TOOL SUPPORT FOR COLLABORATIVE SOFTWARE PROTOTYPING

Elliot A. Shefrin †
James M. Purtilo ‡

Computer Science Department
University of Maryland, College Park, MD 20742

December 1994

ABSTRACT: Prototyping is a means by which requirements for software projects can be defined and refined before they are committed to firm specifications for the finished software product. By this process, costly and time-consuming errors in specification can be avoided or minimized. Reconfiguration is the concept of altering the program code, bindings between program modules, or logical or physical distribution of software components while allowing the continuing execution of the software being changed. Combining these two notions suggests the potential for a development environment where requirements can be quickly and dynamically evolved. This paper discusses *reconfiguration-based prototyping* (RBP), that is, the simultaneous consideration of requirements, software behavior and user feedback within a running system in order to derive a clear specification of an intended product. Tools enabling RBP can coordinate the efforts of designers, prototypers, users and subject matter specialists as they work towards concensus on an application's specification by means of a prototype. The authors describe the scope of the modifications that can be effected by an integration of prototyping and reconfiguration protocols, and they then demonstrate that the technology exists to create such an environment. They conclude by describing a software development environment based on RBP.

KEYWORDS: Reconfiguration, Prototyping, M Technology

† Elliot Shefrin's research has been supported by the Longitudinal Studies Branch, Gerontology Research Center, National Institute on Aging, National Institutes of Health, United States Public Health Service. He can be contacted at `mazel@cs.umd.edu`, (410) 558-8144.

‡ With oversight by the Office of Naval Research, James Purtilo's research has been supported by ARPA in conjunction with the Common Prototyping Language project, contract number N00014-90-C-0015. He can be contacted at `purtilo@cs.umd.edu`, (301) 405-2706.

1 INTRODUCTION

During the development of software systems, a prototyping phase is often employed to fine tune various aspects of the system or to discover user requirements, preferences, or operational imperatives [PFW94]. These aspects may include development of the user interface, testing of various structures for the database, or optimizing internal algorithms. From this viewpoint, prototyping can be regarded as an information acquisition activity whose principal goal is to reinforce the confidence of the system developer and the user in the correctness of the system specification. In [Pur91] the author describes the traditional approach to the tuning procedure as an iterative process of program execution, halting, modification, and restarting. Such a technique can be time-consuming and frustrating, especially when the system takes a while to reach a steady-state or when each iteration of the tuning loop requires many steps.

A pivotal point illustrated in [PFW94] is that a prototype need not implement the underlying algorithms of the system under development. Rather, all relevant data may be loaded from tables and processing on that data may be simulated. Furthermore, not all aspects of a design require a prototype. It is up to the designer to identify the elements of risk — the areas of uncertainty — that can be reduced by the implementation of a prototype. The ingenuity, then, of the system developer lies in deciding which aspects of the design need to be prototyped and what information is to be extracted from the prototype. A further challenge is to involve the user to such a level that they feel they have a vested interest in producing the best product possible. Stated another way, it is the ends and not the means that concern the prototype developer. The observation that platforms and scenarios can be contrived to elicit necessary information leads to the useful conclusion that the tools and vehicles used to construct the prototype do not need to be the same as those used to construct the finished product. Indeed, since it can be built on a simplified testbed, the prototype should be capable of cutting quickly to the core of the problem and can be empowered to use whatever technology is most efficient and effective to obtain the information that is the end product of that particular prototype. This point obtains relevance because we will contend that there is no need to justify the choice of a prototyping host technology or to be concerned with the applicability of the technology used in the prototype to the design project at large.

The traditional approach to prototyping can be compared to attempting to tune the engine of an automobile by the following series of steps:

- start the car
- observe the output on an engine analyzer
- stop the engine
- make an adjustment
- restart the car to see how close we have come to the correct setting.

This affords no ability to receive real-time feedback from the engine — the object under study. In contrast, the way it is done in practice is that the technician observes the output of the engine analyzer while the motor is running in steady-state and he is making incremental adjustments. In this actual practice, the technician is *reconfiguring* the operation of the engine while it continues to run. Applied to software, we believe that the concept of reconfiguration — dynamically modifying an executing piece of software — can be employed to advantage in order to make the tuning process more efficient.

The current state of understanding regarding reconfiguration is summarized in [Pur91]. It is stated that the options for reconfiguration are limited to those anticipated by the developer of the software. Fundamental problems arise because tools for reconfiguration are weak and the steady-state may be easily disturbed by those tools that do exist. However, in this paper, we will describe an approach to reconfiguration that allows a great amount of flexibility. We will outline the framework that should be employed in order to attain this level of adaptability, and we will demonstrate that it is, in fact, achievable. In the next section, we will illustrate our position and motivation regarding the desirability of a synthesis of prototyping and reconfiguration. A discussion of language issues relating to our work will comprise section 3. In section 4, we will describe the elements which underlie the concept of reconfigurability and link them to the prototyping process. Then we will describe a methodology which enables a developer to avoid the requirement of anticipating all possible points of modification and demonstrate how easy it is to accomplish a high level of reconfigurability. Section 5 will describe the course of research by which we are validating our assertions. We will draw our conclusions in section 6.

2 MOTIVATION

We believe that the broadening of the prototyping process to incorporate reconfiguration technology will result in a more efficient discovery of the needed information. Furthermore, if the user can be actively involved in the process, then the outcome is more likely to be acceptable; some of the mystery of problem resolution will have been mitigated and the user will have a personal interest in the quality of the end product. The development cycle that we anticipate is illustrated in Figure 1. Starting from an initial design specification, the developer identifies zones of risk or uncertainty. These are the elements of the design that must be refined in order for the final product to have the greatest acceptability and utility to the user. Having identified these areas, the designer must articulate alternatives along with a methodology for evaluating these alternatives. The goal is to arrive at a refined specification and a decision as to whether the process is complete or needs another iteration.

