The Construction of Meta-Tools for Program Visualization

of

Parallel Software

Ph.D. Thesis Proposal

James Arthur Kohl

February 4, 1992

Ph.D. Committee:
Thomas L. Casavant - Chair
Jon G. Kuhl
Michael R. Lyu
John P. Robinson
Teodor Rus

Abstract

Program visualization has been successfully applied to alleviate the complexities of parallel software development. However, those program visualization tools which utilize the more advanced visualization techniques tend to be too complex and special-purpose to be generally useful. The use of meta-tools, or tools that construct and manipulate other tools, can be applied to simplify the use of complex environments, but meta-tools for program visualization can still require extensive specification for views and information processing details.

This proposal provides a design methodology for program visualization meta-tools for parallel software that simplifies the use of such tools while maintaining a high degree of flexibility and expressive power. The approach is based on a *meta-tool circulation architecture* model that organizes the details of the user specification, and provides a circulation of information which supports a formal means for indicating relationships among that information. The overall user specification is divided into independent modules containing distinct entities, and the relationships among these module entities are identified using a powerful *relationship mapping language*. This language maps conditions on selected entities to manipulations that modify the entities, allowing the state of an entity to be controlled in terms of the state of any other entity or itself. The mapping language supports arbitrary levels of abstraction in manipulating entities, allowing a full range of possible detail. As a result, visual analyses can be specified efficiently, utilizing only the minimum level of detail necessary. To demonstrate the feasibility and usefulness of this approach, a specific program visualization meta-tool design is proposed based on the methodology.

Outline

I. Introduction & Motivation

II. Background Perspectives

III. Problem Statement

IV. Approach

V. Support

VI. Conclusion

VII. References

1. Introduction & Motivation

The use of parallel computer systems is by no means a new or foreign concept, and many multiprocessor architectures of varying sizes and shapes have been in production for several years. However, even after years of contemplation of parallel computation, the true nature of the field is still not well understood. There is no standard approach to viewing the parallel paradigm, and there is no agreement on how best to develop parallel software that efficiently executes on parallel hardware. It comes as no surprise that the tools created to assist in developing and tuning parallel software are likewise not well understood.

The inherent difficulty in developing and tuning parallel software is that the user cannot readily identify the precise internal behavior of the software. **Program visualization** [Mye88] has been successfully applied to alleviate this problem by representing complex internal software behavior in an abstract and intuitive manner using graphics [Koh91,CaK92a,CaK92b]. Yet, no standard approach or paradigm exists for visualizing parallel software. There are many open questions yet to be answered in this area, including some serious, fundamental, unresolved issues. There still are not concrete answers to the questions of which data is most useful for performance tuning and debugging, and, most importantly, how the data should best be displayed. Each user has a unique, personal perspective and intuition on parallel software development, and perceives visualizing the behavior of software differently. As a result, a majority of existing program visualization tools are not sufficiently general and have not gained widespread usage.

In addition, there is a lack in understanding the fundamental, underlying concepts in parallel software design, leaving the visualization tool designer guessing, at best, the needs of the user. Existing tools are either simple and inflexible, based on superficial or aggregate information, or are of comparable complexity to the parallel software itself, and still may not be generally applicable. While the simpler tools are extremely useful in many cases, there are difficult problems that could likely be solved more effectively using advanced visualization techniques. Even so, many of the more flexible and powerful tools are too complex to be used easily, and are typically avoided. There is a need for highly flexible and powerful visual analysis tools that can efficiently utilize more sophisticated graphical techniques.

One approach for providing this power and flexibility is the use of **meta-tools**, or tools which construct and manipulate other tools. The concept of meta-tools is based on generalizing classes of tasks into their fundamental parameters. By specifying and manipulating these parameters, a user can implement a specific task from within the class. A program visualization meta-tool allows users to customize visual analysis environments to suit individual perspectives and intuitions regarding the viewing of parallel software behavior.

It should be noted that constructing a general meta-tool does not necessarily require a full understanding of parallel software and visualization. In fact, a meta-tool can serve as a basis for gaining such understanding. By collecting those aspects of analysis which *are* understood, an environment can be constructed that not only provides useful analysis features, but that also allows the user to arbitrarily experiment within the given paradigm and gain new understanding.

Unfortunately, the use of meta-tools can be complex, requiring extensive specification of views and information processing. As a result, users may discard the approach in favor of simpler, albeit less effective means. This proposal describes a design methodology for program visualization meta-tools that simplifies their use while maintaining a high degree of flexibility and expressive power. The methodology utilizes two concepts:

- Meta-Tool Circulation Architecture
- Relationship Mapping Language

The **meta-tool circulation architecture** organizes the complex aspects of the required user specification. Information circulates in an orderly flow that allows fundamental relationships among various details to be concisely specified. The organization is based on a generalization of the program visualization procedure. By identifying the fundamental parameters of this procedure, specification details are categorized into several key **components**. The details of each component are grouped into individual **entities**. All entities for a component are maintained in a single **module**, such that the specification of details within each module is independent from the other modules.

As information circulates through the meta-tool architecture, user-specified relationships dictate how the state of each entity affects the state of other entities. The **relationship mapping language** provides a precise means for combining the details among modules into a specification of these relationships. Each relationship is specified by a **mapping** that links conditions on module entities to a set of highly flexible operations which in turn manipulate the module entities. The mappings utilize details of the entities at arbitrary levels of abstraction, allowing high efficiency by ignoring unnecessary detail. The mappings do not specify the precise implementation details, but rather the fundamental concepts on which the analysis is based. The result is a powerful, flexible, visualization tool design that relies on high-level directives.

To demonstrate the feasibility and usefulness of this methodology, a specific program visualization meta-tool design is proposed, utilizing the circulation architecture and mapping language concepts. Useful features specific to parallel software visualization are proposed. The resulting customizable tool design will provide a framework for experimenting with visual analyses. In addition, the tool design will be generally useful for a variety of more advanced program visualization analyses.

The following section describes the foundation of the methodology by summarizing several perspectives on program visualization. Section 3 concisely provides the problem statement. In section 4, the proposed design methodology approach is described, along with a brief overview of the demonstration design. Section 5 provides justification of the key features of the methodology, along with several examples illustrating use of the concepts.

2. Background Perspectives

In reaching a better understanding of program visualization, there are at least three perspectives to consider [KoC91c]. These perspectives focus on the user's needs and what the tool designer is capable of providing to satisfy them. The perspectives are:

1. User View
2. Meta-Tool Designer View
3. Toolkit View

The user view emphasizes the qualities and features that a user would need or want in a program visualization system, independent of the feasibility of actually implementing those qualities or features in a real tool. Within the context of program visualization, a meta-tool refers to a visually-oriented tool or interface for specifying and manipulating a visual analysis. The meta-tool designer view focuses on what kinds of features are possible and reasonable in a meta-tool, and explores the extent to which this approach meets the needs and desires of the user. The **toolkit** view poses an alternate approach to the meta-tool view, and leans toward simpler, less flexible visualization environments that consist of collections of fixed, general-purpose views. These toolkits may not provide the customizability of more general visualization tools, but may sufficiently satisfy the needs of the user and are easier to learn.

2.1. User View

There are several issues that are important to the developers of software who use performance tuning and debugging tools, and there are several others that are not. The issues that are important to tool users involve the efficiency and ease of use of the tools, what features and functions are supported, and the amount of control that the user has over operation of the tool. The user does not care how a tool is implemented, how elegant or general an approach it provides, or how complicated and difficult it was to build the tool, as long as the tool helps the user do what must be done. The following sections discuss in more detail those issues of greatest importance to the user.

2.1.1. Customizability

The importance of customizability to the user is surprisingly higher than might be expected, and constitutes one of the most serious problems in designing tools. Each user has a personally tailored set of individual preferences and perspectives on the manner in which software should be developed, debugged, and optimized. As a result, each user will desire a unique collection of options and features that, unfortunately, may expand and change over time as the user gains experience or adopts new perspectives. Any tool developer who has attempted to custom design a system for a specific user has encountered the "Law of Unlimited Options". This "law" states that no matter how many features and options are added to a tool, users will continue to request new options, such that the number of options increases without bound. This scenario likely occurs in part due to the creativity of users, because the installation of new options inspires ideas for other new options.

Another form of customizability results from tedious sequences of tasks that the user is confronted with on a regular basis. Many users will desire continued optimization of these operations to an extreme. Satisfaction of this desire leads to tools that consist of multitudes of special-purpose, custom macros and "single-keystroke" commands. The user will not be completely satisfied unless a tool can be customized to match specific personal preferences and requirements.

PARADISE [KoC91a,KoC91b] provides customizability through custom-designed objects used to construct visual models for analysis. These models specifically cater to the individual user intuition for a given analysis. The Pablo [ReO91], Hyperview [MaR89,Nic90] and Triplex [CoK90,Nic90] visualization tools provide user customizability by allowing the user to organize and control views using filter modules that are graphically interconnected to process and display data. Other tools such as Tango [Sta90], Voyeur [BaS88], BALSA [BrS84,BrS85,Bro88] and Q+ [MeM85,Nic90,FuK91] produce arbitrary animations based on specifications written by the user.

2.1.2. Flexibility and Expressive Power

While desiring the capability to customize views, the user does not want to be forced to provide complex specifications for each different application or analysis. Also, in addition to the complex, customized views, the user is likely to want some very simple alternatives that will provide basic or fundamental information automatically or with little effort. In many ways, this dual approach to visual analyses relates to the expressive power of a tool. The methodology or interface of a given tool must be capable of producing complex, detailed, custom displays, while at the same time supporting certain fundamental analyses without substantial effort. An illustrative example of such an interface is that provided by the C programming language. A program in C can be simple and consist only of the fundamental language constructs, with little more structure than Fortran. Alternately, a program can be written that utilizes complex structuring, unary operators, and pointer manipulations.

Existing tools tend toward opposite ends of the spectrum with respect to flexibility and expressive power. Tools such as PARADISE provide a high flexibility, but without the inherent capability for simpler automatic displays. Tools such as ParaGraph [Hea90,HeE91] and TOPSYS [BeE88,Bem88,BeB90,BeB91] produce displays automatically with only a few mouse clicks, but have fixed sets of views and a more limited flexibility. Some middle ground with respect to flexibility and expressive power needs to be identified, such that the benefits of both of these approaches can be realized in a single unified interface.

2.1.3. Efficiency and Ease of Use

Clearly, one of the most important qualities for any tool is the overall efficiency with which the tool operates, along with the ease of using the tool. The central features and functions that the tool provides must be as automatic as possible, as these will be used most frequently. The interface for specifying the more complex, customizable features and options must be concise and intuitive, and should be as efficient as possible. If the overhead penalty for the more sophisticated features is too great, they will always be discarded in favor of the simpler features, even if the simpler features are not as effective in the long run. Existing tools provide varying levels of efficiency and ease of use. ParaGraph and TOPSYS both provide highly efficient interfaces for producing views, and typically require only a few keystrokes or mouse clicks to arrange an entire analysis. The Pablo/Hyperview and Triplex systems use a simple graphical interface for specifying the filtering architecture, resulting in simple, straightforward view construction. PARADISE reduces some specification overhead of its custom displays by allowing reuse of the more involved portions of the object definitions for the visual models, but requires some involved static textual specification.

