The IMPROV Meta-Tool Design Methodology for Visualization of Parallel Programs*

Thomas L. Casavant: tomc@eng.uiowa.edu *and James Arthur Kohl*: kohl@eng.uiowa.edu
Parallel Processing Laboratory, Department of Electrical and Computer Engineering
University of Iowa, Iowa City, IA 52242

Abstract

A design methodology is presented that simplifies the creation of program visualization tools while maintaining a high degree of flexibility and expressive power. The approach is based on a *circulation architecture* model that organizes the details of the user specification, and provides a formal means for indicating relationships. The overall user specification is divided into independent modules containing distinct, well-defined entities, and the relationships among these module entities are identified using a powerful *mapping language*. This language maps conditions on entities to manipulations that modify entities, resulting in dynamic animations of program behavior. The mapping language supports arbitrary levels of abstraction providing a full range of detail, and allowing efficient view development. To demonstrate the feasibility and usefulness of this approach, a specific program visualization meta-tool design, IMPROV, is described.

Keywords: program visualization, software engineering, parallel processing, human-computer interfaces, debugging, performance tuning.

1. Introduction & Motivation

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. 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 often 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. The user of the meta-tool can implement a specific task from within the class by specifying and manipulating these parameters. A program visualization meta-tool allows users to customize visual analyses and views to suit individual perspectives and intuitions.

It would seem that constructing a general meta-tool would require a full understanding of parallel software and visualization. To the contrary, 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 for the user, but that also allows the designer to arbitrarily experiment within the paradigm and gain new understanding.

* This work is supported by the National Science Foundation under Grant Number CCR-9024484 and by DARPA under Contract Number N00174-91-C-0116.

Unfortunately, using meta-tools can be complex, requiring experience and extensive specification. 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 architecture, user-specified **rules** dictate how the state of each entity affects the state of other entities. These rules are constructed using the **relationship mapping language** as a precise means for specifying the relationships among details in the modules. Each rule consists of 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, 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 illustrate the nature and usefulness of this methodology, a specific program visualization meta-tool design, IMPROV, is briefly described, utilizing the circulation architecture and mapping language concepts. Useful features specific to parallel software visualization are included. The resulting customizable tool provides 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.

2. Background Perspectives

In reaching a better understanding of program visualization, there are at least three perspectives to consider [CaK92,KoC91b]. 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 them in a real tool. 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 tool designer approach and leans toward simpler, less flexible tools with sets of fixed, general-purpose views. These toolkits may not provide the customizability of more general tools, but can sufficiently satisfy the needs of the user and are easier to learn.

2.1. User View

The issues that are important to tool users involve ease of use and ultimate power of the tool, independent of how elegantly the tool is constructed or how complex the underlying system is. The specific issues of concern to the user are customizability, flexibility, expressive power, efficiency, and granularity control. Many of these terms are self-explanatory, and a full description of each of them is provided in [KoC91b,Koh92], but the key points will be discussed briefly here.

Customizability refers to how well the tool can be adjusted to suit specific user preferences, implying the need for some means for specifying default parameter values. This is different than flexibility, which indicates how diverse the capabilities of the tool are. Granularity control refers to how well the user can focus and control the visual analysis, to prevent visual "overload" and to examine details more closely. Tool capabilities which illustrate this concept include animation speed and direction control (running an animation backward can be valuable, especially if the user "blinks" and misses something), abstraction level control, and visual "analysis guidance". The manipulation of abstraction level can greatly reduce the amount of extraneous details, and provide useful generalities concerning program behavior. Visual "analysis guidance" refers to providing the user with extra assistance in locating important aspects of program behavior. This is accomplished by utilizing visual contrasts to attract the user's attention to critical behavior.