The methodology employed by the designer subsumes several critical components. The first is a suite of test data or a hypothetical problem to be solved using each alternative of the system under test. These problems must be formulated to provide the user the opportunity to make contributions to the refinement and evaluation of the alternatives; this is the critical stage where the user becomes a partner in the design process. A means must be specified to quantify the merit of each outcome so that alternatives can be objectively compared. We shall use the term *discriminant function* to refer to this evaluation protocol that will allow the user and the developer to quantify differences in the identified alternatives, assign a merit value to each, and recognize when the objective has been reached. And finally, differing scenarios must be envisioned so that the design will, in fact, meet the specifications over the entire operational domain.

A critical part of this prototyping method is the ability to interactively and dynamically reconfigure the executing software in order to perform the test and evaluation. We are working on a protocol that specifies and implements prototyping in a development environment that incorporates dynamic software reconfiguration. Such a Reconfiguration-Based Prototyping (RBP) model, as described in greater detail in section 5, is being constructed with design concepts and software tools using sufficiently powerful and flexible languages, as discussed in section 3. RBP includes the capability to modify parameters, algorithms, interfaces, and database structure, thereby enhancing the prototyping effort in a manner outlined in section 4.

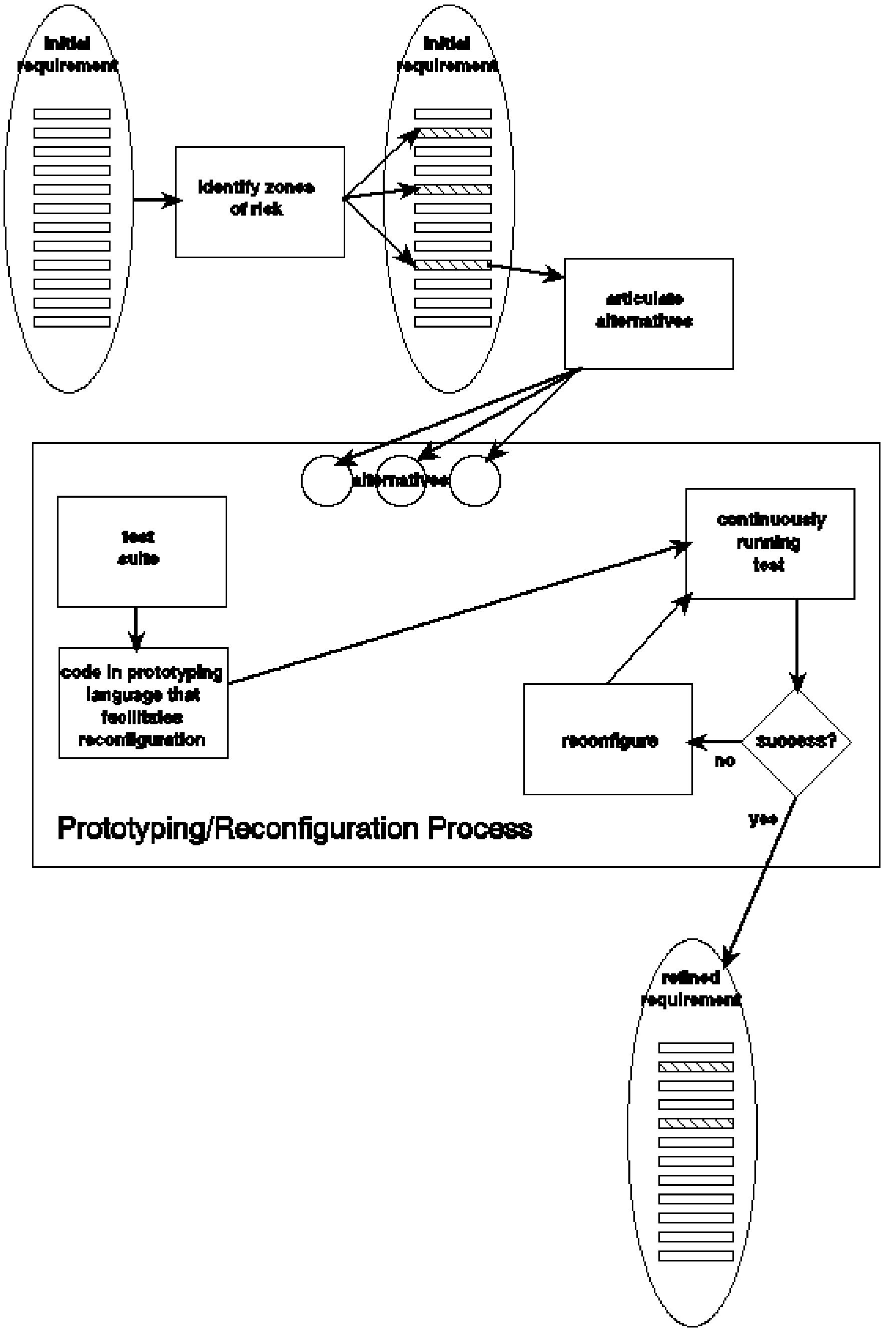

Figure 1: Prototyping Reconfiguration Development Cycle

3 BACKGROUND

The concept of prototyping has been a part of engineering philosophy for many years, and it is only natural that it should have been incorporated into the design approach for software development. Dynamic reconfiguration, on the other hand, is a topic more specific to software systems and we find ourselves earlier in the evolution of this technology. The issue of language choice has been raised in the previous section, and there is much prior information about special capabilities of various languages. In the next part of this section, we look at some of these issues and examine one existing technology that facilitates the work we are doing.