A tool should also include some consideration of default user preferences. For instance, the aspects of views which will be similar for different analyses, due to particular user preferences for color or format, should be automatically instantiated in each new view without explicit specification by the user. Some form of simple database interface for maintaining and accumulating standard user configurations would be desirable. The one-time overhead in specifying such preferences and standard configurations would be reusable many times with only minor adjustments, if the format chosen is complete. The central difficulty with such a database would clearly be deciding precisely which preference information should be maintained. The Pablo/Hyperview systems provide a defaults manager for maintaining a collection of user defaults preferences.

2.1.4. Multiple Granularity Control

There are many aspects of visualization that benefit from some type of "focusing" capability. It is often useful to adjust the amount of visual stimulation to prevent visual "overload", as well as emphasize certain aspects of a view to reveal important or interesting contrasts or relationships.

2.1.4.1. Visualization Speed Control

The nominal speed of an animation or dynamic view can proceed too quickly for full user comprehension, even after repeated viewing of the same sequences. Alternately, long periods with no activity can waste time and wear the user's patience thin. Therefore, some form of speed control would be helpful to allow quick traversal through the display of uninteresting portions of data, and the ability to slow down the display to closely examine the important details. PARADISE, SHMAP [DoB90], and other tools support variable speeds for various aspects of displays, and many other tools support some form of stop/start or single step control. It should be noted that the concept of speed control contradicts the basis of real-time display tools. One of the benefits of post-mortem display tools is the freedom to manipulate the time axis as well as play back program executions in "real-time".

2.1.4.2. Visualization Direction Control

In conjunction with variable speed, the ability to reverse the direction of an animation is useful. By allowing both forward and reverse playback of execution traces, interesting perspectives on the behavior of software can be provided, but more importantly, the overhead in viewing large traces can be greatly reduced. When speeding through long, empty portions of a trace, there is no means for identifying precisely when the next interesting event might occur, especially when it is not known ahead of time what that event might be. This causality problem will likely cause a situation where, by the time a detail is identified, it is too late to stop, and the detail has passed. In this case, being able to rewind the animation or display back in time to the interesting detail is very important. Without this capability, the entire trace may have to be replayed again up to that point to view that detail. Another reason for such reverse playback is the inherent likelihood that the human user will simply miss a desired detail or make a mistake that causes the detail to pass before stopping the display. In this case, the user could simply rewind the trace to catch the missed detail as before, rather than replaying the whole trace again.

This reverse playback concept is reasonably straightforward to include in timeline-oriented tools, by simply keeping track of key pieces of information as the timelines are generated. Several such tools provide forward and reverse scanning of displays. Unfortunately, for more complicated animation systems, where the graphics displayed depends on complex functions, relations, or even programmed behavior, as in PARADISE, it may be nontrivial to implement a reverse display. An inverse function will not likely exist for more complicated analysis or animation functions. There is no standard method for undrawing graphics, especially when different portions of the display dynamically overlap with other portions. For tools with this type of display, the overhead in time and space to maintain the information necessary for reverse playback may be large. Further, note that it is not completely possible to optimize a reverse playback scheme by discarding older information, as the user may very well need to rewind through large portions of a trace, reaching past the earliest point where information has been retained. Indeed, there may be no best "safe" point, past which information need not be saved, as no matter how far back the point is selected, the user might still need to go back before that point.

2.1.4.3. Abstraction Level & Display Size

The last dimension of focusing and granularity control corresponds to overall display presentation. Both the abstraction level and the aggregate size of a display might be adjusted to provide better focus of the desired details and information. It may be difficult to pick out important details if too much information is displayed simultaneously, and if no details are given the display may very well be useless. By adjusting the abstraction level of a display, where appropriate, a user could identify anomalies at a high level and then focus in on details to identify the source of the failure. Similar benefits are obtained by adjusting and scaling the visible perspective on a large display, to show overall behavior versus internal details. As with speed and direction adjustments, a trace can be traversed at a high level until interesting details are identified, and then these details can be more closely examined. PARADISE provides some aspects of abstraction level adjustment, but not yet in an interactive manner. Other approaches to the adjustment of abstraction level have been successfully utilized in the Triplex system, which explicitly identifies contrasts by summarizing or "binning" similar portions of the display into groups, thereby reducing the overall display size and complexity.

All these features reduce sensory overload and allow natural control over the density of detail in a display. Several analogies can be drawn — in human grasping, the limb moves very quickly to the location of a physical object, and then the hand slowly closes around the object to grasp it. Similarly, in learning it is often beneficial to confront a concept first at a higher level and then "slow down" and closely examine the lower level details to gain full understanding. Also, consider Gary Larson’s "Far Side" cartoons, in which the meaning may not be readily evident at first, until some small detail in the cartoon reveals the "punch line".

2.1.5. Animations and Views

There are many unanswered questions regarding what types of displays are most desirable to users in analyzing parallel software. It is not known what level of abstraction or detail is the most effective — another justification for interactive control of these parameters as described above. The problem of scalability for large displays and massively parallel applications is still not well resolved. In some cases, analyses on a small scale are sufficient to describe the potential results on a large scale, hence parallel software can be developed and debugged using smaller numbers of processors. However, many large scale parallel systems require large scale details to fully understand the behavior of the software to adequately tune its performance and debug it. Some of the novel approaches such as the Triplex "binning" feature attack this problem, but more work needs to be applied toward fully resolving this issue.

Another problem with respect to creating views or animations is the lack of a standard approach to displaying information. There is no agreement on a "best" display format. There are many alternatives regarding the use of fixed versus dynamic images, and mobile versus immobile images within a display. In addition, animations can move in specific, discrete steps or can consist of a continuous dynamic flow of graphics. Existing tools support a wide variety of combinations of these choices. PARADISE uses immobile, dynamic images with discrete animation, while TOPSYS uses fixed, mobile images. Tango supports full, continuous animations.

2.1.6. Extra Assistance in Analysis

Another more obscure aspect of the user's view is the desire for a tool to be able to capture the user's attention and draw it to the critical details. While a tool cannot be expected to intelligently direct a user through the software development process, displays can be constructed to emphasize contrasts and relationships in the data in a way that guides the user to important or significant information. The user may not always necessarily know where to begin in identifying a bug or performance bottleneck, and visual tools should be capable of partially leading a user to the source of the problem. Alternately, certain displays might inspire ideas in the user, leading the user to examine particular details more closely by adjusting the view or by creating an entirely new view.

2.2. Meta-Tool Designer View

As opposed to the straightforward, albeit somewhat evasive, perspective of the user, the perspective taken by the meta-tool designer is more complex. The goal of the meta-tool designer is to provide the full benefit of powerful, sophisticated displays while at the same time attempting to reduce the associated overhead to an acceptable minimum. The central task in constructing a meta-tool is to mask as many internal low-level details as possible from the user, and require the user only to specify the necessary high-level details for a given analysis. The user should not be required to understand or manipulate the underlying graphics or simulation systems, but rather should focus on the abstract specification of views, using the higher level tool functions as building blocks.

This masking of low-level internal details is straightforward to implement by utilizing common principles of modularity and layered interfaces, *as long as a suitable means for specifying visual semantics exists.* There must be a clear, concise, formal specification syntax for creating and controlling views — the lack of such a standard "visualization language" to date for parallel software analysis is the central roadblock in designing tools for program visualization. Several **visual programming** [Mye88] and other techniques have been applied to controlling visual analyses, but the problem of actually specifying the visual analysis or view is not yet well understood.

Textual languages are cumbersome for specifying some aspects of visual behavior. What is necessary is an abstract, visually-oriented language that describes the drawing and manipulation of graphics at a high level suitable for a non-graphics expert, and with appropriate semantics for parallel software analysis. Note, however, that the existence of such a language might indeed constitute a solution to the visual programming problem, and even further borders on transcending conventional and natural languages to be an abstract, symbolic, universal language. Clearly, this is not fully possible given current technology. However, within the limited context of program visualization it is feasible, and a sufficient syntax can be constructed.

Some existing visualization tools have explored various alternatives to masking details from the user, as well as different techniques for specifying views. PARADISE utilizes the "plug-in" simulator concept to hide a majority of internal detail from the user for its animated simulations (see Figure 1). Using this approach, those details consistent for all simulations are provided by the tool, and the user simply "plugs in" a particular simulation module to specify the necessary detail for that simulation. This approach hides internal details well, but can require significant specification of the remaining details by the user depending on the user interface. The Tango visualization system utilizes an abstract visual manipulation language to specify the movement of graphic objects in conjunction with a program's execution. A more complete, formal, graphical animation language is necessary in future tools to allow more general animations, and without integration with and annotation of user applications.

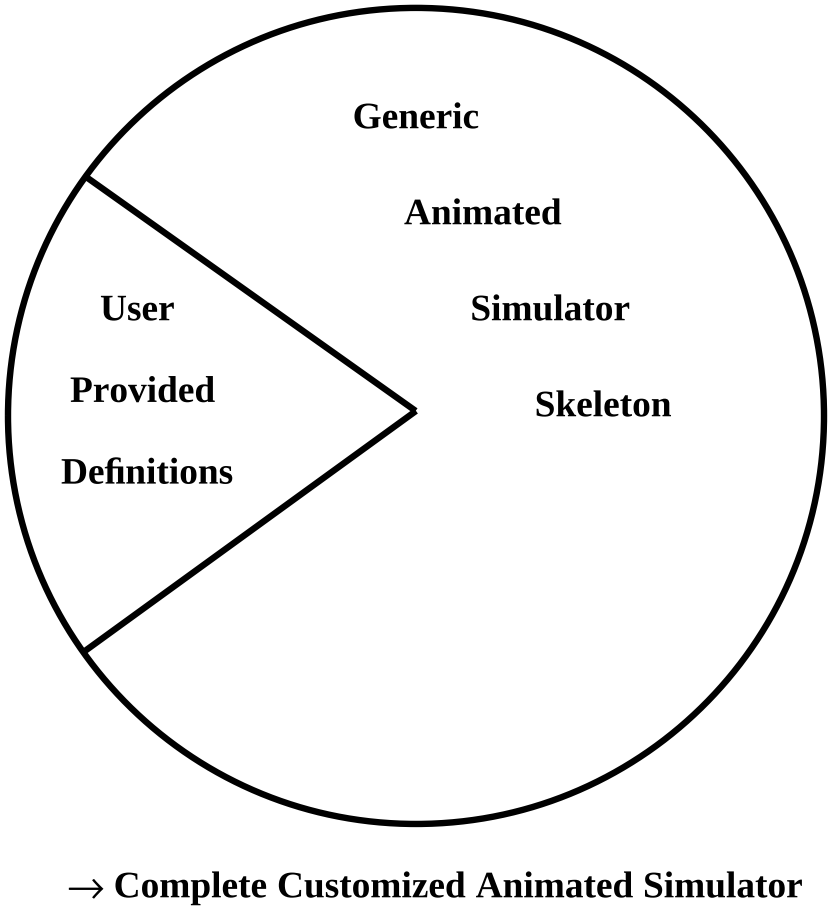


Figure 1: Plug-In Simulator Concept

An important aspect of any scheme a meta-tool designer creates is that the scheme must be customizable to suit the user's particular preferences, as discussed in the previous section. To provide customizability in views, and analysis features and options, it is necessary for the meta-tool designer to generalize and formally classify the fundamental parameters of the analyses.

