Using Model Dataflow Graphs to Reduce the Storage Requirements of Constraints

BRADLEY T. VANDER ZANDEN and RICHARD HALTERMAN
University of Tennessee

Dataflow constraints allow programmers to easily specify relationships among application objects in a natural, declarative manner. Most constraint solvers represent these dataflow relationships as directed edges in a dataflow graph. Unfortunately, dataflow graphs require a great deal of storage. Consequently, an application with a large number of constraints can get pushed into virtual memory, and performance degrades in interactive applications. Our solution is based on the observation that objects derived from the same class use the same constraints and thus have the same dataflow graphs. We represent the common dataflow patterns in a model dataflow graph that is stored with the class. Instance objects may derive explicit dependencies from this graph when the dependencies are needed. Model dependencies provide a useful new mechanism for improving the storage efficiency of dataflow constraint systems, especially when a large number of constrained objects must be managed.

Categories and Subject Descriptors: D.2.2 [**Software Engineering**]: Design Tools and Techniques—*User interfaces*; D.2.6 [**Software Engineering**]: Programming Environments—*Graphical environments, Interactive environments*; D.3.2 [**Programming Languages**]: Language Classifications—*Data-flow languages*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Constraints*; I.1.2 [**Computing Methodologies**]: Algorithms—*Nonalgebraic algorithms*; I.1.2 [**Computing Methodologies**]: Languages and Systems—*Evaluation strategies*

General Terms: Algorithms, Experimentation, Performance

Additional Key Words and Phrases: Language design and implementation, Programming environments, Dataflow constraints, Prototype-instance model, Class-instance model, Storage optimization, Graphical interfaces

This research was supported in part by the National Science Foundation under grants CCR-9633624 and CCR-9970958.

Name: Brad Vander Zanden
Affiliation: Computer Science Department, University of Tennessee
Address: Knoxville, TN 37996-3450; email: bvz@cs.utk.edu
Name: Richard Halterman
Affiliation: School of Computing, Southern Adventist University
Address: P.O. Box 370, Collegedale, TN 37315; email: haltermn@cs.southern.edu

1. INTRODUCTION

Constraints can reduce the complexity of developing highly interactive, graphical applications by allowing a programmer to declaratively express a relationship among objects and then having a constraint solver automatically maintain the relationship. For example, a programmer might center a label within a textbox using the constraint:

$$label.left = textbox.left + textbox.width/2 - label.width/2$$

Whenever any of the variables `textbox.left`, `textbox.width`, or `label.width` change, the constraint will be automatically re-evaluated and the new result assigned to `label.left`. Hence if the textbox is moved or resized, or the label is edited, the label will remain centered within the textbox.

The most commonly used constraints in user interface toolkits are *one-way, dataflow* constraints, of the type shown above. A dataflow constraint (also called a one-way or a spreadsheet-style constraint) is an equation of the form $v = f(p_1, \ldots, p_n)$, where f is an arbitrary function, p_1 through p_n are the parameters to this function, and v is the variable to which the function's result is assigned. The function is re-evaluated each time one of its arguments changes, and the result is assigned to v.

Such constraints are called dataflow constraints because data flows from the variables on the right side of the equation to the variable on the left side of the equation. The constraints are called one-way because it is not permissable to invert the equation and solve for one of the right side variables. If v is changed by the user or by the application, the constraint is left temporarily unsatisfied until one of the p_i's is changed.

The reason for the popularity of one-way, dataflow constraints is severalfold:

(1) They are simple for programmers to learn and use.

(2) Their results are predictable (more complicated constraint systems may have multiple admissable solutions and if the constraint solver chooses one that the user is not expecting, then the user may be confused), and

(3) They support both numeric and non-numeric constraints. One study showed that 36% of the constraints defined in a set of graphical applications were non-numeric, indicating that non-numeric relationships are an integral part of many such applications [Vander Zanden et al.].

Because of their appeal, one-way constraints have been integrated into a variety of drawing packages and interface development toolkits [Myers et al. 1990; Myers et al. 1997; Hudson ; Hill 1993; Hudson and Smith 1996; Barth 1986; Alpert 1993; Hudson 1994; Hudson and Mohamed 1990]. Unfortunately, studies of at least two of these toolkits, Garnet and Amulet, have shown that constraints can exact a significant storage toll on programs [Vander Zanden and Venckus 1996]. Ultimately the execution times of programs managing a large number of constrained objects may suffer since virtual memory must be accessed to meet their storage demands.

One reason dataflow constraints require so much storage is that constraint solvers explicitly represent the relationships among variables and constraints by maintaining a *dataflow graph* (see Section 2). As shown by Table 1, the edges in these graphs

Table 1. Number of bytes required by a dependency in a variety of constraint systems.

System	Bytes Per Dependency
Garnet [Myers et al. 1990]	16
Amulet [Myers et al. 1997]	24
Eval/vite [Hudson]	24
Rendezvous [Hill 1993]	16

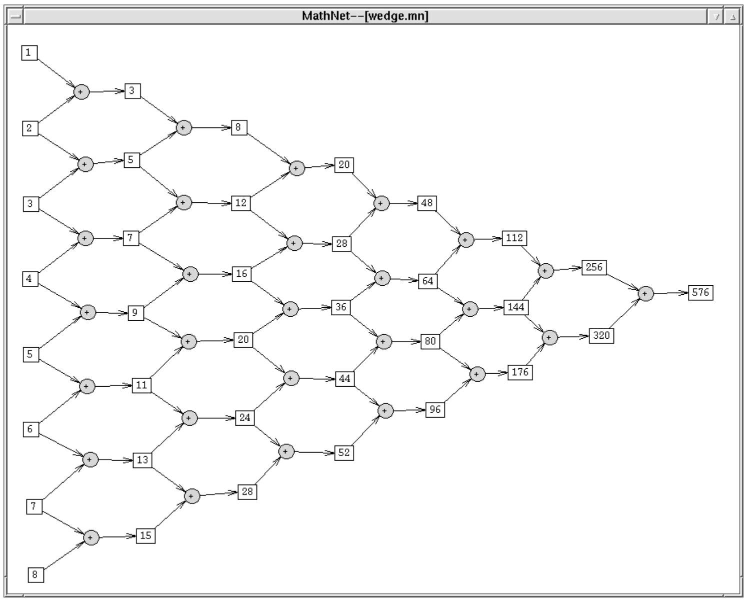


Fig. 1. An application that uses many labeled boxes.

can consume a considerable amount of storage. One study has shown that dataflow graphs account for up to 50% of the storage required by constraint systems [Vander Zanden and Venckus 1996]. Reducing the storage cost of these dataflow graphs is therefore an important goal for constraint researchers.

