Preventative Software Systems

Margaret H. Hamilton

Hamilton Technologies, Inc.
17 Inman St., Cambridge, MA 02139
Voice: 617-492-0058; Fax: 617-492-1727

Abstract

Traditional software environments help "fix wrong things up" instead of "doing them right in the first place". Things happen too late if at all. Quality is compromised. Dollars are wasted.

A promising paradigm is development before the fact *where a system's properties control its own development. The goal is to develop a system with* built-in quality *and* built-in productivity. *Whereas the traditional approach is curative,* development before the fact *is preventative. Its formal language, both function and object oriented, is the key.*

Described herein is the technology and its automation, a full life cycle systems engineering and software development environment. Complete, fully production-ready, integrated code can be automatically generated for any kind of software system as well as be configured for the language and architecture of choice. Also described is users' experience in its application to real world systems.

1: Introduction

Traditional system engineering and software development environments help their users "fix wrong things up" or perform tasks that should no longer be necessary instead of helping them "do things right in the first place". Because critical issues are dealt with *after the fact,* true reuse is ignored. Quality and functionality are compromised. Responding to today's rapidly changing market is not practical. Deadlines are missed. Time and dollars are wasted. The competitive edge is lost.

Take, for example, defining requirements. Incompatible methods are used to capture even part of a definition (e.g., data flow and timing). Once defined, there is no way to integrate the parts. Methods available to them force designers to think this way, leading to further problems. Integration of objects (object to object), modules, phases, or application types is a challenge left to the devices of a myriad of developers well into the development, compounded by the use of mismatched development products. Systems are actually encouraged by these methods to be defined as ambiguous. Interfaces are incompatible and errors propagate throughout development. Again the developers inherit the problem. The system and its development are out of control.

Requirements, focusing on application properties, do not consider that the user will change his mind. Porting involves new development. Maintenance is risky and costly. A distributed system is often first targeted for a single processor and then re-developed for a distributed environment. Another unnecessary development. Insufficient information about a system's run-time performance, including concerning 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. It is not known if a design is a good one until its implementation has failed or succeeded.

Focus for reuse is late into development during the coding phase. Definitions lack properties to help find, create and make use of commonality. Modelers are forced to use the same informal methods for dividing a system into components natural for reuse. Little incentive exists for reuse in today's changing market if a module is not able to be integrated, not portable or adaptable and it is error prone. Redundancy is a way of life.

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, these 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 shell code. 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). The code generated is often inefficient and/or hard wired to a particular architecture or language. Everywhere manual processes complete unfinished automations. Most of the development process is needlessly manual.

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

2: Development before the fact

With *development before the fact* a system is defined with properties which inherently control its own development. An emphasis is placed on defining things right the first time. From the beginning, a system integrates all of its own objects (and all aspects of these objects) and the combinations of functionality using these objects; maximizes its own reliability and flexibility to change; capitalizes on its own parallelism; supports its own run-time performance analysis; and maximizes the potential for its own reuse and

0730-3157/94 $04.00 © 1994 IEEE

automation; it is developed with *built-in quality* and *built-in productivity.*

2.1: Technology

The *development before the fact* technology includes a language, an approach, and a process, 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 and about any 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 can be used to define systems for real time or data base environments with diverse degrees of fidelity and completeness. With this language, function-oriented parts of a system are defined to integrate with object-oriented parts; control hierarchies are defined to integrate with networks of functions and objects. Such a language is always considered a design language, since design is relative; one person's design phase is another person's implementation phase. Semantics-dependent but syntax-independent, its mechanisms are used 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.

The first step in building a *before the fact* system is to manage the system by configuring the process management environment. The next is to define a model. The model is automatically analyzed, statically and dynamically, to ensure that it was defined properly. Management metrics are collected and analyzed. A fully production ready and fully integrated software implementation, consistent with the model, is then automatically generated by a generic generator for a selected target environment in the language and architecture of choice. If the required environment has been configured, it is selected directly; if not, the generator is configured for a new environment before it is selected. The resulting system can then be executed. It becomes operational after testing. Target changes are made to the definition, not to the code. Target architecture changes are made to the configuration of the generator environment, not to the code. Once a system has been developed, the system and the process used to develop it are analyzed to understand how to improve the next round of system development. The process evolves before proceeding through another iteration of system engineering and software development.