Designing a meta-tool in this way requires a full understanding of the needs of both the user and the basic concepts and parameters of the paradigm under analysis. It is the task of the meta-tool designer to organize and partition a problem space into its fundamental characteristics and parameters, and then to construct analysis tools that are generally applicable to these core entities. The operations on these parameters must be understood, and a means for effectively specifying these parameters must be constructed. For customizability, the meta-tool environment must avoid static semantics and predefined configurations, and rely on the user to specify the fundamental parameter values.

Further, to fully utilize and benefit from the generality of the tool, there must be a flexible interface for the user to specify the necessary details. The user interface must be highly interactive and allow dynamic specification of the semantics of analyses. Modification and manipulation of analyses should be possible without great expense to the user, and every attempt should be made to avoid the necessity for re-compilation and restarting analyses from the beginning. Specification of dynamically changing characteristics and parameters should be interpreted by the tool, and static details should be dynamically compiled internally where possible.

Care should be taken in creating any tool to reduce the inherent user overhead in using the tool. Interfaces should be concise and simple, with common operations being optimized down to a minimal number of steps. The aggregate number of keystrokes, mouse clicks, etc., must be minimized by intrinsically including automatic features, as well as the capability for user-defined macros. Especially for analyses of parallel systems, an efficient method for replicating details of a specification is useful. Any aspect of an analysis that can be reused constitutes a substantial savings to the user. A simple storage and retrieval system or a library system for analysis specifications would be beneficial. The user will likely have particular preferences for certain analysis parameters which will remain constant over several analyses. These parameters should have the ability to use default values at the user's discretion. The meta-tool designer must select these candidate parameters and organize them into some form of database that can efficiently maintain the necessary default choices.

An important issue in designing a meta-tool is deciding on the proper model for the underlying simulations or analyses. There are several methods for driving graphical views and animations. For example, a state machine approach may be taken, in which case the meta-tool designer must decide upon the best approach. A Moore approach would result in large collections of states that would simply control the visual output, where a Mealy approach would utilize fewer, more complicated states. Alternately, the use of dynamic filter schemes, to process raw data and channel it to displays, could be used in place of state machines.

For any of these underlying models, it is necessary to decide what types of animations or views will be supported, and how those views can be specified. An animation can be constructed from sequences of discrete images, requiring specification of each such image, or perhaps an animation could be created by completely specifying an initial state, and then utilizing special functions to continuously modify or modulate the view. Another alternative involves using expressions or functions to relate program execution data directly to animation behavior. The best choice for this aspect of program visualization remains unknown.

2.3. Toolkit Alternative

An alternative to the use of meta-tools is the use of toolkits. Toolkits are simpler, though more cumbersome than meta-tools, and consist of a limited set of general-purpose "tools" or analysis features that can be utilized in various combinations and configurations. Rather than attempt to provide the user with the "right tool for the job" in every case, a toolkit instead provides a set of simpler tools which the user can "make work" for each particular "job". Analogies can be taken from Unix hackers, who can accomplish nearly anything with a minimal set of system calls — such as constructing Turing machines using the "vi" editor. A toolkit can also be compared to the "hammer fixes anything" approach in construction and home repair. Several existing, and quite successful, visualization tools could be categorized as toolkits, such as ParaGraph, TOPSYS, IPS-2 [MiY89,MiC90], and Faust [GuG89].

The effectiveness of a toolkit depends on the variety and quality of the tools included. Clearly, the larger the toolkit, the better each resulting solution will be, and the more efficiently the solution will be reached. The issue of cost may be considered in a toolkit when deciding whether or not to add a new tool. If the cost of adding new features to a toolkit is more expensive than making the existing features work for the job, then adding new tools will not be cost-effective for the kit.

The inherent drawback to the use of toolkits is that even when a "good" solution exists, there may be a high cost in overhead to arrive at that solution. By forcing the "wrong" tools to do a particular job, the solution may become complicated and cumbersome, and the resulting overhead will likely have to be repeated for each subsequent analysis. As a user becomes more experienced with the toolkit, the overhead can be reduced as the user learns more effective methods of using the toolkit. The extent to which the development time is reduced depends on the skill of the user and the quality of the toolkit. A more precise explanation is provided by the following expressions:

$$\mathbf{T_{Analysis_1} = T_{Learn\ Tool} + T_{Design\ Analysis_1}}$$

$$\mathbf{T_{Analysis_n} = T_{Design\ Analysis_n} - f(Experience)}$$

The time for the first analysis requires time to familiarize the user with the tool and learn its basic functionality, plus time to actually design the first analysis. After this first analysis, the tool is "learned" and the time for each subsequent analysis is only the time to actually construct the analysis, minus some function of the user's experience with the tool. It is likely that this function has an asymptotic limit that depends on the user's skill and the toolkit's diversity.

The difference between the meta-tool approach and the toolkit approach may be seen by comparing their parameters in the two above expressions. For a meta-tool, the time to learn the complexities of the specification language and familiarize the user with the tool may be much larger than the learning time for a toolkit. However, the solutions generated for analyses using the meta-tool are more specialized and will likely produce results more efficiently. In addition, as a user accumulates a collection of user preferences with the meta-tool, the development time for new analyses will be further reduced, whereas the user must reconstruct each new analysis when using the toolkit. This comparison is illustrated in Figure 2.

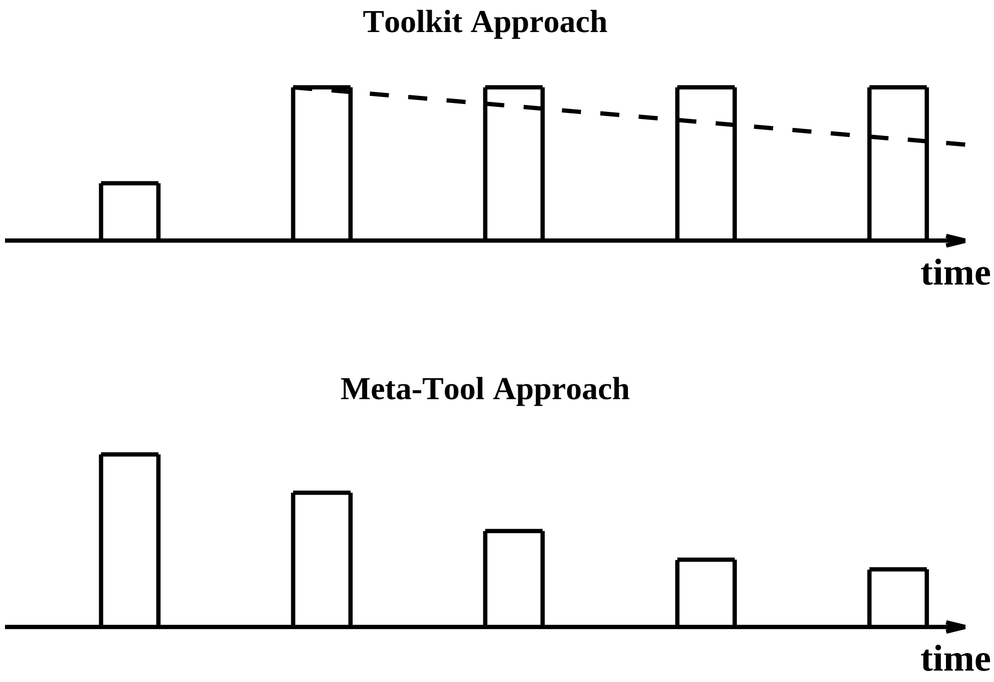


Figure 2: Toolkit & Meta-Tool Development Times

The fundamental trade-off between these two approaches involves the patience of the user in first adopting a particular approach. It would seem that if the high cost of learning a meta-tool is paid, the user benefits from a more powerful, flexible, and efficient analysis system. Otherwise, if the user is not willing to take the time to learn the more complicated meta-tool system, a higher price is paid for each analysis. Each of these approaches is appropriate for particular sets of users and analyses, depending on the long-term needs and requirements for the analysis tool.

An interesting observation is that as a user customizes a meta-tool to suit individual preferences, the user may arrive at several key analysis configurations that are used repeatedly. The resulting default meta-tool configurations could actually constitute a customized toolkit for that user. Perhaps when the analytical needs of parallel software development are better understood, there will be less need for meta-tools, and a basic set of visual analysis techniques will be constructed. If that becomes possible, then general-purpose toolkits will be capable of handling all desired analyses. Until then, if ever, meta-tools will be an essential element in the analysis of parallel software.

3. Problem Statement

The purpose of this thesis is to develop a design methodology for constructing program visualization meta-tools for parallel software analysis. The goal of this methodology is to allow utilization of more advanced visualization techniques while reducing the overhead in creating custom, visual analysis environments. Customizable use of these more sophisticated visualization techniques will provide a framework for experimentation within the visual software analysis paradigm. The methodology will also allow application of these advanced techniques to analyzing parallel software. Based on the methodology, a prototype tool will be implemented to verify the validity of the concepts.

4. Approach

This proposal provides a program visualization meta-tool design methodology for organizing and simplifying the process of creating visual analyses for analyzing parallel software. The central concepts utilized in this methodology are the meta-tool circulation architecture and the relationship mapping language. The architecture organizes the details of the user specification, and provides a framework for combining those details using the mapping language. The following two sub-sections describe the methodology and an example meta-tool design based on the methodology, respectively.

4.1. Methodology

For the purposes of the methodology, the specification details of the visual analysis procedure are viewed as belonging to three **components**:

1. Events
2. Behavior
3. Graphics

Typically, information concerning the execution of parallel software is considered in the form of discrete occurrences, or **events**. Raw event data is captured during program execution to create a trace of information that describes the program behavior during that execution. The events encapsulate data concerning the internal behavior of the software, and identify the time of occurrence of specific actions within the software. The sequences of events produced by the execution of software are utilized in tuning software for performance and for debugging software.

To be utilized in an analysis, the raw event data must often be processed to derive the desired program **behavior** information. Various mathematical functions are applied to convert the individual raw event instances into useful statistics, and to organize the event information. The results reveal high-level features and characteristics of the behavior described by the collection of events.

The software behavior information, once derived from events, must be made available to the user in a useful and meaningful form. In a program visualization system, behavior is rendered visually using **graphics**. Graphics can provide intuitive, high-level interpretations of program behavior by depicting the internal complexities of parallel software using visual analogies.

The central difficulty in using visualization for analyzing parallel software is in integrating these three components in an efficient, flexible manner. To visually analyze software, the event and behavior information must be linked together with the graphics to drive the dynamics of an animation. The possibility for large amounts of detail, combined with a high degree of coupling among the different components, can result in highly complex and cumbersome specifications. The organization of the details and the manner of relating them are critical.

There is no concise specification approach that applies well to details from all of the components. The nature of the information in each of the components is unique, and each component requires a different framework for specifying details. Existing program visualization tools tend to emphasize the specification of only *one* of the components, leading to difficult specification in the remaining components. The proposed design methodology relieves this imbalance by allowing independent specification of the details in different components. Each component utilizes the specification structure which is most natural for the details in that component.

To allow this independence in component specifications, a means is required for efficiently combining the individual representations into a complete, overall analysis specification. This integration of the specifications must be general, to account for the variance in structure among the different components. In addition, there must be a common, underlying foundation among the individual specifications to provide a frame of reference for combining the information. The methodology supports these requirements by introducing a general language for relating the details from different components in terms of fundamental constructs called **entities**. These entities organize the information within each component into distinct sets which contain closely related details. The common entity organization underlying all specifications provides a uniform interface for accessing the information. The entities constitute the "building blocks" for constructing the complete specification.

