A Formal Universal Systems Semantics for SysML

Margaret H. Hamilton and William R. Hackler
Hamilton Technologies, Inc.
17 Inman Street
Cambridge, MA 02139

Copyright © 2007 by Margaret H. Hamilton and William R. Hackler. Published and used by INCOSE with permission.

Abstract. OMG SysML™ is a general purpose systems modeling language adopted by OMG in May, 2006. Used for specifying, analyzing, designing, and verifying complex systems; it provides graphical representations with a semantic foundation for modeling system requirements, behavior, structure, and integration with a broad range of engineering analysis. SysML represents a subset of UML2 with extensions needed to satisfy the requirements of UML for systems engineering. The goal is to enhance systems quality, improve the ability to exchange information among tools, and help bridge the semantic gap between systems, software, and other engineering disciplines (Friedenthal et al. 2006). *This paper provides an analysis of how SysML may be further enhanced by a more formal framework that uses the semantics, based on the axioms of a general systems theory, of the Universal Systems Language™ (USL™), 001AXES™ (Hamilton April 1994). At the same time SysML provides 001AXES with a standardized based approach for capturing this formalism.*

001AXES has had a focus on reliable systems since its inception. 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, that tend to be software centric, becomes intuitive with this approach, which is system centric. 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 then be used as a means to an end—designing and building better systems.

001AXES is used today to address problems considered difficult to solve with traditional approaches (Hamilton and Hackler 1991, 2003-2004); it can be used to address these problems for SysML users as well. Its preventative paradigm and how the 001AXES kernel can provide SysML with a formal foundation will be discussed. Examples show mappings between SysML and 001AXES and how the kernel can be used to support SysML.

Introduction

The assumption is made, here, that the goal of SysML, that includes enhancing systems quality, improving the integration of tools, and bridging the semantic gap between systems and software, could be addressed with a more formal framework. And, further, that the formal semantics of the 001AXES universal systems language could be used in this regard to support SysML. In order to clarify our analysis and findings regarding how this approach would work, we first describe 001AXES, its formal foundations and aspects of 001AXES that would be of most benefit in further enhancing SysML.

001AXES was created to provide a language formalism 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 with 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.

The use of the semantics of the 001AXES language and its associated automation (Hamilton June 1994) based on a technology, in large part derived and evolved from lessons learned from the Apollo on-board flight software effort, is intended to address these objectives. The technology also takes roots from other real world systems, systems theory, formal methods, formal linguistics and object technologies—concepts older (e.g., mathematics) and newer than Apollo; keeping in mind the relevance of a technology is independent of its age.

During Apollo the question was asked; "what should we do better for future systems and what should we keep doing because we are doing it right (Hamilton 1986, 2004)"? The search was for a means to build ultra-reliable systems. Earlier ideas for the technology surfaced as the kinds and causes of problems 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 approximately 75% of all errors found—finding ways to improve the integrity of integration and communication was of the highest priority. And although half of the billions of dollars (by today's standard) spent on the life cycle was devoted to simulation, 44% of the errors were found by manual means (eyeballing)—more automation was needed, especially static analysis 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.

The interface errors were analyzed in more detail first. Each error was placed into a category according to the means that could be taken to prevent it by the way a system was 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 to obtain quality is to not to allow the errors in, in the first place. Correctness is accomplished by the very way 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 it also models 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 developed 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. Instead of object oriented systems, one thinks in terms of system oriented objects; instead of model driven systems, one thinks in terms of system driven models.

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 method 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), DBTF 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 and realization of any kind or size of system.

A formalism for representing the mathematics of systems, 001AXES is based on a set of axioms 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™). The 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 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.

001AXES has been used to define systems ranging from guidance, navigation and control (e.g., (Hamilton and Hackler 1988, 2003-2004), (HTI 1989) (Hamilton 2004)) to commercial applications (e.g., (HTI 1997), (HOS 1980), (Keyes 2000a, 2000b)) to system and software tools, e.g, (HTI 1986-2007). 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). The 001AXES team has recently been involved in analyzing how SysML can support the formal semantics of 001AXES, and identifying what SysML extensions may be required to inherit certain 001AXES properties and its associated automation. The SysML mapping effort will be discussed below. In the following sections the formal semantics of 001AXES will be discussed, followed by a description of a high level mapping between SysML and 001AXES; followed by a

discussion of findings that can be used to identify potential SysML extensions to leverage 001AXES semantics and enhance SysML's formal underpinnings.

Integrated Modeling Environment

001AXES—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).

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 [or perceptions of 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."[1]

001AXES provides a mathematical framework within which objects and their interactions and relationships with other objects may be captured. Its philosophy: 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.

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; e.g., the behavior of a successive lower level (or layer) completely replaces the behavior of that it replaces. 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 a given map is the same kind of object (e.g., a function on every node of an FMap; a type on a TMap). 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.

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

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

(Hamilton and Zeldin 1976, 1979). Each axiom defines a relation of immediate domination of a parent object over its children. The union of these relations is control. Among other things, 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.

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 childrens' 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 function from a lower level is removed and its ancestor still maintains its same mapping, the function at the lower level 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 (ability to 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 parent's own 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 output variables at the parent are a subset of the output variables of the collective children.

Axiom 4 states that a given parent controls the input access rights (ability to obtain the values of variables) to each set of variables whose values define the elements of the input space for each

immediate, and only each immediate lower level child. Axiom 4 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.

Some 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 the variables 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). Axiom 5 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, and, 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, and 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; a process cannot interrupt 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 is an event; 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 Control 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 and FMap definitions. Figure 2.a contains a description of the rules as they apply to an FMap (read left-to-right in this syntactical view). Figure 2.b contains a right-to-left syntactical view. 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 the 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.

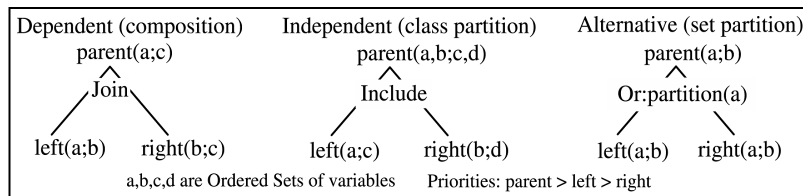

Figure 1: Universal Primitive Control Structures

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 (Hamilton June 1994, HTI 1986-2007), in automatically generating detailed designs (such as for resource allocation) and the software part of the system as required. With a map, everything you need to know (no more, no less) is available. All model viewpoints can be obtained from FMaps and TMaps, including structure (organization of components and their connectivity), behavior (object flow, control flow, state transition, timing), parametrics (constraints), allocations and structures of types and functions. Inherent within each map are features such as polymorphism, encapsulation and inheritance that reside both on the function side as well as the type side of a system; 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. For example, in an FMap, an output variable of any function is fully traceable to all other functions that use the state that variable refers to.

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 its allowed FMap primitive 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. If for example 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 transitions of objects from one state to another state.