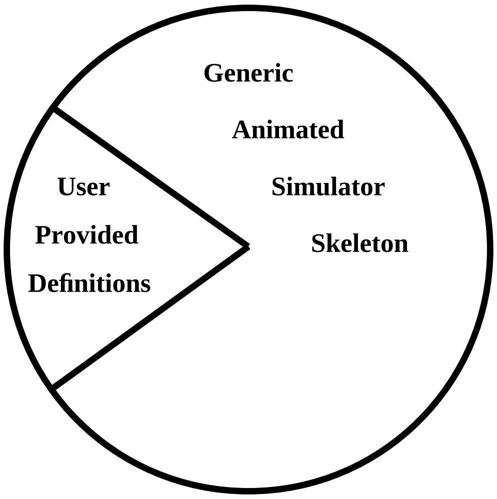
2.2. Meta-Tool Designer View

The goal of the meta-tool designer is to provide the full benefit of powerful, sophisticated tools while attempting to reduce the associated overhead to an acceptable minimum. The central task is to mask as many internal low-level details as possible from the user, and require specification of only the necessary high-level details. 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.

The masking of low-level internal details is accomplished by utilizing common principles of modularity and layered interfaces, There must be a concise, formal specification syntax for creating and controlling views — this requires a suitable means for specifying visual semantics. The lack to date of a standard "visualization language" 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.

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, as shown in Figure 1 [KoC91a, KoC92]. 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. The Pablo [ReO91] and Triplex [CoK90,Nic90] program visualization tools utilize a visual tree structure to process program behavior data and display it. The Tango visualization system annotates user applications using an abstract visual manipulation language to specify the movement of graphic objects in conjunction with a program's execution [Sta90].

Textual languages can be cumbersome for specifying some aspects of visual behavior. An abstract, visually-oriented language is necessary to describe 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 "solve" the visual programming problem, and further borders on transcending conventional and natural languages to constitute an abstract, symbolic, universal language. Clearly, this is not possible given current technology. However, within the limited context of program visualization it is feasible, and a sufficient syntax can be constructed.



→ Complete Customized Animated Simulator

Figure 1: Plug-In Simulator Concept

Another 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, it is necessary for the designer to generalize and formally classify the fundamental parameters of the analyses. Designing a meta-tool in this way requires a full understanding of both the needs of the user and the basic concepts and parameters of the paradigm under analysis. The operations on these parameters must be understood, and a means for effectively specifying these parameters must be constructed. The underlying model on which the meta-tool is based becomes critical.

2.3. Toolkit View

The toolkit view takes a simplistic approach to solving the specification problem, by providing a fixed set, or "kit", of views, or "tools", with limited flexibility. The benefit of such a toolkit is that analyses proceed almost automatically with little user overhead. One successful example is ParaGraph [HeE91]. The drawback lies in the potential for limits in analysis power, depending on the quality and size of the toolkit. Even when a good view is possible, the overhead to construct that view using the tools can be significant.

One of the key issues which distinguishes the meta-tool approach from the toolkit approach involves the learning curve to use each type of tool. As a user becomes more experienced with a tool, the overhead is reduced as the user learns more effective methods of using the tool, as shown in the following expressions:

$$\mathbf{T}_{\mathbf{Analysis}_1} = \mathbf{T}_{\mathbf{Learn\ Tool}} + \mathbf{T}_{\mathbf{Design\ Analysis}_1}$$

$$\mathbf{T}_{\mathbf{Analysis}_n} = \mathbf{T}_{\mathbf{Design\ Analysis}_n} - \mathbf{f(Experience)}$$

The first analysis includes time to learn the tool, along with the time to design the view. Subsequently, the user's experience continues to reduce the development time for each new view. For a meta-tool, the time to learn the complexities of the specification language can be larger than for a toolkit. However, the solutions generated with the meta-tool are more specialized and will likely produce more effective results. With the toolkit, each new view must be reconstructed using the provided tools, imposing a higher design time than a meta-tool in the long term. This comparison is illustrated in Figure 2.