Entities can be arranged in arbitrary structures to suit the specification needs of each component. The entities can be constructed from several layers of low-level building blocks, or **sub-entities**. Each entity consists of one or more **attributes** which represent the individual details that describe the entity. The collection of attributes, and sub-entities with their attributes, actually forms an abstraction hierarchy for each entity. The general language which combines the different specifications can reference entities anywhere in this abstraction hierarchy. Therefore, the uniform interface to entities provides a highly flexible means for relating the independently specified details.

Along with high flexibility, this approach also allows efficient construction of complete specifications from the individual components. Because entities can be arbitrarily referenced within their abstraction hierarchy, unnecessary detail can be ignored. Information among the components can be related using entities and attributes at any level of abstraction, such that the highest possible abstraction level can be applied in every case. This focusing of abstraction level reduces the overall specification overhead.

The entities for each component are collected into a single **module**. The details of each module are specified independently with respect to other modules, with all coupling among modules handled externally. Each module consists of two portions:

1. Specification
2. State

The **specification** portion includes declarations of the essential characteristics of entities and their attributes. These declarations provide structure for instances of the entities, and can specify relationships among the entities within a module. The **state** portion maintains values for specific instances of entities, using the specified structure.

In the **events module**, entities describe the details of raw, or **primitive**, events, including attribute declarations that identify the data to be associated with each event. Entities can also represent complex combinations of events, or **compound** events. The organization of events into compound events can be nested hierarchically using sub-entities. Conditions can be imposed using expressions on the sub-entities and their attribute values, allowing detailed identification of specific occurrences in the software. The state portion of the events module maintains the data associated with specific event instances. These event instances can be input from execution traces or internally generated.

In the **behavior module**, the entities represent all variables necessary to derive the software behavior of interest. Each of these **behavior variable** entities is associated with an appropriate data type, and each can represent scalar values, vectors or multi-dimensional arrays. The declarations of behavior variables can include specifications to indicate static relationships among behavior variables, and to set initial values. The state portion of the behavior module maintains the values of the behavior variables over the course of the analysis.

In the **graphics module**, the entities represent graphical objects. The specification in the graphics module defines a starting point for the visual display, including fixed, background graphics and graphical reference points, along with dynamic graphical objects and their starting or default positions. Sub-entities in the graphics module correspond to any lower-level building blocks, e.g. points, lines, etc., which make up a particular graphical entity. The state of the entities in this module corresponds to the location, rotation, and scaling of the graphical entities, as well as the values of their other graphical attributes.

These three modules are combined and organized using the **meta-tool circulation architecture**, which provides an orderly flow and processing of information among the modules, as shown in Figure 3. State information for entity instances from different modules is circulated together and processed via a set of **relationships**. These relationships identify manipulations of the state information according to precise conditions on the module entities and their attributes, and describe how the entities influence each other. The results of the manipulations are recirculated back to the modules to constitute new entity states.

The relationships applied to the modules integrate the independent module entities into a cohesive specification. Each relationship consists of a **mapping** which associates a set of conditions on the state with a set of manipulations on the state. The mappings are specified using a **relationship mapping language**. A relationship mapping consists of a set of **rules** of the form:

$$\text{Condition} \rightarrow \text{Manipulation}$$

The **condition** or head of a rule is an expression describing the state of entities from any of the modules. The condition can include references to an entity at any of the levels of abstraction defined for that entity, by utilizing its sub-entities or attributes. A condition can also merely indicate the existence or occurrence of particular entities, such as events and boolean behavior variables. The **manipulation** or body of a rule indicates some action to be carried out when the corresponding condition is satisfied. These actions create, destroy, or otherwise manipulate the state of entities, or their attributes, from any module.

The references to entities in the mapping rules can be very high-level and general, or can be very specific and relate to precise attributes, variable values, etc. For example, consider that events can be referenced simply by whether or not they occurred, or more precisely by their timestamp or a particular data value. Similarly, a graphical image, say a rectangle, could be directed to move in its entirety across the display, or could be distorted by moving only a single side or corner point. The benefit of this flexibility is that the user is not required to specify unnecessary details, but can instrument precise control if desired. In addition, the focus of the mapping rules is on module coupling relationships, not implementation details or flow control. The mapping rules describe the fundamental interaction among events, behavior, and graphics.

The general semantics of the mapping rules are simple, and can describe cause-effect relationships, transformations and assignments, and numerical and statistical operations. For example, the occurrence of a certain event might cause the movement of some graphical image across the display, or perhaps a certain program behavior state, in conjunction with a particular condition on the graphical display, might cause the creation of an event instance.

The relationships specified using the mapping language constitute the details for a "plug-in" **simulator**. This simulator processes the state information from the module entities, under the constraints of the relationships, and automatically derives new state information. In addition, the simulator utilizes portions of the state information to produce visual output for analysis. This output is a graphical display that animates the behavior of the software under analysis, based on

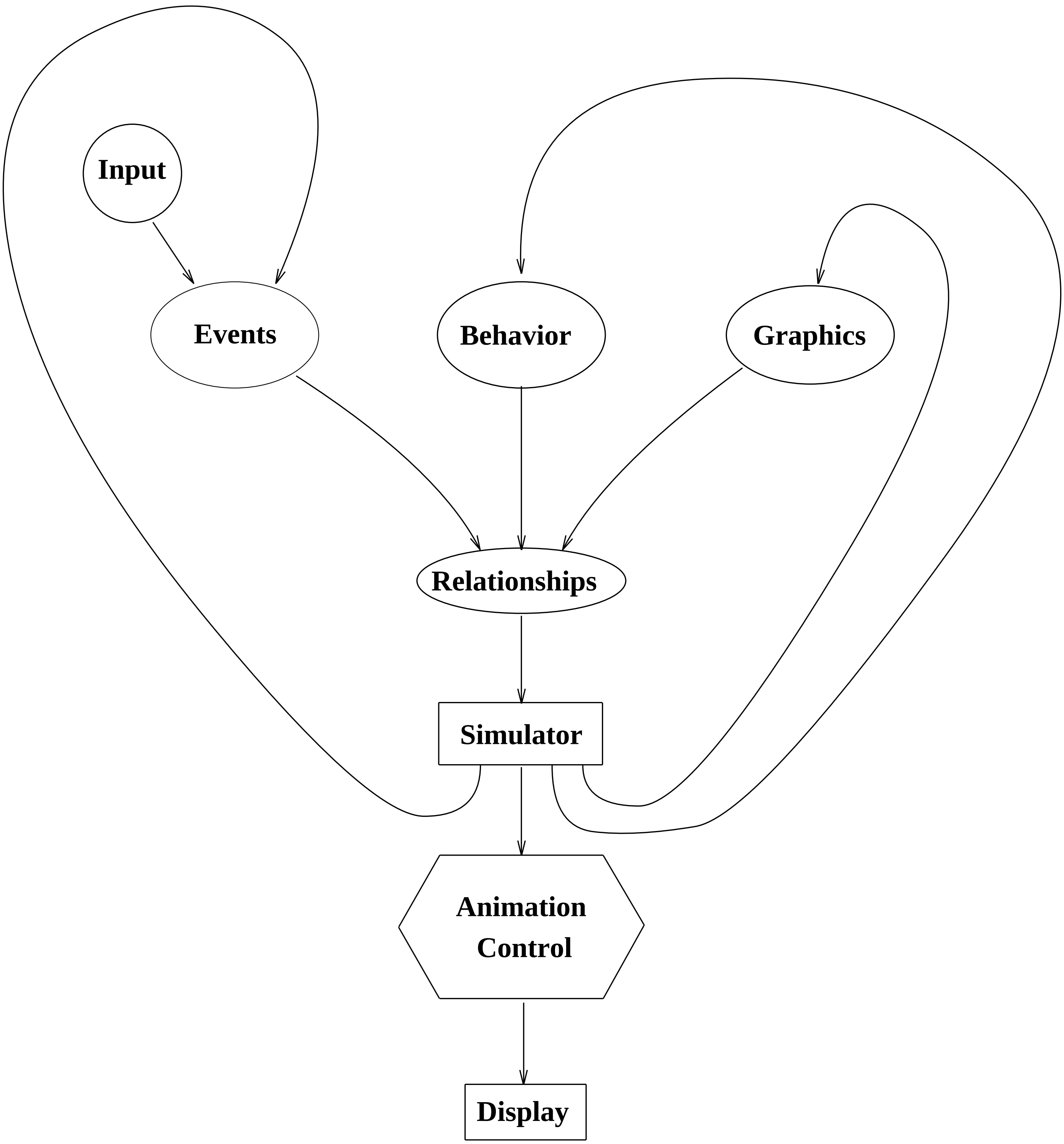


Figure 3: Meta-Tool Architecture

the relationships provided by the user.

The user can adjust and focus the view of the display using the **animation control** interface, which acts like a valve or lens to control the overall perspective of the visual analysis. Various analysis features can be controlled, including the speed and direction of the animation, display size and zooming.

In addition to these post-simulator adjustments, the user can also interactively adjust the parameters of the analysis at any time by modifying the relationships among the modules. For example, the same behavior and graphics configuration could be used with a different set of event definitions to analyze a different aspect of the software behavior using the same animation.

One exception to this modification capability, and to the independence of the specification of modules, involves the nature of the data that the events can represent. Events can be categorized as being either **qualitative** or **quantitative**. Qualitative events correspond to specific conditions, having a value of either TRUE or FALSE, or identifying a specific occurrence that results from passing a particular point in a computation. Quantitative events carry numerical information, other than a timestamp, and describe the value of some program metric or characteristic, or the result of some operation. Quantitative events can also originate from some monitoring systems, where large amounts of data are summarized prior to being dumped into an execution trace.

Because these two types of events inherently describe software behavior from different perspectives, events of different types cannot be exchanged in the event module without affecting the linkage to other modules. Specifically, the behavior and graphics modules must be adjusted to appropriately process and display the correct type of event information. The corresponding mapping rules which relate the events to the behavior and graphics must also be adjusted. The graphics module may or may not necessarily require adjustment, depending on the level of coupling between the graphics and the other modules — the existing graphical specification may, in fact, effectively describe the new event and behavior information.

4.2. Example Design

To assist in describing the details of the design methodology, an example meta-tool design is provided. This design forms the basis for a prototype to be built as a demonstration of the capabilities of the methodology. Discussion here will be limited to an overview of the features and language syntax, emphasizing those details which are most relevant.

It is beyond the scope of this proposal to design a new event description language or contribute to the ongoing research in the monitoring of parallel software. Therefore, the specification of events in this design is accomplished via EDL (Event Definition Language) [BaW83,Bat88]. This language allows primitive and compound events to be defined, along with appropriately typed attributes for organizing data. Several operators are provided for combining events into various types of sequential and concurrent compound events. An example of the format of a compound event definition using EDL is shown in Figure 4.

event <event_name> *is*
    <event1> <op> <event2> <op> ... <op> <eventN>
cond
    <eventI>.<attr1> == 3;
    <eventJ>.<attr2> <= 2 * <event1>.<attr3>;
    . . .
with
    <attr1> = <event2>.<attr3> + 4;
    <attr2> = <event1>.<attr1> - <eventN>.<attr1>;
    . . .
end

Figure 4: Example Compound Event Definition Format

Each event is defined with a unique name, and consists of a combination of one or more other events using a selection of compound operators. These operators indicate temporal relationships, such as concurrent versus sequential, and other groupings. Events can also be declared to be primitive, and can be linked with a corresponding raw event. An event can be defined to occur only under certain conditions, with respect to the attributes of the "sub-events" that define it. These conditions are specified using simple boolean expressions. Each event can have several attributes declared, with corresponding data types. These attributes are assigned values using expressions of the other sub-event attributes.