3.1 LANGUAGE ISSUES

We can envision an environment for reconfiguration-based prototyping. We can articulate the capabilities that separately facilitate prototyping and reconfiguration, combine them, and add any extra features suggested by the merging of the two protocols. For prototyping, we would like to be able to quickly and easily convert ideas and algorithms into operational programs; this implies the use of some higher level computer language. We would like the ability to embed instrumentation in the prototype and to observe the output of that instrumentation in an on-line mode. We would like a shared run-time environment where system state and data can be examined to further understand the operation of the prototype. For reconfiguration, we would like the capability to monitor the currently operating configuration and the state of potential reconfiguration options. Dynamic module bindings would be necessary for interchanging modules. Though not a necessity, modifiable code would open a broad range of revision options. Here, too, a shared run-time environment would empower the designer to manipulate components on the same platform that the reconfiguration is proceeding on. For the combination, it would be desirable to be able to alter the mapping of modules onto nodes, and to be able to radically transform the functionality of any component regardless of its level in the execution hierarchy.

While there may be several tools that provide this desired environment, we maintain that it is only necessary to identify one, since the language of the prototype need not be wedded to the language of the ultimate implementation. With this in mind, we describe an existing technology which provides the facilities that we have identified as empowering. M (formerly known as MUMPS) is a powerful ANSI, FIPS, and ISO standard third-generation programming language. Part of this standard is an assurance that implementations will be platform independent. Thus, prototypes developed under one configuration will be readily portable to other configurations or platforms. Furthermore, since the language is in a continuing state of development, the standards are revised periodically (typically every three or four years) to incorporate new features of interest to developers of state-of-the-art applications. By incorporating new developments, such as M/WAPI (M/Windows Adaptable Programmer Interface) and OMI (Open M Interconnect), into the language standard, the M community is assured of continued consistency and portability. In using these interfaces and standards, the M programmer can write software modules which can function well in a mixed-language environment.

From a prototyping standpoint, M has several areas of strength. A feature of the M Technology is positive control over system resources and devices. Specific units are opened and then used, and the programmer is allowed to manipulate the parameters of these units. By interposing an interface layer, many implementations can be made virtually device independent. This adds to portability as well as consistency between various environments.

Various forms of parameter passing and accommodation of undefined values allow programmers to develop subprograms that are general, robust, and reusable in nature. Modern features of definition of scope, including private and public variables, extend this ability. While this is not a feature unique to M, it does permit independent parallel development of various aspects of a larger module by separate groups of programmers, once the interface, input, and output specifications have been decided upon. However, M is not a recursive language; programs constructed in M rely instead on iteration.

Originally M was strictly an interpreted language. As such, there were no compile or link steps in program development. Modern M implementations perform a compilation (ranging from tokenization to more comprehensive compile), but most still revert to real-time linking. Syntax checking and compilation is often performed as a final step in the program editing process, prior to filing the edited routine. This translates into a rapid development/trial/revision cycle that is ideal for debugging segments of a large system. Furthermore, since M remains essentially interpreted in nature, it is easy to step through routines, examine current values of variables, and set traps using the powerful debugging tools that have been provided by implementers.

Furthermore, M does not require or allow declaration of type, size, or structure of variables. This applies both to local, memory-resident variables and to global disk-resident data structures. While this may pose an intimidating abundance of freedom for novice, inexperienced, or unskilled programmers, it can be a powerful ally for good software engineers. It can provide the capability to quickly and painlessly restructure a database as dictated by evolving development and debugging during the prototyping process. Once established, the finished structure need only be formalized in the documentation. With respect to instrumenting, visualizing, and monitoring the behavior of the prototype, M has the potent property that all of its database globals can, with trivial ease, be made visible to other processes on the system, whether in the same or separate computers. This implies that prototype behavior and status can be observed, recorded, and displayed by non-invasive independent procedures executing on platforms of the implementer's choosing. In some cases, the prototype may be augmented to incorporate reporting of status; however, due to the ease of variable definition, this can be quickly patched in or removed as the situation requires.

The language has the flexibility to deal with novel and current paradigms, such as objects and the relational model. Also, client-server and peer-to-peer networking and distribution of the database are standard features of many of the implementations of the M Technology. Another inherent feature of the M Technology is the capability to spawn background, independent, parallel processes. These jobs can be synchronized and coordinated by use of the global database or by pipes easily established as devices connecting two processes. Coupled with the distributability of the database, this provides a powerful tool and ability to develop parallelism as part of the design of a prototype. In fact, jobs can be spawned on the same processor as the parent program, or they can be started on a server which hosts the data on which the process operates. Therefore, by clever design of the distribution of the database and distribution of the computing load, significant advantages can be had by parallelism, and this is completely under control of the program designer.

From the reconfiguration standpoint, there are two additional key properties of the M Technology that empower the developer: indirection and the capability of executing variable code. For example, the following line of M code illustrates indirection.

```
do @variablesub
```

This statement is a command to invoke the subroutine whose name is stored in the variable named `variablesub` as a text string. In other words, if the current value of `variablesub` is "output" when the above line of code is encountered, then the line is equivalent to

```
do output
```

Furthermore, `variablesub` could be a variable in the local space of the executing module, it could be inherited from a calling module, it could have been passed as a parameter, or, most significantly, it could be a shared global element of the database.

M also includes the ability to execute a variable as an M command. Again, we employ an example.

```
set variablecommand="set x=v*b*a write x,y,z"
xecute variablecommand
```

has precisely the same effect as if

```
set x=v*b*a
write x,y,z
```

had been hard coded into the routine. Analogous to the description of indirection, the executed variable can be local or global, and the effect of executing the command is strictly a function of the value of `variablecommand` at the time it is actually encountered during execution.

4 PROTOTYPING AND RECONFIGURATION

In order to proceed on the combination of the prototyping approach with the reconfiguration capability, we need to understand more fully the interactions and overlaps between the two concepts. The first part of this section explores this in greater depth. Building on this understanding, the second part of section 4 proposes guidelines to ensure that developed prototypes will be able to take advantage of reconfiguration.

