RC22739 (W0303-032) March 7, 2003
Computer Science

IBM Research Report

MindFrames: A Visual Environment for Semantically-Oriented Program Construction

Herb Derby, Robert M. Fuhrer, Donald P. Pazel
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Delhi - Haifa - India - T. J. Watson - Tokyo - Zurich

LIMITED DISTRIBUTION NOTICE: This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. Ithas been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g, payment of royalties). Copies may be requested from IBM T. J. Watson Research Center , P. O. Box 218, Yorktown Heights, NY 10598 USA (email: reports@us.ibm.com). Some reports are available on the internet at http://domino.watson.ibm.com/library/CyberDig.nsf/home.

MindFrames: A Visual Environment for Semantically-Oriented Program Construction

Herb Derby
IBM Watson Research Center
19 Skyline Drive
Hawthorne, NY 10532
914-784-7502
herbd@us.ibm.com

Robert M. Fuhrer
IBM Watson Research Center
19 Skyline Drive
Hawthorne, NY 10532
914-784-7773
rfuhrer@watson.ibm.com

Donald P. Pazel
IBM Watson Research Center
19 Skyline Drive
Hawthorne, NY 10532
914-784-6916
pazel@us.ibm.com

ABSTRACT
The MindFrames project is a research effort addressing the semantic gap between our conceptualizations within application problem domains and implementations about them. At a superficial level, our approach concerns diagramming and the use of domain-specific visualizations to bridge this gap. At a foundational level, it involves deeper principles involving program construction methodology, state reflective capabilities, a rich semantic model for program visualization, and a carefully crafted condensation of abstraction and concretization. We present the vision of our work, and a discussion of the MindFrames prototype. We demonstrate an environment in which constructive, state reflective visualization is the means for programming, and add to that a discussion of advanced topics such as design patterns and partial evaluation. We also discuss design issues regarding our semantic programming model, visual infrastructure, and graphics foundations.

Keywords
Algorithms, Documentation, Performance, Design, Reliability, Experimentation, Human Factors, Languages, Verification.

1. INTRODUCTION
Much of the history of programming language evolution can be viewed as an attempt to bring clarity, speed, and facility to programming applications through abstraction. At a rudimentary level, programming languages have from the beginning shielded programmers from hardware considerations such as register usage. Language structure evolved for organizing processing at a higher level through decomposition into individual procedures. Eventually entity-relationship and object-oriented approaches developed simple data and process encapsulation techniques. All of these developments can be seen as an attempt to bridge concept to programming.

Commensurate with this evolution, programming tools followed its own developmental path. Much of the effort in this area has focused on making source text entry easier and clearer, achieved primarily through smart text assists on syntactic structures and program dictionaries. Recently, tools such as IBM's Eclipse and Idea's IntelliJ offer increasing degrees of program semantic manipulability through refactoring operations, relieving the programmer of tedium during activities such as renaming programming elements or code movement. These tools, along with debuggers and execution profilers, are standard equipment in every programmer's toolbox.

Despite these laudable efforts, programming is still an onerous and therefore costly endeavor, specifically, we argue, because the programmer is left with a large conceptual gap between the problem domain and the artifacts of programming. It is our experience that the most useful problem-solving models remain on white-boards or paper scraps as diagrams, or as conceptual mental images – most of which gets discarded in the course of development. Tooling offers little or no help in building a comprehensive view relating the ideas embodied in the application to the concrete programming entities. As a result, the need to continually translate between problem conceptualization and programming wastes much time and energy.

This paper presents an applicative approach to programming based on several key ideas. First, the kind of intuitive diagrams one draws when working out a solution on paper can in fact become the program. Second, programmers need help understanding the evolution of the data's state throughout a parallel or sequential program's execution. In MindFrames, this help takes the form of "state reflection," a means for reflecting information about a data entity's state in its visualization. Third, domain-specific visualizations are critical to narrowing the conceptual gap. Finally, programming environments need to provide operations for constructing and manipulating code that are semantically aware (cf. textual editing operations like "delete character").

This paper is organized as follows. The next section is a detailed look at the current state of programming and the problems therein. This is followed with a discussion of a conceptual solution to these issues in the spirit of the MindFrames project. This is followed by a close look at the MindFrames programming environment. The architecture of the various components is described at length. Finally, we share our thoughts on future work and direction.

2. Transforming Ideas into Code

Inasmuch as the prior section points out inadequacies in the current state of programming tools, it indicates a better approach to programming, based in part on common engineering practices not strictly limited to programming. These include white-board diagramming, state drawing, design sketching, along with state evolution diagramming in reaction to process.. In this section, we examine a few such approaches and motivate how their use provides a foundation for improving programming tools. We show that these approaches relate intuitively to the methods we use to think about the core problem-solving domains. So, not only do diagrams help us clarify ideas, but also provide a means to facilitate working with ideas, e.g. to change or morph ideas into different or better ideas. The result is an environment wherein the focus is building on the concepts being manipulated, thus facilitating a path from ideas to code, and de-emphasizing the mere management of programming details.

2.1 Diagrams and the Programming Process

The path to learning and understanding a discipline is often rich in visual illustration. Be it in textbooks, the classroom, or industrial practice, graphical depiction is often the best means to convey an idea, related ideas, and their relationships, and further their dynamics.

The similarity in illustration use over many domains is quite striking, but is particularly striking for the mathematical, scientific, and computing disciplines. In mathematics, depicting sums as areas on a Cartesian plot, and abstractions such as sets as Venn diagrams, are standard tools for facilitating conceptual understanding. In physics, drawings illustrate physical elements and states and dynamics in real-world phenomena. In programming, data elements and their relationships are illustrated in many variations of block diagrams. Further, illustrations enhance our understanding when each phase of reasoning about conceptual artifacts is reflected in a graphical transition in the drawing. For example, in mathematics, reasoning steps result in diagrammatic changes relating mathematical objects; the changes due to the onset of physical dynamics is clarified by changes in depictions of affected physical objects; and depictions of data element state transitions clarify the impact of executing processes.

