001: A Rapid Development Approach for Rapid Prototyping Based on a System that Supports its Own Life Cycle

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

I. INTRODUCTION

The need for rapid prototyping techniques arises because there are no conventional techniques for rapid development. With experience, it becomes clear that the reason conventional techniques cannot be used to deliver rapid systems is because they force the users and developers to concentrate on fixing wrong things up rather than on doing them right in the first place. Problems are cured rather than prevented. Every system is defined with a technology which expects a development to be based on "alchemy" and trial and error. When critical issues are dealt with too late or "after the fact", both the quality of a system and the productivity in producing it become unacceptably low. The result is that system integrity is reduced at best. There is a compromise in functionality. Deadlines are missed. Time and dollars are wasted. The competitive edge is lost and opportunities are gone.

If it were possible to develop software based systems with a marked improvement in productivity, there would not be a need to differentiate between methods for rapid prototyping and rapid development. To make this happen, however, it will be necessary to resolve what some believe to be the highest priority issues surrounding the conventional development process. Take, for example, integration. *Integration happens too late into the development process.* During the requirements process, data flow is defined using one method, state transitions another, dynamics another and data structure using still another method. Once these aspects of requirements are defined, there is no way to integrate them. Integration 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 no correspondence to the real world.

There are several other examples: *Errors are eliminated too late.* It is the accepted practice that a system be defined from the beginning to be ambiguous and incorrect. As a result, interfaces are incompatible and there is a propagation of errors throughout development. The system and its development are out of control. Once again the developers inherit the problem. Similarly, *flexibility and the ability to handle the unpredictable are issues that are dealt with too late.* Systems are not defined at the beginning to handle changes or recover from errors. As a result, porting is a new development for each new hardware, operating system or implementation language configuration; critical functionality is avoided for fear of the unknown, and maintenance is the most expensive part of the life cycle. *Preparing for distributed environments happens too late.* Systems are first defined and developed for a single processor environment. They are then re-implemented for a distributed environment. The result is at least one unnecessary development process of the system. *Reusability happens too late.* There are no properties in system definitions to help find, create and use commonality. The result is that redundancy is a way of life. Errors propagate accordingly. *Automation happens too late.* A definition is given to developers to turn manually into code. As a result, new errors are created and a process that could have been mechanized is performed over and over again. *Run-time performance analysis* (the process whereby decisions are made between algorithms or between architectures) *takes place too late.* Insufficient information is used to define system performance. A system is defined without consideration of how to separate a system from the environment in which it operates. The result is that design and implementation decisions depend on analysis of after the fact and ad hoc implementation results. *Design integrity is considered too late.* More often than not, a system design is based on short term considerations. As a result, development is driven in the direction of failure.

II. THE SOLUTION

The solution to the problems discussed above is an approach which expects each development to be based on predictive discipline and an integrated system science. An example is a *Development Before the Fact Approach* which dictates that each system be defined with properties in its definition which support its own development throughout its life cycle. With such an approach, many of the tasks associated with the traditional sequential "waterfall" development process could be performed in parallel. A system would inherently integrate and make understandable its own real world definitions; maximize its own reliability and predictability and its own flexibility to change and the unpredictable; capitalize on its own parallelism; maximize the potential for its own reuse and automation; support its own run-time performance analysis and the ability to understand the integrity of its own design principles. Because of these properties systems would be developed with *built-in quality* and *built-in productivity* assurance. The result would be unprecedented productivity in a system's development.

A language for this approach would have the capability to define any aspect of any system, including its functional architectures, resource architectures and resource allocation architectures throughout all of its levels and layers of definition.[1] Such a language can always be considered a *design* language, since design is a relative term. One person's design phase is another person's implementation phase. One person designs requirements. Another designs specifications. Still another designs code. If this language is truly successful in one phase as a design language, it should be successful in any phase This language would have mechanisms to define mechanisms for defining systems. Although the core language is generic, the user "language", resulting from the use of this mechanism definition aspect for a particular user, can be application specific, since the language is semantics-dependent but syntax- independent. The use of the collective set of mechanisms, so defined, defines each object with respect to its interrelationships to other objects for a given system. As such, using this language to define a system assures that the system will be object-oriented throughout its development, starting at the very beginning of its requirements phase. Every system defined with the use of this language would have built-in properties for supporting its own development.

[1] The *functional architecture* defines what it is the user of a target system wants to do. Associated *resource architectures* define the potential execution environments for the functional architecture. The *resource allocation architecture* is the system that maps the functional architecture to one of a possible set of resource architectures.

The steps for building a Development Before the Fact system would be as follows. First, a model[2] is defined with the language for defining a system with built-in development properties. Next, the model is automatically analyzed to ensure that it was defined properly. A software implementation is then automatically generated consistent with the model. The resulting system is executed. This step is followed by building the real system. If the real system is software, this last step may not be necessary, since a final evolution of the software developed to simulate the model could become the real system.

We will discuss next a technology, 001, which embodies many aspects of this approach in terms of its abilities to approach a solution for each of the problems discussed above. The reason 001 lends itself to the rapid prototyping of a system is that it has been designed, developed and used for the rapid development of systems.

III. 001: A DEVELOPMENT BEFORE THE FACT APPROACH

001 had its beginnings at the time of APOLLO 11 when Hamilton and her staff analyzed the on-board flight software in order to find a way to minimize errors. The result was a theory for defining systems free of interface errors. The first implementation of this theory was a technology which concentrated on defining and developing reliable systems in terms of functional hierarchies [1]. The technology has since evolved into an integrated hierarchical functional and object oriented network technique based upon a unique concept of control [2,3]. Every system defined and developed with 001 and its associated automated tool suite has properties of Development Before the Fact. The 001 tool suite has been defined and generated with itself.

A. Modeling Environment

The 001 modeling environment makes the assumption that 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 systems. The new system becomes a reliable system for building other systems. The building blocks (Figure 1) include *types* of objects; *functions* whose inputs and outputs are members of these types; *structures* which relate a parent type[3] to its children types and a parent function to its children functions; *relations* which link types to types and *constraints* which define the external boundaries within which a system may reside.

