Original cover, Electronic Design's Special Editorial Supplement*, April 4, 1994. Reproduced here for historical context; the article begins overleaf.*

PROGRAMMING

Inside Development Before The Fact

With this preventative approach, each system is defined with properties that control its own design and development.

By Margaret H. Hamilton
Hamilton Technologies Inc.

Today's traditional system engineering and software development environments support their users in "fixing wrong things up" rather than in "doing them right in the first place." Things happen too late, if at all. Systems are of diminished quality and an unthinkable amount of dollars is wasted. This becomes apparent when analyzing the major problems of system engineering and software development.

In defining requirements, developers rely on many different types of mismatched methods to capture aspects of even a single definition. Among other things, data flow is defined using one method, state transitions another, dynamics another, data types another and structures using still another method.

Once these aspects of requirements are defined, there is no way to integrate them. Designers are forced to think and design this way because of the limitations of methodologies available to them.

This leads to further problems. Integration of object to object, module to module, phase to phase, or type of application to type of application becomes even more of a challenge than solving the problem at hand.

This is compounded by a mismatch of products used for design and development. Integration of all forms is left to the devices of a myriad of developers well into the development process. The resulting system is hard to understand, objects cannot be traced, and there is at best little correspondence to the real world.

With these traditional methods, systems are actually encouraged to be defined as ambiguous and incorrect. Interfaces are incompatible and errors propagate throughout development. Once again the developers inherit the problem. The system and its development are out of control.

Requirements are defined to concentrate on the application needs of the user, but they do not consider that the user changes his mind or that his environment changes. Porting becomes a new development for each new architecture, operating system, database, graphics environment, language, or language configuration; critical functionality is avoided for fear of the unknown, and maintenance is both risky and the most expensive part of the life cycle. When a system is targeted for a distributed environment, it is often defined and developed for a single processor environment and then re-developed for a distributed environment—another unnecessary development.

Insufficient information about a system's runtime performance, including that concerning the decisions to be made between algorithms or architectures, is incorporated into a system definition. This results in design decisions that depend on analysis of outputs from exercising ad hoc implementations and associated testing scenarios. A system is defined without considering how to separate it from its target environment.

It is not known if a design is a good one until its implementation has failed or succeeded. The focus for reuse is late into development during the coding phase.

Requirements definitions lack properties to help find, create, and make use of commonality. Modelers are forced to use the same informal and manual methods to find ways to divide a system into components natural for reuse.

Why reuse something in today's changing market if it is not able to be integrated, not portable

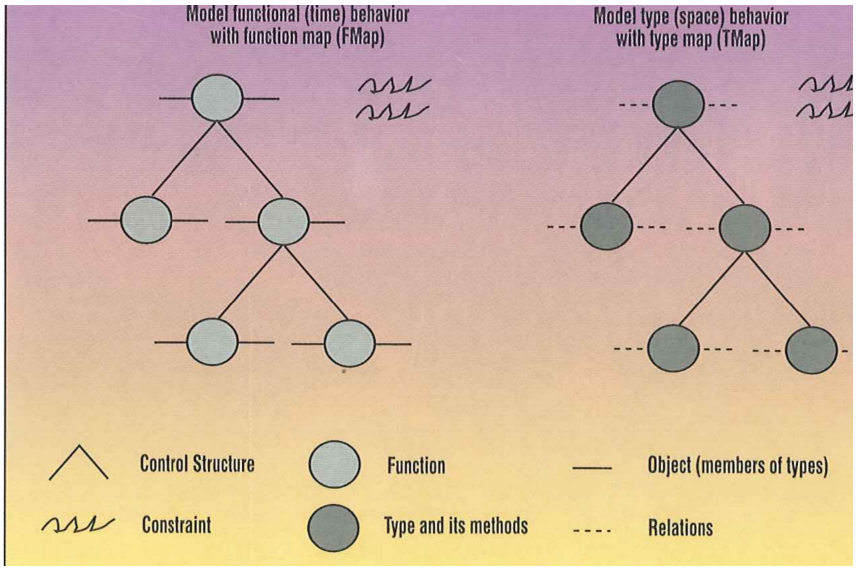

Figure 1. Every model is defined in terms of an integration of Functional Maps (FMaps) for capturing time behavior and Type Maps (TMaps) for capturing space behavior. A map is both a control hierarchy and a network of interacting objects.

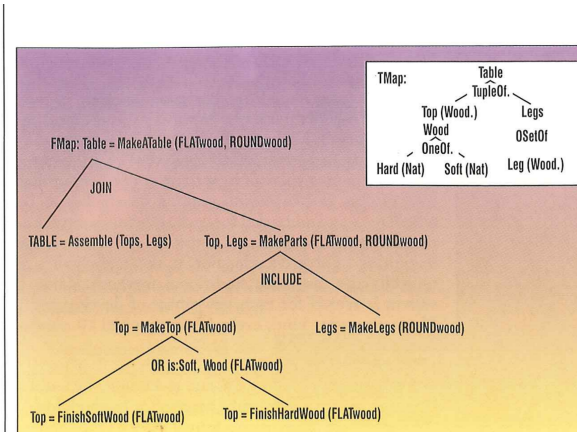

Figure 2. The three primitive structures are ultimately used to decompose a map. The FMap part of the system, MakeATable, is modeled using the JOIN, INCLUDE and OR for controlling dependent, independent, and decision-making functions, respectively.

or adaptable, and it is error-prone?

The result is that there is little incentive for reuse and redundancy is a way of life. Again, errors propagate accordingly.

Automation, itself, is an inherently reusable process. If a solution does not exist for reuse, it does not exist for automation.

Systems are defined with insufficient intelligence for automated tools to use them as input. Too often, automated tools concentrate on supporting the manual process instead of doing the real work.

Definitions supported by automation are given to developers to turn into code manually. A process that could have been mechanized once for reuse is performed manually over and over again.

When automation attempts to do the real work, it is often incomplete across application domains or even within a domain, resulting in incomplete code such as skeleton or shell code.

Manual processes are needed to complete unfinished automations. An automation for one part of a system (e.g., graphics) needs to be manually integrated with an automation for another part of the system (e.g., scientific algorithms) or with the results of a manual process.

The code generated is often inefficient and/or hard-wired to a particular architecture, language, or even a particular version of a language. Most of the development process is needlessly manual. Again, all these manual processes are creating new errors each time.