4.1 INTERRELATIONSHIPS BETWEEN THE TECHNOLOGIES

In [HP93] the authors discuss the need to manage three different types of reconfiguration module — implementations, system logical structure, and geometry. While their discussion is placed in a framework of language independence, we contend that many applications are implemented in a language of choice with very few pieces, if any, written in other languages. A particular example of this class of application is scientific computing. In prototyping for this type of software, there exists the need to reconfigure while executing. This is because it may take a while for the prototype to either reach steady-state or to reach the place that is of immediate interest to the designer. Point by point, we will outline the interrelationship of reconfiguration and prototyping.

The need to modify the implementation of modules encompasses several different activities. It may be necessary in order to replace a module which has been found to contain a bug. It may be driven by the desire to install a module implementing a more efficient algorithm, or a different (slightly or otherwise) functionality. There may be a reason to adjust some parameters embedded in the program. Specifically in the case of prototyping, there may be the need to modify the instrumentation, so that different or additional aspects of program function are made accessible to observers. Relating back to the example, reconfiguration might be applied to the task of substituting different display or interface algorithms, to discern which best conveys the information that is appropriate to the objective.

In a single-thread application, where, traditionally, the entire process is one load module, it is difficult to make modifications while the application is executing and still maintain the ongoing execution state. Concepts of *encode* and *decode* are introduced in [HP93] to capture and then reinstall run-time state when a module is to be reconfigured. However, use of these constructs require the designer to correctly anticipate places in the algorithm where reconfiguration will be desired. Furthermore, special provision must be made to recognize when the program has reached a reconfigurable state so that reconfiguration can be invoked. These considerations are important in the prototyping process, as they are in any reconfiguration, so that the steady-state which has been reached is not disturbed.

The discussion regarding logical structure or topology of the application can be approached in the same manner as changes in implementation. It may be that the attention of the prototyping effort has been turned to performance considerations of how the task is subdivided among subroutines or parallel processes. Observing the response of the system to such a topological reconfiguration could be very useful in the development process.

As for geometry, there are several reasons why the distribution of the processing might be altered during execution, in addition to load balancing, fault tolerance, or resource availability. It may be desired to execute a module closer to the data on which it operates, to measure the performance improvement. It may be useful to direct the operation of a module toward a different local database operating on another machine.

In summary, there exists quite a bit of overlap between the availability of a strong dynamic reconfiguration tool and the efficient pursuit of the prototyping process.

As discussed in the introductory section of this paper, we do not need to justify our choice of a prototyping host technology or to concern ourselves with the applicability of the technology used

```
RECONFIG(ODEV)   ;EAS;03:46 AM  18 Jan 1994
        ;demonstrate M's reconfiguration capability

        ;set up routines to call initially
        set ^SUBNAME="^NOHIST",^STATDISP="^DISPLAY1"

        ;initialize the number of targets to track
        set ^NUMTARGET=5

        ;now just loop forever until termination flag set
        for  quit:$data(^RECONQUIT)  do @(^SUBNAME)

        kill ^RECONQUIT quit
```

Figure 2: Routine RECONFIG

in the prototype to the design project at large. The prototype subject to reconfiguration is a vehicle for refining the requirements of a project. It is not necessarily a means by which program code is developed for the finished product. Since we can and have described a set of problems the solutions to which are enhanced by combining reconfiguration and prototyping, we can proceed in whatever milieu provides the capability to perform such a combination.

4.2 ESTABLISHING RECONFIGURABILITY

In order to initiate reconfiguration, a process must come to a reconfigurable state [HP93]. According to [KM90], such a state exists when a process finishes any communication and has produced all output necessary to allow other processes to conclude their tasks and also reach a reconfigurable state. We believe that this requirement is too restrictive, and that a process can be reconfigured at any time, provided that, with respect to the configuration that it replaces:

- it supplies at least the same functionality to modules above it in the execution hierarchy, and
- it can operate in the same environment.

In order to achieve this degree of flexibility, a program designer should adhere to several basic rules:

- use modular design to as great a degree as possible,
- try to return to a programming depth of one as often as possible,
- ensure that required information is persistent, and
- utilize parameters drawn from the global database wherever there is the slightest chance the parameter might be "tweaked" in any tuning process.

Since the name of a subroutine can be parameterized, using the indirection operation, any branch to a subroutine can be a reconfiguration point. Therefore, the first requirement, using modular design, enables a change of configuration to occur at whatever frequency subprograms are called. If the designer, observing the operation of the prototype, decides to make an implementation or topological change, he could accomplish this by simply replacing the name of the existing routine in the database with the name of the new routine. Then, the next time this call is made, control is transferred to the new module.

This means that reconfiguration can occur at any programming depth. However, following the second rule and returning to the outermost level of program depth implies that the entire module can be swapped during a reconfiguration, while execution continues with the steady-state environment intact. In this way, reconfigurations as dramatic as a complete metamorphosis of software direction, scope, and purpose can be effected.

If the design of the process considers the need to have persistent values, then these values can be set at an outer programming level and will be an inherited part of the operating environment of any subordinate scope, unless explicitly redefined. This ensures that when a reconfiguration is initiated, the successor configuration will inherit the same environment as its predecessor and will be able to maintain a steady-state operation. If the environment is not preserved, then a reconfiguration will have to, indeed, take place only when the system is quiescent so that the appropriate environment can be created from the top down.

The fourth design criterion is stated to empower the prototype designer to make incremental adjustments to the running software, just as an auto mechanic makes small corrections to a running car engine while observing the effect of those adjustments on the instrumentation attached to the car. If a parameter is picked out of the global database each time it is used, then replacing one value with another in that global database will have a near term impact on the operation of the process. It is interesting to note that, if the program is made modular enough, it may not be necessary to anticipate every adjustment point. In the case where an adjustable value is desired where none was provided for, the designer can revise the routine containing the parameter to be adjusted, either making it global or changing its hard-coded value, and then can swap the revised program in at the next iteration, as described above.