Every model is defined in terms of functional hierarchies (FMaps) and type hierarchies (TMaps). All model viewpoints can be obtained from FMaps and TMaps including data flow, control flow, state transitions, data structure and dynamics. On an FMap there is a function at each node. For example, the function, *drive the truck*, could be decomposed into and control its children functions, *turn on engine* and *turn wheels* ; *turn wheels* could have children, *turn wheels left* and *turn wheels right*. On a TMap there is a type at each node. For

[2] A *model* is a tentative definition of a system or theory that accounts for all of its known properties.

[3] Every function on a functional hierarchy is a parent which controls its children functions on the next most immediate lower level nodes. Every type on a type hierarchy is a parent which controls its children types on the next most immediate lower level nodes.

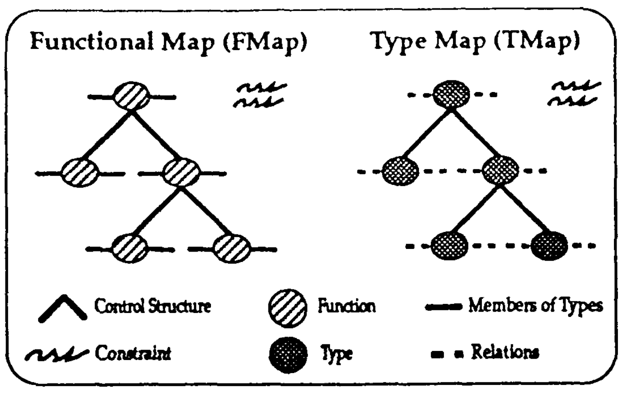


Figure 1: Reliable Building Blocks

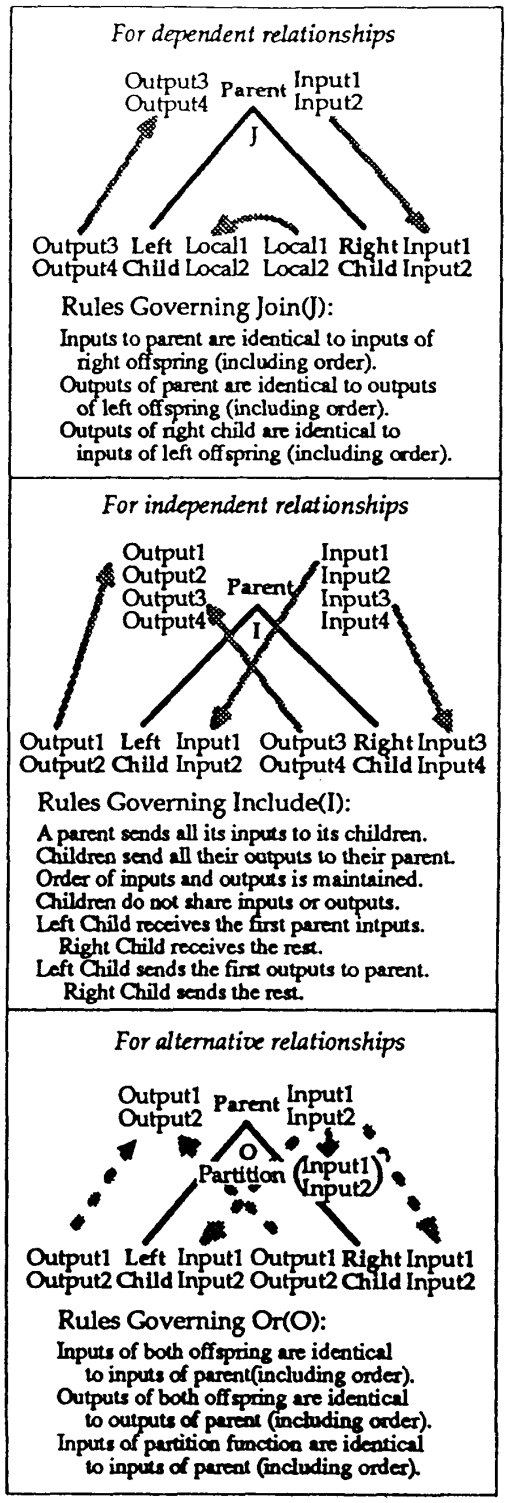
example, type *truck* could be decomposed into and control its children types,*wheels* and *engine*. A higher level function is defined on an FMap in terms of its relationships to lower level functions. A higher level type is defined on a TMap in terms of its relationships to lower level types. For each model, FMaps are inherently integrated with TMaps. Each function on an FMap has one or more objects as its input and one or more objects as its output. FMaps are used to define, integrate, and control the transformations of objects from one state to another state (e.g., a truck with a broken wheel to a truck with a fixed wheel). Each object resides in an object hierarchy (OMap) and is a member of a type from a TMap. The bottom nodes on an FMap contain primitive operations on types which are defined in the TMap. The bottom nodes on a TMap contain primitive types. When a system has all of its object values plugged in for a particular performance pass (i.e., it is executing) it exists in the form of an execution hierarchy (EMap). With these hierarchies, all system viewpoints are integrated. A system is defined from the very beginning to inherently *integrate* and *make understandable* its own real world definitions.

B. The Primitive Control Structures

001 FMaps and TMaps are ultimately defined in terms of three primitive control structures. Structure decompositions overlay the abstract concept of control onto a network of primitive nodes where each node has input and output interconnections to other nodes. A formal set of rules is associated with each primitive structure. The three primitive structures are: the *Join(J)* for defining dependent relationships, the *Include(I)* for defining independent relationships and the *Or(O)* for defining decision-making relationships (Figure 2). With the use of the primitive structures interface errors are "removed" before the fact by preventing them in the first place. Interface errors include data flow, priority and timing errors throughout a system definition to a very fine grained level. It has been the experience of several that interface errors make up approximately 75% to 90% of the errors that are found after implementation with a conventional technique. The use of the primitive structures supports a system to be defined from the very beginning to inherently maximize its own *reliability* and *predictability*.

1) Example of the use of the primitive control structures. A use of the primitive structures is shown in the definition of the FMap for system *IndependentRobots*, which defines a system of two robots synchronized to work in parallel (Figure 3a). In *IndependentRobots*, the top function is decomposed into offspring functions *Finish* and *Continue*. It receives input, *RobotB0* and *RobotA0*, and produces output, *RobotB* and *RobotA*. The *Or* relationship between *IndependentRobots* and its offspring is one of making a decision. The left child of an *Or* is chosen when the result of a partition function is *True*. The right child is chosen when it is *False*. In this example, either *Finish* or *Continue* will be performed. The decision depends on *RobotB0* and *RobotA0*. If processing is to be discontinued, *IsFinished* will select *Finish*. If not, it will select *Continue*. Each of the offspring of *IndependentRobots* takes in the same input and produces the same output, since only one of the offspring will be performed for a given performance pass.