To date, conceptual illustrations as described above have chiefly been utilized effectively for educational or design purposes. In the latter case, the diagrams are tossed aside, in favor of less intuitive artifacts used during the implementation phase. How to turn them into a basis for a pragmatic programming paradigm is less apparent. An appreciation of the fundamental programming activities, however, provides motivation for pursuing this paradigm and clues as to what that paradigm entails. Consider the following typical programming activities:

- Creating or identifying programming objects or data of interest, e.g. variables and data structures, and establishing relationships, e.g. pointer structures, lists.
- Inferring the current state of data of interest, e.g. x is set, unset, possibly set, or pointer p points to a FOO object.
- Reasoning about the correct execution order through repeated application of the above.
- Trying abstract code with concrete values in-situ to gain confidence that the computation proceeds as anticipated.
- Abstracting specific code fragments to more general use.

The above indicates that imperative programming has as a vital component an exercise in state management through an incremental state-transition process. Through pencil and paper, or "in our heads", we conceptualize what is happening at each point in a program, usually through visual representations of data and state, and often through example. The current state of practice in programming tools does not externalize these state management processes, nor the movement from abstraction to concretization. Consequently, at best we store the results of such efforts on secondary media (e.g., paper or whiteboard) so that they exist only "outside the system," or discard them, only to reconstruct them when needed again.

A close look at the above reveals several key facets of the programming process:

- **Construction** – Programming is a process of data state construction and evolution. We carefully choose program operations that transform the data to new desired states.
- **State Reflection** – Knowledge about a program's data state at any execution point is critical to reasoning effectively about what should be done constructively or correctively with the code.
- **Domain Specific Representation** – Application programming is deeply rooted in our conceptualization of the application model, its objects and relationships.
- **Causal Localization** – Operation execution order dependence or independence is a significant factor in state inference and management.
- **Virtualization** – Propositions over abstract state (such as "x is non-NULL"), though important, often don't provide enough leverage to understand program behavior. We need to explore concrete examples within a localized context.

These facets are examined more closely in the following sections from the perspective of using diagrams as a means for facilitating their utilization.

2.2 Construction

A program can be viewed as a constructive process based on the notion of building or managing data state, and evolving state through program operations. The act of programming, however, can be viewed as a constructive activity with a more proactive dynamic in which data state manipulations map onto program operations that achieve that change. This takes the usage of diagrams from being informally descriptive to being the program artifacts themselves.

Key to constructive program development is isolating the data and relationships of interest, and manipulating them to further develop process. From a diagrammatic viewpoint, that means that data and relationships are visualized, and their manipulation results in process, i.e., a program. Viewed from this perspective, a program may be broken into sequences or hierarchies of state goals, each with their own state evolution.

Figure 1 illustrates constructive activity. The example shows several data structures and relationships, along with two visual value assignments (arcs). The two resulting programming statements appear on the lower right. Pointer `p` refers to object instance `A`, whose member `table` refers to array `B` of data elements. Element `C` is the third element in `B`, and has members `x` and `y`. Pointer `r` refers to object `D` with member `z`. There is also a free-standing expression `a+b`.

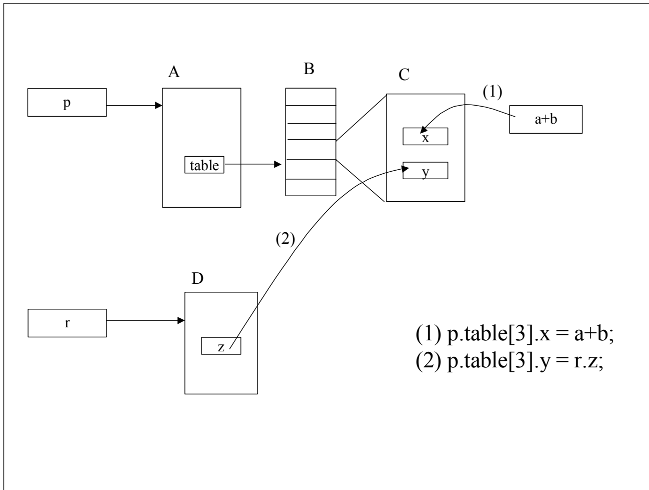

Figure 1- Programming as a constructive process

In this example, the expression `a+b` is assigned to `p.table[3].x`, and the value `r.z` is assigned to `p.table[3].y`. In a visual system, these assignments could be created through drag/drop of value sources onto value destinations. In this case, the expression `a+b` is dropped onto `x`, and `z` dropped onto `y`. With each gesture, the appropriate program statement is generated. The arcs 1) and 2) may remain on the diagram indicating the flow of data.

Notice the conceptual clarity in the rendered data representation, including the levels of pointer indirection. Also note that the use of drag/drop as an assignment gesture is simple, powerful, and appropriate, especially given that drag/drop operations do not disturb the source object, instead moving a copy of the image while interacting with target objects.

2.3 State Reflection

Clearly, the usefulness of diagrams in a constructive programming paradigm increases with the level of useful detail provided about the data and their relationships. The definition of appropriate detail is typically context-dependent. For example, relevant detail might include the state of variable initialization (i.e., set, unset, possibly set). Alternatively, detail could include known "points-to" relationships (reference `p` points to object instance `A`). In state diagrams, detail could refer to the present state or a set of possible present states. This information can be reflected visually using any of a variety of methods, e.g. colorations, arcs, highlights, fill patterns, and so on.

Providing such informative detail about program state is referred to here as *state reflection*. This information is dynamic and changes with constructive activity; thus, it is important that visual cues are updated properly after each constructive action.