Development before the fact is a function and object-oriented approach based upon a unique concept of control [1], [2], [3]. This approach had its earlier beginnings with the Apollo space missions when research was performed for developing man-rated software. This led to the finding that interface errors accounted for most 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. Each error was placed into a category according to the means that were taken to prevent it by the very way a system is defined. A theory was derived for defining a system such that this entire class of interface errors would be eliminated. The first technology derived from this theory concentrated on defining and building systems in terms of functional hierarchies [1]. Having realized the benefits of addressing one major issue (i.e., reliability) just by the way a system is defined, we continued to evolve with this philosophy by addressing other major issues in the same way. Systems can now be defined with *development before the fact* properties in terms of both functional and type maps where a map is both a control hierarchy and a network of interacting objects.

2.2: Integrated modeling environment

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 maps (FMaps) to capture time characteristics and type maps (TMaps) to capture space characteristics. 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 (e.g., data flow and timing) can be obtained from FMaps and TMaps. Maps of functions are integrated with maps of types.

On an FMap each node has a function which is defined in terms of and controls its children functions. On a TMap each node has a type which is defined in terms of and controls its children types. For example, the function, *MakeaTable*, could be decomposed into *MakeParts* and Assemble and the type, Table, into *Legs* and *Top*. Every type on a TMap owns a set of inherited primitive operations. Each function on an FMap has objects as input and as output. Each object, a member of a type from a TMap, resides in an object hierarchy (OMap), an instance of 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 (e.g., 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. A system with all of its object values assigned for a particular performance pass exists in the form of an execution map (EMap), an instance of an FMap. Global and local constraints (which can be defined in terms of FMaps and TMaps) can be defined for both FMaps and TMaps.

Typically, a team of designers will begin system design by sketching a TMap of their application. Here they decide on the types of objects (and the relationships between them) that will be in their system. Often a Road Map (RMap), which organizes all system objects, including FMaps and TMaps, as well as RMaps, themselves, will be sketched in parallel with the TMap. Once a TMap(s) has been agreed upon, the FMaps begin almost to fall into place because of the natural partitioning 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 (e.g., 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 which naturally work together. The system is *integrated* from the very beginning.

2.3: Primitive control structures

FMaps and TMaps are ultimately defined in terms of three primitive control structures: a parent controls its children to have a dependent, independent or a decision making relationship. A formal set of rules is associated with each primitive structure. If these rules are followed, interface errors (up to 90% normally found during testing in a traditional development) are "removed" at the definition phase. A system is defined from the very beginning to *maximize its own elimination of errors.* A use of the primitive structures is shown

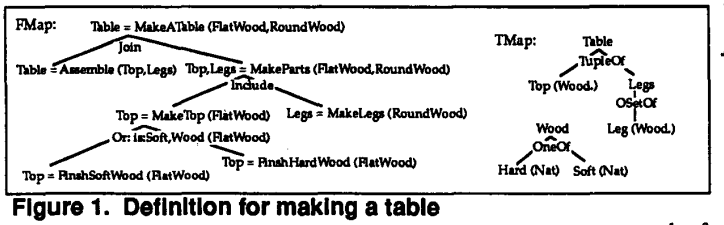

Figure 1. Definition for making a table

in the definition of the FMap for system, *MakeATable* (Figure 1). The top node function has *FlatWood* and *RoundWood* as inputs and produces *Table* as output. *MakeATable*, as a parent, is decomposed with a *Join* structure 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 is decomposed into *MakeLegs* and *MakeTop* who are controlled to be independent of each other with the *Include* primitive structure. *MakeLegs* uses part of its parent's input and *MakeTop* the rest. *MakeLegs* provides part of its parent's output (*Legs*) to its parent and *MakeTop* provides the rest. *MakeTop* controls its children, with an *Or*. Here, both children have the same input and output objects 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. All objects in a system are traceable, since input is traceable down the system from parent to children and output is traceable up the system from children to parent. *MakeATable's* TMap, *Table*, uses non primitive structures, a concept discussed in a later section.