Continue is decomposed into functions, *IndependentRobots* and *Process*. *Continue* controls its offspring in a *Join* relationship where *IndependentRobots* depends on the output of *Process* as input. With a *Join*, a common dependency is realized by the fact that one offspring's output (in this case, *RobotBn* and *RobotAn*) is another offspring's input. *IndependentRobots* under *Continue* cannot complete its performance without *Process* since it depends on the output of *Process* to provide its input; once *IndependentRobots* is finished, it provides its output to its parent *Continue*. Since *Turn* and *Move* are controlled by *Process* as


For dependent relationships
Output3 Output4 Parent Input1 Input2
J
Output3 Output4 Left Child Local1 Local2
Local1 Local2 Right Child Input1 Input2
For independent relationships
Output1 Output2 Output3 Output4 Parent Input1 Input2 Input3 Input4
I
Output1 Output2 Left Child Input1 Input2
Output3 Output4 Right Child Input3 Input4
For alternative relationships
Output1 Output2 Parent Input1 Input2
O
Partition Input1 Input2
Output1 Output2 Left Child Input1 Input2
Output1 Output2 Right Child Input1 Input2

Rules Governing Join(J):

Inputs to parent are identical to inputs of right offspring (including order).
Outputs of parent are identical to outputs of left offspring (including order).
Outputs of right child are identical to inputs of left offspring (including order).

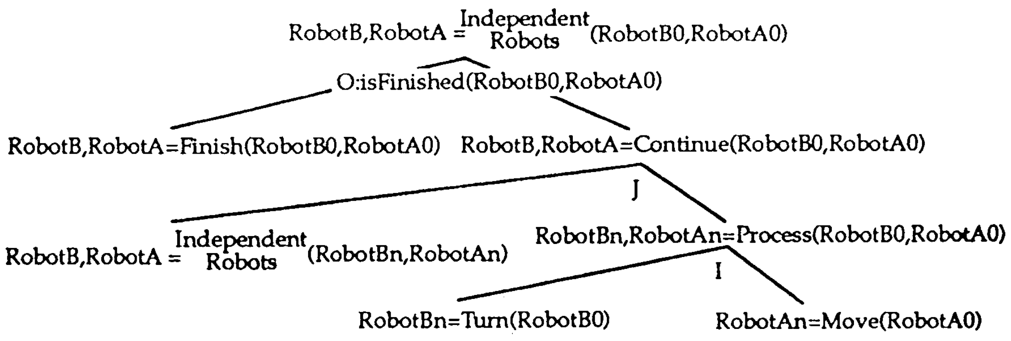
Rules Governing Include(I):

A parent sends all its inputs to its children.
Children send all their outputs to their parent.
Order of inputs and outputs is maintained.
Children do not share inputs or outputs.
Left Child receives the first parent intputs.
Right Child receives the rest.
Left Child sends the first outputs to parent.
Right Child sends the rest.

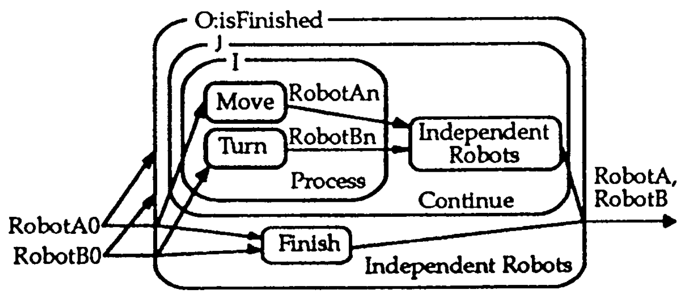
Rules Governing Or(O):

Inputs of both offspring are identical to inputs of parent(including order).
Outputs of both offspring are identical to outputs of parent (including order).
Inputs of partition function are identical to inputs of parent (including order).

Figure 2: The Three Primitive Control Structures.


RobotB,RobotA = Independent Robots (RobotB0,RobotA0)
O:isFinished(RobotB0,RobotA0)
RobotB,RobotA=Finish(RobotB0,RobotA0)
RobotB,RobotA=Continue(RobotB0,RobotA0)
J
RobotB,RobotA = Independent Robots (RobotBn,RobotAn)
RobotBn,RobotAn=Process(RobotB0,RobotA0)
I
RobotBn=Turn(RobotB0)
RobotAn=Move(RobotA0)

a. Control Flow Oriented Graphics



b. Controlled Data Flow Oriented Graphics

Figure 3: Use of the Primitive Structures in a Functional Hierarchy.

independent functions with an *Include*, each takes its own input from its parent. Thus, for example, *Turn* takes in its own input *RobotB0* directly from its parent and produces its own output *RobotBn*, giving it directly to its parent *Process*. *Move*, likewise, takes its own input directly from its parent and produces its own output. In this relationship between the parent and its offspring, the offspring do not communicate with each other. Notice in the use of each primitive structure that inputs are traceable down and/or across (in this example from the right child to the left child) a hierarchy and outputs are traceable across and/or up a hierarchy.

IndependentRobots, a recursive function, controls *continue* which controls *IndependentRobots*. Recursion in a map is simply a repetition of the map inside of itself; it is a shorthand notation for indicating that the lowest *IndependentRobots* in the hierarchy has the same definition as its ancestor; when instantiated, this definition is plugged in at the leaf function and the control hierarchy is extended downwards during execution.

Although the *IndependentRobots* example is shown in terms of a formal graphics tree format which emphasizes control flow, 001 does not dictate a particular syntactic form. It can be defined in some other graphics form, such as controlled state diagrams or controlled data flow diagrams (see, for example, a definition of the same system which emphasizes controlled data flow, in Figure 3b). It can also be defined in textual form, such as in an algebraic or natural language. Any syntax can be used as long as the semantic rules of the technology are adhered to. In all these forms the same information is presented, but that which is highlighted varies.