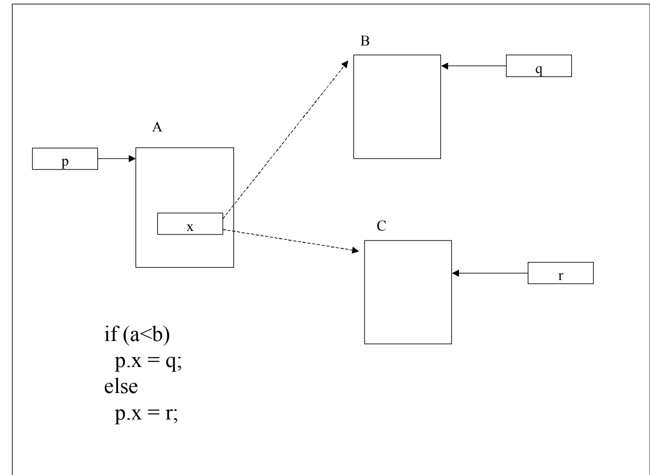

Figure 2 - State reflection of pointer relationships

Figure 2 depicts a constructive state in which pointer `p` refers to an object `A` with pointer member `x`. Pointer q references object `B`; and `r`, `C`. Consider the data state after execution of the "if" statement: `A`'s data member `x` may point to `B` or `C`. This state is "reflected" in the two dashed "possible pointer relationship" arcs. Having this information readily available is critical to making decisions in subsequent programming activities.

2.4 Domain-Specific Representation

Programming is never done in a vacuum, but rather in the context of some conceptual model. Application programming involves the analysis and manipulation of that model to such an extent that we are forced to translate back and forth between program and model. That is, we continually reconstruct the state of the model from the code and visa-versa. Thus, the closer the model is to the application structure, the easier the programming.

In other words, the more accurately the application corresponds to the illustration, the greater confidence we have in its use as a programming medium. The diagrams presented earlier demonstrate an effective use of pointer and array representations which, while useful, utilize traditional programming artifacts. It is possible to gain even more leverage by broadening the scope of visualizations. Figure 3 shows a traditional state diagram, along with the corresponding code fragment. Other useful examples include visuals related to physics, such as spring-mass diagrams, or mathematics, such as Cartesian plots.

We envision many representations specific to application domains that are interactive and state reflective, as well as multiple visualizations for the same domain objects, each with specific capabilities. As importable/exportable conceptual artifacts, such diagrams over time define a "MindShare" or knowledgebase of composable re-usable ideas. The vision is that using this knowledgebase would replace the current preoccupation with

programmatic details with a more focused attention to composing and managing the higher level concepts behind programs.

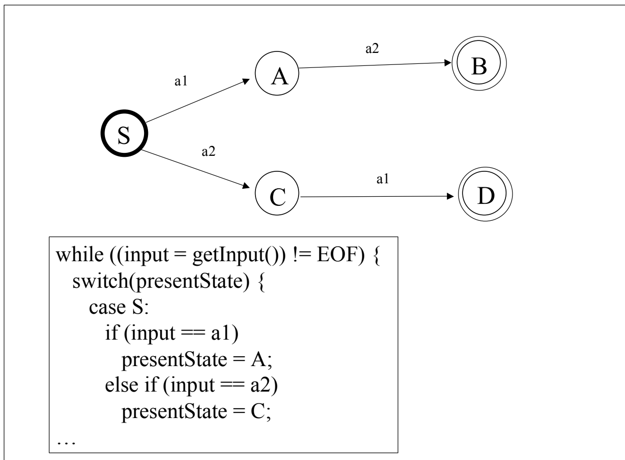

Figure 3 - State diagram as domain-specific visual

2.5 Causal Localization

Visualizing the order of program operation execution is another critical factor in our paradigm, because it is the source of many problems in conventional imperative programming. This happens because the partially gratuitous sequence imposed on a program coded in a linear medium (such as text) obscures its true data-flow and control dependencies. As a result, problems regarding order of value assignments derive from inaccurate analyses of sequentially ordered operations. In fact, it is just as important to know when operations are order-independent as when they are order-dependent. This is especially true in today's increasingly concurrent software. Understanding the flow of data related to a given value assignment is called causal localization.

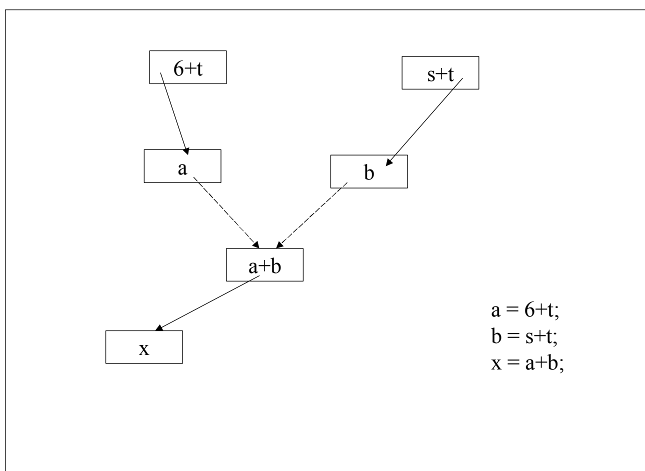

Figure 4 - Causal localization; data-flow

Ferreting out order dependence is laborious and time consuming, but state reflection can help. Figure 4 depicts a constructive drawing accentuating causal localization. The assignments (solid arcs) to `a` and `b` are unrelated, and therefore can be executed in parallel, or in either order. Their values are combined (dashed arcs) into the value assigned to `x`. The assignment to `x` causally succeeds the other two assignments. This is very clear from the diagram, and would remain clear even in significantly more complex diagrams, which cannot be said for the generated code. Having this sort of clarity early in the programming process would avoid errant uses of values.

2.6 Virtualization

Desk-checking or debugging program fragments with actual values are among the most overlooked methods for acquiring program understanding. This process, often affiliated with defect localization, is used in all phases of program development, especially the development phases, to ensure or check that the coding matches intention. However, desk checking is tedious, error-prone, and time-consuming. Debugging traditionally requires execution of a fully built or scaffold system, or at best (when used with unit testing), forces one to decide a priori the granularity at which one must test before building the software.