Following these rules, a prototype designer is empowered to make changes ranging from the complete metamorphosis of a target tracking system into a banking system (though one wonders how the initial requirements statement could be so general as to have such a thorough change be meaningful) to the simple fine-tuning of a computational algorithm. The more far-reaching the reconfiguration, the more likely that different data structures, topologies, and interfaces will be part of the change. A comprehensive level of change would be very spectacular and would dramatically illustrate the power of reconfiguration, and this degree of change is as easily enacted as a minor modification.

This raises the issue, then, of what kinds of change would be found useful. Recalling that our approach to prototyping is as a tool to resolve high risk areas in the specification of a software product, it appears that the changes would be on the line of "variations on a theme," rather than sensational changes in basic goals and purposes. However, within the scope of the risk to be minimized, any degree of change that would be desirable to the resolution would be supported by RBP. The impact would be as dramatic as appropriate to answering the questions at hand.

All of the above are best illustrated by an example. While we could choose to give a case in which an avionics system is changed into a mortuary accounting system, it is more realistic and illustrative to relate an initial example of target tracking to the M Technology. Let us presume that the designer has identified the information displayed to the user as an element of risk, an aspect which should be prototyped. Initially, he has specified as alternatives the option of displaying either current target information only, or current and historical target information. The objective that he has defined for the user is to be able to ascertain when a target becomes a threat. Let us further presume that several utility functions are available:

- `STATUS(target-number,time)` which returns a delimited string describing all known simulated information about the requested target as of the requested time, including location

```
NOHIST  ;EAS;03:44 AM  18 Jan 1994;
        ;demonstrate M's reconfiguration capability
        ;display status on number of targets in ^NUMTARGET
        ;at the current time only
        ;then hang for # seconds in ^PAUSE

        ;fix the current time, in seconds
        set NOW=$piece($horolog,",",2)

        for TARGET=1:1:^NUMTARGET do
           .set STATUSSTRING=$$^STATUS(TARGET,NOW)
           .do @(^STATDISP)

        ;get pause time, use one second as default (if not
        ;defined)
        hang $get(^PAUSE,1) quit
```

Figure 3: Subroutine NOHIST

coordinates, symbol, time of observation, etc.,

- `X(status-string)` and `Y(status-string)` which return, respectively, the X and Y coordinate for the target described by the status string, and
- `LINE(x1,y1,x2,y2)` which draws a line on the display from the point with coordinates `(x1,y1)` to the point `(x2,y2)`.

The plan is, by varying the number of targets, the information displayed, the update frequency, and the history/no history options, to arrive at an improved design specification.

Figure 2 shows a top-level program which repeatedly calls one subroutine. Upon initiation, the output device descriptor is passed to the program `RECONFIG`. This value becomes part of the local environment, which is available in all subordinate scopes. Also, the program initially sets up to call the subroutine `NOHIST`, by storing that subroutine name in the global variable `^SUBNAME`. In the M notation, a carat (^) in front of a routine name indicates that it is external to the current routine, and a carat in front of a variable name means that variable is part of the external, global database. Then, the routine establishes the initial display routine to be used by storing its name `DISPLAY1` into the global variable `^STATDISP`, and the number of targets to initially display. Next, the routine begins repeatedly calling the subroutine named in the global variable `^SUBNAME`, until a flag, also part of the external database, is set directing it to shut down. When this shutdown occurs, the flag is removed from the database, and the routine exits.

Within this subroutine, there are several points of reconfigurability, which will allow the designer to interact with the user to solve the problem. First, the number of targets can be changed, for example to 8, by executing the statement

```
set ^NUMTARGET=8
```

Also, as stated, the names of the two subroutines to invoke are variable, and can therefore be changed to substitute any desired functionality.

The subroutine initially invoked is named `NOHIST` and is shown in Figure 3. This routine inherits its local variables from the calling routine, so `ODEV` is assigned whatever value it acquired in `RECONFIG`, and this value will, in turn, be available to any routines `NOHIST` calls. The first executable statement establishes the current time by retrieving part of the system clock (available in the `$horolog` special variable). The routine enters a loop which will be repeated for targets numbered 1 through the value found in the global variable `^NUMTARGET`. During each iteration, the simulated status is retrieved and displayed. Finally, the subroutine pauses (hangs) for the number of seconds that are stored in the global database variable `^PAUSE`, before returning to the calling routine.

The final piece of the example is the subroutine called `HIST` and shown in Figure 4. Its operation is similar to `NOHIST`. However, in this routine, two statuses are retrieved, the interval between which is parameterized in `^INTERVAL`. For each, the coordinates are extracted and retained while the status is displayed. Then, a line is drawn from prior to current status.

This example provides us with several ways to demonstrate dynamic configuration. After the main routine is invoked, it quickly reaches steady-state. Because it is specifically written to be modular, at the top level it only calls one subroutine. The point at which this subroutine is called, inside a loop structure, becomes a reconfigurable state. This is because the prototype designer can substitute the second alternative he has predefined by executing the command, from his test platform

```
set ^SUBNAME="^HIST"
```

and the reconfiguration is accomplished on the next iteration. Furthermore, while the prototype is calling `DISPLAY1` to paint the status on the screen, the designer can be writing, testing and

```
HIST    ;EAS;04:01 AM  18 Jan 1994;
        ;demonstrate M's reconfiguration capability
        ;display status on number of targets in ^NUMTARGET
        ;at the current time as well as a prior interval
        ;and connect the two with a line
        ;then hang for # seconds in ^PAUSE

        ;fix the current time, in seconds
        set NOW=$piece($horolog,",",2)
        ;and the prior time interval is a dynamic parameter
        set THEN=NOW-^INTERVAL

        for TARGET=1:1:^NUMTARGET do
           .set STATUSSTRING=$$^STATUS(TARGET,NOW)
           .set X1=$$^X(STATUSSTRING),Y1=$$^Y(STATUSSTRING)
           .do @(^STATDISP)
           .set STATUSSTRING=$$^STATUS(TARGET,THEN)
           .set X2=$$^X(STATUSSTRING),Y2=$$^Y(STATUSSTRING)
           .do @(^STATDISP)
           .do ^LINE(X1,Y1,X2,Y2)

        ;get pause time, use one second as default (if not
        ;defined)
        hang $get(^PAUSE,1) quit
```