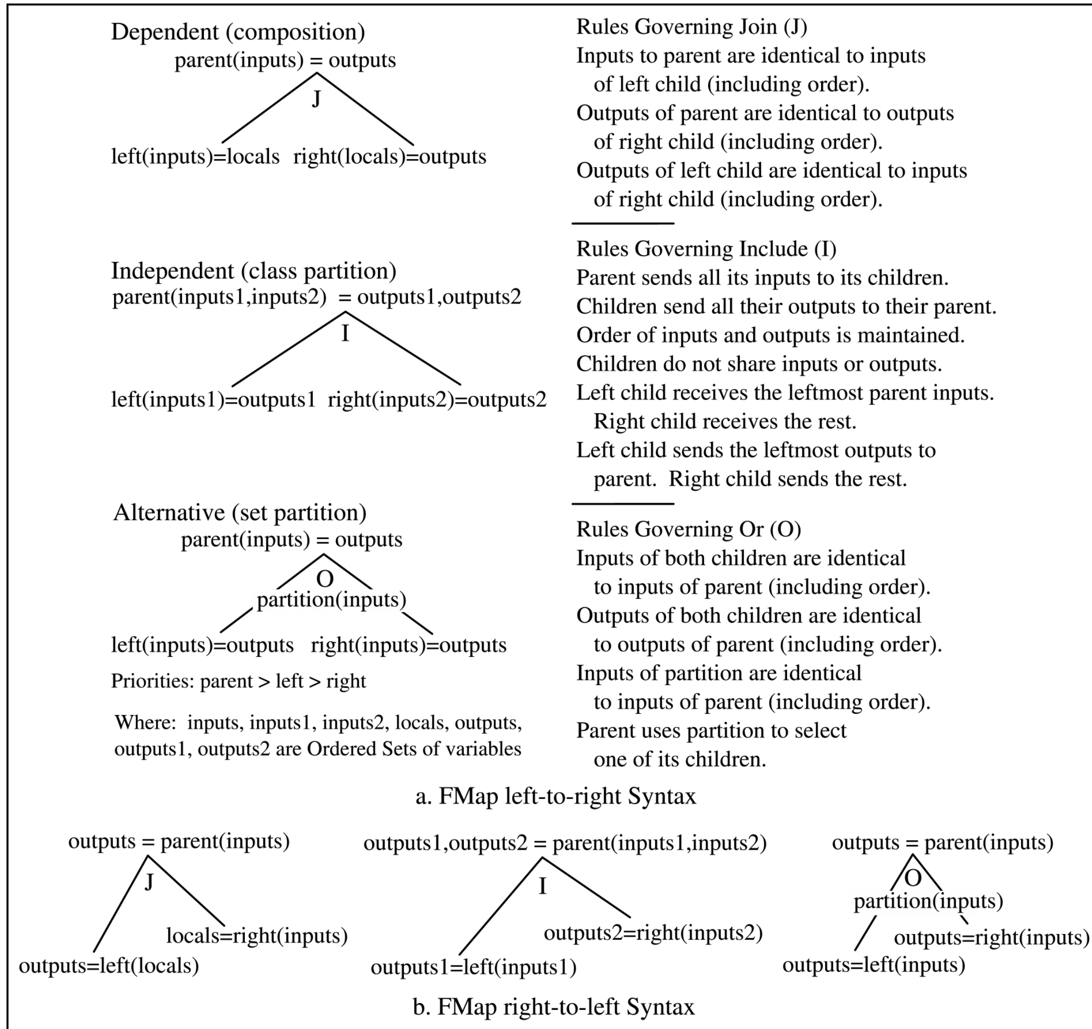

Figure 2: Rules for Primitive Control Structures

A SOO is realized (has all of its values instantiated for a particular performance pass) in terms of *execution maps*™ (EMaps™), each of which is an instantiation of an FMap and OMaps, each of which is an instantiation of a TMap. When an object state 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). 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 to design a system by sketching a TMap; 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) 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 can be derived from the primitive ones; and accelerate the process of defining and understanding a system. Since all non-primitive structures are ultimately derived from the three primitives, they are also governed by the control axioms. Defined structures for both FMaps and TMaps can be created for any kind of system including real time, distributed systems; retrieval and query structures can be defined for more database oriented systems.

The defined structure, a powerful form of template-like reuse, provides a mechanism to define a map without some of its elements being explicitly defined. Whereas an FMap structure has placeholders for variable functions, a TMap structure has placeholders for variable types. Async is an example of a real time, distributed, communicating FMap defined structure with both asynchronous and synchronous behavior. An example of a TMap defined structure is TreeOf, a collection of the same type of objects ordered using a tree indexing system. Each type structure assume its own set of possible 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.

Universal Primitive Operations

The TMap provides universal primitive operations, used for controlling objects and object states, that are inherited by all types (a primitive operation is used as a primitive function(s) in an FMap). 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 also available as types to facilitate the ability of a system to understand itself better and manipulate all objects the same way when it is beneficial to do so. 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). One cannot, for example, 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 they are 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 such as those used to manage a development environment.

Constraints

When designing a system, it is important to understand the performance constraints of the 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 support such flexibility with 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 terms of FMaps and TMaps. If we place a constraint on the definition of a function (e.g., Where sendBy:vehicle takes between 2 and 3 hours), this constraint influences all functions that use this definition. Such a constraint is global with respect to all the functions that use the original function definition.

Global constraints may be further constrained or overridden by local constraints placed on a function that uses this original definition (e.g., where function sendBy:car takes between 4 and 6 hours, overriding the default). The validity of constraints and their interaction with other constraints can be analyzed by static or dynamic means with the automation of 001AXES, the 001 Tool Suite. 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 both the direct and indirect causal effects of functional interactions of constraints.

Automation

Because of a SOO's inherent support of automation; more automation is possible (e.g., much of the system design can be automatically generated from SOOs). Given a set of FMaps and TMaps, 001 has the means to perform requirements analysis; and simulate and observe the behavior of a system as it is being evolved and executed 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 DBTF 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 001 Analyzer 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 of an application 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 001 DXecutor™ component of 001 is a distributed runtime execution engine. 001 DXecutors form a hierarchy, each managing its own resources (e.g., different CPUs) and communicating (e.g., using TCPIP) to other 001 DXecutors. 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 001 DXecutor, that provides for a distributed active object like substrate).

Take also reuse. The more a paradigm supports 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; such reuse can result in many parts of the design and development process to become no longer necessary. Every object is a candidate reusable—and integratable—within the same system, other systems and these systems as they evolve.