The behavior variables for this design are specified using typical variable declaration syntax, such as found in C. The user may specify scalars, vectors, and multi-dimensional arrays using a variety of data types. Some examples are shown in Figure 5. Note that multi-dimensional arrays can be defined with unspecified, or **implicit**, dimension sizes, to allow greater flexibility in array usage. This implicit sizing also proves useful for "sparse" information. This type of declaration is implemented using a table-lookup indexing scheme, to allow efficient storage for structures of unlimited size.

```
double util[128][128];

double x[][][];

char message[256];

int i,j;
```

Figure 5: Example Behavior Variable Declarations

It should be noted that a few analysis details must transcend the independent boundaries of the modules. For example, as illustrated in the above declarations, many analyses will deal with a particular number of processing elements (double util[128][128]). Consideration of this information can be necessary in multiple components, requiring agreement among the independent modules. Such information might be needed to properly size arrays, to indicate the number of graphical objects necessary, or even to control event declarations in some cases.

Depending on the intricacy of the detail desired in the animation for a particular analysis, the amount of this dependent information will vary. In most cases, the amount of such information will likely be small, relating to a few key architectural parameters, such as the number of processors or buffer sizes, or the software structure. Cases of extreme interference due to dependent information could be alleviated by introducing simple constructs for global constants. Another alternative involves utilizing the implicit dimension sizing for behavior variables to allow at least the behavior module to remain independent.

The graphics specification for this design is derived from typical existing interfaces for drawing static graphics, such as are provided in MacDraw [MAC92], xpic [Mor92], and xfig [Sut92]. The initial graphical display for an analysis is specified in much the same way that pictures are created using these standard drawing tools. The user may select from a wide variety of graphical objects, including points, lines, triangles, squares, circles, ellipses, arcs, etc., as well as text, pixel images, and mathematical functions. Each of the object instances may be scaled, rotated, resized, or colored as desired when created.

Each graphical object type implicitly carries a set of attributes that specify the shape, size, rotation, and other defining characteristics for that object type. The set of attributes for a given graphical object type actually overspecify a particular instance, such that any combination which completely defines the object instance is sufficient. For example, a circle could be defined by a center and radius, or a center and diameter. Likewise, a rectangle could be defined by specifying the location of two opposite corners, or by providing a length and width.

The specification language for the graphics is hybrid, using both visual and textual representations. The visual manipulation allows a natural interaction with the graphics, utilizing standard drawing techniques and operations, as well as advanced features specifically for parallel software views. A precise textual representation mimics the graphical manipulations, allowing straightforward integration of the graphics with the other modules. The link between the graphical and textual representations is well-defined and unambiguous, such that graphical specification can be automatically mirrored using the textual specification, and vice-versa. The user can specify details using either representation, according to individual intuition and preferences. Those details best suited to graphics can be specified graphically, and the precision of text can be applied to refine the details.

To identify specific graphical objects for integration into the mapping language, graphical object variables can be assigned to "point" to particular graphical object instances. These variables are used to access the attributes of graphical objects and reveal their state information in a form more readily usable in the textual mapping language. Assignments to these variables can be inserted into the textual portion of the hybrid specification language, such that the identity of graphical objects can be saved upon their creation.

The relationship mapping language for this design combines the various details of events, behavior variables, and graphical objects into a complete analysis specification. These entities can be accessed in terms of high-level entities or low-level attributes, and placed into the condition and manipulation portions of the rules. Both the condition and manipulation portions are specified using expressions that reference the entities. The condition expressions are specified using comparators much like standard C, such as ==, !=, <=, >=, &&, ||, etc. The manipulation expressions consist of one or more statements, each representing some action that manipulates the events, behavior variables, or graphics. Examples of rules are given in Figure 6.

update	→	COLOR {circle}[*update*.pe] *update*.value;
load[i] ≥ 10	→	ROTATE {image}[i] (**load**[i] * **scale**);
{point}[i].x ≤ SIN(**current_time**)	→	*below*; **status**[i] := 3;

Figure 6: Example Relationship Mapping Rules

In the examples, italics is used to represent events, and bold represents behavior variables. The items enclosed in braces ({}) represent graphical references to graphically specified objects. In constructing a rule, the user can graphically specify appropriate details concerning graphical objects and locations. The first example rule indicates that whenever an *update* event occurs, a circle, representing the processing element (pe) that issued the *update*, changes colors based on an event attribute "value". The second rule specifies that whenever an element of the behavior variable vector **load** is greater than or equal to 10, a corresponding graphical "image" is rotated to a position identified by multiplying a behavior variable **scale** by the particular **load** value. The third rule states that when any point from a vector of points crosses a vertical boundary, as determined by comparing the "x" coordinate to a sine wave function of the current time, a *below* event instance is created, and the corresponding behavior variable **status** is updated to 3. It is clear from these rules that very simple, as well as arbitrarily complex, relationships are possible.

A sketch of the grammar for the mapping language is provided in Figure 7. The <rule> consists of the <condition> and <manipulation>. The <condition> is the conjunction, disjunction, or compliment of one or more boolean <predicate> items. Each predicate can be some comparison of two expressions, or can be an event — signifying the occurrence of the event — or a boolean variable or constant. The <expression> is made up of standard binary operators and parentheses — "()" — and functions with expressions as arguments. The <term> items used in an <expression> can be either a numerical constant or an <entity> from one of the event, behavior or graphics modules. If the <entity> references an <event>, an accompanying attribute, <attr>, must be appended to actually reference a data value rather than just an event occurrence,

```
<rule>          ::=  <condition>  →  <manipulation>

<condition>     ::=  <condition> && <condition>  |
                     <condition> || <condition>  |
                     ! <condition>  |  <predicate>

<predicate>     ::=  <expression> <compare> <expression>  |
                     ( <predicate> )  |  <event>  |  <boolean>
                     TRUE  |  FALSE

<compare>       ::=  ==  |  !=  |  <=  |  >=  |  <  |  >

<expression>    ::=  <expression> <binop> <expression>  |
                     ( <expression> )  |
                     <function> ( <arguments> )  |  <term>

<binop>         ::=  +  |  -  |  *  |  /  |  %

<function>      ::=  mean  |  variance  |  min  |  max  |
                     cos  |  sin  |  log  |  exp  |  abs  |
                     sum  |  count

<arguments>     ::=  <expression>, <arguments>  |  <expression>

<term>          ::=  <entity>  |  <constant_number>

<entity>        ::=  <event>.<attr>  |  <variable>

<variable>      ::=  <behavior>  |  <graphics>  |
                     <variable>.<attr>

<manipulation>  ::=  <operation>; <manipulation>  |  <operation>;

<operation>     ::=  <event_op>  |  <assignment>  |  <graphic_op>

<event_op>      ::=  <event>  |  !<event>  |  <event>: <setattr>

<setattr>       ::=  <attr> := <expression>  |  <setattr>, <setattr>

<assignment>    ::=  <entity> := <expression>

<graphic_op>    ::=  CREATE  |  DESTROY  |  COPY  |
                     MOVE  |  ROTATE  |  SCALE  |
                     COLOR
```

Figure 7: Mapping Language Grammar Sketch (Incomplete)

as a numerical value is needed. The behavior and graphics <variable> references can stand alone, or can include an <attr> as well.

The <manipulation> portion of a <rule> consists of one or more <operation> items, each of them being either an event operation (<event_op>), an assignment to a variable (<assignment>), or a graphical manipulation (<graphic_op>). The event operations either create or destroy and instance of an <event>, and on creation may specify the value of particular event attributes. The variable assignment operation sets either a behavior variable or graphical object variable, or an attribute of a variable. The graphical manipulation operations apply one of several graphical functions to particular graphical object instances. The arguments to these functions (not shown) consist of a number of expressions to produce the necessary function parameter values, etc.

Each of the various modules supports different semantics for this grammar. As mentioned above, events may be referenced in two different ways, by their occurrence or by particular attribute values. In a condition expression, an event reference with no attributes indicates that the desired condition is merely the occurrence of an instance of that event at the current simulation time. Otherwise, if an event is referenced with a particular attribute, the desired condition is that the event occurred at the current time *and* that the attribute has some particular value. An exception to this latter case is when an event is referenced with its timestamp attribute. In this case, the condition relates to any events which have occurred, and examines some condition based on the overall event history. For example, a condition might rely on how long ago in simulation time a certain event occurred.

In the manipulation expressions, events references have several different contexts. If an event occurs alone in a statement, without attributes, the operation indicated is the creation of an instance of that event at the current time. An event referenced in this way can also include specific assignments to the new instance attributes. Similarly, a statement containing a single event preceded by a "!" indicates the removal or deletion of an event instance. This feature can be applied to events whose significance does not depend on time, and can scratch events from the event history. Events can also be referenced in expressions as part of functions or operations in other statements, but only when accompanied by references to particular attributes. In this context, the event reference utilizes the value of an event attribute for manipulating another entity.

The use of behavior variables in mapping rules is simpler than the many different event contexts. The use of a behavior variable in a condition expression implies a comparison of the value of that variable to some other value. The use of attributes with behavior variables implies use of the specific attribute values in comparisons. Behavior variables, with or without attributes, which appear in manipulation expressions are used to change the value of particular behavior variables or their attributes, when appearing on the left side of an assignment. Otherwise, the reference implies use of the value of the variable or attribute in a mathematical expression for the right side or an assignment or in another operation.

The references to graphical objects can appear in similar ways to the behavior variables. Graphical object variables can appear in conditions expressions to evaluate comparisons of the location and attributes of graphical objects. In manipulation expressions, references to graphical objects can be used, as behavior variables, as values for mathematical expressions in assignments and other operations. In addition, graphical objects can be part of the special functions and operations that manipulate the graphics to control animation. Graphical objects can be supplied to these functions to be moved, copied, resized, colored, etc. The parameters of these functions, that control the way in which the graphics is manipulated, use the references to the event attributes, behavior variables and attributes, and other graphical objects and attributes as arguments. These functions also handle creating and destroying graphical objects, and can save graphical variable pointers to the objects for subsequent references.

