation in time and space, input and output (domain and codomain), input access rights and output access rights (domain access rights and codomain access rights), error detection and recovery, and ordering during its developmental and operational states. Every system can ultimately be defined in terms of three primitive control structures, each of which is derived from the six axioms—resulting in a universal semantics for defining systems.

Universal primitive control structures

A structure relates each parent and its children according to the set of rules derived from the axioms of control. A primitive structure provides a relationship of the most primitive form (finest grain) of control. All maps are defined ultimately in terms of the primitive structures and therefore abide by the rules associated with each structure: A parent controls its children to have a dependent (Join), independent (Include), or decision-making relationship (Or).

Figure 1 shows the rules used in defining each of the three primitive structures, using a syntax that FMaps and TMaps can share. Because it is defined in terms of these structures, every SOO has control properties, inherently providing seamless integration, maximizing

its own reliability and flexibility to change, capitalizing on its own parallelism, and maximizing the potential for its own reuse and automation. The structures ensure that all interface errors—approximately 75 to 90 percent of all errors normally found during testing in a traditional development—are eliminated at the definition phase.

Although SOOs have properties for systems in general, the properties have special significance for the real-time, distributed behavior of systems: Each system is event-interrupt-driven; each object state is traceable (putting into good use the property of single reference, single assignment), 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 are most beneficial.

Any system can be defined completely using only primitive structures, but less primitive structures defined by and derived from the primitive structures—and therefore governed by the control axioms—accelerate the definition and understanding of a system. The defined structure, a powerful form of template-like reuse, provides a mechanism to define a map without explicitly defining some of its elements. An FMap structure has placeholders for variable functions; a TMap structure has placeholders for variable types; a universal structure has placeholders for variable functions or types. Async is an example of a real-time, distributed, communicating FMap structure with both asynchronous and synchronous behavior.

USL meets the challenge linguists describe as assuring consistency in meaning to approximate the way reality fits together as it comes to us from moment to moment.

An example of a TMap structure is TreeOf, a collection of the same type of objects ordered using a tree indexing system. Each TMap structure assumes its own set of possible relations for its parent and children types. Abstract types decomposed with the same TMap structure inherit 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 TMap's types). As researchers gain experience with new and different types of applications, new reusable structures emerge.

Definition and execution space

We define all functions in a system and their relationships with a set of FMaps. Similarly, we define all types in a system and their relationships with a set of TMaps. FMaps represent the dynamic (doing) world of action by capturing functional and temporal (including priority) characteristics. TMaps represent the static (being) world of objects by capturing spatial characteristics—for example, containment of one object by another or relationships between locations of objects in space.

FMaps define, integrate, and control the transition of objects from one state to another. TMaps define, integrate, and control the potential atemporal relations between states of objects. Each function in an FMap, defined using types of objects in a TMap, has one or more objects as its input and one or more objects as its output. Each object is a member of a type in a TMap and resides in an Object Map (OMap), an instance of a TMap. Each type on a TMap owns a set of inherited primitive operations used as primitive functions by the FMaps using that TMap's objects.

FMaps and TMaps depend on and reuse each other. The primitive operations that belong to types on a TMap used by FMaps within the same layer are themselves defined with FMaps on the TMap's implementation layer and therefore rely on another layer's TMaps. Thus, because an FMap depends on TMaps, it depends on another layer's FMaps; similarly, because a TMap depends on another layer's FMaps, it too depends on another layer's TMaps. Functions depend on types, types on functions. In other words, FMaps and TMaps recursively reuse each other, layer by layer.

Every change to a SOO is traceable throughout the system. The FMaps for a given system are inherently integrated with the TMaps by using their objects and primitive operations, providing the ability to automatically trace within and between a system's levels and layers. If, for example, a type is changed on a TMap, all impacted FMap areas are traceable. In an FMap, an output variable of any function is fully traceable to all other functions using the state that variable refers to.

An FMap is completed when all its leaf nodes (or leaf nodes of the FMaps it uses) are recursive leaf nodes or are primitive function leaf nodes that use primitive operations of types in the TMaps. A recursive leaf node definition has the name and functionality of one of its parent's ancestor definition nodes; see, for example, Jset in Figure 2. The recursive reuse pattern has an Or primitive structure decision node between each of its recursive leaf nodes (each with some input different than the ancestor's input) and the ancestor node. For a recursion to always be able to terminate, at least one of the Or structure's alternatives cannot be (or have a descendent that is) a recursive leaf of the ancestor. A recursive leaf is instantiated using its ancestor definition node, and its execution control is expanded as an acyclic graph of actions (or objects). A set of primitive types and their associated primitive operations provides a mechanism for layered reuse by different domains of applications. Application domains are separated into layers of reuse in which the primitive types of one layer are implemented in terms of reusable FMaps and TMaps of one or more lower-level layers of detail.