A promising solution to these problems is development before the fact. Whereas the traditional approach is after the fact, or curative, development before the fact approach is preventative.

Development Before The Fact

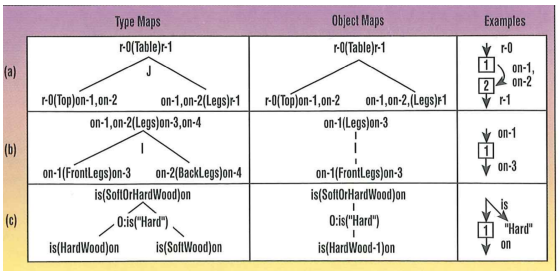

Figure 3. A TMap (and its corresponding OMaps) can be decomposed into its explicit relationships in terms of three primitive structures.

With development before the fact, each system is defined with properties that control its own design and development. With this paradigm, a life cycle inherently produces reusable systems, realized in terms of automation. Unlike before, an emphasis is placed on defining things right the first time. Problems are prevented before they happen. Each system definition not only models its application but it also models its own life cycle.

From the very beginning, a system inherently integrates all of its own objects (and all aspects of and about these objects) and the combinations of functionality using these objects. It maximizes its own reliability and flexibility to change; capitalizes on its own parallelism; supports its own run-time performance analysis and the ability to understand the integrity of its own design; and maximizes the potential for its own reuse and automation. The system is developed with built-in quality and with built-in productivity.

Whereas a curative means to obtain quality is to continue testing the system until errors are eliminated, a preventative means is to not allow errors to creep in, in the first place.

Whereas a curative means to accelerate a particular design and development process is to add resources, such as people or processors, a preventative approach would find a more efficient way to perform this process, such as capitalizing more on reuse or eliminating parts of it altogether, yet still reaching the desired results.

Effective reuse is a preventative concept. Reusing something with no errors to obtain a desired functionality avoids the errors of a newly developed system; time and money will not be wasted in developing that new system. For successful reuse, a system has to be worth reusing and reused for each user requiring functionality equivalent to it. This means starting from the beginning of a life cycle, not at the end, which is typically the case with traditional methods. Then a system is reused for each new phase of development.

No matter what kind, every 10 reuses saves 10 unnecessary developments.

The Technology

Development before the fact technology includes a language, an approach, and a process (or methodology), all of which are based upon a formal theory. Once understood, the characteristics of good design can be reused by incorporating them into a language for defining any system. This language is the key to development before the fact. It has the capability to define any aspect of any system (and any aspect about that system) and integrate it with any other aspect. These aspects are directly related to the real world.

This same language can be used to define system requirements, specifications, design, and detailed design for functional, resource and resource allocation architectures throughout all levels and layers of "seamless" definition, including hardware, software and peopleware. It could be used to define missile or banking systems as well as real-time or database environments. It is used to define and integrate implementation-independent, function-oriented decompositions with implementation-independent, object-oriented decompositions. It defines and integrates these decompositions (control hierarchies) with networks of functions and objects. It can be used to define systems with diverse degrees of fidelity and completeness. Such a language can always be considered a design language, since design is relative; one person's design phase is another person's implementation phase.

This language has mechanisms to define mechanisms for defining systems. Although the core language is generic, the user "language," a by-product of a development, can be application-specific, since the language is

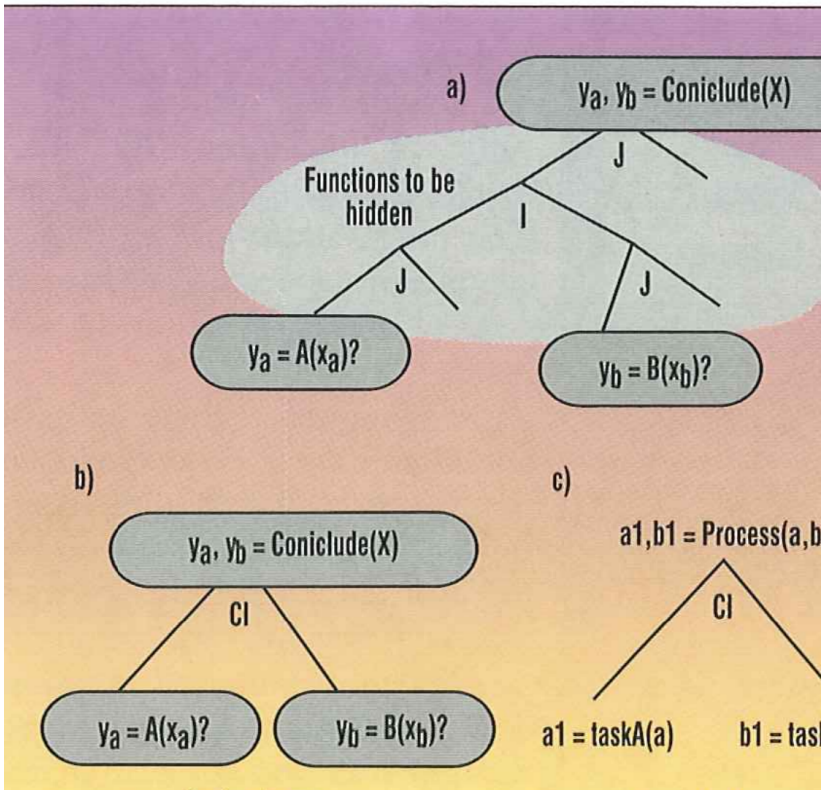

Figure 4. Defined structures are used to define non-primitive structure reusables in terms of more primitive structures. COINCLUDE is an example of a system pattern that has been turned into a defined structure.

semantics-dependent but syntax-independent.

The first step in building a before the fact system is to define a model with the language. This process could be in any phase of development, including problem analysis, operational scenarios, and design. The model is automatically analyzed to ensure that it was defined properly. This includes static analysis for preventative properties and dynamic analysis for user intent properties.

A fully production-ready and fully integrated software implementation for any kind of application, consistent with the model, is then automatically generated by the generic generator for a selected target environment in the language of choice (for example, in C or in Ada) and the architecture of choice. If the selected environment has already been configured, it is selected directly; if not, the generator is configured for a new language (for example, for VHDL, EDIF, and C++) and new architecture before it is selected.