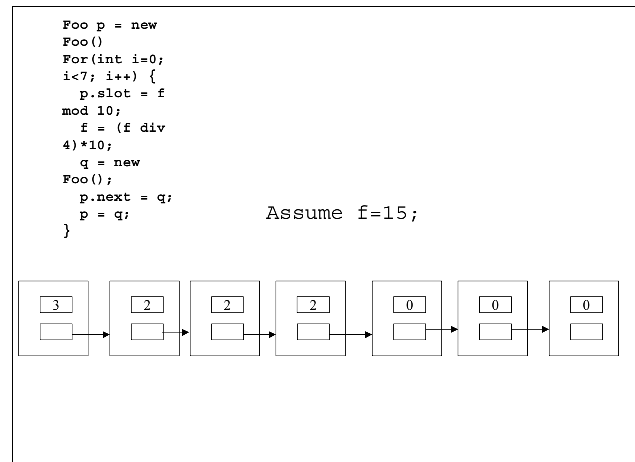

Figure 5 - Virtualized execution

A more effective development environment would allow exercising code with values, without requiring execution from the program's beginning, and without requiring a fully specified program. Allowing "partial evaluations" wherein programs are re-written or simplified with known values for some, but not necessarily all, data elements extends this "virtualization" approach in a most useful way.

Figure 5 illustrates an example wherein a complex value generation scheme is combined with linked list generation. Without concrete values, the behavior of the code is difficult to surmise. In this case, with f set to 15, the value of "slot" stabilizes to 0. The order of elements along with the depiction of next references gives confidence in the correctness of the program.

2.7 Putting the Vision Together

We propose that a programming environment based on these facets provides a positive step in the evolution of the art of

programming. In the following sections, we explore our prototype, MindFrames for Programming, which demonstrates how many of these ideas can be combined effectively.

3. Overview of MindFrames Programming Environment

We built a prototype programming environment to demonstrate many of the ideas exposed in the prior sections. As our focus is on tools that facilitate concretizing ideas and abstracting from implementations, our belief is that many of these ideas transfer naturally to many other disciplines led us to name our programming domain-specific tool "MindFrames for Programming", or MFP. Our infrastructure is designed to support a fluidly dynamic interface to the programming structures to be manipulated. Likewise, the user interface attempts to provide flexible, intuitive and powerful personalized information management. Although MFP itself represents a somewhat neutral imperative language, it is compliant with either Java or C++. In the next sections, both the user interface and infrastructure are explored in more detail.

3.1 MFP – The Environment & Class Construction

The MFP screen shot in Figure 6 shows the construction of a class. The general work area is a free-form graphical area, somewhat similar to what one would find in Smalltalk or Squeak. A number of palettes are available to assist program construction (ref. basic type, expression, and class/package palettes), along with a trashcan for discarding items. The interactive paradigm is primarily drag-and-drop (DnD) along with text focusing visuals

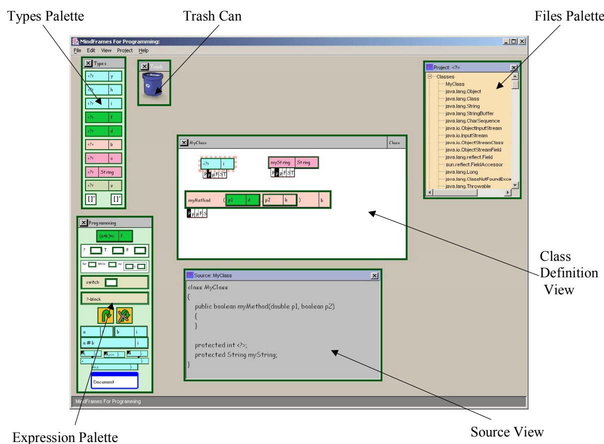

Figure 6 - MFP class definition

allowing text input and change.

The type palette is typically used as a source to construct values, fields or methods, or change types of existing data or methods, through a DnD operation of a type onto an appropriate context. The expression palette provides a set of standard programming constructs such as expressions, if statements, etc., which are similarly instantiated in programs through DnD operations. The file palette provides a list of classes and packages, including those in available class libraries.

In this example, a class definition view for `MyClass` is shown. This view displays declarations for all data and method members in the given class. Populating the definition is generally accomplished by dropping types onto the view. In this example, the class has three members: the method `MyMethod`, the string `myString`, and an unnamed integer field. Unlike most systems, ours does not arbitrarily restrict the usage of incompletely defined elements. In this case, the unnamed `int` data member may be used freely. Mostly likely, however, it should be named at some point; the state reflective dashed highlight indicates so.

Although MFP supports customized visuals of programming elements, a default set of visuals is provided. These are seen in the type and expression palettes, as well as in the class definition view. The default graphics provide some consistency, for example, in the color-coding of types, and the appearance of type information on the right of various programming elements. In this case, the method has two parameters `p1` and `p2`, and returns a `Boolean`. The parameter signature can be changed easily, for example, by dragging items to re-order them, or by dragging them to the trashcan. A set of buttons provides an interactive means of changing an item's access level, e.g. `public`, `protected`, and `private`.

Below the class definition view is a source view, showing the source code corresponding to the class, as either Java or C++. This view is updated with each graphical action. The source view is intended as a "crutch" (the visual *is* the program), to increase the comfort of new users by giving a familiar frame of reference.

3.2 MFP – Method Construction

The MFP screenshot in Figure 7 highlights features for visually programming methods. Here the main window is the coding pane for a method implementing the LZW compression algorithm. The top border displays the name, signature (`void`), and return type (`Boolean`) for the method, and a visual for the "`this`" variable for easy access.

In MFP, a program is constructed as a hierarchy of coding frames. Each frame is a visual work area in which a set of program operations appear. Visual nesting corresponds directly to context scoping. That is, a loop's body and nested contexts within conditionals are represented by frames visually nested within the frame of their outer context. In the figure, the vertical column and horizontal row of frames, somewhat reminiscent of [7], present a portion of the frame hierarchy. The vertical column depicts a program context stack, while the horizontal column displays the sequence of frames at the lowest context level. A border highlight on a miniature depicts it is the "current" frame shown in the middle. The miniatures are "live" in that one can manipulate these frames in the same manner as the "current" frame, and scale automatically when resized by moving the column or row borders.