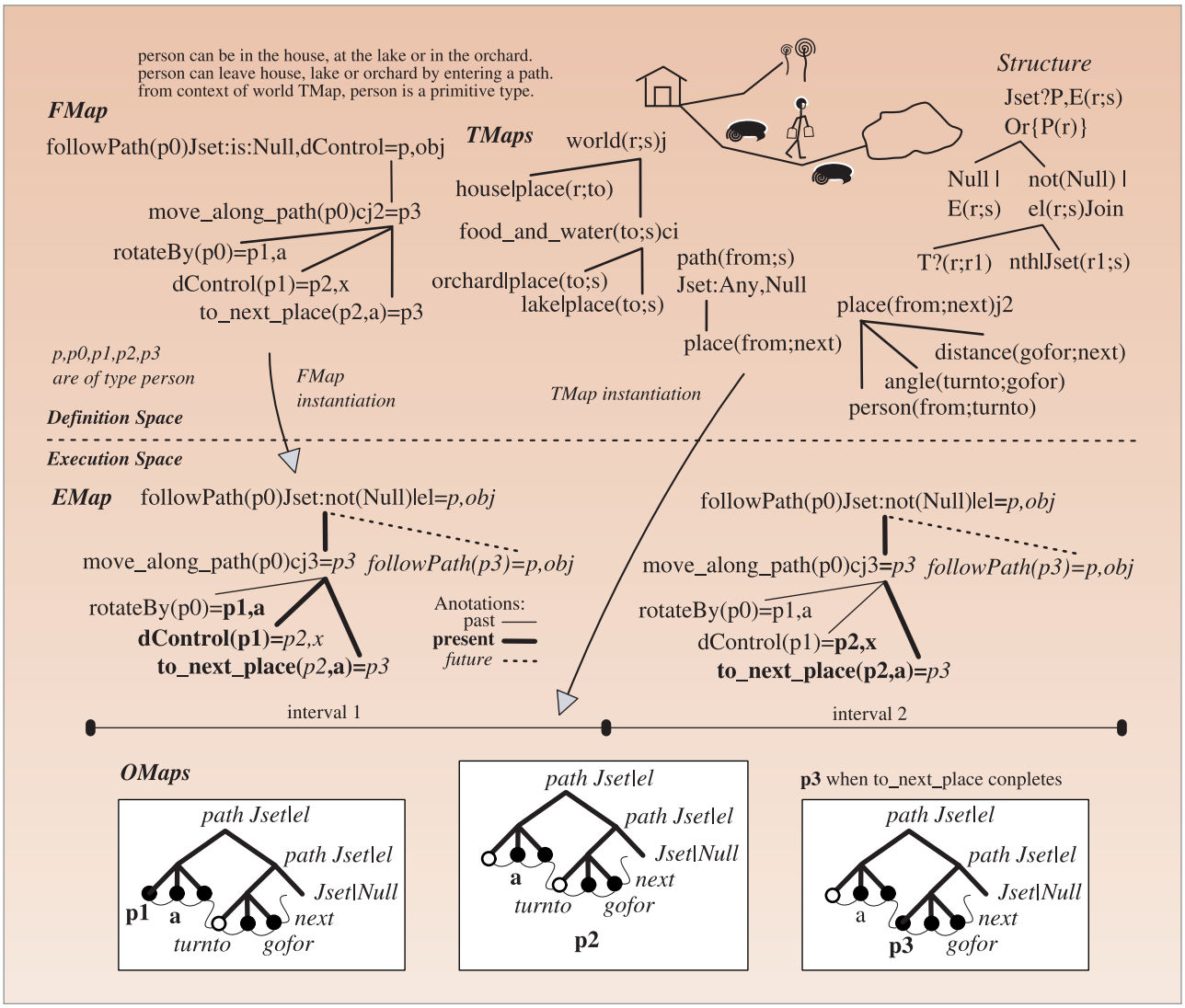

Figure 2. The definition and execution space of a SOO shows an FMap and its TMaps and their instantiation in terms of an EMap and its OMaps over time and space.

Each layer of TMaps and FMaps becomes itself a reusable system to the layer immediately above it, which itself is a system layer. Another special category of reuse is the ability for an OMap to be persistently stored to disk—providing long-term memory—or marshaled to a socket, providing generic transmission of objects between processes. Because everything a designer needs to know (no more, no less) is in a SOO definition, all model viewpoints can be determined from that definition—for example, in terms of projections. Inherent within each map are features such as polymorphism, encapsulation, and inheritance that reside on both the function side and type side of a system.