In this paper we present a solution to the dataflow graph problem that is based on the observation that objects that use the same constraints have the same dataflow graph. Consequently, one can store a pattern of a dataflow graph, a *model dataflow graph*, in a common place and then use the pattern to derive explicit dependencies on demand. Since thousands of objects may be created from the same prototypical object, the storage savings can be considerable. For example, Figure 1 shows a visual arithmetic editor that uses labeled box objects for operands and labeled circle objects for operators. All the operand objects can share the same model dataflow graph. Similarly the operator objects can all share the same model dataflow graph. Thus a considerable storage savings can be realized in a large application that manipulates thousands of these objects.

Our solution is inspired by the Reps *et. al.* idea of using supertree-subtree graphs to implicitly represent a dataflow graph [Reps et al. 1983]. However, Reps dealt with attribute grammars which give rise to restricted types of dataflow graphs and restricted types of edits to these dataflow graphs. In contrast, our problem deals with graphical interfaces which give rise to arbitrary dataflow graphs and arbitrary edits. One of the important findings of this paper is that a simple model dependency scheme of the type presented by Reps does not work well with arbitrary dataflow graphs. Our initial implementation was only able to model 20% of the explicit dependencies using a scheme based on his ideas. When we performed an in-depth analysis of constraints in actual graphical applications, we were able to identify a number of common situations that, when integrated into the model dependency scheme, greatly increased the number of explicit dependencies that could be eliminated. Our experiments show that over 75% of the explicit dependencies in most applications can be eliminated with this more sophisticated scheme. Consequently, while our solution incorporates the Reps, *et. al.* idea of supertree-subtree graphs to reduce the number of explicit dependencies, we have significantly extended it to work in the environment of graphical interfaces.

2. BACKGROUND

This section describes the key components used in this paper, including the object model, composite objects, and dataflow graphs. These three components are all typically present in a toolkit that provides one-way constraints.

2.1 Object Model

The approach described in this paper can be used with either the class-instance or prototype-instance object models. Consequently, we will present a generic object model.

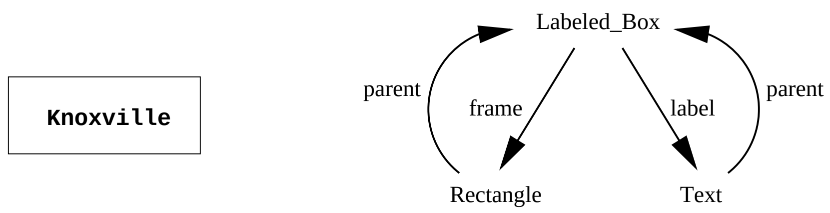
Our object model assumes that there is a template object (a class in the class-instance model and a prototype in the prototype-instance model) from which instance objects can be created. An object's properties, such as its position, size, and color, are specified by slot/value pairs (a slot would be called an *instance variable* in a class-instance model). All slots associated with the template object are inherited by the new instance unless the slot's value is overridden in the instance.

Constraints are inherited just like slots. For example, if the template's `color` slot is computed by a constraint, then the `color` slot of an instance will also be computed by a constraint, unless the programmer provides an alternative value.

2.2 Composite Objects

A *composite object* is an object made up of parts consisting of other objects [Gamma et al. 1995]. For example, Figure 2 illustrates a simple composite object, a labeled box consisting of a text string enclosed within a rectangle.

The labeled box has named pointers to its children (`frame` and `label`) and the children have named pointers to their parent (`parent`). (See Figure 2b.) These pointers allow the labeled box to access slots in its parts and the parts to access slots in their parent and in their siblings. The notion of inheritance can be extended to composite objects in the sense that when an instance of a composite object is created, instances of all its parts are created as well. This type of inheritance is called *structural inheritance* [Myers et al. 1990].



(a) A labeled box (b) The labeled box composition diagram

Fig. 2. A labeled box object (a) and its structural components (b)

label:

--Width depends on the label's text string and the display font

```
C1  self.width = self.compute_width(self.text,
                                    self.font)
```

--Label is centered horizontally within its owner

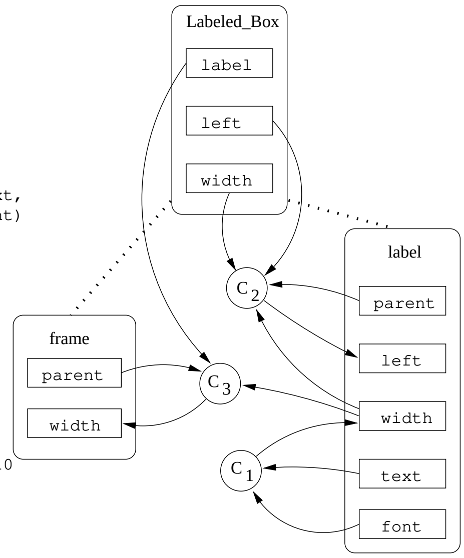
```
C2  self.left = self.parent.left
                + self.parent.width/2
                - self.width/2
```

frame:

--Frame is slightly wider than the label it encloses

```
C3  self.width = self.parent.label.width + 10
```

(a) Constraint equations



(b) Constraint graph

Fig. 3. Constraints that lay out the elements of the labeled box in Figure 2 (a) and the dataflow graph that they generate (b). The constraints make the frame slightly larger than the text label, and center the label within the frame.