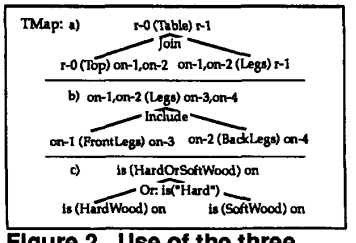

Figure 2. Use of the three primitive structures in a TMap

Each type on a TMap can be decomposed in terms of primitive structures into children types where the relationships between types is explicit. In figure 2a, *Table* is decomposed into *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 between *Table* and *Top* is *r-0*. *Top* depends on *Legs* to stand on to make a *Table*. An independent relationship exists between the front legs and the back legs of the *Table* [Figure 2b]. The *Table* may have *FrontLegs* or *BackLegs*, or both *FrontLegs* and *BackLegs* at once. In figure 2c, which illustrates a decision structure with objects, unlike with the dependent and independent structures, the pattern of the OMap is different than the pattern of the TMap, since only one object is chosen to represent its parent for a given instance.

A system defined in terms of these structures has properties which support real-time distributed environments. Each system is event interrupt driven. Each object is traceable, reconfigurable and has a unique priority. Independencies and dependencies are tractable and can be used to determine where parallel and distributed processing is most beneficial. A system is defined from the beginning to inherently *maximize its own flexibility to change and the unpredictable* and to *capitalize on its own parallelism.*

2.4: Reusables

Any system can be defined completely using only primitive structures, but less primitive structures can be derived from more primitive ones. *CoInclude* is an example of a system pattern that is used often (Figure 3a). In this case, everything stays the same for each use except for the functions at leaf nodes *A* and *B*. Systems like *CoInclude* can become *user defined structures*, derived from more primitive structures. Included with each structure definition is the syntax definition for its use (Figure 3b). Its use provides a "hidden repeat" of the entire system as defined, but explicitly shows only the elements which are subject to change. 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 ultimately from the rules of the primitives.

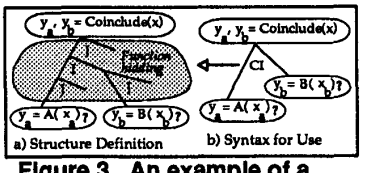

Figure 3. An example of a user defined structure

Async, (Figure 4), is a real-time, distributed communicating structure with both asynchronous and synchronous behavior. *Async* was defined with primitive 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 *Id2* and *Cll* 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 belong to 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. *Coordinate* uses *Async* as a reusable where two robots, *DecideNextStep* and *PerformTask*, are working together to perform a task such as building a table. Here one phase of the planning robot, *Master0* is coordinated with the next phase of the slave robot, *Slave0*.

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 *Table* (Figure 1) uses a set of default parameterized types. *Table* controls *Top* and *Legs*, in terms of a *TupleOf* parameterized type, *Legs*

controls *Leg*, in terms of *OSetOf*, and *Wood* controls *Hard* and *Soft* with a *OneOf*. *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. Defined structures and parameterized types can be created as reusables for asynchronous, synchronous and interrupt scenarios used in real-time, distributed systems. Similarly, retrieval and query structures can be defined for client-server data base management systems. With the use of these kinds of mechanisms, a system is defined from the beginning to *maximize the potential for its own reuse.*

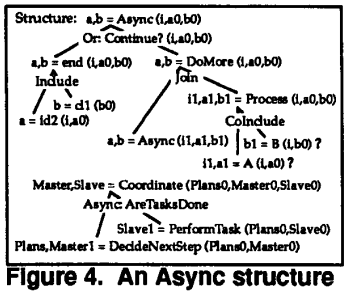

Figure 4. An Async structure

2.5: FMaps and TMaps and their integration

Figure 5 shows a complete system definition for a manufacturing company, defined in terms of an integrated set of FMap(s) and TMap(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 which are decomposed in another TMap.

Is_FullTime_Employee uses objects defined by the TMap to check 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.