A SOO is realized—that is, has all of its values instantiated for a particular performance pass—in terms of an Execution Map (EMap) of actions, an instantiation of an FMap, and its OMaps. Figure 2 depicts a SOO's definition and execution space.[5] It shows a person's house and a path that he can take from the house to get food and water. In this figure, an alternative syntax is used to define FMaps: "function(domain)structure=codomain" instead of "function(domain;codomain)structure."

Part of the FMaps and TMaps of this system are shown here; the EMap and OMaps represent the execution of the FMap and TMaps. Since TMaps are integrated with FMaps, the OMaps are integrated with the EMap. Annotations on the EMap show the functions presently being executed as actions, actions that have occurred in the past, and actions that will occur in the future over two progressive intervals of time. The labels on the OMaps show corresponding variable object states, relation instances between object nodes, and an overall structure inherited from the Jset universal structure map. A Jset is a recursively defined set of dependent elements. It can be interpreted as an FMap-defined structure or a TMap-defined structure. In the FMap,

r1 becomes object state, p3; in the TMap, r1 becomes relational state, "next." OMap, path, is defined by Jset as a set of places each having a "next" relation between them. When Jset is used to define the followPath function, it results in a recursive sequence of move_along_path actions in the EMap that go from one place on the path to another using the "next" primitive action that uses the "next" relation on the OMap. Other defined structures in the figure are also used in the FMap and TMaps (for example, ci, cc, and cj).

When an object state event occurs, each function that depends on that object state is instantiated. This instantiation process results in a totally ordered (in terms of priority) map of function instances; when a function instance becomes ready to execute, it inherently is always correctly scheduled and allocated to the appropriate resource(s).

TMaps provide universal primitive operations for controlling objects and object states (for example, type Any) that are inherited by all types of objects. They offer a means to create, destroy, copy, reference, locate, 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 different types of objects. With the universal primitive operations, building systems can be accomplished uniformly. TMap and OMap, themselves, are available as types to facilitate a system's ability to understand itself better and manipulate all of its objects the same way, when it is beneficial. TMaps ensure proper use of objects in an FMap (for example, objects cannot exist in the same place at the same time; it is not possible to put a leg on a table where a leg already exists; conversely, it is not possible to remove a leg when there is no leg). A reference to an object's state cannot be modified if other references are being (or could be) made to that state; reject values exist in all types, signifying error conditions.

A system can adapt to changing resource requirements if the functional architecture definition is separated from its resource definitions. To support such flexibility with the necessary built-in controls, USL itself is used to define functional, resource, and allocation architectures. It can be used to define global and local constraints for both FMaps and TMaps, with the constraints themselves defined in terms of FMaps and TMaps. If we place a constraint on the definition of an operation (for example, where sendBy:vehicle takes between 2 and 3 hours), this constraint influences all functions that use this definition. Such a constraint can be overridden by a constraint placed on a function in a local context that uses this original definition—where sendBy:car takes between 4 and 6 hours, for example, overriding the default.

Maps guide a designer in thinking through concepts at all levels and layers of system design and the 001 Tool Suite (001), USL's automation,[14,15] in automating the life cycle. Typically, designers begin to define a system by sketching TMaps, where they decide on the types of objects (and their relationships) in the system. Often, a RoadMap (RMap) that organizes all system objects, including FMaps, TMaps, EMaps, OMaps, defined structures, and other RMaps is "sketched" in parallel with the TMap. At each node of an RMap, a reference is made to another map.

Once a TMap has been agreed upon, the FMaps begin to fall into place because of the natural functionality (or groups of functionality) in the TMap system. The TMap provides the structural criteria from which to evaluate the functional partitioning of the system—for example, the shape of the combined patterns of the structural organization of the FMaps is derived from the structural organization of potential objects defined by the TMap. With FMaps and TMaps, a system (and its viewpoints) is divided into components and groups of components that naturally work together.

Correct use of USL eliminates the majority of errors, including all interface errors within a system model and its derivatives.

Automated environment

How can we build a more reliable system and at the same time increase our productivity in building it? Take for example, testing.

Correct use of USL eliminates the majority of errors, including all interface errors within a system model and its derivatives. Our 001 analyzer statically hunts down all errors resulting from the incorrect use of USL. When a TMap is changed, 001 demotes the status of all FMaps impacted by that change; the FMaps are then reanalyzed in light of the TMap changes to reestablish their status. Testing for integration errors is minimized because of the inherent integration of SOOs. A SOO model can be realized by directly interpreting it on an operational runtime environment or by mapping it onto a targeted virtual or real machine architecture and environment. For a virtual machine, simulations of models can be used to quantify characteristics and qualify tradeoffs of the system to be developed.