In the short term, the toolkit requires less overhead to learn and use. In the long term, the meta-tool can provide more

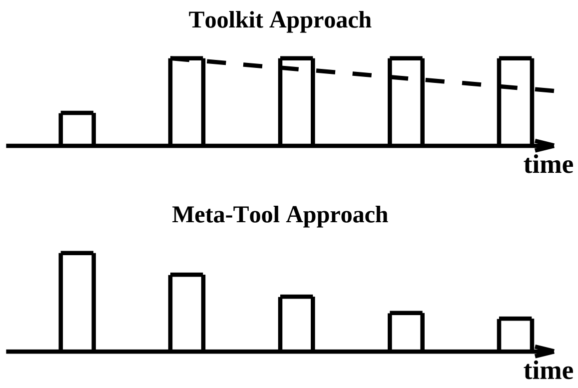


Figure 2: Toolkit & Meta-Tool Development Times

powerful analyses with less effort than forcing the toolkit to do the job. The trade-off resembles the "right tool for the right job" argument, as the overhead to make the toolkit work for a particular analysis can well surpass the meta-tool specification overhead.

3. Approach

The goal of the methodology presented here 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. A prototype tool, IMPROV, is under development to verify the validity of the concepts.

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 subsections describe the methodology and the IMPROV design based on the methodology, respectively.

3.1. 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, into a trace that describes the program behavior. The events encapsulate the internal behavior of the software, including the timing of specific actions. The sequences of events are analyzed in tuning software for performance and for debugging software.

To be utilized, the raw event data must often be processed to derive the desired information. The **behavior** component acts as a buffer between the raw data and the output to the user, and provides for any necessary manipulation or numerical analysis. Various mathematical functions are applied to convert the individual raw event instances into useful statistics, and to organize the event information. Results can be generated which reveal high-level features and characteristics of program behavior.

The software behavior information, once derived, 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. The event and behavior information must be linked 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.

There is no single specification approach that applies equally 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 difficulties in the remaining components. The methodology relieves this imbalance by specifying the fundamental details of each component independently, utilizing the specification structure most natural for that component.

To allow this independence in component specifications, a means is required for efficiently filling in the remaining dependent details which relate the independent components. There must be a common, underlying foundation to provide a frame of reference for combining the diversely structured components. The methodology supports these requirements by introducing a mapping language for relating the details from different components, in terms of fundamental constructs called **entities**. These entities organize and encapsulate information into distinct sets which contain closely related details, and provide a uniform interface for accessing the information. The entities constitute the "building blocks" for constructing a complete specification.

Entities within each component can be arranged in arbitrary structures to suit the specification needs. 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, forms an abstraction hierarchy for each entity. The mapping language can reference entities anywhere in this abstraction hierarchy, providing a highly flexible means for relating the independently specified details.

The entities for each component are collected into a single **module**. 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 structure of raw events and their associated attribute data. Conditions can be imposed using expressions on entities and their attribute values, throughout the entity hierarchies, allowing detailed identification of specific occurrences. The state portion of the events module maintains the data associated with specific event instances. These event instances are input from execution traces or can be generated by the circulation architecture.

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, representing scalar values, arrays, or structures. The declarations can include static relationships among behavior variables, and can set initial values. The state portion of the behavior module maintains the values of the behavior variables throughout 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 the high-level graphical entities. The state of the graphical entities corresponds to the values of graphical attributes, including location, rotation, scaling, etc.

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 circulates and is processed via a set of **relationships**. These relationships identify changes in state of the entities according to precise conditions on the entities and their attributes, and describe how entities influence each other. The new entity states are recirculated back into the modules for the next iteration, and the new graphical entity states are also passed along to update the animation.