2.3 Dataflow Graphs

A one-way constraint solver typically uses a bipartite, *dataflow graph* to keep track of dependencies among variables and constraints (Figure 3). Variables and constraints comprise the two sets of vertices for the graph. There is a directed edge from a variable to a constraint if the constraint uses that variable as a parameter. There is a directed edge from a constraint to a variable if the constraint assigns a value to that variable. Formally, the dataflow graph can be represented as G = {V, C, E}, where V represents the set of variables, C represents the set of constraints, and E represents the set of edges. The edges are often called *dependency* edges because they show how variables and constraints depend on one another. For each edge from an input variable to a constraint many systems also maintain a backward edge that points from the constraint to the variable. These backward edges allow the dependency edges to be removed if a constraint is deleted or allow the dataflow graph to be traversed in the reverse direction.

A constraint solver uses the dataflow graph to locate the constraints that must be resatisfied when a variable is changed by either the user or the application. The algorithms the constraint solver uses for resatisfying constraints are described in Section 2.5.

The dataflow graph can be automatically generated as constraints are being evaluated using the following technique. Before a constraint is evaluated it pushes itself onto a global constraint stack and after it is evaluated it pops itself off the stack. Pushing and popping the constraint in this manner ensures that if the constraint requests any slots, then the constraint will be the top constraint on the stack [Vander Zanden et al. 1991; Hoover 1992; Vander Zanden et al. 1994]. Whenever a slot's value is requested, the stack is checked to see if it is non-empty. If the stack is non-empty, then an explicit dependency from the slot to the stack's topmost constraint is generated by adding the constraint to the slot's dependencies list.

2.4 Basic Data Types

The previous three sections have described the features of a typical constraint system. This section describes the object and constraint data structures that can be used to implement such a constraint system. These data structures provide the basis for the implementation of the model dependency scheme described in this paper. The data structures are presented as a set of classes and the algorithms are presented as a set of methods, although Algol-like pseudocode, rather than a conventional language like C++, is used to enhance readability.

Four primary classes are used to implement the object and constraint model:

(1) **Object**. This class implements the objects used by application programmers. Table 2 describes the `Object` fields and methods.

(2) **Slot**. This class implements the slots used by objects. It contains a `value` field and fields that support constraint satisfaction. A `Slot's` fields and methods are described in Table 3.

(3) **Formula**. This class stores a formula defined by the application programmer. It includes a pointer to a function that computes the formula's value, and, if the formula creates model dependencies, a pointer to a parameter list.

Table 2. Definition of `Object` fields and methods used by the algorithms in Section 4. The boldface items provide support for model dependencies. Part names and slot names are implemented as named integer constants.

`Object` Fields and Methods	
Field	Meaning
`parts`	The set of objects that are children of this composite object; a part named `p` can be accessed via the notation `parts[p]`.
`slots`	The set of slots for this object; a slot named `s` can be accessed via the notation `slots[s]`.
`parent`	A pointer to the object's parent.
`part_name`	The name of this object, if it is a part. For example, the `part_name` for the rectangle in Figure 2.b would be `frame`.
model_subtree	A pointer to the model dataflow graph storing this object's `SELF` and `CHILD` model dependency edges.
model_supertree	A pointer to the set of model supertree graphs for each of the object's parts. A part's model supertree can be accessed via the notation `model_supertree[part_name]`.
common_model_supertree	A pointer to a set of model dependency edges shared by all children (see Section 5.2).
Method	Meaning
create_model_dataflow_graph()	Creates a model dataflow graph for the object (see Figure 12).
inherit_constraints()	Inherits constraints from the object's template (see Figure 14).

Table 3. Definition of `Slot` fields and methods used by the algorithms in Section 4. The boldface items provide support for model dependencies.

`Slot` Fields and Methods	
Field	Meaning
`value`	The value of the slot.
`object`	The object to which this slot belongs.
`name`	The slot's name.
`valid`	True, if the slot's value is up to date; otherwise, false.
`explicit_dependencies`	A set of pointers to constraints that depend on this slot; these pointers constitute the slot's *explicit* dependencies.
`constraints`	The set of constraints that can be used to compute the value of this slot.
Method	Meaning
`get()`	Returns the slot's value; if necessary, get will first bring the slot's value up-to-date by evaluating any out-of-date constraints.
`set()`	Sets the slot's value and invalidates all constraints and slots that depend either directly or indirectly on the slot.
`invalidate()`	Marks the slot invalid and each dependent constraint invalid (see Figure 15).
get_model_dependencies()	Returns the slot's model dependencies (used by the algorithm in Figure 15).

Table 4. Definition of `Formula` fields used by the algorithms in Section 4. The `model_parameters` list provides support for model dependencies.

`Formula` Fields and Methods	
Field	Meaning
`function`	A pointer to the function that computes this formula's value.
model_parameters	The formula's set of model parameters.

Table 5. Definition of `Constraint` fields and methods used by the algorithms in Section 4. The boldface items provide support for model dependencies.

`Constraint` Fields and Methods	
Field	Meaning
`valid`	True, if the constraint is up to date; otherwise, false. A slot may have multiple constraints so even if a slot is marked invalid, it is still necessary for each constraint to have a valid field so the appropriate invalid constraint can be found and re-evaluated.
`slot`	A pointer to the slot computed by this constraint.
modeled	True, if the constraint was created from a formula with a non-empty set of model parameters; otherwise, false.
generate_dependencies	True, if the constraint should create explicit dependencies, even if it is modeled; otherwise, false (the initial, default value is true).
needs_parent	True, if one or more of the constraint's model parameters defines a path through the parent; otherwise, false.
`formula`	A pointer to the `formula` object that computes this constraints's value.
Method	Meaning
`invalidate()`	Marks the constraint invalid and the slot that the constraint computes invalid (see Figure 15).
`create_copy()`	Makes a copy of the constraint.
`evaluate()`	Calls the constraint's `formula` function.

A `Formula's` fields and methods are described in Table 4.