The resulting system can then be executed. If the desired system is software, the system can now be tested for further user intent errors. It becomes operational after testing. Target changes are made to the requirements definition, not to the code. Target architecture changes are made to the configuration of the generator environment, not to the code. If the real system is hardware or peopleware, the software system serves as a simulation upon which the real system can be based.

Development before the fact is a function- and object-oriented approach based upon a unique concept of control. The foundations are based on a set of axioms and on the assumption of the existence of a universal set of objects. Each axiom defines a relation of immediate domination. The union of the relations defined by the axioms is control. Among other things, the axioms establish the relationships of an object for invocation, input and output, input and output access rights, error detection and recovery and ordering during its developmental and operational states. Table 1 summarizes some of the properties of objects within development before

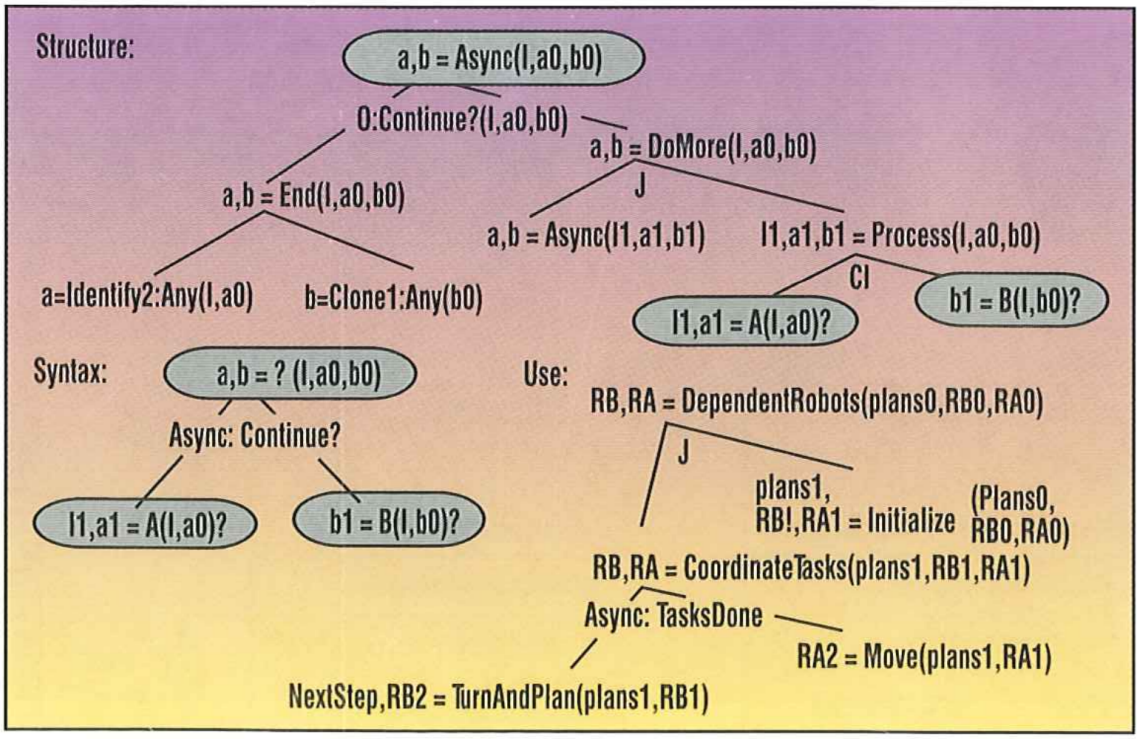

Figure 5. Async is a defined structure that can be used to define distributed systems with both synchronous and asynchronous behavior.

the fact systems.

This approach is used throughout a life cycle starting with requirements and continuing with functional analysis, simulation, specification, analysis, design, system architecture design, algorithm development, implementation, configuration management, testing, maintenance, and reverse engineering. Its users include managers, system engineers, software engineers, test engineers, as well as end users.

The development before the fact approach had its earlier beginnings in 1968 with the Apollo space missions when research was performed for developing software for man-rated missions. This led to the finding that interface errors accounted for approximately 75% of all errors found in the flight software during final testing. They include data flow, priority, and timing errors at both the highest and lowest levels of a system to the finest grain detail. Each error was placed into a category according to the means taken to prevent it by the very way a system was defined. A theory and methodology was derived for defining a system such that this entire class of interface errors would be eliminated.

Integrated Modeling Environment

The first technology derived from this theory concentrated on defining and building reliable systems in terms of functional hierarchies. Since that time this technology has been further developed to design and build systems with development before the fact properties in terms of an integration of both functional and type maps where a map is both a control hierarchy and a network of interacting objects.

The philosophy behind this approach is inherently reusable where 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. The new system becomes a reusable for building other systems.

Every model is defined in terms of functional hierarchies (FMaps) to capture time characteristics, and type hierarchies (TMaps) to capture space characteristics (Figure 1). FMaps and TMaps guide the designer in thinking through his concepts at all levels of system design. With these hierarchies, everything you need to know (no more, no less) is available. All model viewpoints can be obtained from FMaps and TMaps,

Table 1: Object-oriented properties of development before the fact

Control: The ability to have an object control and be controlled throughout all phases of its development and its birth, life, and death of operation.
Creation: The ability to create an object.
Reference: The ability to use or mention an object.
Uniqueness: The ability to select or find an object.
Destruction: The ability for an object and all of its influences to be destroyed.
Identification: The ability to identify an object with respect to its structure, its behavior, its relationships in development and in operational real time and real space with respect to an integrated set of all aspects of control.
Classification: The ability to belong to the same class with other objects, each of which shares a common set of properties.
Traceability: The ability to trace the birth, life, and death of an object and its definitions, as well as its transitions between definition and instantiation; trace changes and know their effects; trace patterns (e.g. distributed patterns), control, and function flow (including data, priority access rights and timing).
Accessibility: The ability to safely access an object in all of its states of existence.
Boundary conditions: The ability to safely exclude invalid states of an object.
Security: The ability for an object to keep its behavior and structure secure (e.g. communication of one function with another function always takes place at the same level in a hierarchy and a given object has no knowledge of a higher level object.)
Belonging: The ability of an object to belong to, for example, a parent or a set of values.
Having: The ability for an object to have for example, a child, sibling, a set of proper values, and an architecture.
Timing: The ability to instantiate an object at a given time or a given event.
Modularity: The ability for an object to be portable, flexible, and reusable.
Healthy existence: The ability for an object to live a full life throughout all of its states of being and doing (persistence).
Completion: The ability to determine when an object is completely defined, used, or instantiated.
Inheritance: The ability of an object to derive behavior in terms of other objects.
Real-time and space constraints: The ability to realize an object in terms of its physical existence.
Minimality: The ability to define necessary and sufficient information about an object.
Containment: The ability of an object to have an inside and an outside (encapsulation). The outside view of an object may be completely replaced by the inside view of the object in question.
Representation: The ability for an object to have a natural correspondence to the desired aspects of the real-world object of which it is a model.
Ordering: The ability to establish a relation in a set of objects so that any two object elements are comparable in that one of said elements precedes the other said element.
Priority:The ability to determine which object is more important than any other, given constraints such as time, priority, order, events, and architecture.
Relativity: The ability for an object to change roles depending on how it is being used (polymorphism) or viewed. This includes being vs. doing, controller vs. controllee, parent vs. children, requirements vs. implementation.
Structure: The ability to distinguish between properties of dependence, independence, and decision-making within, between and about objects.
Predictability: The ability for the behavior and structure of an object to be understood in terms of its relationships without ambiguity.