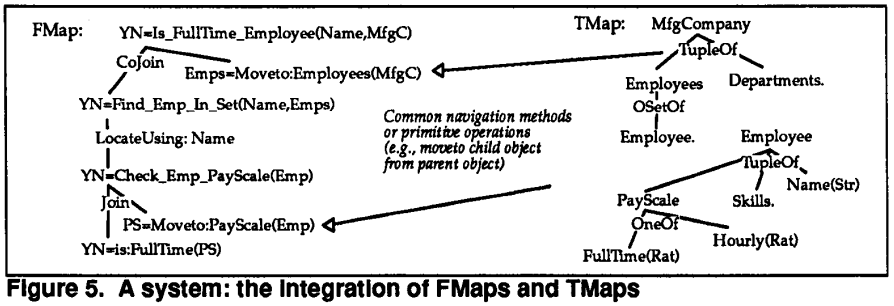

Figure 5. A system: the integration of FMaps and TMaps

Abstract types decomposed with the same parameterized type inherit the same primitive operations. For example, *MfgCompany* and *Employee* use *MoveTo* which is inherited from *TupleOf*. Each use of the *MoveTo* instantiates the *Child=MoveTo:Child:(Parent)* operation of the *TupleOf* parameterized type. For example, *Emps=MoveTo:Employees(MfgC))* allows one to navigate to an employees 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 which is defined in another type subtree (e.g., *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).

The TMap system provides universal primitive operations, inherited by all types, for controlling objects and object states. Used to 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 in order 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 since a TMap has a corresponding set of control properties for controlling spatial relationships between objects. One cannot, for example, put a leg in a position 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.

With *development before the fact*, systems engineering and software development are merged into one discipline. A system is by its very nature an integration of being function oriented and of being object oriented. Definitions are independent of particular function or object oriented implementations. Classical object oriented system properties such as inheritance, encapsulation, polymorphism and persistence are supported with the use of generalized functions on OMaps and TMaps. With systems constructed in a tinker toy-like fashion, reuse naturally takes place throughout the life cycle. Functions and types, no matter how complex, are reused in terms of FMaps and TMaps and their integration. Objects are reused as OMaps and scenarios are reused as EMaps. Architecture configurations are reused as RAT environments.

As experience is gained with different types of applications, new reusables emerge. For example, a set of mechanisms has been derived 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 defined structures that can be represented using a graphical syntax. 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.

3: An automation of the technology

The 001 tool suite is an automation of *development before the fact* [3]. A full life cycle systems engineering and software development environment, it begins with the definition of the meta process and the definition of requirements. Using FMaps and TMaps any kind of system can be designed and any kind of software system can be automatically developed resulting in complete, integrated and fully production ready target system code (or documentation) configured for the language and architecture of choice. The tool suite also has a means to observe the behavior of a system as it is being evolved and executed in terms of OMaps and EMaps. Every system developed with the tool suite is a *development before the fact* system, including the tool suite itself, since it was used to define and generate itself. A discussion of its major components follows.

The most important component in the tool suite is the 001 AXES systems language. Every system defined with 001 AXES inherits the *development before the fact* properties providing a basis for the unification of systems in general. This same language can be used to develop systems from a systemic viewpoint or from an internal or localized procedural viewpoint (e.g., a system model versus a software model). A unique aspect of this language is its ability to represent structure as an inherent part of the definition of a system. The primitive control structures define the fundamental separation between that which is considered to be inside the boundaries of a system and that which is to be considered as a part of the environment in which the system is embedded. The interfaces between a system and its environment are formally defined in terms of the objects and the structure of those objects (as defined by a TMap) that are created, evolved, and deleted as the system interacts (or reacts) to its environment.

Virtual Sphere (VSphere) is a component of the tool suite that supports a layered system of distributed hierarchical abstract managers. It provides the user the ability to define object relations that are explicitly traceable. Any relationship between objects can be defined providing the user the ability to query on relationships (e.g., between a set of requirements and its supporting implementation). The generalized manager, Manager(x), based on VSphere, allows the user to tailor his own development environment, a manager, itself. To tailor a manager, a user defines his process needs with FMaps and TMaps and then installs them into Manager(x). Since the tool suite is a Manager(x) configuration, a user can extend the tool suite environment itself. These extensions might be interfaces to other tools or tools which he develops with the tool suite to support his specific process needs. Although VSphere supports Manager(x) in providing a user callable and distributable object management system layer, a user can directly use VSphere as a data type from within his applications to provide control and distribution of his objects. This also allows the user's application to behave as an object manager for users of that user's application.