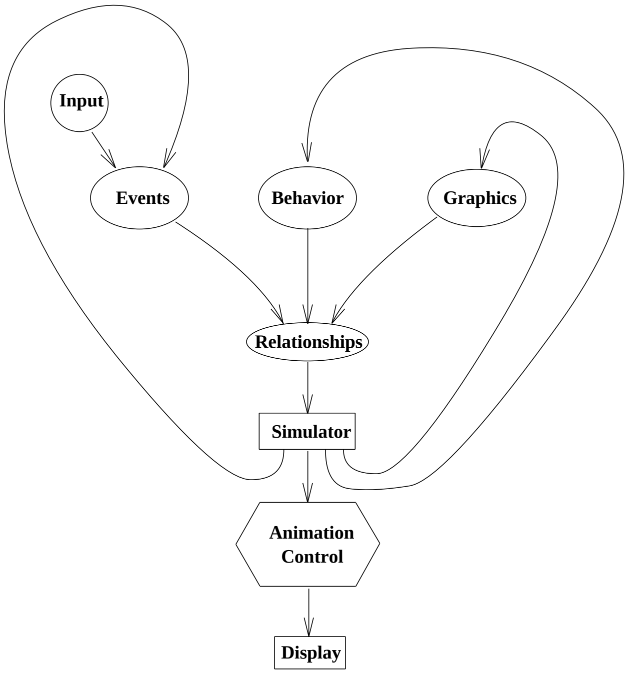


Figure 3: Meta-Tool Architecture

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**, and consist of a set of **rules** of the form:

Condition → Manipulation

The **condition** or "head" of a rule is an expression imposing some restriction on the state of entities. 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.

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 relationships specified using the mapping language constitute the details for a **plug-in simulator**. This simulator applies the state information from the entities to the conditions of the relationships, and automatically derives new state information using the manipulation functions. The simulator also passes along a copy of the graphical state to produce the animation.

The user can adjust and focus the animation using the **animation control interface**, which acts like a valve or lens to control the overall perspective onto the visual analysis. The speed and direction of the animation can be set, as can 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 entities. 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 events can represent. Events are categorized as being either **qualitative** or **quantitative**. Qualitative events correspond to specific conditions, having a value of either TRUE or FALSE, such as having passed a particular point in a computation. Quantitative events carry numerical information, other than the timestamp, that describes the value of some program metric or characteristic, or the result of some computation. The use of these two types of events can be substantially different, requiring more than trivial adjustments in the behavior and graphics modules when the event type is switched. For example, instantiation of a metric value event in place of a boolean event would likely require the addition of some numerical processing.

3.2. Example Design

This section describes the example design for the meta-tool prototype IMPROV, based on the methodology. The discussion here will be limited to an overview of the key features and language syntax.

The specification of events in IMPROV is based on EDL (Event Definition Language) [BaW83,Bat88]. This language allows primitive and compound events to be defined, along with appropriately typed attributes. 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
      <event1>.<attr2>  ==  3;
      <event2>.<attr3>  <=  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 **sub-events**, possibly "primitive". This combination can indicate temporal relationships, such as concurrent versus sequential, and other standard groupings. Any attributes of the event are also defined, with appropriate data types and values derived from the sub-event attribute values.

The behavior variables for IMPROV are specified using typical variable declaration syntax, such as found in C. Behavior variables can be scalars, vectors, and multi-dimensional arrays of 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 interfacing. This implicit sizing also proves useful for "sparse" arrays by utilizing a table-lookup indexing scheme to avoid wasted storage.

It should be noted that the use of **explicit** array sizing may

```
double util[128][128];
double x[][][];
char message[256];
int i,j;
```

Figure 5: Example Behavior Variable Declarations

be necessary in some cases, e.g. to indicate the number of processors used in an execution. Any such explicit specification requires agreement among the modules. To alleviate this problem, IMPROV is designed to reduce or extend arrays as necessary should a mismatch occur, and can instantiate implicit arrays appropriately for interfacing with explicit ones.