Mapping 001AXES to SysML: 001AXES Semantic Kernel for SysML

The 001AXES kernel (001AXES formal semantics) could provide SysML with a universal system formalism that can reduce semantic ambiguity in the OMG SysML specification (OMG 2006). The following section, that provides an initial mapping of 001AXES to SysML structure, behavior, parametrics/constraints, and allocations, discusses the feasibility of such an approach. The block definition diagram (bdd) and associated internal block diagram (ibd) along with other SysML constructs provide a foundation for the mapping from the SysML perspective. Instead of using the default syntax(s) for 001AXES (as exemplified in Figure 1), a 001AXES kernel block diagram (kbd) syntax will be used that is a syntactic integration of the SysML bdd and ibd. Figure 3 contains a description of the three primitive control structures using the kbd syntax. This syntax can be used to define both the structure and behavior of any system, because it is a syntax variation of (and isomorphic to) the 001AXES map syntax that has the underlying 001AXES kernel semantics.

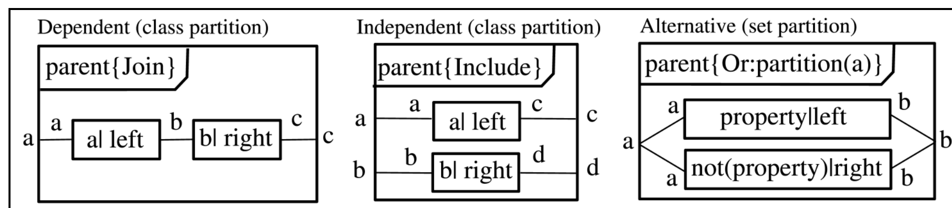

Figure 3: kbd Primitive Control Structure Syntax

The kbd syntax will be used in the following examples to represent SysML-like syntax for the following system viewpoints: structure, behavior (e.g., activities), constraints (e.g., parametrics) and layered architectures (e.g., allocations). Use of the kbd among other things will provide SysML specifications with the characteristics of:

- being executable, and therefore translatable (e.g., to software) or interpretable (e.g., used as an embeddable real time operating control system)

- having built in causality and total ordering of timing in terms of priority
- having built in schedulability in terms of active distributed resources
- automatable design based on implications of active resource allocations (e.g., fully automated distributed communications)
- automatable systems analysis based on optional resource architecture allocations (e.g., based on definable engineering characteristics such as: cost, risk, time, energy, reliability)

Structure: Mapping TMaps to SysML Structure Diagrams

A system has both static and dynamic structure. In 001AXES, the same structure of a system can be interpreted statically in the context of a TMap or dynamically in the context of an FMap (Hamilton and Hackler 2007). In SysML, block definition diagrams and internal block diagrams are used to define the static aspects of a system (see Figure 4) and activity diagrams may be used to define the dynamic aspects of a system (see next section). Figure 4.a shows a bdd and its ibd used to define the structure of a vehicle (OMG 2006). A TMap in 001AXES default syntax that corresponds to this is shown in figure 4.b. Figure 4.c shows the kbd for just the Anti-Lock Controller portion. The reference, s1, of the original bdd is defined in the kbd TMap (figure 4.b) using the relation, rvel. Figure 4.b shows the path of the rvel relation from its sensor source, s, part of the HubAssy, up and then down into the TractionDetector. This corresponds to the flow implied by the ibd. Flow directionality is captured in a TMap by a relation variable, having a "<" or ">" (or as arrows). This specification directionality will constrain its realization by some resource (e.g., sensor signals).

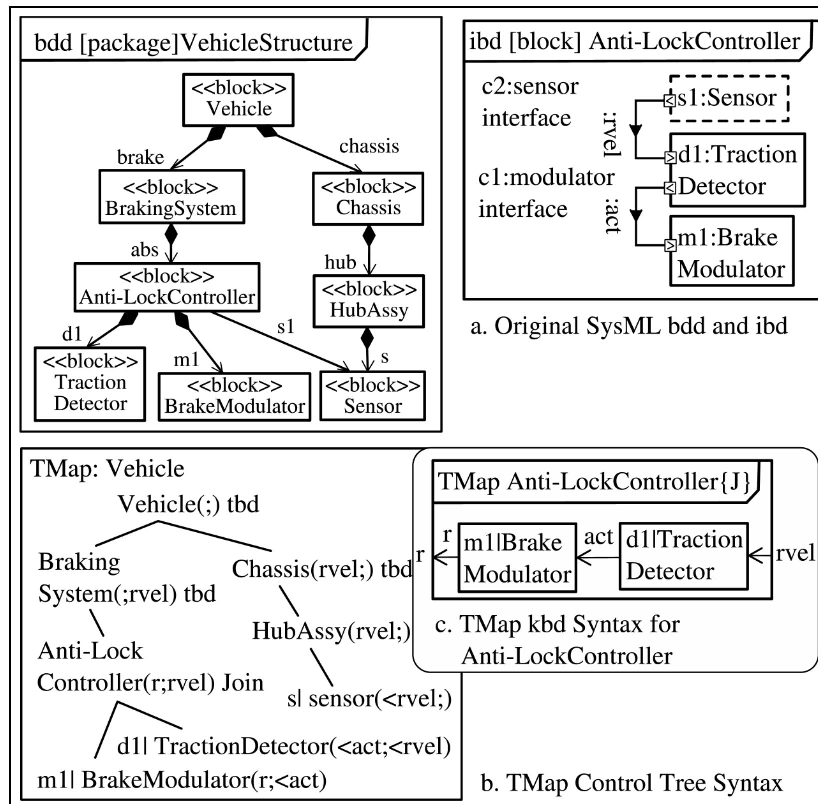

Figure 4: TMap Default Syntax and kbd Syntax

Behavior: Mapping FMaps to SysML Activity Diagrams

In SysML, behavior can be represented using activity diagrams, sequence diagrams and state machine diagrams. Activities in SysML, are classifiers that can be represented as blocks. FMaps are used to define the behavior of a system and incorporate behavioral aspects found in SysML activity, sequence and state machine diagrams. An FMap node integrates the aspects of control, function, causality and time. An FMap node's function corresponds to a SysML action. The bdd representation of activities as classifiers syntactically resembles the 001AXES control tree FMap representation. The FMap is more compact since all nodes are assumed to be functions; no stereotype like mechanism is needed. In a bdd, "bdd[act]" could implicitly mean that all blocks within the bdd have <<activity>> stereotypes. By default composition in these bdds could be simplified by simple lines between parent and children blocks (i.e., leaving off black diamonds and multiplicities). Composition associations would now be more like FMap control line relations as in Figure 2. Instead of representing inputs and outputs of an activity or action by further block decomposition, they would be listed; inputs to the left and outputs to the right of a block. At this point the bdd[act] diagram would syntactically be fairly close to early FMap graphical representations (HOS 1980-1985). The kbd syntax could have been designed with a control tree-like syntax in mind; however, a data flow-like representation, more characteristic of an ibd, was chosen as the kbd syntax to represent the kernel semantics in the examples that follow.