(4) **Constraint**. This class stores the constraint that gets created when a formula is assigned to a slot. A constraint object records whether the constraint's value is up-to-date and to which slot the constraint is attached. Constraint objects share formula information by pointing to the same formula object. Table 5 describes the relevant fields and methods associated with `Constraint` objects.

Figure 4 provides a concrete example illustrating the relationships among `Objects`, `Slots`, `Formulas`, and `Constraints`. This figure shows an abbreviated picture of the implementation of the labeled box from Figure 2.

2.5 A Constraint Satisfaction Algorithm

The model dependency scheme described in this paper works with the two most commonly used constraint satisfaction schemes—*mark-sweep* and *topological ordering*. It would be confusing however to try to present the model dependency scheme

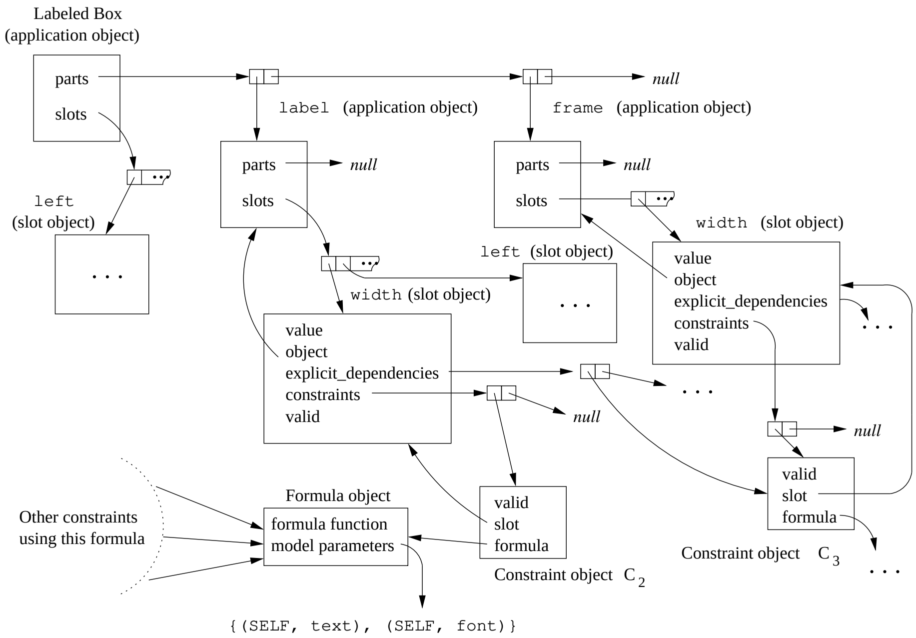


Fig. 4. The runtime data structures implementing a portion of the labeled box object. The composite object for the labeled box is drawn at the upper left corner of the diagram. It has two children in its **parts** list, a label and a frame. Neither the label nor the frame have any children of their own. The label's **width** slot shows that its value is determined by exactly one constraint, C_2, and that its value is used by constraint C_3 (one of the constraints in its **explicit_dependencies** list).

and simultaneously show how it works with both types of algorithms. We have therefore chosen to illustrate how it works with a mark-sweep algorithm. We have chosen a mark-sweep algorithm over a topological ordering algorithm because 1) it is the most widely used algorithm in user interface toolkits [Hill 1993; Myers et al. 1990; Myers et al. 1997; Hudson ; Hudson and Smith 1996; Hudson and King 1988], 2) it is the easier algorithm to implement, and 3) it is the most efficient, at least within the context of graphical interfaces [Hudson 1991; Vander Zanden et al.]. Although it will not be shown, the depth-first search that is presented in this paper's mark algorithms can be easily adapted to handle the depth-first search that a topological-ordering scheme uses to assign order numbers to constraints (an order number represents a constraint's position in topological order).

A mark-sweep algorithm has two phases:

(1) *mark* phase: This phase uses a depth-first traversal of the dataflow graph to find all the constraints that are reachable from a changed slot and marks them `invalid`. Figure 5 formalizes the mark procedure.

(2) *sweep phase*: This phase brings invalid constraints up-to-date by evaluating their formulas [Demers et al. 1981; Reps et al. 1983; Hudson 1991; Vander Zanden et al. 1994]. The sweep phase can use either *eager evaluation* or *lazy*

```
Method Slot.set(Value new_value)
self.value ← new_value
self.invalidate()
—self.invalidate sets the valid field to false so set it back to true
self.valid ← true

Method Constraint.invalidate()
if self.valid = true :
    self.valid ← false
    self.slot.invalidate()

Method Slot.invalidate()
if self.valid = true :
    self.valid ← false
    for each constraint cn ∈ self.explicit_dependencies :
        cn.invalidate()
```

Fig. 5. Methods used to implement the mark portion of the mark-sweep algorithm. The keyword **self** is a pointer to the object on which the method is invoked.

```
Method Slot.get(Boolean create_dependency) returns Value
—If a dependency should be established or the top constraint on the constraint
—stack is flagged to generate a dependency, then establish a dependency
—to the top formula since it has requested this slot's value
if not constraint_stack.empty() :
    top_cn = constraint_stack.top()
    if create_dependency = true or top_cn.generate_dependencies = true :
        self.explicit_dependencies ← self.explicit_dependencies ∪ top_cn
—update the slot's value if any of its constraints are out-of-date. In this
—algorithm the last invalid constraint on the constraint list determines
—the slot's value. Other rules can be used just as easily.
if self.valid = false :
    self.valid ← true
    for each constraint cn ∈ self.constraints :
    if cn.valid = false :
        cn.valid = true
        self.value ← cn.evaluate(self.object)
return self.value

Method Constraint.evaluate(Object obj) returns Value
—Push this constraint on to the constraint stack so that a slot accessed by this
—constraint's formula function will know that this constraint requested its value
constraint_stack.push(self)
—obj is the object which owns the slot to which the constraint is attached.
result ← self.formula.function(obj)
—Remove this constraint from the stack
constraint_stack.pop()
return result
```

Fig. 6. Methods used to implement the sweep portion of the mark-sweep algorithm.

evaluation. The eager strategy re-evaluates constraints as soon as the mark phase is completed. The lazy strategy defers a constraint's re-evaluation until its value is requested. Figure 6 shows a lazy version of the sweep algorithm. The algorithm has two items of interest:

(a) The algorithm allows multiple constraints to be attached to a slot. Some systems, such as Rendezvous [Hill 1993] and Amulet [Myers et al. 1997], support this feature.