2. *Real-time distributed properties of the primitive control structures.* Although the primitive control structures were derived from a theory which was intended for defining systems to be free of interface errors, we discovered later that a hierarchy defined with the primitive control structures also results in properties for supporting real-time parallel environments. Each 001 system is event interrupt driven based on its functional architecture relationships and demand driven based on its resource architecture relationships. An object is therefore always both demand driven and event driven at the same time. Primitive functions are available for activation when all of their input events are available. An abstract function has a lifetime that contains the lifetimes of its immediate children.

Each function in a hierarchy has a unique priority; the existence of an event can cause a system to automatically reconfigure and execute a higher priority function. If, in the use of the *Include* structure, the left offspring always has a higher priority than the right offspring in system *IndependentRobots*, both offspring can independently begin when each of their inputs becomes available if there are two processors. If one were to simulate the behavior of two robots, failure of *Turn's* processor forces an interrupt of *Move's* processor if *Move* is processing; *Turn* can starve *Move* of its processor resources; if *Move* is not processing, then if *RobotA0* becomes available, *Move* must wait until *Turn* is finished processing. Given one processor, if neither offspring has been initiated, *Turn* can initiate when *RobotB0* becomes available or *Move* can initiate when *RobotA0* becomes available; if both arrive simultaneously, *Turn* initiates before *Move*. Concurrent processing can also take place within a *Join* structure when more than one input is being processed. In this example, the decision function *IndependentRobots* uses the function *IsFinished* to decide whether to *Finish* or *Continue*. Since *IsFinished* needs both its inputs to execute, the decision is blocked until both inputs are available.

C. Reusability with Defined Structures

Any system could be defined completely in terms of only primitive structures, but there is often a desire to use less primitive structures to accelerate the process of defining and understanding a system. The degree to which a system has symmetry or asymmetry determines the amount of reusable patterns that can be derived in terms of structures. Non-primitive structures can be defined in terms of the primitive structures or in terms of other non-primitive structures. *Coinclude* is an example of a system hierarchy pattern which happens often when using primitive structures (Figure 4a). Its FMap was defined with primitive structures, *Include* and *Join*. Within the *Coinclude* pattern, *A* and *B* are the only leaf node functions that change. 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 which are subject to change (i.e., functions *A* and *B*). The *Coinclude*

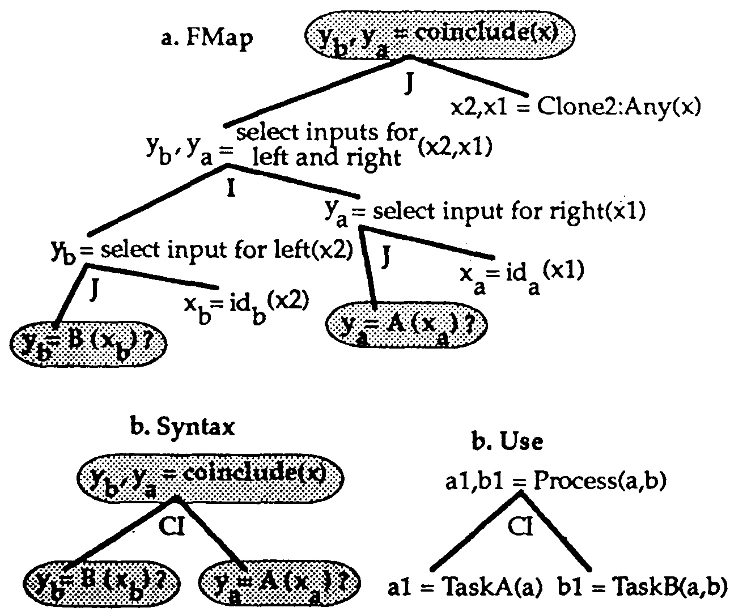
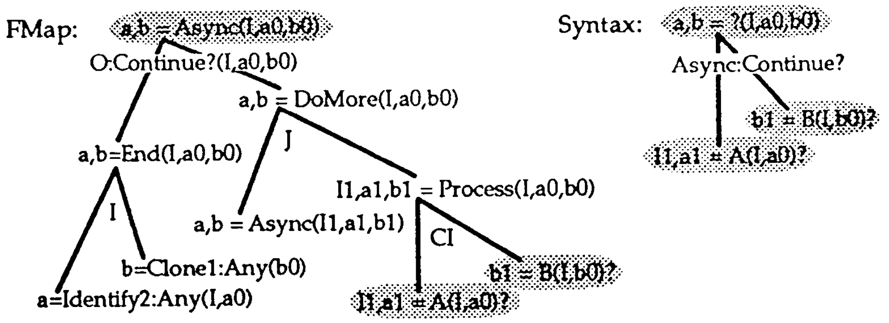


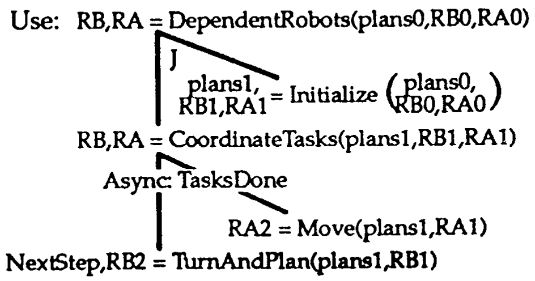
Figure 4: Co-Include Structure Definition.

structure is used in a similar way to an *Include* structure except with the *Coinclude* the user has more flexibility with respect to repeated use of an object state, ordering of objects 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 derived from the rules of the primitives. 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*.

Async, shown in Figure 5, is a real-time, communicating, concurrent, asynchronous structure. The *Async* system was defined (Figure 5a) with the primitive *Or, Include* and *Join* structures and the *Coinclude* non-primitive structure. It cannot be further decomposed, since each of its lowest level functions is either a primitive function or 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. A use of *Async* is shown in Figure 5b. Whereas *Turn* and *Move* are independent functions in system, *IndependentRobots*, they are dependent, communicating, concurrent and asynchronous functions in system, *DependentRobots*. In this example the *Turn* function has been updated to incorporate a planning capability into part of *RobotB0's* function, *TurnAndPlan*, so that the two robots could work together to perform a more complex task. Here one phase of the planning robot is coordinated with the next phase of the slave robot, *RA2*.



a. Definition of Async Structure.