Table 1. Object-oriented properties of development before the fact (reproduced from the original sidebar).

including data flow, control flow, state transitions, data structure, and dynamics. Maps of functions are integrated with maps of types.

On an FMap there is a function at each node, which is defined in terms of and controls its children functions. For example, the function—build the table—could be decomposed into and control its children functions, make parts and assemble. On a TMap there is a type at each node that is defined in terms of and controls its children types. For example, type, table, could be decomposed into and control its children types, legs and top.

Every type on a TMap owns a set of inherited primitive operations. 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 hierarchy (OMap) and is a member of a type from a TMap. FMaps are inherently integrated with TMaps by using these objects and their primitive operations. FMaps are used to define, integrate, and control the transformations of objects from one state to another state (for example, a table with a broken leg to a table with a fixed leg). Primitive operations on types defined in the TMap reside at the bottom nodes of an FMap. Primitive types reside at the bottom nodes of a TMap.

When a system has all of its object values plugged in for a particular performance pass, it exists in the form of an execution hierarchy (EMap).

Typically, a team of designers will begin to design a system at any level (this system could be hardware, software, peopleware or some combination) by sketching a TMap of their application. This is where they decide on the types of objects (and the relationships between these objects) that they will have in their system. Often a Road Map (RMap), which organizes all system objects including FMaps and TMaps, will be sketched in parallel with the TMap.

Once a TMap has been agreed upon, the FMaps begin almost to fall into place for the designers because of the natural partitioning of functionality (or groups of functionality) provided to the designers by 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 structural partitioning of the FMaps is balanced against the structural organization of the shape of the objects as defined by the TMap). With FMaps and TMaps a system (and its viewpoints) is divided into functionally natural components and groups of functional components that naturally work together; a system is defined from the very beginning to inherently integrate and make understandable its own real world definition.

Primitive Structures

All FMaps and TMaps are ultimately defined in terms of three primitive control structures: a parent controls its children to have a dependent relationship, an independent relationship, or a decision-making relationship. A formal set of rules is associated with each primitive structure. If these rules are followed, interface errors are "removed" before the fact by preventing them in the first place. As a result, all interface errors (75% to 90% typically found during testing in a traditional development) are eliminated at the definition phase. Using the primitive structures supports a system to be defined from the very beginning to inherently maximize its own elimination of errors.

Use of the primitive structures is shown in the definition of the FMap for system, MakeATable (Figure 2). The top node function has FLATwood and ROUNDwood as its inputs and produces Table as its output. MakeATable, as a parent, is decomposed with a Join into its children functions, MakeParts and Assemble. MakeParts takes in as input FLATwood and ROUNDwood from its parent and produces Top and Legs as its output. Top and Legs are given to Assemble as input. Assemble is controlled by its parent to depend on MakeParts for its input. Assemble produces Table as output and sends it to its parent.

MakeParts, as a parent, is decomposed into children, MakeLegs and MakeTop, who are controlled to be independent of each other with the Include primitive control structure.

MakeLegs takes in part of its parent's input and MakeTop takes in the other part. MakeLegs provides part of its output (Legs) to its parent and MakeTop provides the rest. MakeTop controls its children, FinishSoftWood and FinishHardWood, with an Or. Here, both children take in