Figure 4: Subroutine HIST

debugging DISPLAY2. When he is satisfied that the new display subroutine is correct, he can execute the command, from his test platform

```
set ^STATDISP="^DISPLAY2"
```

and this new routine will be invoked at the next call. Finally, the designer can dynamically alter the time interval encompassed by the two status points in HIST by varying the number of seconds in the global ^INTERVAL. Similarly, he can tune the time between updates by dynamically changing the value of the variable that directs the pause time.

```
set ^PAUSE=3
```

This will increase the hang time to three seconds at the next iteration of whichever subroutine is in the execution path. Also note that the routine will run forever unless the tester issues the command

```
set ^RECONQUIT=""
```

which, by creating and defining the global variable ^RECONQUIT, directs the main routine to terminate. This, of course, is also a form of reconfiguration.

In this example, the only entity that is not reconfigurable is the main routine RECONFIG. Below that level, any amount of change is possible, simply by reconfiguring the pointer to which subroutine is called. For example, we are not limited to sending bells to a device or to only one level of depth. The routine pointed to by ^SUBNAME could perform a variety of functions or call other subroutines which, themselves could be variably invoked.

Now that the methodology has been crudely demonstrated, we would like to offer another very simplified example, again clearly demonstrating the interaction between this reconfiguration capability and prototyping. Departing from our previous example of user interface design, we will draw upon a scientific computing scenario. Assume that we are developing a long-running mathematical analysis program. After it reaches steady-state, we observe that the results are not converging as fast as we would like. If written according to this protocol, we could set different values for some of the computation parameters and observe the effect on the speed of convergence. Furthermore, suppose that we, as observers of the instrumentation built into the prototype find that we would like to watch the values taken on by a particular variable that was not initially anticipated. We could edit a renamed copy of the routine, and insert a command such as

```
set ^NEWOBSERVATION=VARIABLEOFINTEREST
```

When the revised copy was completed and filed, we would modify the global software switch that would trigger its invocation in place of the old copy. This would provide us with a new view into the operation of the routine, allowing us to display the most recent value of the variable we have specified. To do this, we would issue the command

```
write ^NEWOBSERVATION
```

and the current value would be presented to us.

These examples have been presented using the features and the syntax of the M Technology. In a prototyping effort that is based on RBP and M, it would be necessary to have a member of the development team with a basic understanding of the M Technology. This requirement is further discussed in section 5.1.1.

5 UNDERTAKING THE RBP PROJECT

We are implementing the RBP prototyping process in an environment that employs reconfiguration technology rather than the start/stop/change/restart technique. This includes investigation into the several aspects of the RBP protocol as described in section 2. This work will increase understanding of the applicability of the various prototyping and reconfiguration techniques, and the synergism and interplay between them. We intend to incorporate the results of our research into a draft guide to the use of RBP.

5.1 RESEARCH PLAN

From a spectrum of research questions and issues, we have selected the area that we believe has the broadest implications. The resolution of RBP implementation and applicability issues constitutes the bulk of our work. In the first major step, we are building the reconfiguration-based prototyping environment, so that its viability and features can be demonstrated and experimented with. Details of this implementation are described in section 5.1.1.

As the second principal part of our work, we will use this model to develop an understanding of the applicability of RBP and cost overhead created by its use. As discussed previously, there are several distinct types of and motivations for prototyping, and different facets to reconfiguration. We will explore a variety of situations to be able to evaluate to what extent different combinations are enhanced by the RBP protocol. A more specific discussion of this undertaking is presented in section 5.1.2.