Requirements Traceability (RT(x)) is an example of a tool within the tool suite which is a Manager (x) configuration. RT(x) provides the user with more control over his own requirements process. It generates metrics and allows the user to enter requirements into the system and trace between these requirements and corresponding FMaps and TMaps throughout system specification, detailed design, implementation and documentation. With RT(x) traceability is backwards and forwards from the beginning of a life cycle to operation and back again. Traceability also exists upwards and downwards since requirements to specification to design to detailed design is a seamless process.

The tool suite is configured with its own configuration of Manager(x) which it provides as a default to its users. With this configuration the tool suite includes besides Manager(x), itself, the Session Manager for managing sessions, the Project Manager for managing projects, the Library Manager for managing libraries within one project, and the Definition Manager for managing definitions within a library. The Definition Editor of the Definition Manager is used to define FMaps and TMaps in either graphical or in textual form. Each manager manages a Road Map (RMap) of objects, including other managers, to be managed. An RMap provides an index to the user's system of definitions and supports the managers in the management of these definitions, including those for FMaps, TMaps, defined structures, primitive data types, objects brought in from other environments as well as other RMaps. Managers use the RMap to coordinate multi-user access to the definitions of the system being developed. Each RMap in a target system is an OMap of the objects in the system used to develop that target system within each particular managers domain. The Road Map Editor is used to define RMap hierarchies.

At any point during the definition of a model, it may be submitted to the Analyzer which ensures that the rules for using the language are followed. After successful analysis, the model is handed to the Resource Allocation Tool (RAT) for automatic generation of source code. The RAT can be configured to interface with language, data base, graphics, client server, legacy code, operating system and machine environments of choice. The Type RAT generates object type templates for a particular application domain from a TMap(s). The Functional RAT generates source code from an FMap(s). The code generated by the Functional RAT is automatically connected to the code generated from the TMap and code for the primitive types in the core library, as well as, if desired, libraries developed from other environments. To maintain traceability, the source code generated by the RAT has the same names as the FMaps and TMaps from which it was generated.

The generated code can be compiled and executed on the machine where the tool suite resides (the tool suite currently resides on the HP 700 series, IBM RS 6000, SunOS 4.X/Solaris, and Digital Alpha UNIX, X Window, Motif, C, FORTRAN and Ada environments); or, it can be ported to other machines for subsequent compilation and execution. Once a system has been RATted, it is ready to be compiled, linked and executed. The RAT provides some automatic debugging in that it generates test code which finds an additional set of errors dynamically (e.g., not allowing putting a leg on a table where it already had one). The developer is notified of the impact in his system of any changes and those areas that are affected (e.g., all FMaps that are impacted by a change to a TMap) are demoted. If a change is made, only that part of the system that is demoted needs to be regenerated.

The next step is to execute and test the system. In order for the user to interact with the system being developed, the tool suite generates default interfaces that obtain the data from the user and display results. For data types such as boolean and string, this interface is a simple prompt asking the user to enter the data. For complex data structures, however, the user uses Datafacer as an

object editor to provide comprehensive object viewing and editing.

Datafacer is a run-time subsystem that automatically generates a user interface based on the data description in the TMap. It uses the structure described in the TMap to generate appropriate modes of visualization for the specific data in an OMap. For example, an ordered set is visualized in Datafacer by default as a scrollable list. Users can configure alternative visualizations. When given an OMap instance of the ordered set, Datafacer fills this list with the actual OMap elements. The resulting interfaces are forms-entry screens, much like conventional database screen painters, but with support for the full semantic capabilities of TMap. Datafacer will generate screens for arbitrary depth type hierarchies and has full support for parameterized types.