Interrupt, an important FMap defined structure for real time systems (here, represented in kbd syntax), allows object states to be used to interrupt an ongoing executing function (Figure 5). Flows within a kdb FMap have no arrows because flow is always in one direction, left-to-right, from the function that could produce the object state to the function(s) that could receive the object state. A flow at the left border of a kbd function block always has a flow property of "in"; while a flow at the right border of a function block always has a flow property of "out". Kbd function blocks never have in/out flow properties. This is because kbd supports the property of single reference/single assignment for a variable and that a variable represents an object state, not a location in memory.

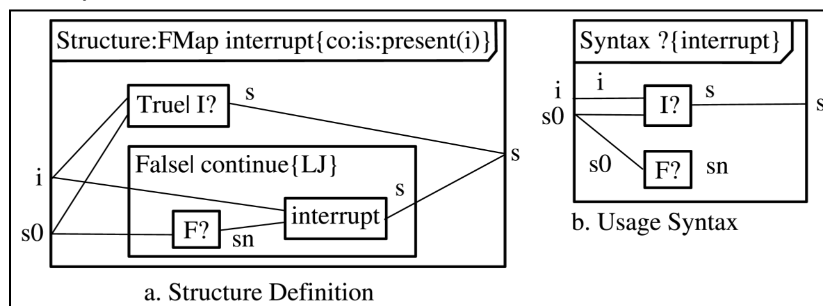

Figure 5: Interrupt Structure Definition

The topmost block input is i and s0; and the output is s. The decision structure, co, defined in terms of Or, uses the primitive function is:present(i) to select one of its sub-blocks only when one or more of the sub-blocks could be invoked by a variable having an object state (i.e., a value) that can be used. When either i or s0 has an object state, is:present can be asynchronously evaluated. This means that at this decision point both inputs are not required for a decision to be made; when only one of i or s0 is has a value, then is:present immediately chooses the alternative. When i has an object state, is:present returns True; the function replacing I? in the use of the structure will be invoked. When the object state of s0 exists and the object state of i

does not exist, is:present returns False; the function that replaces "F?" will be invoked. The interrupt innermost block function within the continue block is a recursive leaf definition (i.e., one using a definition of a containing block, here the outermost diagram block for Structure:FMap interrupt). When an execution instance reaches this point, it uses its recursive ancestor function with the same name (here, the outermost block, topmost function) to determine what to do next. The block family interface pattern for the use of this structure is defined by the usage syntax (figure 5.b). "?{interrupt}" corresponds to this ancestor outermost block function in the definition. It will become the parent of the family using the structure. See the run block in the ignition_on block in figure 6. This run function will be the one used by the recursive interrupt leaf function (hidden within this specific interrupt structure usage) to determine what to do next during the invocation of an execution instance.

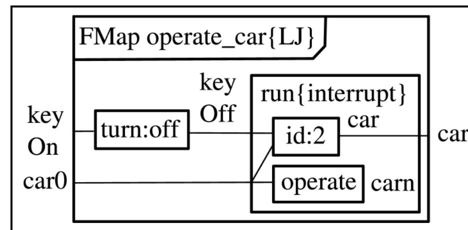

Figure 6: FMap for operate_car

The operate_car kdb FMap has roughly the same behavior as the activity OperateCar in (OMG 2006). To operate car0, its key is turned on. car0 invokes the function, operate, within the run{interrupt} block. Operate is repeatedly invoked until the key is turned off. The is:present(keyOff) function (hidden inside the interrupt structure) checks the state of keyOff before invoking a new instance of Operate. When keyOff has a "turned off" value, the id:2 universal primitive operation is selected by is:present(keyOff) to return the current state of the car, carn, to be its final state, car.

An FMap variable by default always implies a fully asynchronous stream-like behavior. It declares a relationship between potential producer functions and potential receiver functions anywhere in the system that need that variable's object state. As an event, this relationship is logically simultaneous. However, this relationship is constrained by its application on some resource architecture. Its instantiation initiates an invocation process that integrates the functional architecture and the resource architecture invoking (based on priority) the most important functions of the functional architecture onto the most important available resources of the resource architecture. The parent of the producing child determines if any other of its children need the newly created object state. If so, the child with the highest priority is given control of the invocation process. The selected child continues the invocation process; itself, as a parent over its children (again, following the priority rules). When a primitive function is reached it becomes ready to execute based on how it has been allocated. Which inputs are needed before execution can start is determined by the allocated resource. Any function with an output variable that has a newly assigned object state "notifies" its parent who, being in control, determines what to do with it. This results in object states flowing up the chain of command and control with each parent along the way making a decision locally about its own needs. When all functions that need the object state have been invoked, the invocation process is completed. An object state remains active until all functions that need it have completed. An input object state is released when it is no longer needed to produce some output object state. When all output object states have been produced and all input object states have been released, the object state is released.

In Figure 6, when car0 arrives at the operate_car interface, it goes (asynchronously, stream-like) directly to the run interface. The run interface is a usage of the interrupt structure in which the decision partition function becomes: is:present(keyOff)[2]. Assuming the key has not been turned off (i.e., keyOff has no value), car0 invokes operate and any other function inside of operate car as needed. This kind of decision is non-blocking, since the partition function chooses one of its children whether keyOff has a value or not.

Constraints: Mapping Constraint Maps to SysML Parametrics

In SysML, the constraint block and parametric diagram are used to specify constraints among properties of a system and its environment. Figure 7 shows a constraint definition (in both control tree and kbd syntax) of the functional relationships between the fahrenheit and centigrade temperature scales (Steele 1980). A constraint is defined as a set of mappings; in which each domain (and range) is a proper subset of the objects named by the variables at a node (those to the left and right of the semi-colon). A constraint is defined as a special type of FMap structure having a constraint at each node.

A primitive constraint is defined as a set of primitive mappings. Given the constraint, add(a,b;c), when a domain is identified from the variables, "(a,b;c)", the following mappings emerge: add(a,b)=c, sub(c,a)=b, sub(c,b)=a. Both forms of the constraint statements “add(a,b;c)” and “add(c;a,b)” imply these three mappings.