The graphical entities are statically defined in IMPROV, such that the user need only indicate the creation of desired instances. Graphical entities implicitly carry attributes that specify the shape, size, rotation, and other defining characteristics for the given graphical object. The set of attributes for a graphical object actually overspecify the state of a particular instance, such that any combination which completely defines the 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. A precise textual representation mimics the graphical manipulations, allowing straightforward integration of the graphics with the other modules. There is a direct, unambiguous mapping between the visual and textual representations, with graphical variables corresponding to graphical object instances. This allows automatic mirroring of one form while the user is manipulating the other. 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.

The relationship mapping language for IMPROV combines the various details of events, behavior variables, and graphical objects into a complete analysis specification. Entities can be accessed on a high level or using low-level attributes. The conditions and manipulations of the rules are expressions on the entities. Condition expressions are specified using comparators much like standard C, such as ==, !=, <=, >=, &&, ||, etc. Manipulation expressions consist of one or more statements, each representing some action that manipulates the entities. 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 example, italics is used to represent events, bold represents behavior variables, and the items enclosed in braces ({}) indicate references to graphical objects. The first example rule indicates that whenever an *update* event occurs, a circle, representing the processing element that issued the *update*, changes colors based on the 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 computed using the behavior variable **scale**. 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. It is clear from these rules that very simple, as well as arbitrarily complex, relationships are possible.

Each of the modules supports specific syntax subsets within the mapping language. Details of the full mapping language grammar are presented in [Koh92]. As mentioned above, events may be utilized in two different ways, by their occurrence or by specific attribute values. In a condition expression, an event reference with no attributes indicates that the desired condition on the event 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 has not only occurred, but that the given attribute has satisfied some restriction on its value. An extension to this latter case occurs when an event is referenced by its timestamp attribute. In this case, the condition relates to the entire event history, and examines some subset of events in a particular time frame. In the manipulation expressions, events can occur without attributes, indicating the creation or deletion of an instance of that event. Events attributes can also be referenced in manipulation expressions as part of functions or calculations. In this context, the event reference utilizes the value of an event attribute for manipulating another entity.

The use of a behavior variable in a condition expression implies a comparison of the value of that variable, or its attributes, to some other value. Behavior variables in manipulation expressions can indicate either modification of the variable or attribute values, or their use in manipulating other entities. The references to graphical objects appear in similar ways to 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, like 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.

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 handle many replicated instances of entities, as are likely to be found in analyses with multiple identical processors, functional units, etc., a function is introduced for manipulating and constructing large sets of entities. The "REPLICATE" function organizes very large, complex arrangements of state and graphics, reducing the tedium in repeatedly specifying these details by hand. For example, this function can produce large collections of graphical entities in arbitrary patterns and layouts.

4. Illustrative Example

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

4.1. 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 7 through 10 below.

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

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

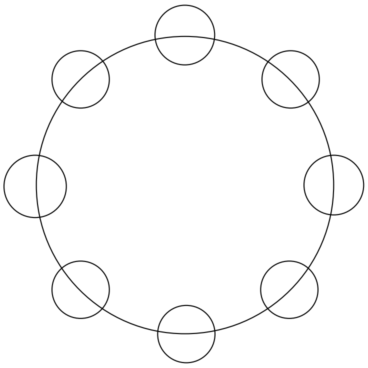
event send *is* primitive "raw_send"
with
    int src;
    int dst;
end

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

Figure 7: Event Specification

None Necessary.

Figure 8: Behavior Variable Specification



CREATE {circle} radius := 10;

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

Figure 9: Graphical Specification

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 10: 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. Conclusion

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

References

[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.

[CaK92] 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.

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

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

[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, "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.

[KoC92] J. A. Kohl, T. L. Casavant, "A Software Engineering, Visualization Methodology for Parallel Processing Systems," Proceedings of the Sixteenth Annual International Computer Software & Applications Conference (COMPSAC), Chicago, Illinois, September 1992, pp. 51-56.

[Koh92] J. A. Kohl, "The Construction of Meta-Tools for Program Visualization of Parallel Software," Technical Report Number TR-ECE-920204, Department of ECE, University of Iowa, Iowa City, IA, 52242, February 1992.

[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.

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