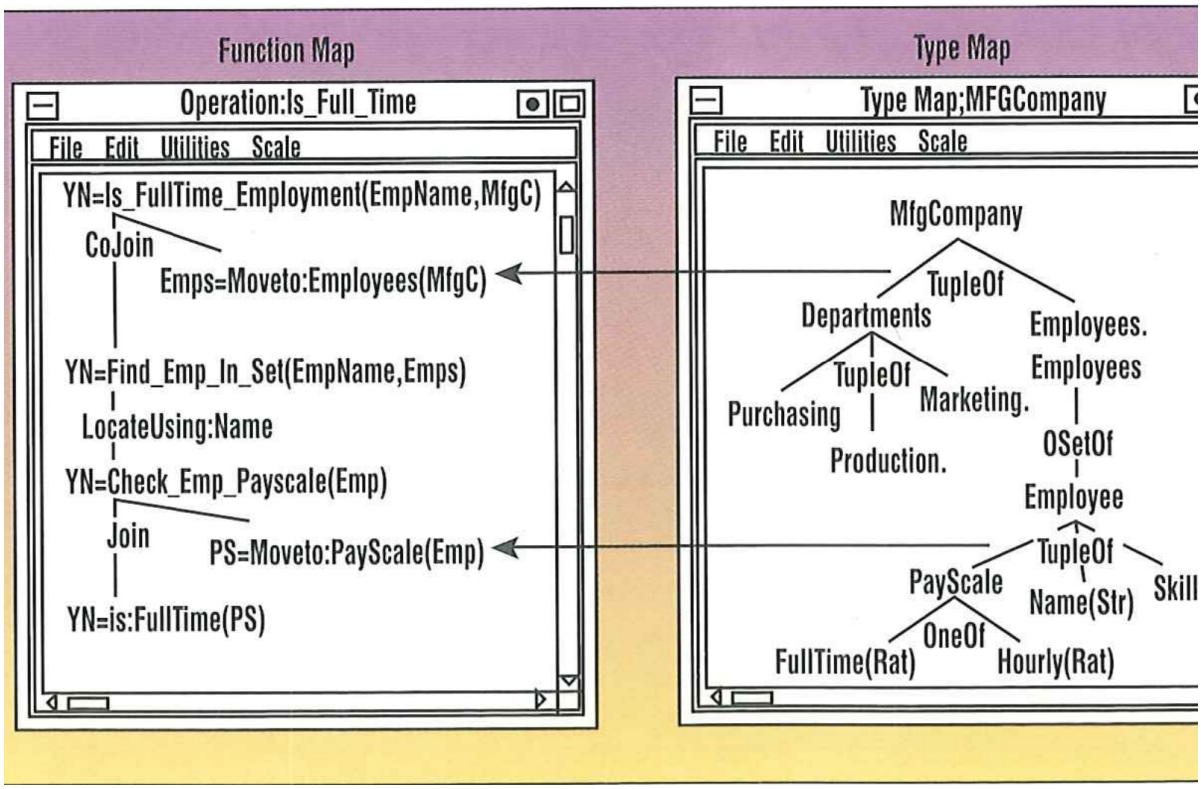

Figure 6. A complete system definition is an integration of FMaps that have been decomposed until reaching primitive functions on types in the TMaps and TMaps that have been decomposed until reaching primitive types.

the same input and provide the same output since only one of them will be performed for a given performance pass. FinishSoftWood will be performed if the decision function is:Soft,Wood returns true; otherwise, FinishHardWood will be performed. Notice that input (for example, FLATwood) is traceable down the system from parent to children and output (for example, Table) is traceable up the system from children to parent. All objects in a development before the fact system are traceable. MakeATable's TMap, Table, uses non-primitive structures called parameterized types, a concept discussed in a later section.

Each type on a TMap can be decomposed in terms of primitive structures into children types where the defined relationships between types are explicit. In Figure 3, Table as a parent has been decomposed into its children, Top and Legs, where the relations between Top and Legs are on-1 and on-2, respectively, the relation between Table and Legs is r-1 and the relation between Table and Top is r-0. Notice that Top depends on Legs to stand on to make a Table (Figure 3a). On the other hand, an independency relationship exists between the front legs and the back legs of the Table (Figure 3b). The Table may have FrontLegs or BackLegs, or both FrontLegs and BackLegs at once. In Figure 3c, which illustrates a decision structure with objects, unlike with the dependent and independent structures, the pattern of the OMap is always different than the pattern of the TMap, since only one object is chosen to represent its parent for a given instance.

It can be shown that a system defined with these structures results in properties that support real-time distributed environments. Each system is event interrupt driven. Each object is traceable, reconfigurable, and has a unique priority. Independencies and dependencies can readily be detected and used to determine where parallel and distributed processing is most beneficial. With these properties, a system is defined from the very beginning to inherently maximize its own flexibility to change and the unpredictable

and to capitalize on its own parallelism.

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. Non-primitive structures can be created for asynchronous, synchronous, and interrupt scenarios used in real-time, distributed systems. Similarly, retrieval and query structures can be defined for client-server database management systems. Non-primitive structures can be defined for both FMaps and TMaps.

CoInclude is an example of a system pattern that happens often (Figure 4a). Its FMap was defined with primitive structures. Within the CoInclude pattern, everything stays the same for each use except for the functions at leaf nodes A and B. The CoInclude pattern can be defined as a non-primitive structure in terms of more primitive structures with the use of the concept of defined structures. This concept was created for defining reusable patterns.

Included with each structure definition is the definition of the syntax for its use (Figure 4b). Its use (Figure 4c) provides a "hidden repeat" of the entire system as defined but explicitly shows only the elements that are subject to change (that is, functions A and B). The CoInclude structure is used in a similar way to the Include structure except with the CoInclude the user has more flexibility with respect to repeated use, ordering and selection of objects. Each defined structure has rules associated with it for its use just as with the primitive control structures. Rules for the non-primitives are inherited ultimately from the rules of the primitives.

Async (Figure 5) is a real-time, distributed, communicating structure with both asynchronous and synchronous behavior. The Async system was defined with the primitive Or, Include, and Join structures and the CoInclude user-defined structure. It cannot be further decomposed, since each of its lowest level functions is either a primitive function on a previously defined type (see Identify2:Any and Clone1:Any under End, each of which is a primitive operation on any type), recursive (see Async under DoMore), or a variable function for a defined structure (see A and B under Process). If a leaf node function does not fall into any of these categories, it can be further decomposed or it can refer to an existing operation in a library or an external operation from an outside environment.

DependentRobots uses Async as a reusable where TurnAndPlan and Move are dependent, communicating, concurrent, synchronous, and asynchronous functions. The two robots in this system are working together to perform a task such as building a table. Here one phase of the planning robot, RB, is coordinated with the next phase of the slave robot, RA.

Reusability can be used within a TMap model by using parameterized types. A parameterized type is a defined structure that provides the mechanism to define a TMap without its particular relations being explicitly defined. TMap Table (Figure 2) uses a set of default parameterized types. Table as a parent type controls its children types, Top and Legs, in terms of a TupleOf parameterized type, Legs controls its child, Leg, in terms of OSetOf, and Wood controls Hard and Soft with a OneOf. A TupleOf is a collection of a fixed number of possibly different types of objects, OSetOf is a collection of a variable number of the same type of objects (in a linear order), and OneOf is a classification of possibly different types of objects from which one object is selected to represent the class. These parameterized types, along with TreeOf, can be used for designing any kind of TMap. TreeOf is a collection of the same type of objects ordered using a tree indexing system. With the use of mechanisms such as defined structures, a system is defined from the very beginning to inherently maximize the potential for its own reuse.