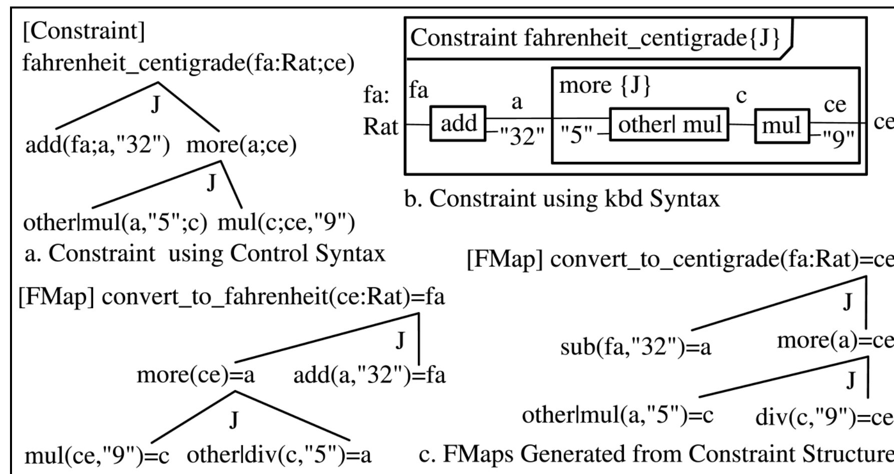

Figure 7: Constraint Structure: Default and kbd Syntax

A constraint structure definition provides for the possibility of generating one of a possible set of FMaps. Identifying a sufficient set of constraint map variables as domain variables determines the selected FMap; a domain/codomain mapping selection and codomain tracing process is used. When the domain variables of a primitive constraint have been determined a mapping is selected from the set of possible primitive mappings. The variables of a primitive constraint not identified as domain variables become codomain variables. Tracing each codomain variable to its related role as a domain variable in other primitive constraints leads to the determination of the domains of these other primitive constraints. There are several ways to evaluate (or compute) a constraint structure. One way is to apply the process to all constraints of

[2] This non-blocking decision point relates to <<interruptibleRegion>> in SysML. A major difference is that SysML is value driven while 001AXES is object state driven with the ability to ask if an object state exists or not. See section Preliminary Findings for more detail.

the structure resulting in the generation of an FMap (e.g., figure 7.b). The FMap is then executed with a standard 001AXES execution engine. How the FMap is generated is described below. Another option is to apply the life cycle process completely at each individual primitive constraint when its mapping has been selected; when the initial domain variable(s) are identified they are given value(s) and when a primitive constraint's mapping has been determined, it is immediately executed, resulting in codomain values. When all individual constraints have been evaluated the constraint structure evaluation is completed. In either option, when this process is completed the values of the variables (object states) are related to each other in terms of the constraint map.

Identifying one variable of the fahrenheit_centigrade constraint map (i.e., fa, ce, a, or c) to be the initial (and sufficient) domain allows for the determination of the primitive constraint mappings and the values of all of the other variables. Figures 7.b and 7.c show two different FMap definitions. given the initial constraint map domain variables of ce and fa respectively. For any abstract constraint, the set of mappings is determined by the combination of variables from the left and right sets. Each variable combination (as with primitive constraints described above) uniquely identifies a mapping. The abstract constraint, fahrenheit_centigrade, has two mappings: mapping1(fa)=ce and mapping2(ce)=fa. When fa is given a value by the user, fa becomes the domain and ce the range; when ce is given a value, ce becomes the domain and fa the range. Each mapping and its consequences may be used to transform the constraint map into a corresponding FMap. One of these, convert_to _centigrade, can be used to convert a fahrenheit value to centigrade value (fa to ce) with the formula: (fahrenheit - 32) * (5/9) = centigrade; another, convert_to_fahrenheit, can be used to convert a centigrade to a fahrenheit value (ce to fa) with the formula: (centigrade * (9/5)) + 32 = fahrenheit. The consequence of identifying fa as a domain variable (and eventually giving it a value) can be seen in convert_to _centigrade. The form of the "add(fa;a,"32")" constraint is "add(c;a,b)"; and since the values of c (here, fa) and b (here, "32") are known, the mapping form "sub(c,b)=a" is used to transform "add(fa;a,"32")" into "sub(fa,"32")=a". The codomain, a, as a domain variable of "other|mul(a,"5";c)" completes the identification of the mapping to be used: "mul(a,"5";c)" is transformed into "mul(a,"5")=c". Finally, with c and "9" as the identified domain of the constraint "mul(c;ce,"9"), it is transformed into "div(c,"9")=ce". With either life cycle evaluation approach, if fa was given the value, "32", the other values would be: a="0", c="0" and ce="0".

System Architecture: Mapping Layering to SysML Allocations

A SysML specification defines allocation mechanisms to support the allocation of behavior to structure. One of the most basic concepts in modeling systems is the separation of the functionality of a system from its realization in terms of concrete resources. This concept is embodied in 001AXES as a layering mechanism (Figure 8). With layering, an allocation architecture (AA) maps the functional architecture (FA) onto a subset of one of a possible set of resource architectures (RAs) resulting in a system architecture (SA). A system (and its behavior) is defined by the mapping of the structure of the FA into the structure of the RA, the mapping of the functionality of the FA into the functionality of the RA and the mapping of the constraints of the FA being upheld by the RA. Each layer provides more system detail in terms of resources used to realize the more abstract layers. At the lowest layer of detail, the real world provides this detail as actual reality.

Each architecture is defined with FMaps and TMaps. The lower boundary of a layer is defined with a set of primitive types each of which has a set of primitive operations and a set of axioms (or constraints). The set of primitive operations defines the mappings and the axioms define conditions that have to be met by a particular RA to be a valid realization. Each primitive operation in the FA is allocated to an RA operation (e.g., an FMap operation). The input and output types in the RA operation become primitive types in the input and output of the FA primitive operation. In 001AXES, an FA is independent of any of its RAs. The allocation mechanisms of SysML, allocatedFrom, allocatedTo and the allocated stereotype to support the allocation of behavior to structure (OMG 2006), most closely relate to 001AXES layering mechanisms. From a 001AXES layered architecture perspective, these SysML allocation statements should be provided in a separate allocation diagram defining the mapping between the functional and resource layers of a system; maximizing the independence of these layers.

Scheduling discipline is inherent in kbd specifications. An SA completely determines the scheduling constraints placed on the system by elements of the FA as realized by its RA. This is because every control node within the system has a unique priority assignment. The Include structure holds the key to concurrency, parallelism and timing of independent actions. In Figure 8, the timing relationships of the Include are expressed in terms of seven predicates (Allen 1991). The thirteen interval time relationships defined by these predicates imply the possible ways in which an active resource can realize an FA function in real time. Because of the Include structure and the fact that c is the highest priority output, the left function is a higher priority than the right function. Although all pairs are potential schedules some are more likely than others depending upon available resources. If only one resource was available and it is assumed that the resource can only perform one function at a time, the only possible schedule would be: before(left,right) or meets(left,right). In addition to the constraints on the before time interval (as seen in figure 8), the meets schedule further assumes that b needs to be available before or at the same time that the resource finishes the action of the left function. The equal(left,right) potential schedule is more subtle and could be applied when the resource has the ability to simultaneously perform both left and right. Because c is more important than d, it would be required to use its capabilities to try to produce c before d.