Figure 7 presents several uses of array. The array visual is highly interactive, providing convenient means for building array-based expressions, e.g. `first`, `last`, `length`, and for producing iterations. The iteration depicted in the top live miniatures represents a forward traversal of the array. The dark square represents the iteration body that is further hierarchically expanded in the miniature below it. The other two uses of array within the nested frame, indicates array slots indexed by

"hashcode" within each. These are the targets of a number of

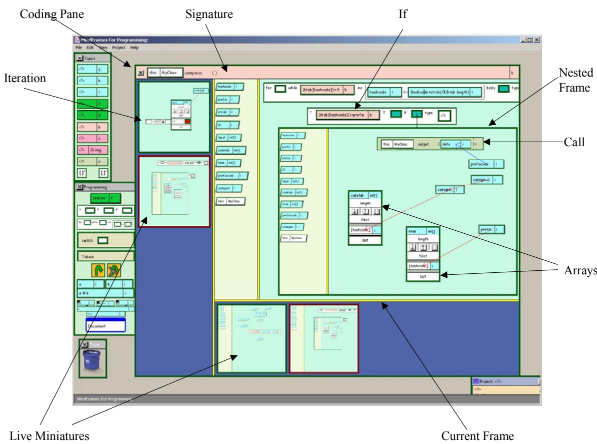

Figure 7 - MFP method construction

data assignments indicated by the red arrowed lines.

The iteration visual uses a very compact visual cue (a small asterisk) to indicate the most common type of traversal (forward end-to-end by one). This conveys in a tiny space what in source code would consume an entire line of text. More important, perhaps, is the cognitive burden of the verbose textual representation: a programmer must mentally scan source text consisting of 10 tokens or so to conclude that a loop indeed represents an iteration. Moreover, it is possible to create an iteration with a single action, whereas in a textual form many errors are possible due to the complexity of the representation.

Unfortunately, due the limitations of printed text, it is difficult to discuss the dynamics of this figure. It should be noted that through simple graphical gestures, it is easy to traverse the frame hierarchy, to make data assignments, to produce or remove variables, and much more. Further, through the use of a sophisticated underlying program model, it is very easy to make changes that accurately permeate the entire program, include operations such as renaming variables or type changes.

3.3 MFP - Additional Features

3.3.1 Partial Evaluation

The snapshots in Figure 8 illustrate a user interface for partial evaluation functionality, seamlessly incorporated into the code editing environment. The first snapshot shows the result after selecting "Instantiate" from the context (right-mouse) menu of an if-then-else visual. The pane below the if-then-else is an inspector, which collects all of the free variables in the if-then-else construct, and offers an editable value field for each. In this case, there is only 1 free variable, p1, defined in an outer scope.

The value shown in the edit field, 0, is an initial value taken from the "default value" for double-precision floats (the declared type of p1). After typing 3 and hitting return, the "Replace" function is activated, causing the if-then-else to be evaluated (constant-folded and simplified) and replaced by the resulting expression, which is simply 6.0. Hitting the "Cancel" button restores the target expression to its original state, that of the if-then-else construct.

Any valid expression can be entered as the value of a free variable (not merely literal expressions), and will be evaluated in the

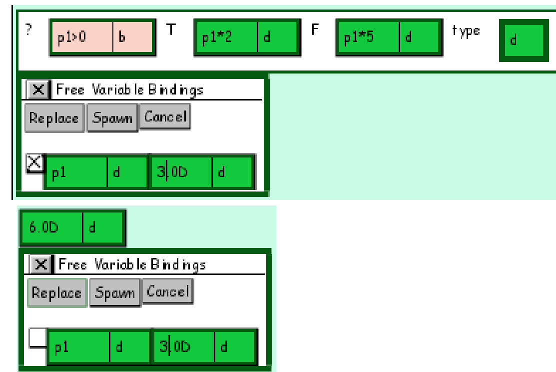

Figure 8 - User interface components for partial evaluation

context of the given construct. For example, one can substitute the expression `f(p2-5)` for `p1`, obtaining an if-then-else expression for the result, rather than a literal value. Such evaluation is available for all constructs, including loops, array operations, and so on. Moreover, certain algebraic simplifications are performed, such as x-x=0. It is our belief that this set of capabilities is very useful in exploring the boundary between the abstract and concrete. More details are provided in the section "Support for Partial Evaluation and Abstract Interpretation."

3.3.2 Design Pattern Interface

The snapshots in Figure 9 depict a simple experimental user interface for design patterns, which has been implemented roughly as a composition of the user interface elements for Class, Type,

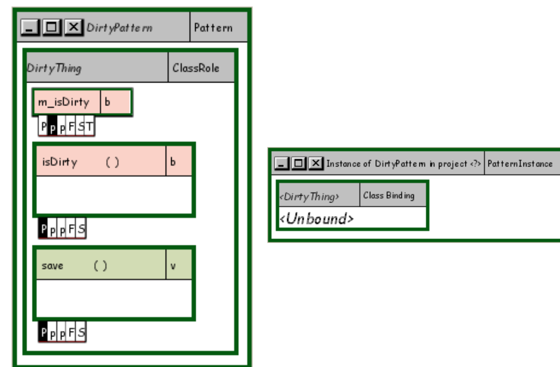

Figure 9 - User Interface components for design patterns

and Expression. A single pattern, `DirtyPattern`, is shown, containing a single class role, `DirtyThing`, which in turn has 1 data member, `m_isDirty`, and 2 methods, `isDirty()` and

`save()`.[1] Below the pattern is an instance of that pattern, containing a role instance for `DirtyThing`, presently unbound.

Interacting with these components is straightforward. To create a role in an existing pattern, simply drop a type, method signature, or an expression on the pattern view. To add, delete, or change the members in a ClassRole, the interactions are the same as for an ordinary Class view. To bind a class to the role instance, a class is dropped onto the role instance

4. MFP Infrastructure

In this section, the infrastructure that enables the MindFrames environment's differentiating features is described.

4.1 The Semantic Programming Model

Underlying all of the programming visuals in the MindFrames environment is a set of objects that model the program under construction. To first order, the model is that of a generic object-oriented language, with objects describing program entities such as packages, classes, methods, expressions, object references, and so on. However, it does not limit itself to strictly object-oriented structure, and so is able to represent compilation units, free-standing functions, lambda functions as first-class values, closures, and other higher-order concepts. The model also provides support for design patterns (manifested as a form of generalized template with constraints), evaluation, and partial evaluation.