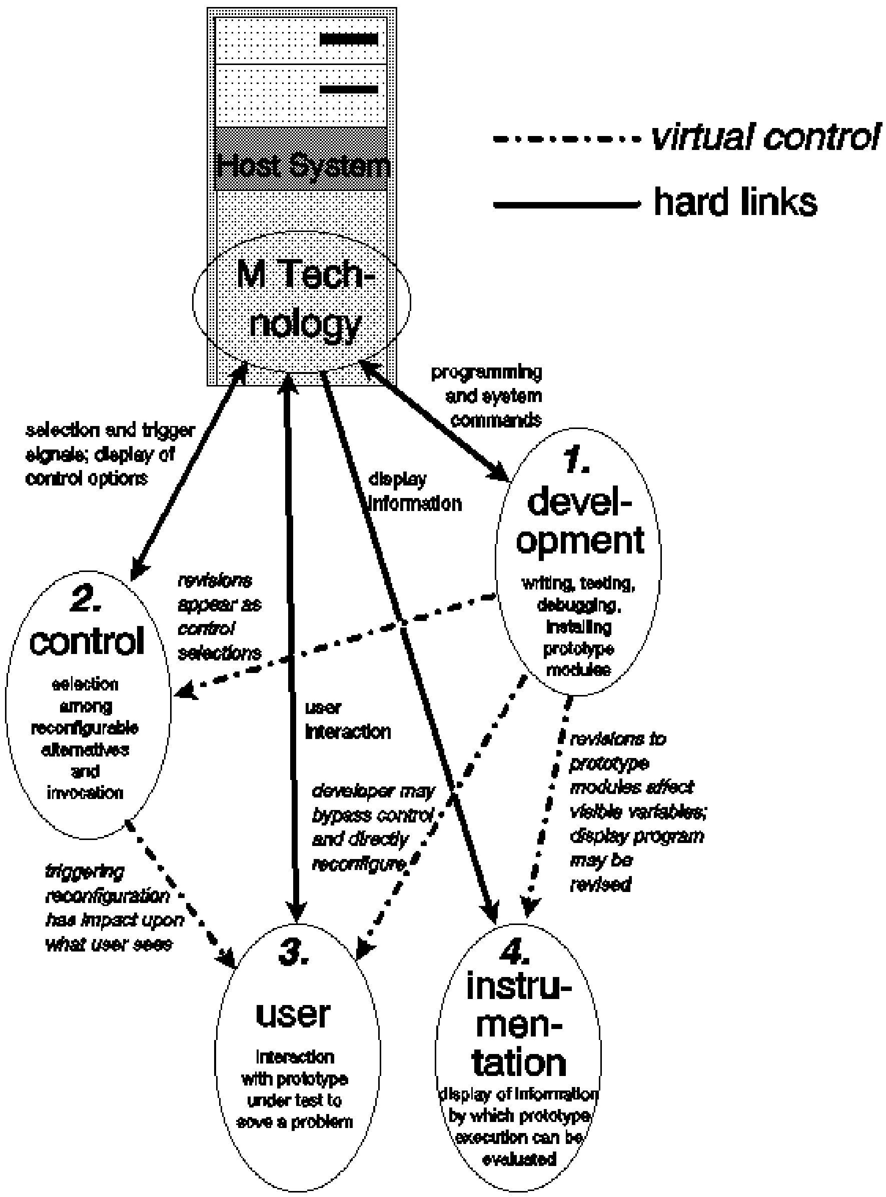

Figure 5: Implementation Concept

5.1.1 DESIGN RBP PLATFORM

The successful design and implementation of the RBP platform, as described in this section, will confirm the first part of our hypothesis: we will substantiate the feasibility of the RBP concept. The applicability of the M development environment will be affirmed.

The model will be implemented in the M Technology on a host system, as illustrated in Figure 5. M has been chosen as the development medium because of the empowering features of the language that have been discussed in section 3.1. Although we recognize that M is not the language of choice in the academic community, we have presented that the prototyping effort itself can be independent of the language of the overall project, and M offers an environment which is conducive to both the development, exercise, and demonstration of the RBP protocol. There are four major components to the design. These may be envisioned as separate windows being driven by the host. Each of these parts is attached to the host by a hard communications link, with the processing being performed internally to the host. However, a virtual control hierarchy is in place by which RBP proceeds.

The governing process, shown as "development" (1.) in the figure, is the window at which the designer works. This individual would be the developer cited in [FD89] and [MC83]. His task is to interact with the user to tailor the software under test to best address the problem to be solved or risk to be reduced. As described in section 2, his goal in employing the RBP methodology is to reduce lag time between proposal and presentation of alternative ideas. From this "development" position, the designer can achieve any extent of reconfiguration that is consistent with the objective of refining a requirements specification to reduce risk. Of course, as stated before, more dramatic reconfigurations are always possible, but are moot if they are not oriented toward solving the problem at hand. From this logical position, the designer has the capability to

- interact with the host system at the executive level,
- initiate the execution of the prototype,
- write, test, and debug new modules (while the prototype is continuing to execute) prior to releasing them for insertion into the prototype,
- control the "control" (2.) window by causing alternatives and reconfigurable parameters to be shown as options,

- optionally, exercise direct control over the "user" (3.) window by bypassing the "control" and directly manipulating the environment, and
- control the "instrumentation" (4.) by means of either the display code or global variables built into the prototype, or by dynamically revising the module that is tasked with manipulating the instrumentation window.

It is this member of the prototyping team that must have a working knowledge of the fundamentals of the underlying software. In the case of our examples, it would be this individual who knew how to use the resources of the M Technology.

The control panel process, shown as "control" (2.) in the figure, is the window at which prototype alternatives are shown as selectable options. This position may be considered a liaison function, much as the architect in [MC83]. This member of the development team would have the responsibility for timing the institution of changes and of manipulating the detailed parameters that would be provided as options. Working at this window, a designer may

- select from prototype alternatives in the form of different modules, routines, or versions that can be reconfigured into the executing model,
- establish values for global parameters which will have an impact on the performance of the prototype,
- terminate the prototyping session, and
- exercise control over the "user" (3.) window by invoking the configuration that has been selected.

The participant who is performing the test of the prototype by attempting to reach the established goal operates at the "user" (3.) window. We have previously discussed the perceived value of having the end users involved in the evolution of specifications for a product, a factor often overlooked in practice [LSZ93]. By including this station as an integral component of the RBP protocol, we are ensuring that user input will be considered. The duties of this individual are to

- interact with the prototype presented to him, attempting to solve an experimental problem or reach an articulated goal,

- evaluate available alternatives as they are invoked from the "control" (2.) window, and
- indicate improvements that the designer, working at the "development" (1.) window, can dynamically incorporate into the prototype.

The fourth part of this prototyping paradigm is the "instrumentation" (4.) window. When we consider a prototype from the engineering perspective, it seems natural that the capability exist to extract performance information from the executing program, whether it is considered an experiment, as in [CPP94] and [PLC91], or a model. Observing the output at this window, the designer is able to

- observe and analyze data extracted from the executing prototype, including operational parameters, performance indicators, and whatever other instrumentation is incorporated in to the prototype, and
- track the progress of the prototype.

5.1.2 TEST AND EVALUATE RBP PLATFORM The second part of our hypothesis concerns the flexibility of the model and the ways in which RBP can benefit the computer software development process. Consideration of this issue is the area in which our work has the potential to make a contribution to the body of knowledge in computer science. Our goal in this part of our project is to provide new insights into prototyping and reconfiguration that can be used as reference material for further experimentation with the RBP approach.