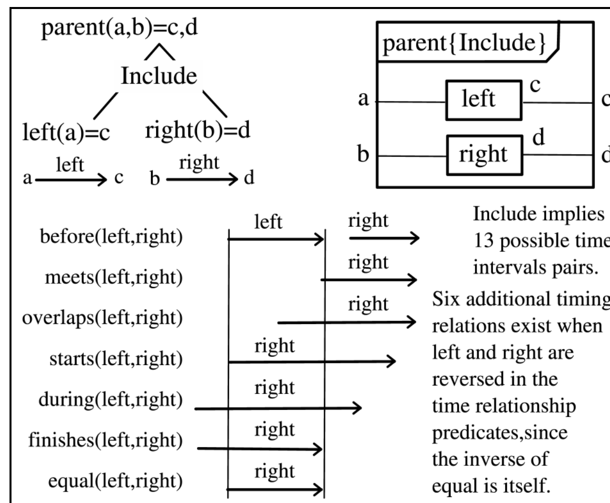

Figure 8: Include: Potential Parallelism

Figure 9 shows an example system (with just the FMaps) of an FA, AA and RA resulting in an SA. The FA (figure 9.a) is simply a function, ship_boxes, that includes two box shipping

functions labeled A and B (corresponding to their roles). The RA (figure 9.c) has a universal FMap that supports transporting a box by any kind of vehicle. For example, a box could be loaded into a car with load:car or into a plane by load:plane when vehicle is replaced by car or plane respectively. The AA (figure 9.b) knows about both the FA and RA, defining a mapping that results in an SA. The AA provides a mapping that can be used in a process of integrating a given FA onto a subset of the selected RA to obtain an SA. The highest priority shipment, A, is allocated to be sent by plane, with "[allocatedTo]sendBy:plane"; and the lower priority shipment to be sent by car, with "[allocatedTo]sendBy:car". Two uses of the sendBy:vehicle universal FMap operation are used to complete the SA. In one case, veh0, is replaced by plane as the resource to be used to transport boxA and in the other, veh0 is replaced by car to transport boxB. The Include structure of ship_boxes in the FA is mapped to the Include structure of ship_boxes in the RA. The behavior of shipping boxA and boxB comes about because of the differences between the resources, plane and car. Parallel behavior results from this mapping of the FA Include structure onto the RA Include structure.

From an FA perspective, in the FMap invocation process, a resource boundary may be crossed. This crossing signals an "allocatedTo" like assignment that brings to bear the characteristics of the new resource, such as its time behavior, into the process. The invocation process provides for the control of fully distributed heterogeneous active objects (i.e., resources) in which all communications and scheduling may be automated. When an FA object state (or flow token) becomes realized by a resource (such as water flowing through a tube or TCPIP communications), its realization by the resource will persist in time and have behavior. The functional object state exists and remains constant over this period of time. Resource boundaries may be crossed within and between layers. Resources are released according to the functional architecture control requirements placed upon the resource architecture.

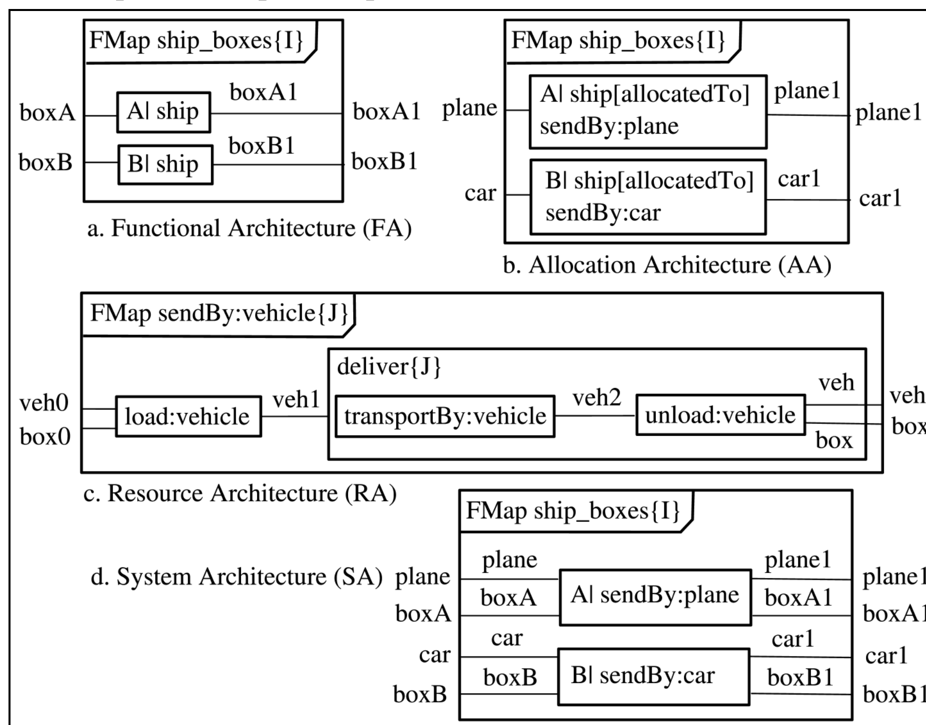

Figure 9: Layered System Architecture for Shipping Boxes

A distributed (or active) object can be thought of as a performance/region (or place) resource while communications can be thought of as transport/path (or connection) resource. Regions perform transformations on objects (and information) and paths transfer objects (and information) from one region to another. Communication between distributed objects can be automated by having fully traceable object states (or flows) between functions of the FA. Given this traceability and the allowed transport mediums between regions (e.g., TCP/IP or a highway), static or dynamic artifacts can be used to mechanize the region/transport design interface. This mechanization process takes into account the performance/region resource boundaries and available transport/path resources. For example, the sends and receives needed for TCP/IP communications can be automatically generated (as distributed design artifacts) into the code or dynamically used by an execution engine when communication between performance regions is needed. From the perspective of a particular execution engine some implementation details that need to be taken into account can be found in (OMG 2005a). This type of dynamic communications strategy for the development of distributed systems can be incorporated into a given execution engine (e.g., the 001 DXecutor) for a given application domain.

From a conceptual perspective, the resource layer provides a map or system of places and connections between places. The functional layer as a model of the current relationships the system has with its environment identifies what will take place at any moment in time within (and constrained by) the resource map, RA.

Preliminary Findings

The SysML/001AXES project was begun by first comparing SysML syntax with 001AXES syntax (while the kernel semantics of 001AXES is syntax independent, a syntax is required to express it). An analysis revealed that there was good support for many of the formalisms, but in other areas, the seemingly natural correspondences between the 001AXES kernel syntax and the SysML syntax was sometimes misleading. In these cases, the underlying semantics of SysML (Bock 2005) was found to be quite different than the semantics of 001AXES. What follows is a discussion of some of the findings regarding the analysis of the correspondences between SysML and 001AXES. The results are preliminary and will require further review with SysML experts to both validate the findings, and determine potential enhancements to SysML to address them.

Finding (variables): In UML2, a variable refers to a physical location such as a memory location (Selic 2004). This is essentially the same semantics used in traditional programming languages such as C or Java. This type of variable semantics often results in misunderstood behavior and systems with side-affects. Some of the issues raised are those regarding resource allocation, consistent naming and traceability of object states. Any SysML specification that uses a UML2 runtime semantics runs the risk of having this type of variable semantics. In DBTF, axioms 3 and 4 do not allow a function to assign a value to one of its input states. This property of kernel variables is called single reference/single assignment. For example, add(x,”1”)=x (or as in a programming language: x=x+1), would mean that the add function modifies its input state x by adding one to it. Any other statement that relied on the value of the original state of x would be side affected. This would violate the traceability of changes to the states of objects and would further not allow for the generalized scheduling of distributed systems resources. The semantics of the kernel eliminates these types of problems.