Given a set of FMaps and TMaps, 001 can generate much of the design and all of the RMaps, perform requirements analysis, and simulate and observe a system's behavior as it is being executed in terms of EMaps and OMaps. For software, 001 can use the same FMaps and TMaps to automatically generate all of the code including its documentation.

001's requirements analysis component automates the process of going from requirements to design to code and back again. Because it has an open architecture, 001's

generator can be configured to generate one of a possible set of implementations for an architecture of choice (or to interface with any outside environment—for example, communications package, Internet interface, database, graphics, operating system, a language, or the users' own legacy code).

Maintenance shares the same benefits. The developer doesn't ever need to change the code, since application changes are made to the specification—not to the code—and target architecture changes are made to the configuration for the generator environment and not the code. Only the changed part of the system is regenerated and integrated with the rest of the application—again, the system is automatically analyzed, generated, compiled, linked, and executed without manual intervention. Just as with the systems it is used to develop, 001 is completely defined with itself, using USL, and is completely and automatically generated with itself. It therefore has the same before-the-fact properties that all USL systems have.

USL as a formal foundation for other languages

Diverse mappings (several automated) exist that go from a given syntax and semantics to USL or from USL to one of a possible set of syntactical forms (and semantics).[9,17,18] The USL team recently performed an analysis of how the USL formal semantics could provide SysML/UML2 with a universal system formalism that can reduce semantic ambiguity in the OMG SysML specification[19-21] and significantly simplify the UML2 specification standard.

Most of today's systems are defined with languages originally intended for software. These systems are built using a programming or specification language created specifically for a computer—a syntax-first, syntax-dependent approach. USL, based on a formal systems theory derived from real-world systems—a semantics-first, syntax-independent approach—was originally created for defining systems in general, where the goal was to combine mathematical perfection with engineering precision.

Unlike languages where language mechanisms, rules, and tools are added ad hoc and after the fact as more is learned about a class of systems, USL derives its language mechanisms and tools from its core set of primitive mechanisms. Because of this flexibility, USL can be used as it gracefully evolves and it also can lend its formal support to other languages. By inheriting its preventive philosophy, the potential exists to "solve" (prevent) a given problem as early in the life cycle as possible.

Automatic static analysis of the specification model is more preventive than static analysis of after-the-fact code. Preventing a problem by the way a system is defined is even better. The goal is to apply this philosophy across the board, including systems and software, unifying their understanding by a formal means with a commonly held set of system semantics.

We inadvertently discovered during the Apollo error study that there was a universal way to prevent errors by the way a system is defined, addressing the issue of reliability head on. While searching for mechanisms to define error-free systems, we unexpectedly found patterns with properties that addressed other issues as well. Among other things, these patterns always present in FMaps and TMaps inherently support asynchronous and distributed behavior within all objects.

Whereas on Apollo it was necessary to manually program the scheduling of the processes and the assignment of priorities to each function to capitalize on the asynchronous operating system, FMaps and TMaps inherently make this happen from the beginning, starting in the models themselves.[4] On Apollo, some tailored lock mechanisms appeared to be at cross purposes with other concerns. What would have been solved with locks then can now take place implicitly and generically through the mechanisms in FMaps and TMaps. What was previously a manual life cycle process can now be automated. Further, because USL maximizes inherent reuse, the larger and more complex a system, the higher the productivity. Unlike before, it is now possible to increase a software system's reliability and at the same time increase the productivity in its development.[22,23]

Used in research and development, the next step is to bring USL into a larger community of users. Analysis of lessons learned from each of its evolving states continues in a manner not unlike the empirical Apollo studies—build on what has been beneficial and eliminate the rest. Each time we learn from experience, we evolve accordingly, maximizing the degree of preventiveness.

Although the Apollo software was developed long ago, we continue to reflect on its lessons. It is the hope that its legacy will continue. The goal is that the systems of today inherit the best of yesterday, and systems of tomorrow inherit the best of today. ■