The model has the following core features:

- Support for interacting with the editing environment
- Extensibility
- Semantically-aware manipulation operations
- Program analysis to support the above

For editing support, notifications are broadcast on changes to relevant objects, e.g., from a bound (possibly) symbolic reference to its defining entity upon a type change. These notifications permit the model to update the derived type of an expression as changes are made to variables, fields, or member function signatures. Also, each model object maintains an "edit state" property which reflects the presence of various kinds of errors, e.g. type errors, dangling references, write access to a read-only entity, and so on. This state is automatically updated as changes are made to a model object's structure (e.g. by setting the left-hand side of a binary expression) or to things upon which it depends. The edit state is made accessible to clients to be used in composing higher-level constructs from lower-level ones, and for visualization in the user interface.

The model is designed to be extensible, and beyond the existence of a very simple base class (SemanticObject), no set of programming entities are distinguished from any others. As a result, any application or domain can define and create additional program entities (types, code, or meta-level structures) that stand on equal footing with the constructs provided in the MindFrames base. Using this facility, higher-level concepts such as state machines, physical simulation building blocks, and the like, can all be defined for use by the program under construction. As a simple example, we have implemented a simple set of bit-field manipulation operations (such as concatenation and extraction) in this manner, layered on top of the more basic primitives.

Semantically-aware operations are the norm in the MindFrames environment, which clearly places demands on the basic operations provided by model entities. The operations fall into two categories: semantic-preserving and non-semantic preserving. The operations are given this semantic awareness in two ways: (1) by the basic structure of the model (e.g. the use of direct, non-symbolic references to various entities whenever possible), and (2) by engineering the operation to "consciously" respect or affect the semantics in specific ways. The net effect of this infrastructural capability is that, unlike the case with textual source code editors, refactoring is "always on".

In support of the refactoring operations, relevant program analysis is performed incrementally, lazily and directly on the model, rather than whole-sale and over basic blocks or other intermediate representations, roughly as done in [8]. At the moment, the analysis consists of a set of flow-sensitive data flow analyses, such as use-def chains and basic "points-to" analysis, all using a common infrastructure based on the use of work-lists to implement the fixed-point computations. The infrastructure factors out the common structure of the various analyses, such as the interface for defining the flow equations for each construct, the interface for defining the results of the computation (expressed as a set of objects), the traversal of the model, and the top-level fixed-point algorithm. To fit within this framework, each model construct is responsible for implementing certain basic analysis operations, such as initialization of the result set, definition of the generate, kill, and propagate flow equation for that kind of construct, and the constructs on which the results depend.

Finally, in support of state reflection, certain "facts" are computed, as a by-product of program analysis, that identify important conditions about the data state at the various points of execution. A fact is represented as a predicate/truth-value pair. Predicates include facts such as "this variable is uninitialized at this point", or "variable A and variable B point to this object." Three-valued logic is used to represent a fact's truth value, so that "unknown" is a valid value for any given fact. This feature is especially important, since many useful forms of predicates are undecidable in general. The model associates a possibly-empty set of facts with each execution instant, where an execution instant is defined as an instance of a state-altering operation. Since a given fact may be true across several execution instants, facts can be shared across instants.

The information encoded in a fact can be presented by visuals in various ways; for example, an "uninitialized" fact concerning a particular variable may manifest as a highlighted border in a particular style around the visual for that variable. Likewise, a "points-to" fact is typically represented by a directed arc from a reference variable to the quantity to which it refers.

Facts are computed for one of two possible reasons. First, the analysis can choose to always compute certain kinds of facts, such as the initialization state of variables and fields. Second, a client can explicitly query the model for the value of a particular predicate at a particular instant, which is either retrieved if already known (and cached), or computed on demand otherwise. Much work remains to be done to extend and refine the set of

[1] See the section entitled "Design Pattern Support" for a detailed description of the MindFrames model for design patterns.

expressible facts, and to determine the most effective computation strategy for generating them.

4.2 Design Pattern Support

A design pattern is defined in the MindFrames model to be a set of roles, each of which has a kind, an arity, and a set of zero or more constraints. To use a design pattern in MindFrames, the pattern is instantiated create a pattern instance. For each role in the pattern, there is a corresponding binding in the pattern instance which associates the role with one or more entities in the program. The role's kind, arity and constraints constrain the entities which may be bound into the role.

The role kind restricts the sort of entities that can be bound into an instance of the given pattern. A role kind is one of type, code, or value. Each kind requires more specific additional information relevant to that kind of role. For example, a type role is one that can be bound only to types, and requires a base type that constrains the set of types that can be bound into that type role. If the type role happens to be a class role, it may define data members and methods. A code kind can be bound only to functions or methods, and has a signature that defines its parameter and return types. Finally, a value kind can be bound only to an expression (i.e., any computation that results in a value, whether it is statically evaluatable or not). Value roles also have a base type that constrains the type of value to which they can be bound.

Each role in the original pattern must be bound to one or more entities in the program; a type role must be bound to a specific type (which may in the case of a class role be an abstract class). A class role's data and method members are inserted into any class bound to the corresponding role instance when the pattern gets instantiated. If the class role has no members, then presumably the base class for this class role defines members used in other parts of the pattern, say, in some other class role that does define method members.

Because of the direct and live linkage between referencing objects and the objects to which they refer, changing a class role's characteristics has an immediate effect on any classes bound to that role. For example, removing a data member from the class role will result in the deletion of this member from all classes bound to that role.

In this framework, it is easy to define patterns corresponding to all of the classical design patterns from [3]: singleton, factory, proxy, visitor, and so on, as well as other, perhaps more domain-specific patterns that have yet to be catalogued.

Note that, unlike other environments, MindFrames maintains patterns and pattern instances as first-class entities that do not suffer from the classic problem of "disappearing into the code". That is, in those IDE's that provide even wizard-based support for using design patterns, the norm is that, once the pattern has been applied to the source code, the only way to tell that the pattern is still there is to inspect the source code of all classes, and attempt to determine whether some pattern is in use. In MindFrames, all patterns and pattern instances are explicitly represented, so that their use is obvious. Moreover, pattern instances have appropriate portions (e.g. members originating in a class role) locked , so that the pattern's implementation cannot be unintentionally corrupted by stray edits.