Finding (invocation policy): In SysML, the default invocation strategy resides within a default synchronous framework. This reflects an imperative orientation of "controlling some machine (such as a car or robot)" as opposed to a declarative statement about the system and how

it interacts with or relates to its environment. ParametricConstraints go a long way in recognizing this declarative emphasis. However, the use of UML2 within SysML specifications may embed unwanted and unnatural system semantics when just the system and its relationship to its environment is to be considered. The result may be that systems are built to run machines with a limited characterization of the abstract functionality of the system and the resource related problem conditions of the environment the machine implementation is intended to solve. Given SysML, without kdb-like enhancements, it is not clear that a completely functional and asynchronous system specification can be defined from a 001AXES perspective.

With the kernel, the functional architecture is defined by default with an asynchronous invocation strategy that is driven by discrete events, under control[3]. The functional architecture is completely declarative as a set of causal relationships and constraints it has with its environment. The conditions of its environment are specified as constraints on its implementation (which could directly be the real world). The relationship between a kernel functional architecture and its environment is the set of constraints collected from each of its successive layer boundaries, down to a primitive layer that directly interacts with reality (e.g., a robot sensory motor subsystem).

Finding (activity control mechanisms): In SysML, control mechanisms are not directly aligned with the decomposition of an activity, control priority is not assigned to every action in the system, and an asynchronous fully traceable event driven variable invocation policy (as described above, integrated with parent/child lines of control) is not the default. Because priority is not used to totally order the system, automated resource allocation with automatic scheduling is not viable; and, other conflicts can arise that need to be dealt with by users. The propensity for synchronization mechanisms seems to stem from the above shortcomings. For example, the SysML-join control node synchronizes multiple object flows into a single sequential flow. The sequential order is based upon token arrival. However, the order was found not to be specified when tokens arrive at the same time (OMG 2005b). A composite flow (like parallel wires in a tube) of independent asynchronously flowing tokens was found not to be supported.

With the kernel, all these shortcomings are integrated into a system of control; the necessity for a host of control node mechanisms is eliminated; e.g., initial node, final nodes (flow and activity) and control flow aspects of the fork node. The kbd syntax is significantly more compact than an activity diagram. Control (in the kbd sense) is implicitly represented at each parent for its children without the need for any other explicit graphical syntax. The priority of independent children is implicit in the vertical placement of a node within its parent's block. 001AXES universal primitive mappings, clone2 and include (Hamilton and Hackler 2003-2004), closely relate to the behavior behind the fork and SysML-join control nodes. The clone2 mapping (like fork) provides for copies of (multiple references to) object states. The include mapping provides for the grouping of independent asynchronous object states into a single composite state. The independent object states may be allocated as sequential or parallel flows depending on their layered resource allocations.

---

[3] In SysML, this would mean always using two tokens (a control flow token and an object flow token to symbolize a discrete event); that when used together can determine which SysML action to invoke. Both tokens have a multiplicity of exactly 1. When an object (value) is produced, these tokens would follow the path of the variable from activity to activity, stream-like, through all interfaces from the source active object through all active objects to the target action(s). Using the kernel, however, no multiple tokens, signal, synchronous or asynchronous communication calls are needed by a user.

Finding (existence of object states): In SysML, control can be treated like an object flow, as a control flow. However, this treatment relies on a variable's having a value (e.g., enable or disable). In the kernel event driven system, it is important to be able to determine if an object state event has occurred or not (e.g., Does a variable have a value or not?). No mechanism in SysML or UML2, was found that could provide for this type of behavior. A SysML-join statement provides for synchronization or waiting until certain object flows have been given values; it is completely value driven. It does not support asking a question like: "Does A or B have a value?", it can only do something when A or B has a value. The kernel provides special system functions (e.g., is:present) to ask questions about the existence of a variable's object state.

Finding (complete mappings): In SysML, activities are allowed to not have inputs and outputs. In addition, internal to the activity, a synchronous, asynchronous or signal variable may be used to pass information from one active object to another. This results in the loss of traceability of these variables to the real interfaces imposed by the model. An event or signal variable may appear and disappear without being traced to higher level activity interfaces. Using these kinds of variables has the effect of embedding and early binding of operating system-like behavior into the specification, limiting it in many respects (e.g., portability) and allowing many control issues to surface (e.g., those to do with resource scheduling, deadlocks and priority inversion). These types of issues are characteristic of sequential programming languages with communication applications interfaces for passing information between distributed processes.

With the kernel, every function must have inputs and outputs; and all variables are traceable. This among other things allows for the full automation of the scheduling of resources with no deadlocks and no priority inversion of processes. As described earlier in the system architecture section, by maintaining traceability, appropriate automations can be used to automate the communications design process (e.g., for defining or initiating synchronous or asynchronous sends and receives). In a distributed system a variable traced from one distributed (or active) object to another can be replaced with an invocation process mechanism to notify the target of an object shipment and a data transport mechanism to ship the object when the target signals it is ready for the shipment (e.g., by automatically generating standard ports for the models).

Finding (active objects): UML2 is based upon passive and active objects (Selic 2004) with a weak layering philosophy. A SysML specification using UML2 inherits these properties. Inter-object communication implies that implicit or explicit resource allocations have taken place. A specification with an embedded synchronous or asynchronous send or receive communication function restricts that specification from a functional architectural perspective. To change the communication strategy, one has to change the specification. This means the specification is not functionality separated from the communications strategy; it has embedded resource allocation artifacts. For example, allocatedFrom, input/output port synchronization, signals and words such as "parallel" used in a functional specification model all indicate some aspect of embedded allocation (or a misuse of terminology). A functional specification never exhibits parallelism (it may denote independence, however); parallelism in a system only comes about through the realization of a functional architecture on a resource architecture. Portability suffers, for example, an "allocatedFrom" statement embedded in an activity diagram limits its portability.

With the kernel, an active object is always associated with a resource allocation mapping. Given the kernel, any active object can be built as a layered system. Communication mechanisms are part of the resource architecture. The allocation architecture specifies how the communications mechanisms are applied to the functional architecture. From an architectural perspective, portability is maximized.

Conclusion

Unlike having first created a programming language(s) for defining software systems specifically for a computer (a syntax first, syntax dependent approach); 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. It is this very flexibility of the language that gives it the ability to lend its formal support to SysML.