(b) The algorithm gracefully handles cycles in the dataflow graph. In particular, the algorithm uses a technique called once-around evaluation. Once-around evaluation means that a constraint in a cycle is evaluated at most once. If the constraint is asked to evaluate itself a second time it simply returns its original value.

To illustrate the lazy version of the mark-sweep algorithm, consider the constraint formulas and graph of Figure 3. Suppose the user edits a label in one of the boxes. This edit changes `label.text` and initiates the mark phase. The mark phase performs a depth-first traversal that invalidates 1) constraint C_1 and `label.width` (the label's width will be changed), 2) constraint C_3 and `frame.width` (the frame's width will be changed) and 3) constraint C_2 and `label.left` (the label will be re-centered).

Now assume the value of `frame.width` is requested. This request initiates the sweep phase. `frame.width` is invalid so C_3's formula is called. C_3's formula requests the value of `label.width`, which is out of date. Hence, `label.width`'s formula, C_1's is called, bringing `label.width` up to date. `label.width` returns its value to C_3 which finishes executing. Its updated value is assigned to `frame.width`, which returns this new value to the application. Note that `label.left` remains out of date because its value was not requested.

3. THE MODEL DEPENDENCY PARADIGM

This section provides an overview of the basic model dependency scheme. It begins by describing the types of dependencies supported by the basic scheme and how an algorithm can use these model dependencies to perform a depth-first search. It then describes how a formula generates model dependencies and how the scheme handles slots with multiple constraints. Finally it concludes with a discussion of how explicit and model dependencies can co-exist.

3.1 Types of Model Dependencies

The basic scheme allows four relationships to be expressed with model dependencies: `SELF`, `CHILD`, `PARENT`, and `SIBLING`. These four relationships were chosen after a survey of applications written in the Amulet user interface development toolkit [Myers et al. 1997] showed that over 50% of all constraint formulas use these four relationships exclusively. Figure 7 shows how these relationships are defined.

A model dependency consists of either a two-tuple or a three-tuple that represents a path in the composition hierarchy from a given slot to its dependent slot. Given a slot, an object, and a model dependency edge, a dependent slot can be located as shown in Figure 7. For example, if the model dependency (`SIBLING`, `C`, `y`) is stored

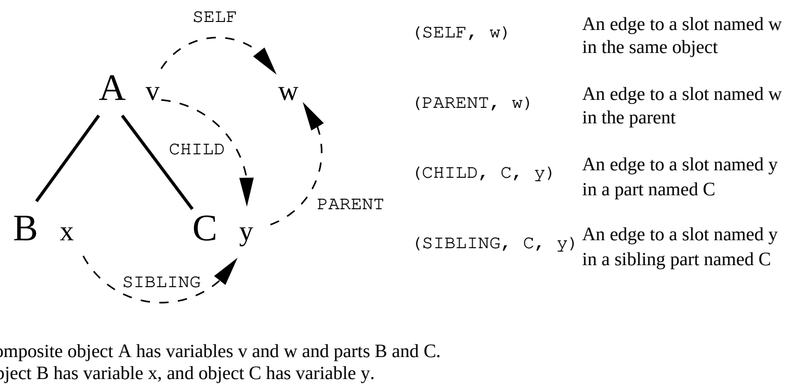


Fig. 7. Model edge relationships

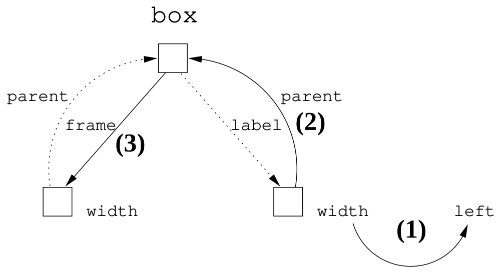


Fig. 8. Illustration of how the mark phase can use model dependencies to perform a depth first search. Suppose that the `label.width` slot is changed in a labeled box. `label.width` has the model dependency edges {(`SELF, left`), (`SIBLING, frame, width`)}, which are derived from the formulas defined for constraints C_2 and C_3. The `(SELF, left)` edge tells the mark algorithm to look in the same object for the `left` slot **(1)**. The `(SIBLING, frame, width)` edge tells the mark algorithm to locate the `width` slot by following the `parent` pointer **(2)** to the parent, and then the `frame` pointer to the sibling **(3)**. The mark algorithm invalidates the constraints that are found at the label's `left` slot and the sibling's `width` slot, which are C_2 and C_3 respectively.

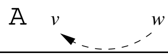
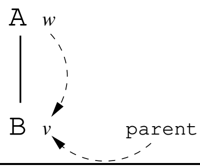
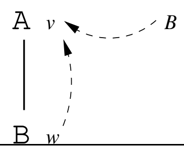
with the slot `B.x`, then the dependency defines an edge from `B.x` to `C.y`. Figure 8 illustrates how a depth-first search would use model dependencies to locate and invalidate constraints.

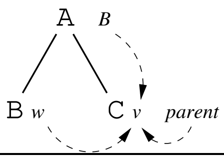
3.2 Model Parameters

In order for the model dependency scheme to work, it needs to know how to generate model dependencies for each modeled formula. The programmer provides this information by providing a list of *model parameters* for each formula. Each parameter specifies a path to one of the slots accessed by the formula. Like a model dependency, a parameter path consists of either a two-tuple or three-tuple and can specify a `SELF`, `PARENT`, `CHILD`, or `SIBLING` slot. For example, the list of parameters for constraint C_2's formula in Figure 3 is {(`PARENT, left`), (`PARENT, width`), (`SELF, width`)}.