FMaps, TMaps and Their Integration

Figure 6 shows a complete system definition for a manufacturing company, which has been defined in terms of an integrated set of FMap(s) and TMap(s). This company could be set up to build tables with the help of robots to perform tasks using structures such as those defined

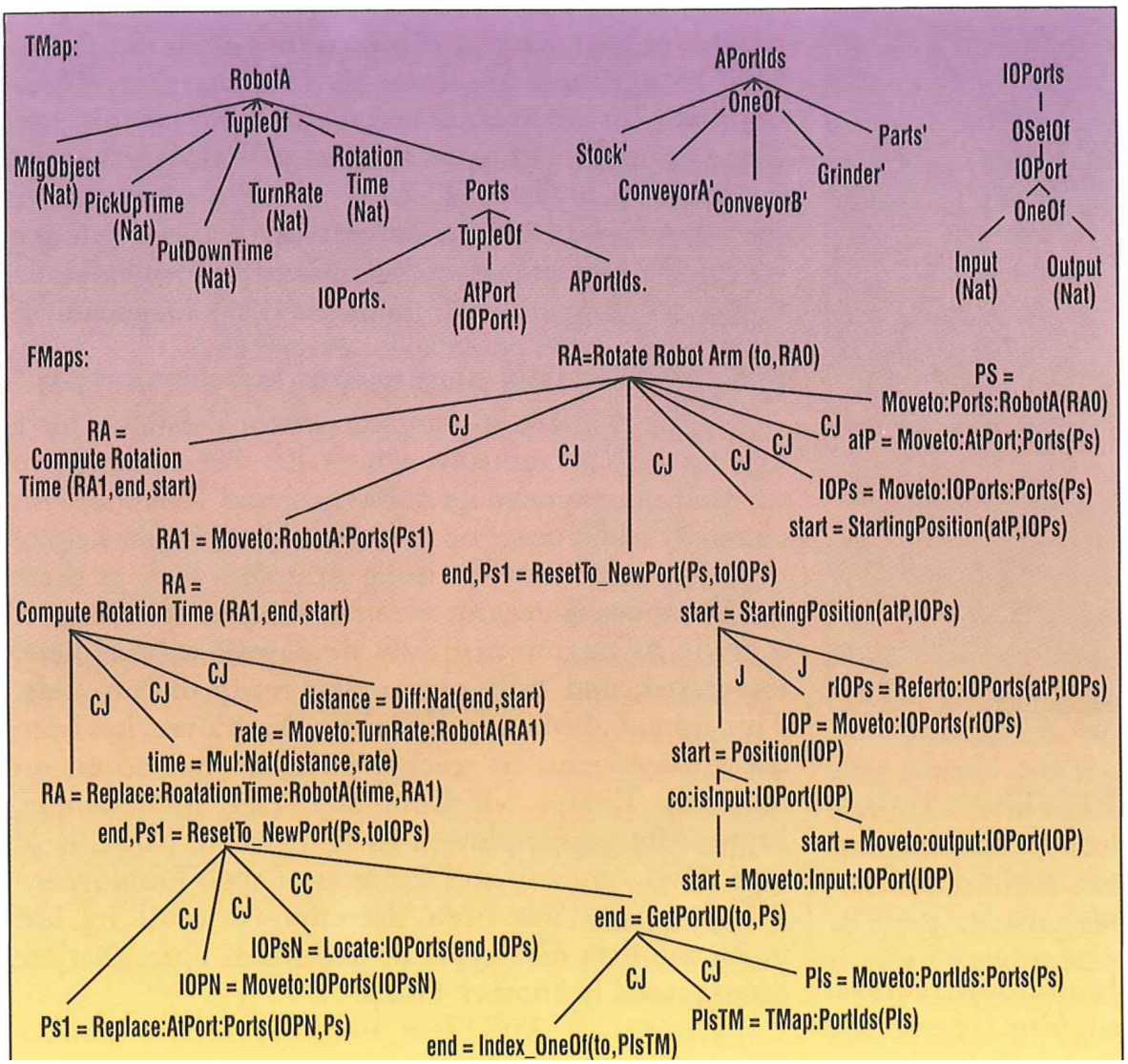

Figure 7. Any kind of system can be defined with this language, including software, hardware, and peopleware. RotateRotateArm is an example of a hardware system defined in FMaps and TMaps.

above. Since this system is completely defined, it is ready to be automatically developed to complete, integrated, and fully production-ready run code. This system's FMap, Is_FullTime_Employee, has been decomposed until it reaches primitive operations on types in TMap, MfgCompany. (See for example, Emps=Moveto:Employees(MfgC) where MfgC is of type MfgCompany and Emps is of type Employees.) MfgCompany has been decomposed until its leaf nodes are primitive types or defined as types that are decomposed in another TMap.

System, Is_FullTime_Employee, uses objects defined by TMap, MfgCompany, to check to see if an employee is full or part time. First, a move is made from the MfgCompany type object, MfgC, to an Employees type object, Emps. The defined structure, LocateUsing:Name, finds an Employee based on a name. Once found, a move is made from Employee, Emp, to PS of type, Payscale. The primitive operation YN=is:FullTime(PS) is then used to determine from PS if Emp is full time or part time.

Each parameterized type assumes its own set of possible relations for its parent and children types. In this example, TMap, MfgCompany is decomposed into Departments and Employees in terms ofTupleOf. Departments is also decomposed in terms of TupleOf into Purchasing, Production and Marketing. Employees is decom-

posed in terms of OSetOf. One of the children of Employee, PayScale, is decomposed in terms of the parameterized type, OneOf.

Abstract types decomposed with the same parameterized type on a TMap inherit (or reuse) the same primitive operations and therefore the same behavior. So, for example, MfgCompany, Departments, and Employee inherit the same primitive operations from parameterized type, TupleOf. An example of this can be seen in the FMap where both types, MfgCompany and Employee, use the primitive operation, MoveTo which was inherited from TupleOf.

Here each use of the MoveTo is an instantiation of the Child = MoveTo:Child(Parent) operation of the TupleOf parameterized type. For example, Emps=MoveTo:Employees(MfgC) allows one to navigate to an employee's object from a MfgCompany object. A type may be non-primitive (e.g., Departments), primitive (e.g., FullTime as a rational number), or a definition that is defined in another type subtree (for example, Employees). When a leaf node type has the name of another type subtree, either the child object will be contained in the place holder controlled by the parent

object (defined as, such as with Skills) or a reference to an external object will be contained in the child place holder controlled by the parent object (forming a relation between the parent and the external object).

Universal Primitive Operations

The TMap provides universal primitive operations, which are used for controlling objects and object states that are inherited by all types. They create, destroy, copy, reference, move, access a value, detect, and recover from errors and access the type of an object. They provide 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.

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.