Aside from the above functions for relating and manipulating events, behavior, and graphical entities, one special function is also included specifically for parallel analyses. To assist in manipulating replicated or multiple copies of information, as are likely to be found in analyses with multiple identical processors, functional units, etc., a function is introduced for manipulating and constructing large arrays of behavior variables and graphical objects. The "REPLICATE" function allows very large, complex arrangements of state and graphics for massively parallel purposes. This function substantially reduces the tedium in specifying these details by hand. The REPLICATE function may be applied within either the behavior or graphic specification modules, as well as externally in mapping language rules. An example of the syntax and use of this function is given in Figure 8.

```
REPLICATE {triangle} IN {square} 0 TO 127

        LABEL "PE $$" AT {triangle}[$$].p1
        COLOR $$ / 8
        ANGLE 90 + (($$ / 127) * 360):

    SET ptr[$$] := {triangle}[$$];
    SET status[$$] := 0;
```

Figure 8: Example Replication Syntax

This example instance of a REPLICATE function, or a **replication**, specifies that a vector of 128 triangles should be arranged along the perimeter of a square, and assigned index values from 0 to 127. Three options of the replication also define special instructions for the construction of the replication. Labels are to be placed at the first point of each triangle, with the index number of the triangle included in each label. Note that the special "$$" symbol is used as an array index, to represent *each* of the index values within the index range, such that the statement will be applied once to every occurrence, or **replicate**, in the replication. The triangles are to be colored in groups of 8 according to their index in the replication, specified by setting each color to the index divided by 8. Note in this option that the triangles in the replication are referenced graphically, as indicated by the braces, and that upon identification of the desired graphical object, the index to the particular replicate is appended as if the triangle were represented textually. The triangles are each placed at rotation angles corresponding to their index, specified by computing the proportion of each index in the range. Two "sub-statements" in the replication indicate other functions that should be executed in creating the arrangement. One sub-statement indicates that each element of the graphical object **ptr** array points to a corresponding triangle, for each index. The other sub-statement sets the behavior variable **status** to 0 for each triangle.

As an aside, regarding the independence of modules and replications, the inclusion of both behavior and graphical variables in this replication require it to reside at the mapping language level. An alternative specification could divide this replication into two separate replications, one specifying the behavior details and the other specifying the graphics, to allow independent specification in different modules. In addition, as replications themselves can be considered a complex type of compound graphical object, the graphical aspects of the replication could have been defined in the graphics module and then the behavioral detail added via a reference to the replication in a mapping rule.

5. Support

This section provides arguments and evidence supporting the decisions made in constructing the methodology for designing meta-tools for program visualization of parallel software. In the first sub-section, comparisons will be made to existing approaches, and the reasons behind several key deign decisions will be discussed. In the second sub-section, four examples of views constructed using the methodology will be presented as evidence of validity and feasibility.

5.1. Justification

There have been many attempts at providing a powerful, flexible, and efficient interface for program visualization of parallel software. Some tools, such as ParaGraph and TOPSYS, have taken the simplistic approach by providing a fixed collection of views with limited control and customizability. Tools such as this are very useful for a limited set of analyses, but, as has been discussed previously, there are limits to the generality and power of these tools. Another approach is needed to realize the full benefits of visual analyses.

The first step away from the simpler tools involves the utilization of more sophisticated techniques to specify the details of views. While maintaining similar types of fixed views as found in the simpler tools, the manipulation and use of those views is made completely customizable. Tools such as Pablo, Hyperview, and Triplex implement elegant graphical programming techniques for controlling the processing of information, and directing the results to configurable displays. All of these tools use graphical "filter-trees" to indicate the flow of information from its "root" source, through various filter "nodes" that apply mathematical, statistical and reduction transformations, to route appropriate data to "leaf" display nodes where the results are visually displayed or animated. Although the customized data processing capabilities of these filter-based tools are unequaled to date, the sophistication of the graphics and animation are no better than that supported by the simpler tools. The filter approach, while well-suited to organizing the data processing details, fails to apply well to the manipulation of complex graphics.

Other tools have approached the problem from the opposite direction, and have designed very powerful graphics and animation systems for analyzing parallel software. In fact, some of the earliest work in visualization took this approach. The BALSA and BALSA-II systems and Tango are the more prominent examples of tools of this type. These tools produce arbitrary animations of parallel software behavior by instrumenting the user application with animation directives. Tango added the extra assistance of several abstractions for manipulating graphics, including *images*, *locations*, *paths*, and *transitions*, to reduce the complexity of the graphics specification.

The clear drawback of this animation directive approach is that the scheme relies heavily on modifying the user application — an intrusion that can directly result in a perturbation of synchronization and program behavior, as well as violate the software readability by scattering distractions through the user's source code. While many monitoring schemes typically introduce some intrusion due to physical constraints, there is no reason to force this extra modification of the user application for visual analysis purposes. This extra intrusion can be especially damaging due to the potential delays for the graphical I/O devices. Another drawback to the animation directive approach is that, much as the filter-based tools were limited in their ability to specify complex animations, these animation systems are not at all capable of the complex data manipulations and statistical analyses supported by the filter-based tools.

Another approach, between the filter-based and animation directive tools, is that used by PARADISE. This approach uses the meta-tool concept to specify the relationship between program behavior and simple animations based on construction of a visual model, made up of individual objects. This different means for specifying the data processing details extends the meta-tool concept of the filter-based tools to include graphical manipulations. The data processing and graphics are encapsulated together in discrete visual objects that interact in the visual model. These objects process the execution trace data and maintain state information concerning the behavior of the software. The internal state of each object is mapped directly to an externally visible effect in the animation.

The important move made by the PARADISE approach was bringing more powerful graphics into the analysis without directly interfering with the user application. Unfortunately, while PARADISE was able to integrate information processing and graphics well, neither aspect was very sophisticated, and the specification of each analysis was tedious. Each analysis effectively required development of a customized data processing and animation program that had to be compiled with the meta-tool to perform the analysis.

From the experiences with the PARADISE prototype, and from consideration of the drawbacks of other existing visualization tools, it is clear that one central problem with visualization systems in general is the lack of any separation of concerns in organizing the analysis process. The existing tools have all attempted to specify everything simultaneously in a single, neat package, only to find that "everything" was too complex for that to be possible. The alternatives tried in response to this obstacle have ranged from eliminating useful capabilities to simply dealing with the high complexity and overhead — neither choice being entirely successful or desirable. The proposed design methodology attacks this problem at a high level by separating the visual analysis process into its fundamental components. The specification of events, behavior, and graphics details in independent modules allows modular development of analyses. This arrangement reduces the complexity by properly organizing the many complex details.

It should be noted that the details of monitoring and interfacing with the user application, though important, are not relevant to the details of mapping the resulting events to program behavior and visualization, and so are completely removed from consideration. Such details can be handled by appropriate front-end packages, and the task of interfacing to such packages is well within the limits of current interfacing technology.

The remaining components, events, behavior, and graphics, are all included in the design architecture, but *independently* at the highest level. Much of the difficulty in specifying visual analyses is immediately alleviated by separating these concerns. The overhead and complexity of specifying any two of these components together are significant and unjustified. If it were possible to combine all of the necessary details easily, it would be preferable to have only one component to specify. Unfortunately, to date this has not been possible, and will likely never be possible, due to the high level of detail involved.

The obstacle faced by this approach, given the proposed separation of specification details, is the problem of specifying the abstract relationships that re-combine the details of the various components into a coherent analysis. Though the filter-based scheme works extremely well in mapping events to behavior for information processing, a more powerful representation is necessary to manage the additional details of the mapping from behavior to graphics. In addition, this mapping language must be simple enough to be easily understood and utilized by even the novice user.

This approach merges the mappings from events to behavior and from behavior to graphics into a single mapping language. This allows added flexibility by providing the capability to map events directly to graphics without the need for behavioral derivations. In addition, arbitrarily complex relationships can be specified allowing information to be mapped from the modules back into manipulations of the modules. This *circulation* capability, as denoted in the meta-tool architecture, yields ultimate flexibility by providing a means for feedback in controlling the visual analysis.

The relationship mapping language also provides a straightforward means for specifying these relationships by arranging the complex details into simple "cause-effect" or "if-then" rules. This binary arrangement is effective for describing any desired relationship mapping — "if this happens, then do this" or "this situation should cause this result". This type of binary mapping relationship has been utilized in several existing technologies, including high-level programming languages, computer-human interfaces, finite state machines, and even an existing visualization tool, Pavane [CoR91]. This tool uses a similar mapping language to specify aspects of its "proof-based" visualization, that relates abstract properties of computation to views.

By restricting the head or condition expression in rules to not modify any entities, but only describe their value or situation, the interpretation and parsing of the rules is simplified. This restriction enforces some organization on the specification of relationships, and prevents the design of confusing rules with ambiguous semantics. Relationships constructed with manipulations and conditions mixed on different sides of rules would serve only to convolute the fundamental relationships and confuse the user.

Instead, maximal flexibility and expressive power are provided by *not* restricting the placement of module details in rules. Details from any module can be used to control manipulations of any of the modules. An interesting ramification of this freedom is that the very sequence of event instances used as input to the simulator can be manipulated using rules. This means that the rules can actually be used to specify feedback relationships to the system input, and the system can alter and introduce information into its own input stream.

Finally, because the mapping language can access and reference information from each of the modules at arbitrary levels of abstraction, the level of detail in the resulting rules is as simple as possible. The user need not include cumbersome characteristics and attributes when only the high-level information is necessary, hence the rules specified are minimal with respect to unnecessary specification overhead. Likewise, if the user needs absolute control over the low-level details of an analysis, the capability exists and can be exploited.

5.2. Examples

This sub-section provides a small, representative collection of visual analysis examples that are specified using the proposed meta-tool design and mapping language. For each example, the module details are provided, along with the necessary mapping language rules for specifying the functionality of the visual analysis.

5.2.1. Example 1: Interprocess Communication Analysis

This analysis depicts the behavior of multiple concurrent processes in a message passing environment. Each process is represented by a circle, and these process circles are arranged in a larger circle. As the processes execute, they enter various states, that are identified by *update* events. For each state, the corresponding process circle should change to an appropriate color. The processes may communicate messages with other processes, as identified by *send* and *receive* events. For each such message, an arrow is to be created from the sender to the receiver process circle, to be present during the time while the message is in transit (between the *send* event and the matching *receive* event). This display is essentially that used by the Ravel visualization tool developed at Argonne National Laboratory. The specification for this example is given in Figures E1.1 through E1.4 below.

Event Specification:

event update *is* primitive "raw_update"
with
    int pe;
    int state;
end

event send *is* primitive "raw_send"
with
    int src;
    int dst;
end

event receive *is* primitive "raw_receive"
with
    int src;
    int dst;
end

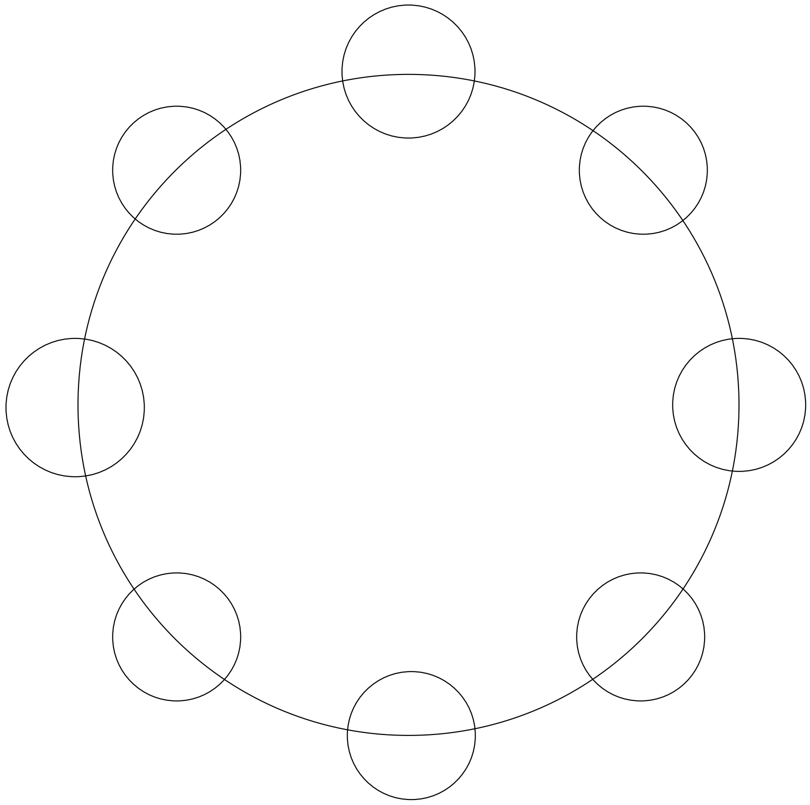
Figure E1.1: Interprocess Communication Event Specification

Behavior Variable Specification:

None Necessary.