Use: RB,RA = DependentRobots(plans0,RB0,RA0)
J
plans1, RB1,RA1 = Initialize (plans0, RB0,RA0)
RB,RA = CoordinateTasks(plans1,RB1,RA1)
Async: TasksDone
RA2 = Move(plans1,RA1)
NextStep,RB2 = TurnAndPlan(plans1,RB1)

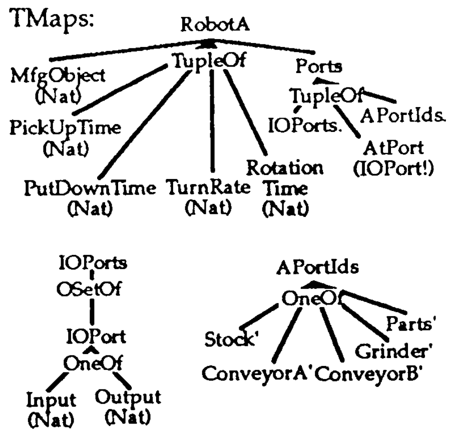
b. Use of Async Structure.

Figure 5: An Async Structure and Its Use.

D. Definition of Objects with Parameterized Types

Reusability can also be used within a TMap model by using parameterized types. A *parameterized type* is a defined structure which provides the mechanism to define a TMap without its particular relations being explicitly defined. Each parameterized type assumes its own set of possible relations for its parent and children types. TMap, *RobotA*, part of a simulator definition (Figure 6), is decomposed in terms of parameterized type, *TupleOf*, into its offspring types *Ports, RotationTime, TurnRate, PutDownTime, PickUpTime and MfgObject; Ports* in terms of *TupleOf, IOPorts* in terms of *OSetOf,* and *APortIds* in terms of *OneOf.* A *TupleOf* is a collection of a fixed number of possibly different types of objects; An *OSetOf* is a collection of a variable number of the same type of objects (in a linear order); a *OneOf* is a classification of object types of different types from which one type is selected. Each parameterized type has a set of primitive operations associated with it for its use. Abstract types decomposed with the same parameterized type on a TMap inherit the same primitive operations. FMap, *StartingPosition* (Figure 6b) uses the *Moveto:Output:IOPort* operation (see Figure 6d) (an instantiation of the *Moveto:Child:Parent* operation of the *OneOf* parameterized type in Figure 6c) with abstract type, *IOPort. Is:Input:IOPort* (an instantiation of the *is:Child:Parent* of *OneOf*) is used to determine which class of *IOPort* object is to be processed.

A type may be non-primitive (e.g., *Ports*), primitive (e.g., *PickUpTime* as a natural number), a reference to a type outside of its parent domain (e.g., *IOPorts*), a



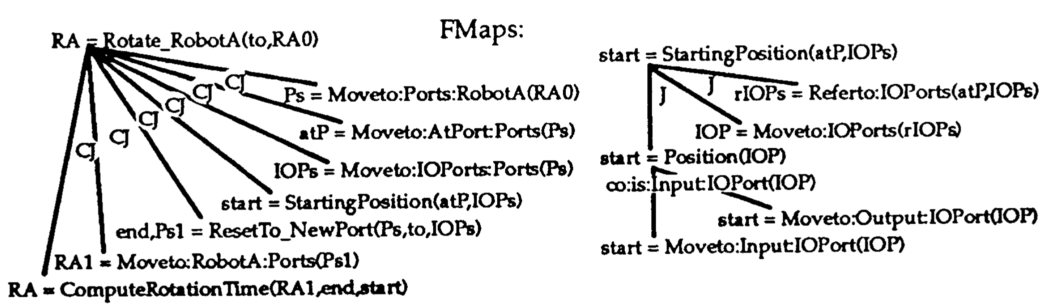
a. Type Inheritance and Decomposition.

Type: OneOf: Parent,Child.
Primitive Operations:
{Abstract | Primitive | Value}
Type = K:Child:Type(TMap);
Type = Copy:Type(Type);
{Abstract}
Type = Moveto:Child:Type(Type);
Boolean = is:Child:Type(Type);
...

c. Generic Parameterized Type.

Type: IOPort.
Primitive Operations:
IOPort = K:Input:IOPort(TMap);
IOPort = K:Output:IOPort(TMap);
IOPort = Copy:IOPort(IOPort);
Nat = Moveto:Input:IOPort(IOPort);
Nat = Moveto:Output:IOPort(IOPort);
Boolean = is:Input:IOPort(IOPort);
Boolean = is:Output:IOPort(IOPort);
...

d. A OneOf Abstract Type Instance.


FMaps:
RA = Rotate_RobotA(to,RA0)
CJ
Ps = Moveto:Ports:RobotA(RA0)
atP = Moveto:AtPort:Ports(Ps)
IOPs = Moveto:IOPorts:Ports(Ps)
start = StartingPosition(atP,IOPs)
end,Ps1 = ResetTo_NewPort(Ps,to,IOPs)
RA1 = Moveto:RobotA:Ports(Ps1)
RA = ComputeRotationTime(RA1,end,start)
start = StartingPosition(atP,IOPs)
J
rIOPs = Referto:IOPorts(atP,IOPs)
IOP = Moveto:IOPorts(rIOPs)
start = Position(IOP)
co:is:Input:IOPort(IOP)
start = Moveto:Output:IOPort(IOP)
start = Moveto:Input:IOPort(IOP)

b. Functional Specification:

Figure 6: Excerpts of a Simulation Implementation of Primitive Operation: RobotA = Rotate:RobotA(APortId,RobotA).

reference to a type which is defined elsewhere but is still part of its parent's domain or recursive. Each parent on a TMap and its children are used as parameters to a parameterized type that is used to decompose that parent into its children. *IOPort* therefore, inherits all of the operations of *OneOf*.

Copy (Figure 6c), a primitive operation associated with all parameterized types, is a universal primitive operation. The universal primitive operations are used for controlling objects and object states. 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.

The TMap properties ensure the proper use of a TMap by an FMap. A TMap has a corresponding set of control properties for controlling spatial relationships between objects. One cannot, for example, put an object into an object structure where an object already exists (one cannot put a wheel on a truck where a wheel already exists); conversely, one cannot remove an object from a structure where there is no object; 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.