A parameter edge can be automatically inverted to create a model dependency edge. For example, a `(SELF, width)` parameter can be inverted to create a `(SELF,`

Table 6. Model dependency edge installation for the four fundamental edges. Formula slot is the slot to which the formula is attached. Parameters are the parameter variables accessed by the parameter edge. Dependency edges (the dashed lines) are the model dependency edges generated from the parameter edge. Each dependency edge is associated with the parameter variable on the same line. For example, the dependency edge `(SELF, v)` would be attached to the parameter variable `A.w`.

Formula slot	Parameter edge	Parameters	Dependency edges	
`A.v`	`(SELF, w)`	`A.w`	`(SELF, v)`	
`B.v`	`(PARENT, w)`	`A.w` `B.parent`	`(CHILD, B, v)` `(SELF, v)`	
`A.v`	`(CHILD, B, w)`	`B.w` `A.B`	`(PARENT, v)` `(SELF, v)`	

`C.v`	`(SIBLING, B, w)`	`B.w`
		`C.parent`
		`A.B`
			`(SIBLING, C, v)`
			`(SELF, v)`
			`(CHILD, C, v)`
				

`left)` model dependency edge. Similarly, the `(PARENT, left)` parameter for constraint C_2 would be inverted to create a `(CHILD, label, left)` model dependency edge and a `(SELF, parent)` model dependency edge. Both dependencies are necessary because the label's `left` slot must be notified if 1) the `left` slot in its parent changes, or 2) if the parent is replaced (that is, the label is removed and placed in another composite object). In the latter case, the change of parents may mean that the new parent's slot has a different value and so C_2 must be re-evaluated.

Table 6 formalizes how each parameter is inverted to create one or more model dependency edges.

3.3 Formula Signature

The traversal process for model dependencies described in Section 3.1 assumes there is only one constraint per slot. If a slot can have multiple constraints, then a model edge must store a pointer to a formula so that the proper constraint can be located. This pointer is called a *formula signature*. The mark phase compares the edge's formula pointer with the formula pointer in each of the slot's constraints. The constraint with the matching formula pointer is invalidated.

3.4 Mixed Dependencies

The model dependencies described thus far cannot model every relationship that might be desired by a programmer. For example, suppose a programmer used the following constraint to center the x-coordinate of an arrow's tail within another object:

$$arrow.tail\_x = self.from\_node.left + (self.from\_node.width)/2$$

In this example, each arrow contains a `from_node` slot that points to the object from which its tail originates (it also has a corresponding `to_node` that guides its tip). The `from_node` slot references an object that is not part of an arrow's composition hierarchy. Consequently, model dependencies cannot be used to model this relationship and an explicit dependency must be used.

Our scheme accommodates the need for explicit dependencies by requiring each slot reference within a formula to indicate if the constraint system should establish an explicit dependency. In our implementation, the programmer passes a flag to a slot's get method that indicates whether or not an explicit dependency should be generated. The implementation makes use of C++'s default parameter mechanism, and by default an explicit dependency is generated. Hence the programmer only passes a flag if a model dependency should be generated. As discussed in Section 8.1, we eventually hope to be able to infer model dependencies, in which case the use of this flag could be eliminated.

4. IMPLEMENTATION

This section describes the implementation of the basic model dependency scheme, and the next section describes some extensions that were made to the basic scheme. It begins by describing the model edge data structure. It then discusses the creation of model dataflow graphs and describes algorithms for traversing model dataflow graphs. Finally it ends by discussing how model dataflow graphs can accommodate edits to an object after the object has been created.

4.1 The Model Edge Type

Model dependency edges and model parameters were informally described in Sections 3.1-3.3. The two types are similar enough that they can both be implemented using a `Model_Edge` class. The class hierarchy for model edges is shown in Figure 9. `Model_Edge` is an abstract superclass that defines the interface to which all model edges must conform. Any `Model_Edge` object must be able to:

—take a slot, invert itself to generate an appropriate set of model dependency edges to that slot, and add the edges to the appropriate model dataflow graph (`install()`–Section 4.2.2),

—take a slot, follow a path through a composition hierarchy and produce the constraint that depends on that slot (`traverse()`–Section 4.4), and

—take a path and create a new `Model_Edge` (`create()`).

The subclasses `Self_Model_Edge`, `Child_Model_Edge`, `Parent_Model_Edge`, and `Sibling_Model_Edge`, are used to specify model parameters.

The subclass `Model_Dependency_Edge` specifies a model dependency edge. A model dependency edge is like a model parameter edge in that it specifies a path

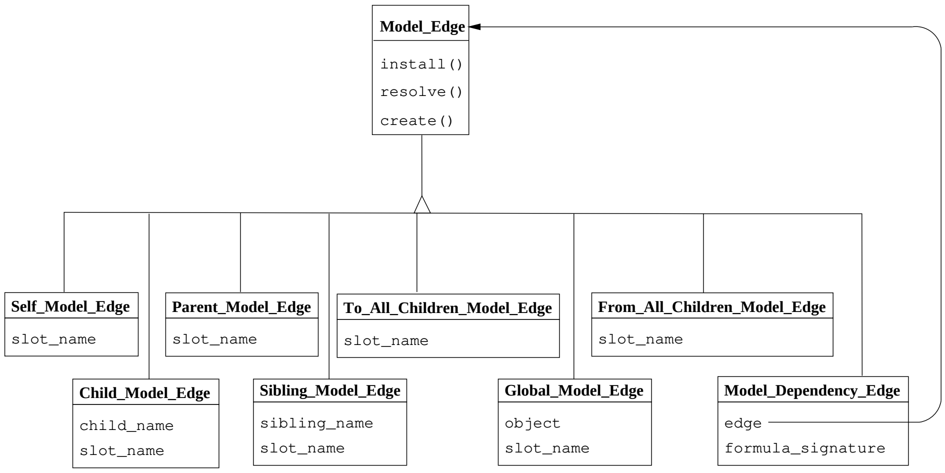


Fig. 9. The class hierarchy for model edges. `Model_Edge` is an abstract base class that defines the `Model_Edge` method interface. The `slot_name` field indicates the parameter slot for a parameter edge or dependent slot for a dependency edge. The `child_name` and `sibling_name` provide the required additional path information for `CHILD` and `SIBLING` edges. The object field for a `GLOBAL` edge provides the required pointer to an object.

to a slot. However, since a model dependency edge also requires a formula signature, it is necessary to define an additional subclass for model dependency edges. The `Model_Dependency_Edge` class has two fields, an `edge` field that points to the appropriate model edge, and a `formula_signature` field that points to the formula signature. The `traverse()` message is delegated to the model edge. The `install()` message is defined to do nothing because the `install()` method is only meant to be used with model parameter edges.