The same types of definition mechanisms are used to define RotateRotateArm, a hardware system (Figure 7), as the software system above. Note that this system also includes the use of primitives for numeric calculation. In this system, the rotation of the robot arm is calculated to move from one position to another in a manufacturing cell to transfer a part. In this example the universal operation (an example of another form of reusable), Replace, is used twice. Each use of a universal operation has function qualifiers that select a unique TMap parent–child combination to be used during the application of the function.

Figure 8 has a definition that takes further advantage of the expressive power of a TMap with the option of using explicitly defined relations. In this example, a stack of bearings is described. A bearing in the stack may be under (with relation on-0) or on (with relation on-1) another bearing object in the stack as defined by the DSetOf structured type. A bearing object is decomposed into a Cap, a RetainerWithBalls, and a Base. Object relationships at this level show that the Cap is above the RetainerWithBalls, which is in turn above the Base. Further detail reveals that a Retainer has (with the has-n relation) some number of RetainerHoleWithBall. The set of RetainerHoleWithBall objects are independent of each other, defined by the ISetOf structured type. This structure allows for physically independent relations on the objects in the set. Here, different portions of the Cap surface are independently related (with the on-Balls relation) to each individual Ball object (with the on-Ball relation).

As experience is gained with different types of applications, new reusables emerge for general or specific use. For example, a set of reusables

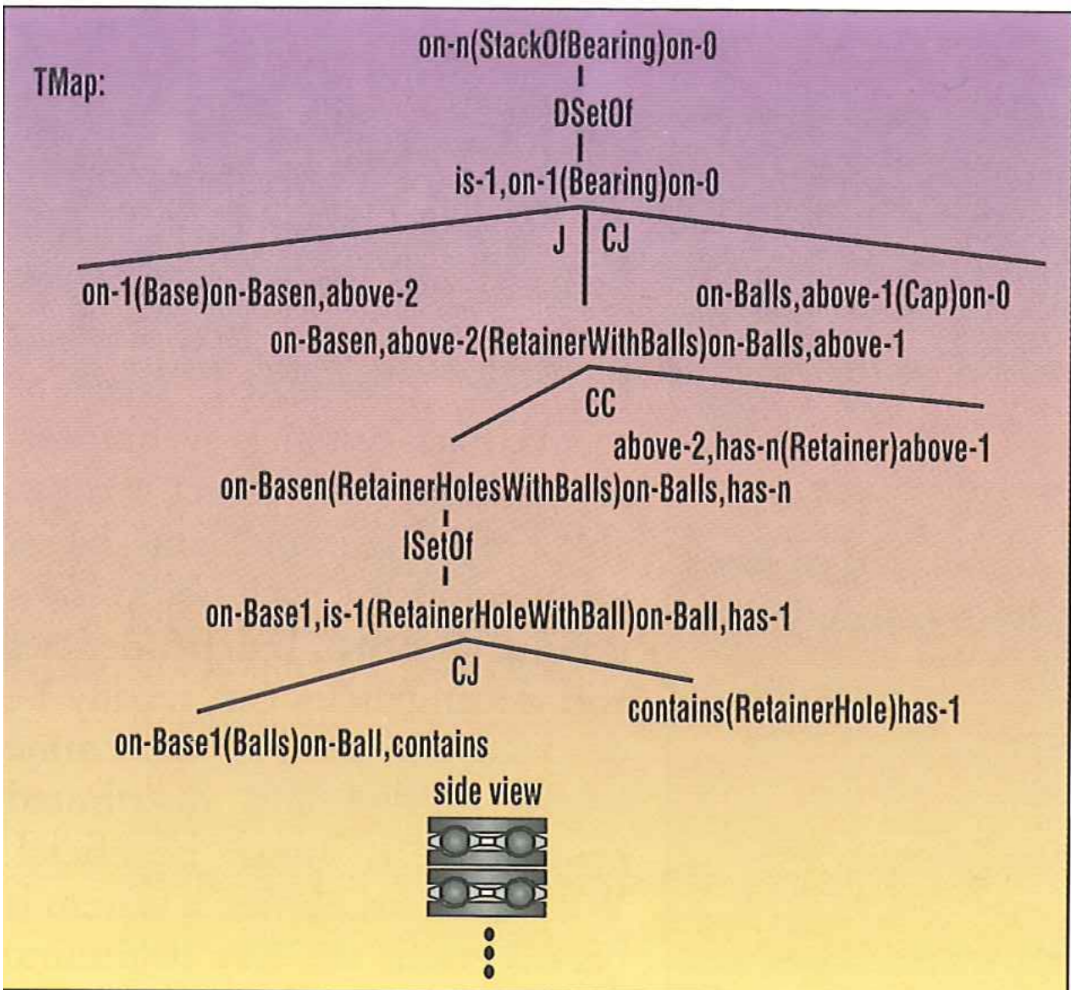

Figure 8. Explicitly defined relations can be used to take further advantage of the expressive power of a TMap. Here, a TMap is used for defining a bearing manufacturing process.

has been derived to form a higher level set of mechanisms for defining hierarchies of interruptible, 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 that can be represented using a graphical syntax (Figure 9).

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. In this example, the robot controller may apply an arm controller or a sensor controller. If the arm controller is activated, the two grippers may concurrently, using an Include, hold two ends of some object. If the sensor controller is activated, a sensor unit senses some image followed, using a Join, by an image unit matcher. These reusables can also be used to manage other types of processes such as those used to manage a software development environment.

Performance Considerations

When designing a system environment, 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, with development before the fact, the same language is used to define functional, resource, and allocation architectures. The meta-