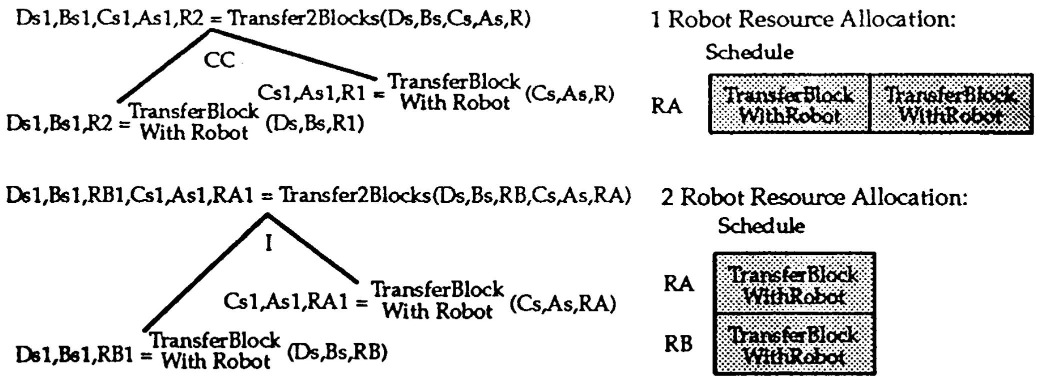
E. Constraint Specification and Analysis

Constraints can be defined for both FMaps and TMaps. The meta-language properties of 001 can be used to define global and local constraints. If we place a constraint on the definition of a function (e.g., *Where F takes between 2 and 5 seconds*), then this constraint will influence 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 which uses the original function definition (for example, where function *B* uses *F Where F takes 3 seconds).* Function *F* could have a default constraint which holds for all uses such as *Where Default:3 secs.* If however, *B* is defined to take 2 seconds , then *B* overrides *F.* The validity of constraints and their interaction with other constraints can be analyzed by either static or dynamic means. The 001 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.

F. Resource Architecture and Resource Allocation Definitions and Their Relationship to Run-Time Performance Analysis.

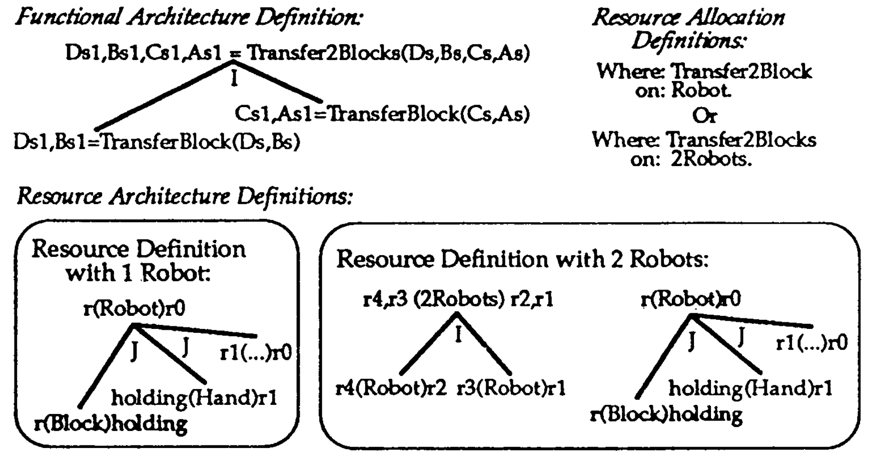
Having a technology is one thing. Using it most effectively is another. Take, for example the process of creating a complete system definition. There are many ways to define a system. Figure 7 shows three different definitions for system *Transfer2Blocks.* The first two definitions are architecture dependent. The third one is not. The first system *Transfer2Blocks* implicitly assumes 1 Robot as its resource. If more robots become available, the functional architecture with it's implicit resource architecture and resource allocation architectures would have to be changed to take advantage of its new resources. Such is the case with the second model which was defined to implement the system with 2 Robots. It will not work with one robot or any set of robots greater than two. In both the first and second models, a change from one architecture version of the system to another affects the input and output list of every function. There is also a structural difference in the FMaps for the two implicit resource allocations of the first two models. One has a *CC* control structure and the other has a *I* control structure.

The third system model provides a solution that is more flexible to changing requirements than the first and second models, since changes made to the third model based on changing user needs do not affect other architecture models. The functional architecture definition can remain the same for any resource architecture. The alternate resource architecture models can also remain unchanged to perform the user's requirements. A resource allocation architecture definition defines the way in which the resources of the resource architecture are applied to the functional



Ds1,Bs1,Cs1,As1,R2 = Transfer2Blocks(Ds,Bs,Cs,As,R)
CC
Cs1,As1,R1 = TransferBlock With Robot (Cs,As,R)
Ds1,Bs1,R2 = TransferBlock With Robot (Ds,Bs,R1)
1 Robot Resource Allocation:
Schedule
RA
TransferBlock WithRobot
TransferBlock WithRobot
Ds1,Bs1,RB1,Cs1,As1,RA1 = Transfer2Blocks(Ds,Bs,RB,Cs,As,RA)
I
Cs1,As1,RA1 = TransferBlock With Robot (Cs,As,RA)
Ds1,Bs1,RB1 = TransferBlock With Robot (Ds,Bs,RB)
2 Robot Resource Allocation:
Schedule
RA
TransferBlock WithRobot
RB
TransferBlock WithRobot

a. Resource Architecture Dependent Definitions.



Functional Architecture Definition:
Ds1,Bs1,Cs1,As1 = Transfer2Blocks(Ds,Bs,Cs,As)
I
Cs1,As1=TransferBlock(Cs,As)
Ds1,Bs1=TransferBlock(Ds,Bs)
Resource Allocation Definitions:
Where: Transfer2Block on: Robot.
Or
Where: Transfer2Blocks on: 2Robots.
Resource Architecture Definitions:
Resource Definition with 1 Robot:
r(Robot)r0
J
J
r1(...)r0
holding(Hand)r1
r(Block)holding
Resource Definition with 2 Robots:
r4,r3 (2Robots) r2,r1
I
r4(Robot)r2
r3(Robot)r1
r(Robot)r0
J
J
r1(...)r0
holding(Hand)r1
r(Block)holding

b. Resource Architecture Independent Definitions.