4.2 Creation of Model Dataflow Graphs

A model dataflow graph for a template object is created by examining each of its constraints, and, if the constraint has a list of model parameters, inverting each of the parameters to create a set of model dependencies.

4.2.1 *Division of Responsibility.* A model dataflow graph must be divided into two parts, which are called the `model subtree graph` and the `model supertree graph` respectively. The reason is that the four basic dependency types fall into two distinct groups. `SELF` and `CHILD` dependency edges can be installed and traversed independently of a parent. These edges represent *parent-independent* edges. In contrast, both `PARENT` and `SIBLING` dependency edges can be installed and traversed correctly only if a parent object is present. These edges represent *parent-dependent* edges. As Figure 10 illustrates, parent-independent edges denote *subtree* relationships, and parent-dependent edges denote *supertree* relationships. The model subtree graph represents an object's `SELF` and `CHILD` model dependencies while the model supertree graph represents an object's `SIBLING` and `PARENT` dependencies.

An object maintains a direct pointer to its subtree graph but relies on its parent to provide a pointer to its supertree graph. For example, Figure 11 shows how the

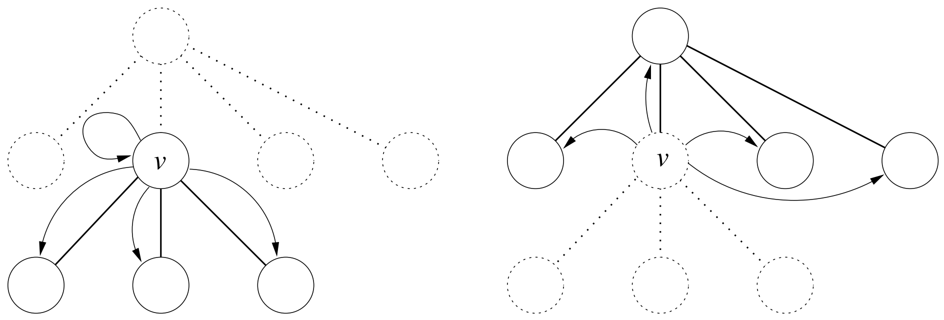


Fig. 10. Subtree versus supertree relationships. `SELF` and `CHILD` relationships refer to slots in object v itself or its children and hence are called **subtree** relationships. `PARENT` and `SIBLING` relationships refer to slots in object v's parent or siblings and hence are called **supertree** relationships. The path for a subtree edge does not pass through the parent while the path for a supertree edge does.

division of responsibility would be handled for the labeled box object. Note that the labeled box stores a collection of model supertree graphs, one for each part.

The reason that a parent stores its parts' supertree graphs is that the `PARENT` and `SIBLING` dependency edges are intrinsic not to the part but rather to the composite object to which they belong. For example, consider the `PARENT` edge from `C.y` to `A.w` in Figure 7 that would be created by a constraint like `A.w = C.y`. If part `C` is removed from the composite object and replaced with another part named `C`, then `A.w` will depend on the new part's `y` slot. Hence, the new part will assume the `PARENT` edge. Similarly, consider the `SIBLING` edge from `B.x` to `C.y` in Figure 7 that is created by a constraint like `C.y = C.parent.B.x`. Again, if part `B` is removed from the composite object and replaced with another part named `B`, `C.y` will now depend on the new part's `x` slot. Hence, the new part will assume the `SIBLING` edge. Consequently, the `PARENT` and `SIBLING` edges are not intrinsic to part `B` but rather to the composite object to which they belong.

This distributed responsibility for model dependency maintenance allows parts to be dynamically added to or removed from composite objects. If a composite object is created with one or more missing parts, it is still possible to create a model supertree for the missing parts. In Figure 7 for example, even if part `B` is missing, the model supertree for `B` can be created and the edge denoting the sibling relationship between `B.x` and `C.y` can be stored in the graph. When the missing part `B` is later added to `A`, it will assume all the `PARENT` and `SIBLING` dependencies associated with that part. Similarly, when a part is removed from a composite object, it automatically jettisons its model supertree graph but it is still possible for it to make use of its model subtree graph.

4.2.2 *Model Dependency Creation.* The fact that some model dependencies require a parent means not every constraint which is capable of generating model dependencies should be included in the model dataflow graph. In other words, it is sometimes necessary to force a constraint to generate explicit dependencies. This

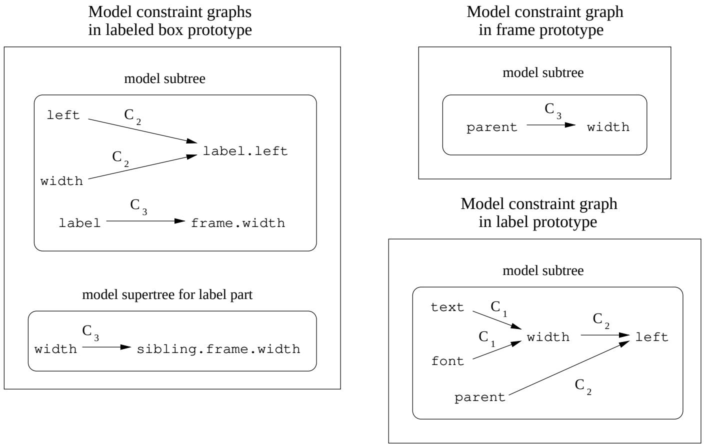


Fig. 11. The labeled box's model dataflow graphs. C_i refers to the constraint that creates the dependency. Note that the model dependencies for a formula can be distributed across two different graphs, as is the case for label's C_2 constraint. C_2 has two references to its parent (**`parent.left`** and **`parent.width`**) and one reference to itself (**`self.width`**). The parent references generate `CHILD` edges that are placed in the parent's model subtree and a `SELF` edge (**`parent`** $\rightarrow$ **`left`**) that is placed in the label's model subtree. The self reference generates a `SELF` edge (**`width`** $\rightarrow$ **`left`**) that is placed in the label's model subtree.