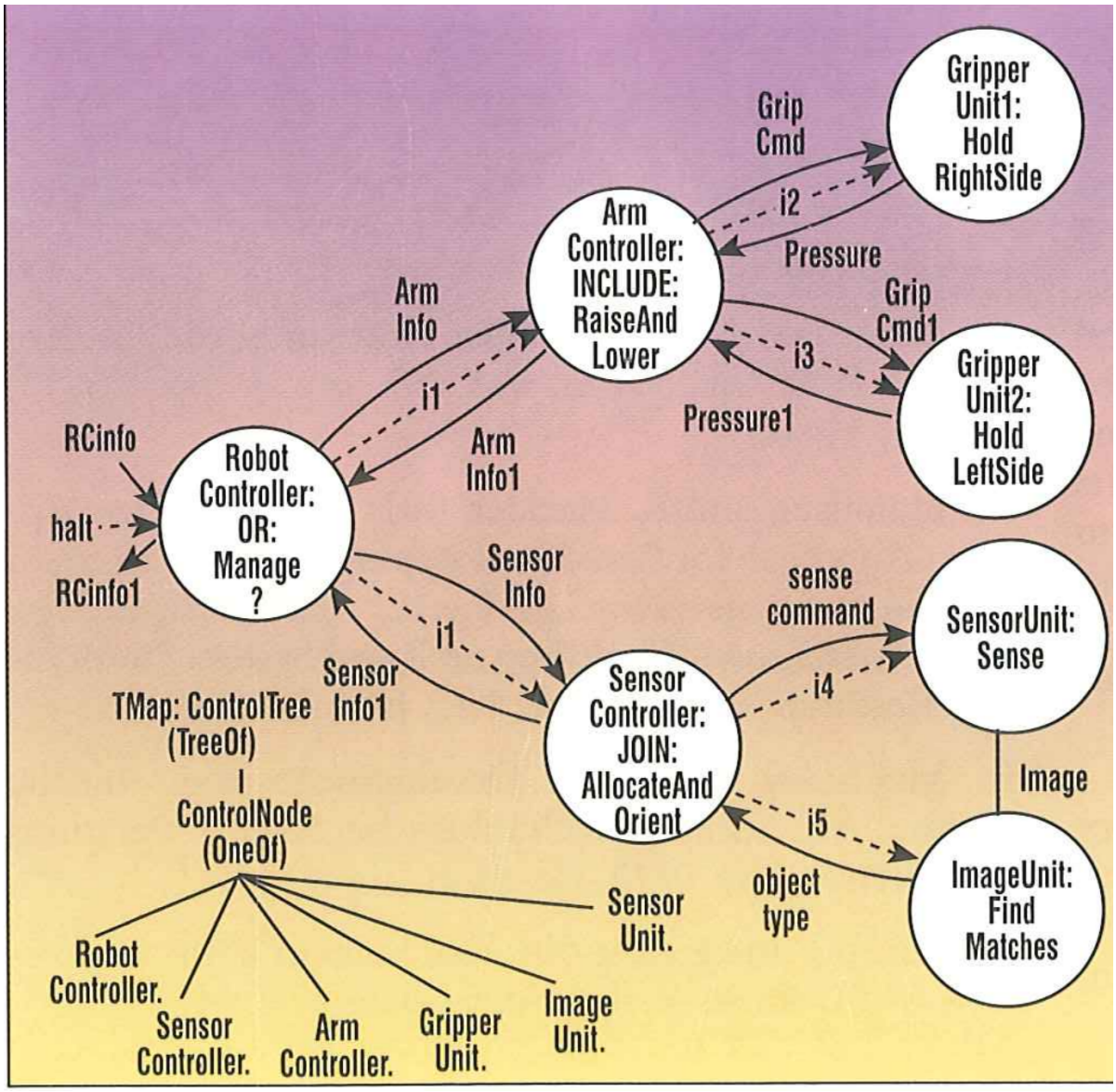

Figure 9. A second order control system has been derived that parallels the primary control system to form a powerful set of reusables for defining hierarchies of interruptible, asynchronous, communicating, distributed controllers.

language properties of the language can be used to define global and local constraints for both FMaps and TMaps. Constraints, themselves, can be defined in terms of FMaps and TMaps. If we place a constraint on the definition of a function (for example, Where RobotA takes between 2 and 5 seconds), then this constraint influences all other functions that use this definition. Such a constraint is global with respect to the uses of the original function.

Global constraints of a definition may be further constrained by local constraints placed in the context of the definition that uses the original function definition (for example, where function RobotB uses F Where F takes 3 seconds). Function F could have a default constraint which holds for all uses such as Where Default:3 seconds. If, however, RobotB is defined to take 2 seconds, then RobotB overrides F. The validity of constraints and their interaction with other constraints can be analyzed by either static or dynamic means. The property of being able to trace an object throughout a definition supports this type of analysis. This property 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 effects of functional interactions of constraints.

Function-Oriented, Object-Oriented

A development before the fact system is by its very nature an integration of being function oriented and being object oriented from the beginning. The definition space is a set of real-world objects, defined in terms of FMaps and TMaps.

Objects, instantiations of TMaps, are realized in terms of OMaps. An execution, an instantiation of an FMap, is realized in terms of an EMap. Building block definitions that focus more on objects than on functions are independent of particular object-oriented implementations. Properties of classical object-oriented systems such as inheritance, encapsulation, polymorphism, and persistence are supported with the use of generalized functions on OMaps and TMaps.

The development before the fact approach derives from the combination of steps taken to solve the problems of the traditional "after the fact approach." Collective experience strongly confirms that quality and productivity increase with the increased use of development before the fact properties. A major factor is the inherent reuse in these systems, culminating in ultimate reuse, which is automation itself.

From FMaps and TMaps any kind of software system can be automatically developed, resulting in complete, integrated, and production-ready target system code and documentation. This is accomplished by the 001 Tool Suite, an automation of the technology that will be discussed in an article in the June 13 issue of this *Software Engineering* Supplement of *Electronic Design*. The tool suite also can observe the behavior of a system as it is being evolved and executed in terms of OMaps and EMaps. The article will discuss some of the systems that have been designed and developed with this paradigm in a wide range of environments, including manufacturing, aerospace, software tool development, database management, transaction processing, process control, simulation, and domain analysis. ES

References

M. Hamilton, "Zero-Defect Software: the Elusive Goal," *IEEE Spectrum*, vol. 23, no. 3, pp. 48–53, March 1986.

M. Hamilton and R. Hackler, "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, June 4, 1990.

B. McCauley, "Software Development Tools in the 1990s," *AIS Security Technology for Space Operations Conference*, July 1993, Houston, Texas.

B. Krut, Jr., "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.

Software Engineering Tools Experiment—Final Report, Vols. 1, Experiment Summary, Table 1, Page 9, Department of Defense, Strategic Defense Initiative, Washington, D.C., 20301-7100.

Margaret H. Hamilton is CEO of Hamilton Technologies Inc. (HTI), Cambridge, Mass., which provides systems engineering and software development products. Before this, she was CEO of Higher Order Software, responsible for the development of the first comprehensive CASE tool. Earlier, as the head of software engineering at MIT's Draper Lab, she was the director of the Apollo on-board flight software project and created Higher Order Software, a systems design theory.