4.3 Support for Partial Evaluation and Abstract Interpretation

The model supports partial evaluation, i.e., the specialization of a given code construct under a set of one or more bindings of free variables, and a currently very limited form of abstract interpretation. Partial evaluation is accomplished by the use of one of the following pair of API functions defined on the Expression class :

- instantiate()
- instantiateWith(Bindings)

The purpose of instantiate() is to non-destructively return a distinct Expression of identical structure, replacing all free variable references with literal values. The net effect of this operation is to "concretize" the given Expression, and, in concert with evaluation, can aid in understanding the behavior of a piece of code.

instantiateWith() is similar to instantiate(), but accepts an explicit set of bindings for the free variables in the given Expression. Each binding is also in the form of an Expression, which is substituted for each appearance of the corresponding free variable.

Abstract interpretation takes the form of the following calls, also defined on the Expression class:

- evaluate()
- simplify()

evaluate() performs constant folding as much as is possible and returns the resulting Expression. This is also a non-destructive operation that always returns a distinct Expression structure. In general, the result will not be a literal, but rather some form of Expression.

simplify() is a simple-minded implementation of an abstract interpretation which takes advantage of several of the most basic laws of algebra and Boolean operators, e.g. x+0=x, x*1=x, x-x=0, x OR 0 = x, x AND false = false, x XOR x = false, and so on. At present, no attempt is made to rearrange the expression using the commutative or distributive nature of the various algebraic operators in order to achieve the above simplifications, but in principle such things can be readily done.

The purpose behind all of these operations is to provide the programmer with a toolkit for exploring the boundary between the abstract specifications that constitute the program, and the specific instances of concrete invocation that are often more easily understandable.

4.4 Visual Framework

The MindFrames visual environment imposes demanding requirements on its software design. The main challenges are:

Visual Independence: The environment should freely accommodate new visuals with unique behavior. At the same time, there needs to be sufficient framework or grounding in rules and protocols to avoid chaotic interaction or behavior. This is true of the environment itself as well as across visuals. Further, the environment and the visuals should not hold direct references to each other by explicit type.

Visual Relationships: The environment should allow for the visual representation of relationships among visuals, independent of their visual type. For example, binary relationships, such as

data assignments, have sources and targets anchored on other visuals. However, the source and target visuals can be of arbitrary type, or nested inside arbitrary visuals. As a concrete example, consider representing assignment to an array slot (`a[i] = x`); the target visual (`a[i]`) is deeply nested within the array visual.

Restoration: The environment requires a mechanism for reconstructing the visuals comprising a view in a visual independent manner. For performance reasons, it must take advantage of the fact that not all visuals are visible at all times.

Reactive Update: Since a visual may represent one or more model objects, it must react to relevant model changes. On the other hand, a model object can be changed by more than one visual. Also, visuals may need to coordinate updates of non-functional (model-independent) visual attributes. For example, several live miniatures may show the same frame at the same time. These live miniatures thus need to coordinate updates for model-independent properties, such as hiding, positioning and colors.

4.5 The MindFrames Visual System

A primary goal in designing the MindFrames prototype was to ensure visual type independence. We allowed visual definitions and their implementations to be dynamically loaded. However, we wanted to avoid visual type dependencies both in the system and across visuals themselves. This posed challenging questions regarding visual identity, restoration, defining and maintaining visual relationships, and visual interaction.

In part, the solution is to use a standard MVC paradigm with the program model as the reference model for the system. Visuals typically reference one or more model objects, and base their visual characteristics and behavior on the model's state. The basic interactive cycle is well defined and is depicted in the Figure 10:

- Visuals change model objects by direct user interaction
- Model object changes may result in changes to other model objects
- Changed model objects notify referencing visuals to update visual state

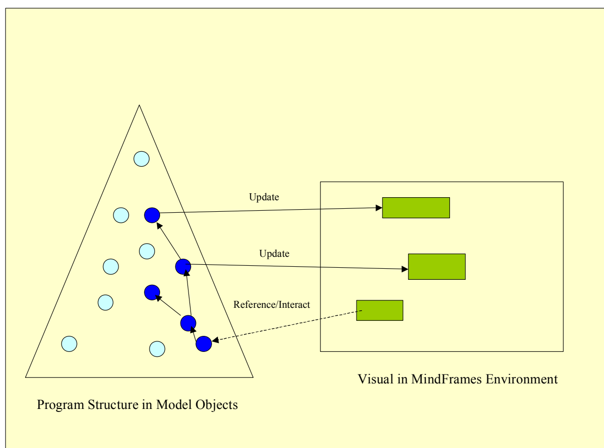

Figure 10 - Visual update

The MVC approach provides an added benefit, namely, that the model is the "language" for communication at the visual level. We found, for example, that using model objects as the drag-and-drop (DnD) payload served as a consistent and simple means for interaction among visuals. Additionally, the system defines a default visual type to represent each model object type, so that an appropriate visual can be constructed for any given model object. Thus, visuals embody, react to, and interact with the lingua franca of model objects, providing visual type independence.

There remained issues concerning visual restoration, as well as the thorny issues of how visual relationships, e.g. pointers, could exist in the presence of visual type independence. Our system defines a base class called VisualRestoreObject (or VRO), which is sub-classed for each visual class, and which holds all information needed to restore an instance of the visual class. It also identifies the visual class, allowing the VRO to instantiate the visual, allowing for overriding the default visual representation. The VRO is in essence a proxy for the visual. VROs may also reference other VROs, as means to facilitate population of a complex visual with other visuals. See Figure 11.

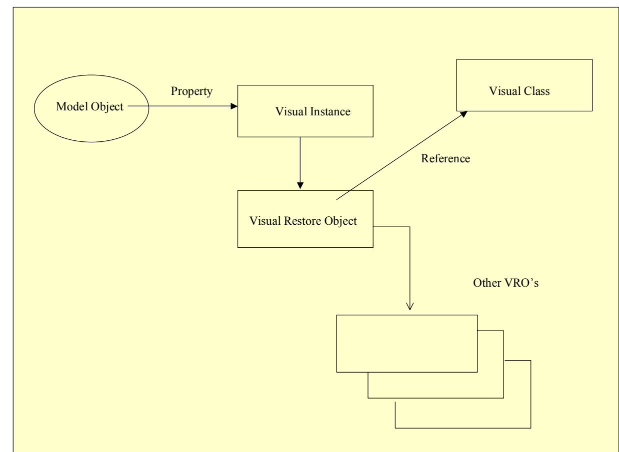

Figure 11 - VRO structure

The model objects used in the context of a given visual presentation each point to a visual instance. For this, we add a property to the model object's "property bag" (a dictionary of key/value pairs). Properties can have values of arbitrary type, and can be persistent or transient, allowing a wide variety of uses.