Central to our contention that prototyping can be more effectively undertaken using our model is the presentation of a case-in-point. We will draw an example from the Baltimore Longitudinal Study of Aging (BLSA), which is an ongoing research project of the National Institute on Aging of the NIH. Initiated in 1958, this study of human aging has generated a huge database and an accompanying library of research and administrative software. Within the scope of this computer system, we will demonstrate the solution of a design problem using our RBP approach.

The choice of a particular existing problem on which to demonstrate RBP will not, however, allow us to generalize about the places and situations where RBP methods may be brought to bear. There are several schools of thought on the motivation for prototyping and different approaches to each [LSZ93]. For example, prototyping may be for risk reduction [PLC91] or interface design

[MC83]; it may be rapid or evolutionary [WK92]. Similarly, reconfiguration technology includes the aspects of topologic, geometric, and implementational change [HP93]. An understanding of the interactions of these factors will be afforded by observation of our RBP model, as it pertains to both our BLSA example and other situations. We may find, in the exercise of our RBP model, that certain reconfiguration options combine differently with prototyping methodologies, and that these combinations are more or less effective depending on the goal of the prototyping effort. Of particular value to the field of computer science would be a classification that could be employed as reference material for these areas of design. Our objective is to assemble and deliver such a reference.

We are going to do this by testing configurations and using what we learn to populate a table of results. We will assemble a series of practical software development scenarios and examine them to determine what types of prototyping method and goal are called for; we will decide which facets of reconfiguration are applicable. Then we will use our model to simulate each situation and extract information about performance under these conditions. We expect that we will be able to construct a three dimensional grid that will serve as a guide to the appropriateness of our RBP protocol.

- The axes of this grid will be prototyping method, prototyping goal, and reconfiguration options.
- As noted above, the values along the prototyping method axis will include rapid and evolutionary; the values along the prototyping goal axis will include risk reduction, interface design, algorithm refinement, and efficiency optimization; and the values along the reconfiguration axis will include topology, geometry, and implementation.
- The cells of the grid will provide an indication of how much advantage might be gained by using the RBP protocol in this situation.
- Further cell content will give an indication of the circumstances under which an application might locate in this cell.

We believe that this taxonomy will provide an insight and guide to the utilization of prototyping, reconfiguration, and RBP. Once a draft of this guide is assembled, we can apply it to the BLSA examples. In so doing, we will be able to refine the information so that other investigators can experiment independently with RBP.

6 CONCLUSIONS

We have briefly discussed and illustrated the benefits that can be had by bringing dynamic reconfiguration technologies to bear on the task of prototyping. We have seen that there are areas relating to fine tuning of algorithms as well as coarse tuning of methodologies that can be facilitated if a software developer has the power to modify executing software in place.

We have described a methodology which will allow a designer to include the user in the refinement of design specifications, in a manner to reduce the uncertainties of the final design. We have made the proposition that we can define the technology necessary to implement these protocols.

In support of this thesis, we have introduced the M Technology as a powerful platform which provides the capabilities to make real time modifications to software without, in most cases, the need to stop and restart. Rather than losing the steady-state, it is preserved and execution can proceed unimpeded.

Finally, we have described a configuration in which the RBP method is being implemented, tested, and evaluated. The outcome of this research will be a taxonomy that will serve as a guide to the further use and refinement of reconfiguration-based prototyping.

REFERENCES

[CPP94] Chen Chen, Adam Porter, and James Purtilo, Tool Support for Tailored Software Prototyping, Proceedings of Symposium on Assessment of Quality Software Development Tools, pp.171-181, June 1994.

[FD89] Daniel A. Fern and Scott W. Donaldson, Tri-Cycle: A Prototype Methodology for Advanced Software Development, Proceedings of the Hawaii International Conference on System Sciences, vol. 22:2, pp. 377-386, January 1989.

[HP93] Christine Hofmeister and James Purtilo, A Framework for Dynamic Reconfiguration of Distributed Programs, Computer Science Technical Report Series, CS-TR-3119, Department of Computer Science, University of Maryland, August 1993.

[KM90] Jeff Kramer and Jeff Magee, The Evolving Philosophers Problem: Dynamic Change Management, IEEE Transactions on Software Engineering, vol. 16, no. 11, pp. 1293-1306, 1990.

[LSZ93] Horst Lichter, Matthias Schneider-Hufschmidt, and Heinz Zullighoven, Prototyping in Industrial Software Projects Bridging the Gap Between Theory and Practice, Proceedings of the 15th International Conference on Software Engineering, pp. 221-229, May 1993.

[MC83] R. E. A. Mason and T. T. Carey, Prototyping Interactive Information Systems, Communications of the ACM, vol. 26, no. 5, pp. 347-354, 1983.

[PLC91] James Purtilo, Aaron Larson, and Jeff Clark, A Methodology for Prototyping-In-The-Large, Proceedings of the 13th International Conference on Software Engineering, pp. 2-12, May 1991.

[PFW94] James Purtilo, Charles Falkenberg, Elizabeth White, William Andersen, and Tess Ollove, An Exercise with Prototyping Technology, 1994.

[Pur91] James Purtilo, Dynamic software reconfiguration supports scientific problem-solving activities, Invited paper, Proceedings of IFIP Conference on Programming Environments and High-Level Scientific Problem Solving, September 1991. Also appears in IFIP Transactions, ed.Gaffney and Houstis, North Holland, pp.245-254, 1992.

[WK92] David P. Wood and Kyo C. Kang, A Classification and Bibliography of Software Prototyping, Technical Report CMU/SEI-92-TR-13, Software Engineering Institute, Carnegie Mellon University, April 1992.