Figure 7: Separation of Functional and Resource Architectures.

architecture. Two different resource allocation architectures are shown here which have been defined based on the alternative resource architectures. The only system that would change to switch from one robot to two robots or more would be the resource allocation specification. In this example only the *Where* statement would change. This model is an example of a technique that can be used to define a system to inherently support its own *run-time performance analysis.*

G. Design Integrity of a System.

Properties of 001 systems can be used to evaluate the integrity of a system design. The "goodness" of "badness" of a system design can be evaluated based upon attributes of the particular FMaps and TMaps used to define a system. Attributes such as the number of layers in a TMap for a particular sysιem , the degree of strong typing used in that TMap, the number of inputs and outputs associated with each function in the FMap of the system, the size of the FMaps and

TMaps, degree of movement around a TMap to accomplish each functional task in an FMap all come into consideration for such an evaluation. It becomes more clear with each new development experience using 001 that a system can be defined from the very beginning to inherently support its own *analysis for design integrity.*

IV. THE 001 TOOL SUITE: AN AUTOMATION OF THE TECHNOLOGY

The purpose of the 001 tool suite is to ensure that the 001 technology is used correctly. It has been defined with, implemented with and automatically generated by itself. It is layered onto primitives which are implemented in a language for a given native computer environment. A developer can use the tool suite either to prototype a system or fully develop that system resulting in production quality code.

The tool suite provides a menu system interface for communicating with users and an editor for defining FMaps and TMaps and their integration in either graphical or in textual form (Figure 8). The capability exists for a user to define his own libraries with a *Road Map (RMap)* hierarchy that provides an overview (and index) of the library of FMaps, defined structures and TMaps.

At any point during the definition of a model, it may be submitted to the *Analyzer* which automatically determines from examining the definitions of FMaps, TMaps and user defined structures whether objects are used in a consistent and logically correct manner. The Analyzer ensures that the rules for using the mechanisms described above are followed correctly. As a result, all interface system errors (the majority of the errors which are normally found during testing in a

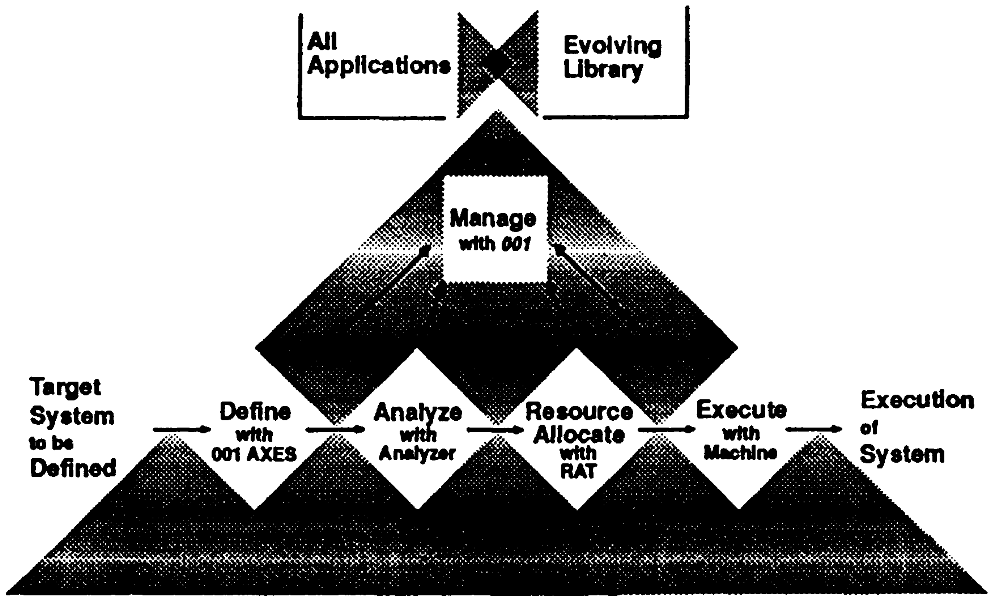


Figure 8: The Definition and Development Process with the 001 Tool Suite.

conventional development) are eliminated at the requirements/specification stage of development instead of later during testing.

When a model has been decomposed to the level of objects designated as primitive and it has been successfully analyzed, it can be handed to the *Resource Allocation Tools (RATs)* which automatically generate source code from that model. These RATs are generic in that they will interface with diverse language, operating system and machine environments. The *Type RAT* generates a system of object type templates for a particular application domain from a type hierarchy. The *Functional RAT* generates source code from the integration of type templates generated by the Type RAT and the functional maps. The code generated by the Functional RAT is automatically connected to the generated type primitives and previously existing function and type primitives in the core library, as well as, if desired, libraries developed from some other environment. The generated code can be compiled and executed on the machine where the tool suite resides (the tool suite currently resides within the VAX/VMS based environment); or, it can be ported to other machines for subsequent compilation and execution. User-tailored documents, with selected portions of a system definition, implementation, description and projections (e.g., parallel patterns, decision trees and priority maps) can also be automatically generated by the RAT.

With the use of the tool suite, a development process is automated within each phase and between phases except at the very beginning of the life cycle where the user inputs his thoughts and at the end of the development process when the developer tests the results of the user's ideas. The order of development is efficient, since phases begin as soon as possible and many phases can proceed in parallel (e.g. analysis for errors begins before implementation). Only one semantics language is used throughout development. Each development phase is implementation independent. A system can be automatically "RATted" to various alternative implementations such as selected language, documentation, projection and computer environments without changing its original definition. The 001 suite takes advantage of the fact that a 001 system is defined from the very beginning to inherently maximize the potential for its own *automation.*

V. RESULTS

Several systems have been developed with 001, including those which reside within manufacturing, aerospace, software tool development, data base management, process control and simulation environments. The definition of these systems began either with our defining the original requirements or with requirements provided to us from others in various forms. They have varied from one extreme of interviewing the user to obtain the requirements to the other of receiving written requirements which were far too detailed. Written requirements were also provided to us in English, 2167A format or in terms of other requirements/specification languages. We have analyzed our results on an ongoing basis in order to understand more fully the impact that properties of a system's definition have on the productivity in its development.

A. Analysis of Recent Systems Developed with 001

Some of our most recent experiences where we analyzed systems, with respect to productivity, include the development of the DETEC, OTD and Executor systems[3,4].