The VRO concept was valuable in designing relationship visuals. In this case, the relationship maintains a list of the related VROs. This mechanism was vital to visual type independence.

Finally, the VRO mechanism was valuable in maintaining visual consistency when several instances of the same visual or view appear. This occurs, e.g., when a live miniature frame and the current frame are the same. The VRO maintains a list of concurrent visual instances derived from it. In reaction to various types of updates, the VRO publishes an update notification to the other visuals. In this way, for example, when a visual position changes or resizes, all sibling visuals update appropriately.

4.6 Graphics Foundations

It is known that highly effective user interfaces assist the user in maintaining "flow state", in which the flow of creative thought is unbroken, because tools behave as a seamless extension of the user's intentions. This higher-level thread of activity is easily broken by latency, gratuitous modality and other interface flaws.

Thus, to fully leverage the programmer's focus on the semantic level, the graphics framework must provide fluid, amodal interactions, and support the many unique kinds of visuals our system uses.

Examples of the kinds of interactions permeating the MindFrames environment that are a challenge to the graphics toolkit include full-detail dynamic-update move, resize and drag-and-drop operations. In all of these, full detail is superior to wire-frame representation, which obfuscates an operation's effect. Likewise, the common use of complex, large, nested program elements requires the graphics framework to make dragged components translucent, so that the drop context can be seen.

The impacts of these design goals are several. Clearly, in such an environment, the graphics framework's performance in several areas is a critical component of system responsiveness. Unfortunately, investigation into many graphics systems found none to meet our needs; hence, we built the MindFrames graphics toolkit directly on top of the Java2D library. Our toolkit relies heavily on Java2D's 2D transforms and alpha blending, creating performance bottlenecks where Java2D renders slowly. To solve this problem we designed a unique caching system to minimize the amount of work done to refresh the screen.

Although we have not compiled performance data, as a subjective indication of the MindFrames' toolkit performance, the graphics system was very responsive and smooth with several hundred components on a 600MHz Windows 2000 machine.

5. RELATED WORK

Program visualization is traditionally related to dynamic program animations [1]. While this type of visualization could add value to our environment, the key problem we address is program semantic visualization.

Visual representations of data types are found in [5], applied to a data-flow paradigm. In contrast, our visualizations of data are operational, rather than merely declarative, and thus participate in semantic interactions during construction of program behavior.

A seamless environment supporting execution, visualization, and multiple views can be found in [4]. Our work developed along many of the same lines, but with a stronger focus on the construction of detailed behavior, and the semantics of program operations, which affect editing manipulations. Further, MindFrames offers an extended notion of program execution and exploration through partial evaluation and abstract interpretation.

MindFrames is a significant extension of many ideas found in [9].

6. CONCLUSIONS AND FUTURE WORK

Designing and implementing a multi-domain-specific programming environment embodying the many ideas mentioned earlier is a daunting task. The challenges derive from the need for extensibility without crippling semantic interoperability. In particular, the visual system supports type-independent composition, while permitting very different visuals to interact in a well-defined manner. The program model provides complete information about a program's structure and semantics in a largely language-independent manner. It provides for manipulation at both high and low levels, along with a notification system rich enough to express a wide variety of changes and editing states. The model also provides support for moving across the boundary between abstraction and concreteness.

Thus far, we have produced a functional prototype demonstrating many of our goals. Further development would extend the model's semantic manipulation and refactoring abilities, along with corresponding presentation and manipulation support in the visual domain. Causal dependencies should be better exposed, taking advantage of data-flow and control-flow analyses. Finally, the visual framework should be enhanced to more easily define new visuals and the semantic connections between them and the model.

Another open question concerns ease of development of new visuals. It is highly desirable for visual representations to be as composable as the model objects that they represent. For example, if the model of an expression permits operands of *any* expression type, it should be similarly easy to use expression visuals of *any* type in creating the composite expression visual. Because the widget class hierarchy generally does not mirror that of the model; however, such interoperability does not come for free.

We believe that with refinement, MindFrames will contribute significant progress to state-of-the-art software development.

7. ACKNOWLEDGMENTS

This work has been supported in part by the Defense Advanced Research Projects Agency (DARPA) under contract No. NBCHC020056. The views expressed herein are not necessarily those of DARPA or IBM.

8. REFERENCES

[1] Bazik, J., Tamassia, R., Reiss, S.P., van Dam, A., "Software Visualization in Teaching at Brown University", Software Visualization, MIT Press, 1998, pp, 383-398.

[2] Eclipse Platform Technical Overview, http://www.eclipse.org

[3] Gamma, E., Helm, R., Johnson, R., Vlissides, J. Design Patterns, Addison-Wesley, 1995.

[4] Grundy, J. and Hosking, J., "Connecting the Pieces", Visual Object-Oriented Programming, Manning Publications, 1995, pp. 229-252.

[5] Ibrahim, B., "Diagrammatic Representation of Data Types and Data Manipulations in a Combined Data- and Control-Flow Language", 1998 IEEE Symposium on Visual Languages, Halifax, Canada, pp. 262-269.

[6] IntelliJ IDEA Overview, http://www.intellij.com/docs

[7] Kurlander, D., "Graphical Editing by Example", PhD Thesis, Columbia University, NY. 1993.

[8] Morgenthaler, J., "Static Analysis for a Software Transformation Tool", PhD Thesis, University of California at San Diego, 1997.

[9] Pazel, D.P., "The Effigy Project – Moving Programming Concepts to a Visual Paradigm, The Visual End User Workshop at VL2000, Seattle, 2000.