References

1. M. Hamilton, "Inside Development Before the Fact," cover story, editorial supplement, 8ES-24ES, *Electronic Design*, Apr. 1994
2. M. Hamilton and W.R. Hackler, "Universal Systems Language for Preventative Systems Engineering," *Proc. 5th Ann. Conf. Systems Eng. Res.* (CSER), Stevens Institute of Technology, Mar. 2007, paper #36.
3. M. Hamilton, "Zero-Defect Software: The Elusive Goal," *IEEE Spectrum*, Mar. 1986, pp. 48-53.
4. M. Hamilton, "The Heart and Soul of Apollo: Doing It Right the First Time," *Proc. 7th Int'l Military and Aerospace Pro-*

grammable Logic Devices (MAPLD) Conf., NASA Office of Logic Design, paper S216, 2004.
5. M. Hamilton and W.R. Hackler, "Reducing Complexity: It Takes a Language," *Innovations in Systems and Software Eng. J.*, NASA, to be published by Springer Verlag, 2009.
6. M. Hamilton, *Shuttle Management Memo #14*, Charles Stark Draper Laboratory, Cambridge, Mass., 1972.
7. M. Hamilton, *What Is an Error?*, tech. note, HTI, Cambridge, Mass., 1991.
8. M. Hamilton and S. Zeldin, "Higher Order Software—A Methodology for Defining Software," *IEEE Trans. Software Eng.*, vol. SE-2, no. 1, Mar. 1976, pp. 9-32.
9. B. Krut Jr., *Integrating 001 Tool Support in the Feature-Oriented Domain Analysis Methodology*, tech. report CMU/SEI-93-TR-11, ESC-TR-93-188, SEI, Carnegie Mellon Univ., 1993.
10. M. Ouyang and M.W. Golay, *An Integrated Formal Approach for Prototyping High-Quality Software of Safety-Critical Systems*, tech. report MIT-ANP-TR-035, MIT, 1995.
11. Software Productivity Consortium, "Object-Oriented Methods and Tools Survey," SPC-98022-MC, v.02.00.02, Dec. 1998.
12. M. Hamilton and W.R. Hackler, "Deeply Integrated Guidance Navigation Unit (DI-GNU) Common Software Architecture Principles," DAAAE30-02-D-1020 and DAAB07-98-D-H502/0180, Picatinny Arsenal, NJ, 2003-2004.
13. J. Keyes, *Internet Management*, chapt. 30-33 on 001-Developed Systems for the Internet, pp. 391-511, Auerbach, 2000.
14. HTI, 001 Tool Suite (1986-2008); www.htius.com and http://icb.nasa.gov/001 or htius.com/001_nasa.
15. M. Hamilton, "Development Before the Fact in Action," cover story, special editorial supplement, 22ES-30ES, *Electronic Design*, June 1994.
16. D. Bolinger and D.A. Sears, *Aspects of Language*, Harcourt Brace Jovanovich, 1981, p. 109.
17. M. Hamilton and W.R. Hackler, "Towards Cost Effective and Timely End-to-End Testing," HTI, prepared for Army Res. Lab., contract no. DAKF11-99-P-1236, 17 July 2000.
18. S. Cushing, "A Note on Arrows and Control Structures: Category Theory and HOS, Candidate BMD Data and Axioms," contract no. DASG60-77-C-0155, HOS, prepared for Ballistic Missile Defense, Advanced Technology Center, June 1978.
19. S. Friedenthal, A. Moore, and A. Steiner, "OMG Systems Modeling Language (OMG SysML) Tutorial," *Proc. 16th Ann. Int'l Symp. INCOSE 2006*, INCOSE, 2006.
20. M. Hamilton and W.R. Hackler, "A Formal Universal Systems Semantics for SysML," *Proc. 17th Ann. Int'l Symp. INCOSE 2007*, INCOSE, 2007, paper #8.3.2.
21. Object Management Group, "Systems Modeling Language," v. 1.0, 2006; www.omgsysml.org.
22. Department of Defense, "National Test Bed Software Engineering Tools Experiment—Final Report," vol. 1, Experiment Summary, Table 1, p. 9. Dept. of Defense Strategic Defense Initiative Organization, Washington, D.C., Oct. 1992.
23. M. Schindler, *Computer-Aided Software Design*, John Wiley & Sons, 1990.

Margaret H. Hamilton *is the founder and CEO of Hamilton Technologies, Inc. She was responsible for the Apollo and Skylab onboard flight software effort while Director of the Software Engineering Division at MIT's Charles Stark Draper Laboratory. Her research interests are preventive systems (languages, methods, and tools), operating systems/man-machine interfaces, and error detection and recovery (including secure systems). She received an AB in mathematics and philosophy from Earlham College. Contact her at mhh@htius.com.*

William R. Hackler *is the lead engineer for 001's development at Hamilton Technologies, Inc. His research interests are neuroscience, metamorphic systems (robotics and hardware), and synergetics (Buckminster Fuller)/tensegrity systems. He studied mathematics and logic at Mesa College and 12-tone music composition with composer Mertin Brown. Contact him at ron@htius.com.*