We have shown examples of mappings between SysML and 001AXES (see above) and determined during our analysis that it is possible to support SysML with the 001AXES formal semantics kernel such that SysML systems will be able to benefit significantly by inheriting some of the properties provided by the kernel. We have described the semantics of 001AXES and shown by example some ways in which the 001AXES kernel would work for SysML. We have discussed possible extensions for SysML in order for it to take advantage of the kernel properties. The next step is to take a subset of SysML, incorporate into it the necessary syntactical extensions for it to layer onto the kernel and demonstrate its behavior by hooking into some of the kernel's existing automation.

With the kernel as its semantics foundation, SysML has the potential to eliminate well known problems; because of the properties that in essence "come along with the territory". To name a few: 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 increase with the increased use of properties of preventative systems. Compared to traditional techniques, the productivity of DBTF designed and developed systems has been shown to be significantly greater. See for example, (DoD 1992), (Krut 1993), (Ouyang 1995) (Keyes 2000a), (Schindler 1990), (SPC 1998), (www.htius.com 1986-2007), (http://icb.nasa.gov/001). Upon further analysis, it was 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 DBTF's support of reuse. The larger a system, the more it has the opportunity to capitalize on reuse. As more reuse is employed, productivity *continues* to increase (e.g., less testing is necessary with 001AXES's inherent "reuse" of each new DBTF capability). Measuring it becomes a process of relativity—that is, relative to the last system developed.

By inheriting the preventative philosophy of the kernel, SysML users will have the potential to "solve" (prevent) a given problem as early in the life cycle as possible. Static analysis is more before the fact than dynamic analysis. Preventing a problem by the very way a system is defined is even more before the fact. Better yet, not having to define (and build) it at all. The ultimate reusable is in the application of the kernel to both systems and software; unifying their understanding by a formal means, with a commonly held set of system semantics.

References

Allen, J., Kautz, H., Pelavin, R., Tenenberg, J., Reasoning About Plans, by Morgan Kaufmann Publishers, Inc., 1991.

Bock, Conrad, "SysML and UML 2 Support for Activity Modeling", Wiley InterScience, Nov. 4, 2005.

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

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

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

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

Hamilton, M., “Inside Development Before the Fact”, Electronic Design, April 4, 1994, ES.

Hamilton, M., “Development Before the Fact in Action”, Electronic Design, June 13, 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 and 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, Defensive Technology Evaluation Code (DETEC) Conceptual Model, 1988.

Hamilton, M. and Hackler, W. R., “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, 1990.

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

Hamilton, M. and 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. and 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, 2003-2004.

Hamilton, M. and Hackler, W. R., “Universal Systems Language for Preventative Systems Engineering”, 5th Annual Conference on Systems Engineering Research (CSER), March, 2007.

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

Hamilton, M. and 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.

Hamilton Technologies, Inc. (HTI), 001 Tool Suite Reference Manual, Introduction to the 001 Tool Suite, Cambridge, MA, 1986-2007.

Hamilton Technologies, Inc. (HTI), Final Report: Object Tracking and Designation (OTD), Architecture Independent Operating System (AIOS) and REBEL, prepared for Strategic Defense Initiative Organization (SDIO) and Los Alamos National Laboratory, Los Alamos, NM 87545, Order No. 9-XG9-F5131-1, December 1989.

Hamilton Technologies, Inc. (HTI), Accident Record System Interim Solution (ARS-IS) Requirements/Design Specification, Prepared for Massachusetts Highway Department Traffic Operations, February 10, 1997.

Higher Order Software (HOS), Application of a Formal Systems Methodology to Civil Defense, Prepared for Defense Civil Preparedness Agency, Wash., D.C. 20301, March 1980.

Higher Order Software (HOS), USE.IT Reference Manual, Cambridge, MA, 1980-1985.

http://icb.nasa.gov/001

http://www.htius.com or http://world.std.com/~hti, What Others Say, 1986-2007.

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, Software Engineering Institute, Carnegie Mellon University, 1993.

Object Management Group (OMG), UML™ Profile for Schedulability, Performance, and Time Specification version 1.1 formal/05-01-02, Available from www.omg.org 2005a.

Object Management Group (OMG), Unified Modeling Language: Superstructure version 2.0 formal/05-07-04, Available from www.omg.org 2005b.

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

Ouyang, M., Golay, M.W., 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, 1995.

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

Selic, B., "On the Semantic Foundations of Standard UML 2.0", in Bernardo, M., and [corradini] Corradini, F. (eds.), Formal Methods for the Design of Real-Time Systems, Lecture Notes in Computer Science vol. 3185, Springer-Verlag, 2004.

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

Steele, Guy Lewis, Jr., The Definition and Implementation of a Computer Programming Language Based on Constraints, AI-TR-595 Massachusetts Institute of Technology artificial Intelligence Laboratory, Cambridge, MA., 1980.

001 Tool Suite, 001, Development Before the Fact, DBTF, Universal Systems Language, USL, Function Map, FMap, Type Map, TMap, Object Map, OMap, Road Map, RMap, Execution Map, EMap, Resource Allocation Tool, RAT, System Oriented Object, SOO, 001AXES, Xecutor and DXecutor are all trademarks of Hamilton Technologies, Inc. OMG SysML and UML are trademarks of the Object Management Group.

Biography

Margaret H. Hamilton (mhh@htius.com) is the founder and CEO of Hamilton Technologies, Inc. (HTI) in Cambridge, MA. She created the DBTF paradigm for designing systems and developing software (along with its associated formal systems language, 001AXES and its automation, 001). She is responsible for the 001 Tool Suite product line which includes components for a full system design and software development life cycle.

Hamilton was the founder and CEO of Higher Order Software, Inc. (HOS), where she was responsible for the first Computer Aided Software Engineering (CASE) product line in the industry (called USE.IT) and the evolving theory behind it (HOS).

Hamilton was responsible for the Apollo (and Skylab) on board flight software while Director of the Software Engineering Division at Charles Stark Draper Laboratory. She created and developed the mathematical theory based on empirical studies of the Apollo on board flight software which formed the earlier beginnings for the DBTF systems systems paradigm.

William R. Hackler (ron@htius.com) is Director of Development at Hamilton Technologies, Inc. (HTI) based in Cambridge, MA. He is the lead engineer for the development of the 001 Tool Suite; using it to define and generate itself. Hackler has been responsible for evolving language concepts and many other 001 designed and developed systems including the development of a simulator for the University of California Los Alamos National Laboratory; a missile tracking simulation for a large aerospace company (HTI was nominated for SBA Subcontractor of the Year as a result of this effort); several asynchronous real time distributed applications for SDI; a factory model for an aerospace manufacturing plant; and several Internet related applications including an accident record system for state highway departments and a trading system for a financial systems.

Hackler was Director of Advanced Concepts at HOS where he developed technologies based on the foundations of the HOS theory. He was responsible for many applications applying this theory and its automation, USE.IT (many components of which he was responsible for developing). These applications included the development of systems in the areas of battle management and aerospace manufacturing. He studied composition with composer Martin Brown in Charles Ives lineage of music and applied music theory to the objects of mathematics.