In addition to visualizing the structure and values of the data, Datafacer automatically manages creation and modification of the data by the user. It binds the primitive operations appropriate for each data item to graphical controls that the user can activate. For example, a button labeled with a question mark activates a creation function when the user double clicks on the button. Similarly, editing a text field would update the string data for that item.

The capabilities of Datafacer are also available to developers to build interfaces that are more than just test harnesses. The primitive data type DFACE provides an API to Datafacer that gives the developer complete control over visualization and data modification from within his application. Here, the developer can add functions to capture run-time data events (like trigger functions), perform constraint checking, data analysis and specialized graphics manipulations. In addition to data specification, the developer has access to many graphical configuration options. Some of these may be carefully controlled while others may be left for users to change.

The Xecutor supports the analysis of the behavior (as a simulation) of a system with a hypothetical environment, or the control (as a real-time executive) of a system with a real environment. As an executive, the Xecutor schedules and allocates resources in order to activate primitive operations which interact with the environment.

The Xecutor executes directly the specification (as FMaps and TMaps) of a system by operating as an asynchronous run-time executive having multiple lines of concurrent control over these activated functions. Activated primitive functions send their input events out into the environment and then after some period of time receive their outputs as incoming events from the environment. These events will then trigger other primitive functions in the system causing an internal reaction resulting in local system concurrency given that the functional architecture and the allocated resource architecture align to produce system parallelism. The Xecutor at any point is able to activate dynamically bound executable functions (which have been defined in FMaps & TMaps as a software specification and implemented in some programming language) to support its own extension as an execution environment as well as environment specific interaction functions.

As a simulator/planning tool, the Xecutor records and displays information to the user for the evaluation of the resources which have been allocated to the functions of the functional architecture. It understands the real-time semantics embedded in a 001 definition. It allows the user to execute or simulate a system before implementation in order to observe characteristics such as timing, cost and risk based upon a particular allocation of resources. If the model being simulated by the Xecutor has been designed to be a production software system, then the same FMaps and TMaps can be RATted for production. The Xecutor can be used to analyze processes such as those in a business (enterprise model), manufacturing or software development environment (process model) as well as detailed algorithms (e.g., searching for parallelism). Because a system is defined from the very beginning to *support its own run-time performance analysis*, the Xecutor is able to perform many powerful functions for the systems engineer.

The Baseliner provides version control and base lining for all RMaps, FMaps, TMaps and user defined reusables, including defined structures. The Build Manager's primary role is to manage all entities which are used in the construction of an executable.

The tool suite environment provides a new set of alternatives for disciplines associated with the traditional development process. Take for example, reverse engineering. Redevelopment is a more viable option, since a system can be developed with higher reliability and productivity than before. Another alternative is to develop main portions of a system with this approach but interface to existing legacy libraries at the core primitive level. In the future, however, for those systems originally developed with the tool suite, reverse engineering becomes a matter of configuring and/or selecting the appropriate RAT configuration and then RATting to the new environment. In such a way the tool suite takes advantage of the fact that a system is defined from the very beginning to inherently *maximize the potential for its own automation*.

4: Results

Many systems have been designed and developed with this paradigm, including those which reside within manufacturing, aerospace, software tool development [4], data base management [5], transaction processing, process control, simulation [6] and domain analysis environments [7]. One of these systems is the 001 tool suite, itself [4]. Approximately 800,000 lines of C code were automatically generated for each of 4 platforms by the tool suite to create itself. Over 7 million lines of code have been generated by the tool suite to generate its 3 major versions on these platforms. Contained within the tool suite are many kinds of tools (applications) that have been automatically generated as integrated systems, including database management, communications, client server, graphics, software development tools, and scientific systems. They are inherently integrated as part of the same system.

Recently, the tool suite was part of a National Test Bed experiment [8]. The same problem was provided to each of three contractor/vendor teams. The application was real time, distributed, multi-user, client server, and was required to be defined and developed under government 2167A guidelines. All teams completed the definition of preliminary requirements, two teams continued on to complete detailed design and one team, the 001 team, continued on to automatically generate complete and fully production ready code; this code (both C and Ada were generated from the same definitions) was running in both languages at the completion of the experiment. 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. Productivity was analyzed with several systems. Compared to a

traditional C development where each developer produces 10 lines of code a day, the productivity of 001 developed systems varied from 10 to 1 to 100 to 1. (In the eighties, 10 lines of code a day per person was the government expected output. Now it is more likely to be 2 to 5 lines a day). Unlike with traditional systems, the larger a system, the higher the productivity. This is in large part because of the high degree of reuse on large systems.