Figure E1.2: Interprocess Communication Behavior Variable Specification

Graphical Specification:



CREATE {circle} radius := 10;

REPLICATE crc[$$] := {circle} IN {circle} 0 TO 7;

Figure E1.3: Interprocess Communication Graphical Specification

Mapping Rules:

update → COLOR crc[*update*.pe] *update*.state;

send → ln[*send*.src][*send*.dst] := CREATE {line}
crc[*send*.src] TO crc[*send*.dst]
ARROW end;

receive → DESTROY ln[*receive*.src][*receive*.dst];

Figure E1.4: Interprocess Communication Mapping Rules

The three necessary events, *update*, *send*, *receive*, are all primitive mappings from raw execution trace events. No behavior variables are necessary for this analysis, only two graphical object variables, "crc" and "ln", for keeping track of and referencing the pe circles and message arrows. The display is constructed by creating a single circle and then replicating it in another circle, using indices from 0 to 7. The mapping rules state that an *update* event causes a re-coloring of the originating pe's circle. A *send* event creates an arrowed line from the source to the destination pe circle, saving a pointer to the line in "ln". Note that the line pointer is created with implicit array dimensions, as supported by the behavior variable declarations, to avoid allocation of the large $\mathbf{n^2}$ full array. The *receive* event destroys the corresponding arrowed line created by the send.

5.2.2. Example 2: Multiprocessor Machine Array Analysis

This analysis is especially flexible for large, massively parallel, multiprocessor systems. The display consists of a two-dimensional array of points, each of which represents the state of a single processor. As software executes, each processor proceeds through a sequence of states which indicate idle and busy portions of a computation, as well as particular stages through the computation. For each state, a color is assigned to the pe's dot in the array. The result is similar to scientific visualization of patterns in a solid. Displays such as this have been developed by Rover [RoC91,RoW90,Rov89]. The specification for this analysis are shown in Figures E2.1 through E2.4 below.

Event Specification:

event update *is* primitive "raw_update"
with
    int pe;
    int state;
end

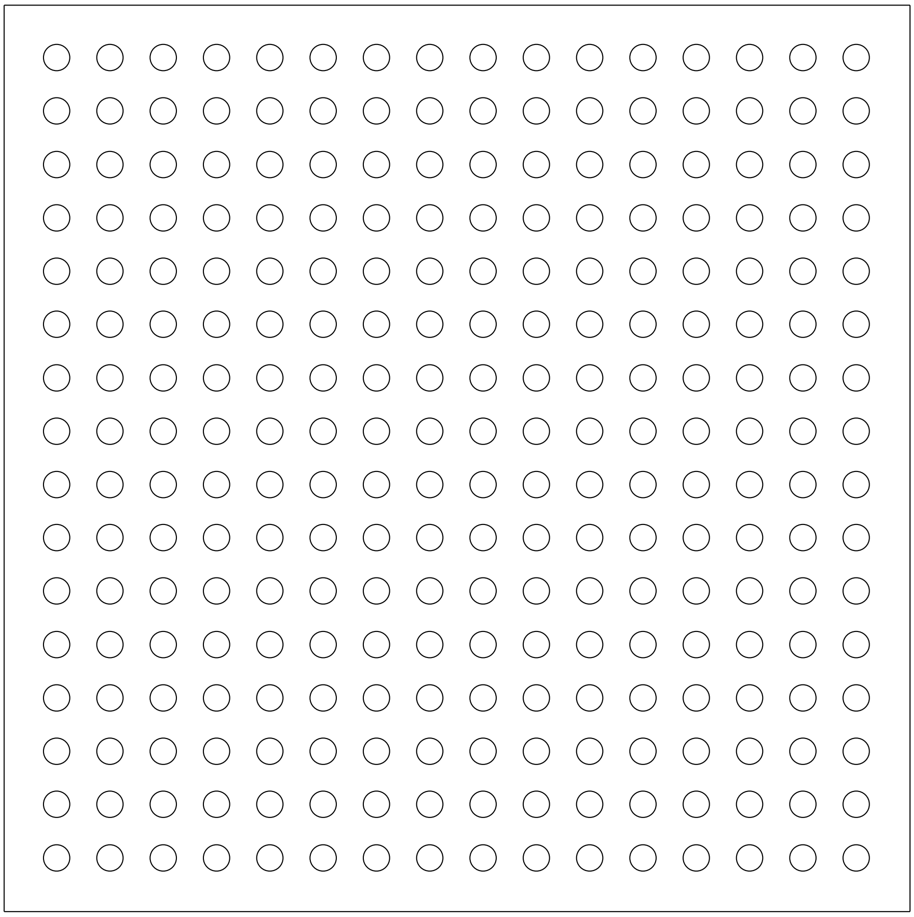
Figure E2.1: Machine Array Event Specification

Behavior Variable Specification:

None Necessary.

Figure E2.2: Machine Array Behavior Variable Specification

Graphical Specification:



REPLICATE pnt[$$] := {point} IN 16 X 16 ARRAY 0 TO 255 ROWMAJOR;

Figure E2.3: Machine Array Graphical Specification

Mapping Rules:

update → COLOR pnt[*update*.pe] *update*.state;

Figure E2.4: Machine Array Mapping Rules

The single *update* event corresponds directly to the raw execution trace event. No behavior variables are needed, and the single graphical object variable "pnt" represents the array of pe points. The display is constructed by replicating a generic point in a two-dimensional array, saving a pointer to each point in "pnt". The only mapping rule states simply that each point should be colored based on the state of the corresponding pe, as set on the occurrence of each *update* event.

5.2.3. Example 3: Kiviat Diagram Analysis

The Kiviat diagram [Mor74] is well known for providing intuitive rendering of the relative balance or variance among the processors in a multiprocessor system. This type of diagram is most useful for load balance and utilization information. The display consists of a wheel with spokes, where each spoke represents the measure of a single pe's condition. A point on each spoke indicates a value for the desired analysis metric, The points are placed along the spokes at distances from the center of the wheel, proportionate to the particular value, where the maximum value corresponds to the rim of the wheel. A set of lines connect the points on each spoke into a flexible ring that mutates according to the distribution of pe values. The roundness of this ring qualitatively indicates the balance or variance among the processors. The specification of the Kiviat Diagram analysis is shown in Figures E3.1 through E3.4 below.

Event Specification:

```
event busy is primitive "raw_busy"
with
        int pe;
end

event idle is primitive "raw_idle"
with
        int pe;
end
```

Figure E3.1: Kiviat Diagram Event Specification

Behavior Variable Specification:

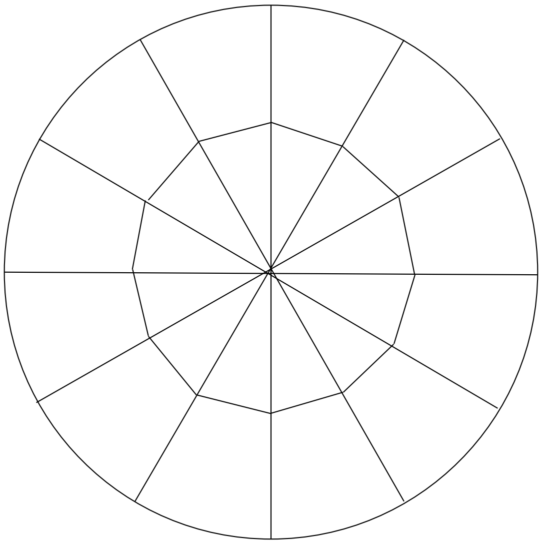
```
boolean busy[16];

long busytime[16];

float util[16];
```

Figure E3.2: Kiviat Diagram Behavior Variable Specification

Graphical Specification:



CREATE {circle} diameter := 200;

CREATE {line}
    length := 100
    {line}.p1 := {circle}.center;

REPLICATE ln[$$] := {line} IN {circle} 0 TO 15
        ROTATE ($$ / 16) * 360:
    pnt[$$] := CREATE {point} AT ln[$$].center;
    CREATE {line} pnt[$$] TO pnt[($$ + 1) % 16];

Figure E3.3: Kiviat Diagram Graphical Specification

Mapping Rules:

busy	→	**busy**[*busy*.pe] := TRUE;
idle	→	**busy**[*idle*.pe] := FALSE;
busy[i]	→	**busytime**[i]++;
(TRUE	→	SET) **util**[i] := **busytime**[i] / **currenttime**;
(TRUE	→	SET) **p**[i] := **util**[i] ALONG **ln**[i];

Figure E3.4: Kiviat Diagram Mapping Rules

The events utilized for this example are *busy* and *idle*, to determine the utilization of each processor. Different events could be "plugged in" to use the behavior and graphics specification for load balance, etc. The behavior variables for this analysis are **busy**, **busytime**, and **util**, which keep track of whether particular processors are busy or idle, and for how long, to maintain a utilization value. The graphical object variables "pnt" and "ln" keep track of particular graphical objects for manipulating the display, such that the processor points can be placed proportionately along their spoke lines in the display. The graphical specification includes drawing of the wheel and spokes, and the identification of the points and spokes, and creates the lines that connect the different points. The main portion of the graphical specification is encapsulated in a single replication that specifies all the graphics and the assignments to graphical object pointers. The mapping rules identify the relationship of the behavior variables to the events and graphics. When a *busy* event occurs, the corresponding **busy** flag for that processor is set to TRUE, engaging a counter rule that keeps track of how long the processor is busy, until the next *idle* event resets the flag. Another rule defines the utilization in terms of the counter and the current elapsed simulator time. The last rule links each current utilization value with the position of the corresponding point on its spoke.

5.2.4. Example 4: Gantt Chart Analysis

The Gantt chart [Gan19] is another well-known visual representation that effectively displays the behavior and synchronization of concurrently executing processes or processors. The displays consists of a vertical stack of multi-colored, horizontal bars along the time axis, one bar for each processor. The color of a given bar at a given point along the time axis corresponds to the particular processor state at that time instant. The relative synchronization among processors is identified by the skew in alignment among the colors in different processor bars. In addition, the correct and efficient operation of each processor can be verified by examining the order and duration of the different colored states in each bar. The specification of the Gantt Chart analysis is shown in Figures E4.1 through E4.4 below.