1) The Defensive Technology Evaluation Code (DETEC) Demonstration System: On this project a discrete event simulator for simulating real world object interactions (e.g., battle managers, sensors and weapons), was defined and developed for Los Alamos National Labs as an asynchronous, event-driven real-time multiprocessing environment [3]. The first effort on this project was spent developing approximately half of the system to an executable design level to demonstrate prototyping at a high level. The second effort on this project was spent in developing the other half the system to its lowest levels to demonstrate the complete generation of a production ready system. Figure 9 summarizes the productivity results on this project. The productivity was lower for the first effort than for the second since the process of understanding the requirements for the entire system took place in the first effort. The productivity in developing the DETEC system with 001 varied from 14:1 to 48:1 when comparing it to conventional C system developments (as measured in [5]) starting with a basic compiler, linker, and standard run-time libraries to work with (Basis 1); but we determined that if only C experts with access to a specialized reusable library equivalent to the core library of 001 were to develop this same system then the productivity could have varied from 2:1 to 8:1 (Basis 2).

2) The Object Tracking and Designation (OTD) Project: The intent of this project was to demonstrate the effectiveness of 001 as a development environment for OTD related systems within SDI [4]. This system performs track initiation updates, discrimination and prediction computations based on sensor data and navigation information. The OTD effort was a ten man week effort starting with

	Processing	Controller
Total C Source 001-Generated	22,000	26,300
Actual 001 Life Cycle (man-months)		
Original Spec to Final Code	7.00	2.75
Estimated Conventional Life Cycle (man-months)		
Original Spec to Final Code, Basis 1	100	132
Original Spec to Final Code, Basis 2	17	22
Productivity Gains		
Basis 1	14:1	48:1
Basis 2	2:1	8:1

Figure 9: DETEC Demonstration System Productivity Statistics.

learning the requirements and ending with a round of performance testing[4]. The system developed with 001 is comparable to a 25,500[5] statement system in C source lines developed in a highly structured and modular conventional development environment or a 15,000[5] statement system in C source lines being produced by very experienced C experts with a generic reusable library equivalent to that of 001.

3) The Executor Prototype System: This system was developed (with a preliminary round of tests) using 001 [4]. It provides the ability to observe the real time behavior of a 001 system. The Executor Prototype system is approximately one half the size of the OTD system. Once the requirements were defined for this system, this system was developed and tested with two man weeks of effort. It generated approximately 13,000 lines of C code from the 001 model. The Executor interfaces with other portions of the 001 tool suite to complete its functions; it therefore is a larger and more complex system than the size of the set of new models developed would suggest. It is our opinion that the productivity in this case, from the completion of the requirements definition throughout development, was even higher than the productivity for OTD largely due to the fact that the requirements for the Executor were well understood by the developer when he began to develop his system.

B. Next Step

Future enhancements of the tool suite will further take advantage of the properties of 001 defined systems. For example, the tool suite is being enhanced to further narrow the possibility of errors in the user intent domain, i.e., in the "remaining 10% to 25%" category. An architecture independent operating system (AIOS) [4] is now being developed which will have the capability to understand 001 systems and use this knowledge to support automatically the integration of all of the architectures in a complete system engineering environment. The inherent parallel patterns of the 001 defined requirements in both functional and resource architectures allow for automated capabilities that were not thought to be possible before. The AIOS will be used to align dependencies and independencies in the functional architecture with those in its associated resource architectures. It will then automatically find best case parallelisms and automatically allocate the functional architecture to the best case resource architectures found. In one sense, the system design engineer who performs run-time performance analysis (such as one who compares the results of an algorithm on one hardware distributed architecture with the results of that same algorithm on another architecture) is a human AIOS. The AIOS will automate these tasks of the design engineer which are manual today.

VI: SUMMARY

Our experience has shown that the productivity in developing a particular system increases to the degree that Development Before the Fact properties exist and are capitalized on in that system. A Development Before the Fact system is an

[4] Since 001 interface errors are found before implementation, it means that approximately 75% to 90% of the "testing" of the 001-defined and developed OTD system was completed before implementation.

[5] This figure was obtained by counting the number of C lines of code automatically generated by 001 and reducing it by a factor of approximately forty to sixty percent to account for a typical programmer's stylistic preferences and removal of comments. (It should be noted that the executables of a 001 developed system are normally smaller than the executables for a conventionally developed system of the same functionally.)

intelligent system in that it contains built-in properties to provide itself the best opportunities that are available for its own development. It provides the opportunity to begin its own development tasks as soon as possible and to *check as you go* throughout its own development process. It is an *independent* system in that it is not locked into obsolescence.

The 001 technology directly addresses the issues of Development Before the Fact systems. With 001, *integration happens early* with the use of TMaps and FMaps; *errors are eliminated early* when, for example, the primitive structures are ultimately used to eliminate interface errors); *flexibility and the ability to handle the unpredictable are issues that are dealt with early* since system definitions based on the three primitive structures have properties of single-reference and single assignment, ensuring traceability and safe reconfiguration; *preparing for distributed environments happens early*, with explicit delineation of independencies, dependencies and decision making; *reusability happens early* with the use of mechanisms such as defined structures and parameterized types; *automation happens early* , since TMaps, FMaps and their associated instantiations support automated tools with sufficient and necessary information to understand formally both a system and its definition; *run-time performance analysis happens early* with the use of techniques which allow a clear separation of architectures as well as an automation which can interpret the meanings and applicability of these architectures and the relationships between them; and *design integrity is considered early* with the use of the TMaps and FMaps to facilitate the process of understanding the relationships between a system and its implementations and executions. Because of these properties, a system can be developed with high quality, high reliability and rapidly with a productivity that is in a category of its own. With this kind of productivity there is no longer a need to differentiate between those techniques for rapid prototyping and those for developing production ready systems.

REFERENCES

[1] M. Hamilton, "Zero-defect Software: the Elusive Goal," IEEE Spectrum, vol. 23, no. 3, pp. 48-53, March, 1986.

[2] M. Hamilton, "Towards Ultra Reliable Medical Systems," Invited paper at *Proceedings, IEEE Symposium on Policy Issues in Information and Communication Technologies in Medical Applications*, Rockville, Maryland, September 29, 1988.

[3] M. Hamilton and R.Hackler: "Prototyping: An Inherent Part of the Realization of Ultra-Reliable Systems" in Final Report to University of California Los Alamos National Laboratory Contract No. 4-X28-8698F-1: Defensive Technology Evaluation Code (DETEC) Conceptual Model, 1988.

[4] Final Report: Object Tracking and Designation (OTD), Architecture Independent Operating System (AIOS) and Run-Time Ensemble Benchmark Environment Language (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.

[5] B. W. Boehm: Software Engineering Economics, Prentice-Hall, Englewood Cliffs, N.J., 1981.