5: The next step

The tool suite has evolved over years based upon user feedback and a continuing direction of capitalizing more on advanced capabilities of *development before the fact*. Datafacer and DFACE are examples of newer tools to be made recently available to external users. This was after the developers of the tool suite used them for several months to develop the most recent versions of the tool suite. Manager(x), an even newer capability, is going through a similar evolution with the developers of the tool suite.

New components to be added to the tool suite environment are the generic Anti-RAT and the architecture independent operating system (AIOS) [9]. With the anti-RAT legacy code and legacy definitions can be reverse engineered to FMaps and TMaps and become a *development before the fact* system before proceeding through the RAT process to generate (regenerate) the target system in the language and architecture of choice. The amount of user interaction after the FMaps and TMaps have been generated will depend on how formal the legacy code was in the first place and on the degree to which the user would like to change or raise the level of his specification. The tool suite currently has an instance of the Anti-RAT in that it can generate FMaps and TMaps from equations. The user has the ability to attach equations to the bottom nodes of FMaps making use of this capability. Upon comparing the anti-RAT reverse engineering approach with the reverse engineering approaches discussed above there are advantages and disadvantages to each one depending on the requirements of the user.

The AIOS will have the intelligence to understand the semantics of functional, resource and resource allocation architectures since all of these architectures can be defined in terms of FMaps and TMaps. It will make use of the information in their definitions (including the matching of independencies and dependencies between architectures) to automatically determine sets of possible effective matches between functional and resource architectures. The Distributed Xecutor, a module of the AIOS, will provide for real time distributed object management capabilities where the user's application will be fully transparent to client server programming techniques and communication protocols.

6: Summary

As was shown above, collective experience 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. Effective reuse is a preventative concept. 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 ten reuses saves ten unnecessary developments.

The paradigm shift occurs when a designer realizes that many of the things he used before are no longer needed. Techniques for reconciling multiple incompatible techniques and bridging the gap from one phase of the life cycle to another become obsolete. Testing procedures and after the fact tools for finding the majority of errors are no longer needed because these errors no longer exist. The same is true for the majority of tools developed to support programming as a manual process. In the end, it is the combination of the technology and the process that executes that technology that forms the foundation of successful software. Software is so ingrained in our society that its success or failure will change dramatically the way businesses and governments are operated as well as their overall success. It is for this reason that the decisions made today relating to systems engineering and software development will have such far reaching effects.

It is within our mistakes that the answers for success often exist. The first step is to recognize the root problems. They can then be understood in terms of how to prevent them in the future. This is followed by the derivation of practical solutions. The process is then repeated by looking for problem areas in terms of the new solution environment. In contrast to the just-in-time philosophy, the preventative philosophy is to solve a given problem such as finding or preventing errors *as soon as possible* (*ASAP*). Finding an error statically is an earlier process than finding it dynamically. Preventing it by the very way a system is defined is even earlier. Not having to define (and build) it at all is earlier yet. The answer continues to be in the results.

References

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

[2] 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.

[3] Electronic Design, "Inside Development Before the Fact", Part I, April 4, 1994, P 8 ES; :Development Before the Fact in Action", Part II, June 13, 1994, P 22 ES.

[4] The 001 Tool Suite Reference Manual, Version 3. Cambridge, MA, Hamilton Technologies, Inc., January 1993.

[5] The OpenINGRES Object Generator reference manual, Version 1, Alameda, CA, ASK Group Ingres, June 1994

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

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

[8] 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.

[9] Hamilton Technologies, Inc., "Final Report: AIOS Xecutor Demonstration", Prepared for Strategic Defense Organization (SDIO) and Los Alamos National Laboratory, Los Alamos, NM 87545, Order No. 9-XG1-K9937-1,1991.