Event Specification:

event $state_1$ *is* primitive "raw_state_1"
with
    int color;
    int pe;
end

⋮

event $state_n$ *is* primitive "raw_state_n"
with
    int color;
    int pe;
end

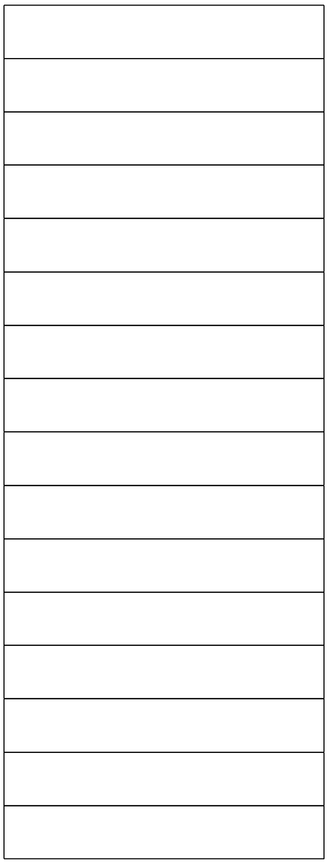
Figure E4.1: Gantt Chart Event Specification

Behavior Variable Specification:

int scale = 10;

Figure E4.2: Gantt Chart Behavior Variable Specification

Graphical Specification:



CREATE {rectangle};

REPLICATE rect[$$] := {rectangle} IN {line} 0 TO 15
COLOR rect[$$] 0;

Figure E4.3: Gantt Chart Graphical Specification

Mapping Rules:

(TRUE →) rect[i].x2 := **scale** * **currenttime**;

EVENT → COPY rect[EVENT.pe] tmp;
rect[EVENT.pe].x1 := tmp.x2;
COLOR rect[EVENT.pe] EVENT.color;

Figure E4.4: Gantt Chart Mapping Rules

As an alternate means for specifying processor state events, for this example, each event is defined individually rather than using attribute values in a single *update* event. Each of the *state* events includes an attribute for the originating pe and a corresponding color to represent the state in the chart. The only behavior variable for this analysis is a **scale** variable for scaling the width of the display versus time. The graphics specification for this analysis is simply the column of rectangles which sets up the initial display. This column is created by a replication that assigns graphical object variables to point to each rectangle and colors each rectangle in a neutral, initial color (0). The mapping rules expand on this initial graphics specification by indicating that the right side of each rectangle should expand linearly with the scaled current time. Note that this movement is independent of the occurrence of any events or other conditions, and the condition expression has a constant value of TRUE. The remaining mapping rule states that when *any* event occurs, denoted by the generic keyword "EVENT", that a new colored rectangle should be appended to the end of the bar for that processor. This is accomplished by copying the current rectangle into a temporary graphical object variable "tmp". The dynamic rectangle pointed to by "rect" is then adjusted to have its left side on the right side the old "tmp" rectangle, and a new color is instantiated. In this way, every new event effectively changes the color of the rectangle bar at the appropriate point on the time axis.

6. Conclusion

The proposed meta-tool design methodology presents an efficient, flexible, and powerful framework for constructing visual analyses for parallel software development. The architecture separates the concerns of the analysis specification into independent modules, and the relationship mapping language represents the relationships among the modules efficiently and naturally. The demonstration implementation is feasible and can mimic existing visualization views, as well as supercede the existing technology by providing a customizable paradigm.

References

[BaS88] M. L. Bailey, D. Socha, D. Notkin, "Debugging Parallel Programs using Graphical Views," *Proceedings of the 1988 International Conference on Parallel Processing*, Volume II, St. Charles, Illinois, August 1988, pp. 46-49.

[BaW83] P. C. Bates, J. C. Wileden, "High-Level Debugging of Distributed Systems: The Behavioral Abstraction Approach," The Journal of Systems and Software, Volume 3, 1983, pp. 255-264.

[Bat88] P. C. Bates, "Distributed Debugging Tools for Heterogeneous Distributed Systems," *Proceedings of the Eighth International Conference on Distributed Computing Systems* June 1988, pp. 308-315.

[BeE88] T. Bemmerl, N. Erl, O. Hansen, "Menu and Graphic Driven Human Interfaces for High Level Debuggers," Microprocessing and Microprogramming, The Euromicro Journal, Volume 24, 1988, pp. 153-160.

[Bem88] T. Bemmerl, "An Integrated and Portable Tool Environment for Parallel Computers," *Proceedings of the 1988 International Conference on Parallel Processing*, Volume II, St. Charles, Illinois, August 1988, pp. 50-53.

[BeB90] T. Bemmerl, A. Bode, P. Braun, O. Hansen, P. Luksch, R. Wismüller, "TOPSYS - Tools for Parallel Systems," Users's Overview and User's Manuals, Technische Universität München, Institut Für Informatik, TUM-19047, December 1990.

[BeB91] T. Bemmerl, P. Braun, O. Hansen, T. Treml, R. Wismüller, "The Design and Implementation of TOPSYS," Version 1.0 Release Notes, Technische Universität München, Institut Für Informatik, May 1991.

[BrS84] M. H. Brown, "A System for Algorithm Animation," Computer Graphics, Volume 18, Number 3, July 1984, pp. 177-186.

[BrS85] M. H. Brown, R. Sedgewick, "Techniques for Algorithm Animation," IEEE Software, Volume 2, Number 1, January 1985, pp. 28-39.

[Bro88] M. H. Brown, "Exploring Algorithms Using BALSA-II," IEEE Computer, Volume 21, Number 5, May 1988, pp. 14-36.

[CaK92a] T. L. Casavant, J. A. Kohl, Y. E. Papelis, "Practical Use of Visualization for Parallel Systems," Invited Keynote Address Text for 1992 European Workshop on Parallel

Computers, Barcelona, Spain, March 23-24, 1992 (15 pages).

[CaK92b] T. L. Casavant, J. A. Kohl, Y. E. Papelis, "Practical Use of Visualization for Parallel Systems," Technical Report Number TR-ECE-920102, Department of ECE, University of Iowa, Iowa City, IA, 52242, (full version of EWPC 92 paper).

[CoK90] A. L. Couch, D. W. Krumme, "Monitoring Parallel Executions in Real Time," To appear in *Proceedings Fifth Distributed Memory Computing Conference, IEEE Computer Society Press, 1990.*

[CoR91] K. C. Cox, G.-C. Roman, "Visualizing Concurrent Computations," *Proceedings of the 1991 IEEE Visual Languages Workshop,* Kobe, Japan, October 1991, pp. 18-24.

[DoB90] J. J. Dongarra, O. Brewer, J. A. Kohl, S. A. Fineberg, "A Tool to Aid in the Design, Implementation, and Understanding of Matrix Algorithms for Parallel Processors," Journal of Parallel and Distributed Computing, June 1990, pp. 185-202.

[FuK91] C. A. Funka-Lea, T. D. Kontogiorgos, R. J. T. Morris, L. D. Rubin, "Interactive Visual Modeling for Performance," IEEE Software, Volume 8, Number 5, September 1991, pp. 58-68.

[Gan19] H. L. Gantt, "Organizing for Work," Industrial Management, Volume 58, August 1919, pp. 89-93.

[GuG89] V. A. Guarna, Jr., D. Gannon, D. Jablonowski, A. D. Malony, Y. Gaur, "Faust: An Integrated Environment for Parallel Programming," IEEE Software, Volume 6, Number 7, July 1989, pp. 20-27.

[HeE91] M. T. Heath, J. A. Etheridge, "Visualizing the Performance of Parallel Programs," IEEE Software, Volume 8, Number 5, September 1991, pp. 29-40.

[Hea90] M. T. Heath, "Visual Animation of Parallel Algorithms for Matrix Computations," Mathematical Sciences Section, Oak Ridge National Laboratory.

[KoC91a] J. A. Kohl, T. L. Casavant, "Use of PARADISE: A Meta-Tool for Visualizing Parallel Systems," Proceedings of the Fifth International Parallel Processing Symposium, Anaheim, California, May 1991, pp. 561-567.

[KoC91b] J. A. Kohl, T. L. Casavant, "PARADISE: A Meta-Tool for Program Visualization in Parallel Computing Systems," Technical Report Number TR-ECE-901011, Department of ECE, University of Iowa, Iowa City, IA, 52242, Revised December 1991, (first revision submitted IEEE Transactions on Software Engineering).

[KoC91c] J. A. Kohl, T. L. Casavant, "Methodologies for Rapid Prototyping of Tools for Visualizing the Performance of Parallel Systems," Presentation at Workshop on Parallel Computer Systems: Software Tools, Santa Fe, New Mexico, October 1991.

[Koh91] J. A. Kohl, "Visual Techniques for Parallel Processing," Written Comprehensive Examination, University of Iowa, Department of Electrical and Computer Engineering, ECETR-910726, July 1991.

[MAC92] Apple Computer Corporation, Macintosh Software Application, 1992.

[MaR89] A. D. Malony, D. A. Reed, "An Integrated Performance Data Collection, Analysis, and Visualization System," *Proceedings of the 4th Conference on Hypercubes, Concurrent Computers and Applications*, 1989.

[MeM85] B. Melamed, R. J. T. Morris, "Visual Simulation: The Performance Analysis Workstation," IEEE Computer, Volume 18, Number 8, August 1985, pp. 87-94.

[MiC90] B. P. Miller, M. Clark, J. Hollingsworth, S. Kierstead, S.-S. Lim, T. Torzewski, "IPS-2: The Second Generation of a Parallel Program Measurement System," IEEE Transactions on Parallel and Distributed Systems, Volume 1, Number 2, April 1990, pp. 206-217.

[MiY89] B. P. Miller, C.-Q. Yang, "Performance Measurement for Parallel and Distributed Programs: A Structured and Automatic Approach," IEEE Transactions on Software Engineering, Volume 15, Number 12, December 1989, pp. 1615-1629.

[Mor92] Xpic Graphics Tool, Created By Mark Moraes, CSRI, University of Toronto, 1992.

[Mor74] M. F. Morris, "Kiviat Graphs - Conventions and Figures of Merit," Performance Evaluation Review, Volume 3, Number 3, October 1974, pp. 2 - 8.

[Mye88] B. A. Myers, "The State of the Art in Visual Programming and Program Visualization," Report Number CMU-CS-88-114, Computer Science Department, Carnegie Mellon, 1988.

[Nic90] K. M. Nichols, "Performance Tools," IEEE Software, May 1990, pp. 21-30.

[ReO91] D. Reed, R. Olson, R. Aydt, T. Madhyastha, T. Birkett, D. Jensen, B. Nazief, B. Totty, "Scalable Performance Environments for Parallel Systems," *Proceedings of the Sixth Distributed Memory Computing Conference*, IEEE Computer Society Press, April 1991.

[RoC91] D. T. Rover, M. B. Carter, J. L. Gustafson, "Performance Visualization of SLALOM," *Proceedings of the Sixth Distributed Memory Computing Conference*, 1991.

[RoW90] D. T. Rover, C. T. Wright, "Pictures of Performance: Highlighting Program Activity in Time and Space," *Proceedings of the 5th Distributed Memory Computer Conference*, IEEE Computer Society Press, 1990.

[Rov89] D. T. Rover, "Visualization of Program Performance on Concurrent Computers," PhD Thesis, Iowa State University, Ames, Iowa, 1989.

[Sta90] J. T. Stasko, "Tango: A Framework and System for Algorithm Animation," IEEE Computer, Volume 23, Number 9, September 1990, pp. 27-39.

[Sut92] Xfig Graphics Tool, Created By Supoj Sutanthavibul, Copyright 1985.