situation arises if the constraint uses at least one `PARENT` or `SIBLING` model parameter and its object does not have a parent. Such a constraint must generate explicit dependencies since a parent would be required to store the resulting `CHILD` and `SIBLING` dependencies. For example, in Figure 7, if part `B` is created independently of the composite object `A` and added at a later time, then part `B` has no place to put any `CHILD` or `SIBLING` edges when it is created[1].

The method that creates a model dataflow graph handles this situation by checking whether a constraint has a parameter that requires a parent, and if so, checking whether the object has a parent. The constraint generates model dependencies only if the parent exists or if none of its parameters require a parent. If the constraint generates model dependencies, its `generate_dependencies` flag is set to false. Figure 12 formalizes the procedure used to create a model dataflow graph.

The procedure examines each model constraint in the object and its parts. If the constraint passes the parent test (only constraints in the top-level object will not pass this test), each parameter in the constraint's parameter list is asked to generate an appropriate set of model dependencies. This is done by calling a parameter's `install` method. Table 6 in Section 3.2 shows the set of model dependency edges that each type of parameter should generate. Figure 13 shows the `install()` method for two subclasses, `Self_Model_Edge` and `Sibling_Model_Edge`. The `install()` methods for the other two subclasses, `Child_Model_Edge` and `Par-`

[1]Even when `B` is later added to `A`, it should not add its `CHILD` and `SIBLING` edges to `B`'s model supertree, because other instances of `A` might have a different type of part for `B`. For example, a labeled box might use either a text part or a bitmap part as its label. Since the text part and the bitmap part might use different constraints, they should not be allowed to add their `CHILD` and `SIBLING` edges to the labeled box's model dependency graphs.

```
Method Object.create_model_dataflow_graph()
for each slot s ∈ self.slots :
    for each constraint cn ∈ s.constraints :
        —Some model parameters, such as a PARENT or SIBLING model
        —parameter, require a parent in order to be inverted so
        —check to see whether a parent is needed, and if so,
        —check whether the parent exists
        if cn.modeled = true and (cn.needs_parent = false or self.parent ≠ null)
            cn.generate_dependencies = false
            for each parameter param ∈ cn.formula.model_parameters :
                param.install(self, s, cn.formula)
for each child pt ∈ self.parts :
    pt.create_model_dataflow_graph()
```

Fig. 12. Model dataflow graph creation algorithm. Observe that the pointer to the constraint's formula serves as the formula signature.

`ent_Model_Edge`, are similar.

4.3 Constraint Inheritance

In order to use a model dataflow graph, an instance must copy pointers to the model subtree and supertree graphs from its template, and then inherit the constraints. Figure 14 formalizes the algorithm for handling constraint inheritance when an object is created.

4.4 Traversing the Dataflow Graph

Once the model dataflow graph has been created, the constraint solver can use it to find and invalidate slots and constraints. When a slot is changed, the constraint solver locates the slot's model dependencies and uses these dependencies to traverse the composition hierarchy and find the dependent slots. Formula signatures are used to locate the proper constraint. The invalidation procedure for model dependencies was illustrated in Section 3.1. Figure 15 formalizes the invalidation process.

4.5 Edits After an Instance has been Created

In most constraint-based object systems an instance can be edited after it has been created. The model dependency scheme must therefore be resilient in the face of editing operations. The following edit operations are supported by constraint-based object systems and must be handled correctly:

(1) a constraint is assigned to a slot,
(2) a constraint is removed from a slot,
(3) a part is added to a composite object, and
(4) a part is removed from a composite object.

In a prototype-instance system, it is also possible to edit the prototype. In this case, the prototype's model dataflow graph may also have to be adjusted to reflect the edit. For example, when a part is added to a prototype, the prototype's model dataflow graph should be augmented to reflect any model dependencies the new part may require.

```
—The self install method creates a SELF model dependency edge from the
—parameter slot to to_slot and places it in the to_obj's model
—subtree. The name of the parameter slot can be retrieved from
—Self_Model_Edge's slot_name field.
Method Self_Model_Edge.install(Object to_obj, Slot to_slot, Signature sig)
  graph ← to_obj.model_subtree
  new_edge ← Self_Model_Edge.create(to_slot)
  new_dependency ← Model_Dependency_Edge.create(new_edge, sig)
  graph[self.slot_name].insert_edge(new_dependency)
```

```
—The sibling install method creates model dependency edges from a SIBLING
—parameter (see Table 6). The method takes as arguments the object and the
—slot whose constraint is requesting the parameter. For example, assume
—that the parameter edge is (SIBLING, B, w) and that C.v is requesting
—B.w (as in Table 6). C and v will be passed in as the to_obj and the
—to_slot respectively. The install method will create three model
—dependency edges: 1) a SIBLING edge from B.w to C.v, a CHILD edge
—2) from A.B to C.v, and 3) a SELF edge from C.parent to C.v.
Method Sibling_Model_Edge.install(Object to_obj, Slot to_slot, Signature sig)
  parent ← to_obj.parent
    —Add the SIBLING edge from B.w to C.v by placing the edge in B's model
    —supertree. self points to the parameter edge. Hence
    —self.sibling_name refers to B and self.slot_name refers to w.
  graph ← parent.model_supertree[self.sibling_name]
  new_edge ← Sibling_Model_Edge.create(to_obj.part_name, to_slot)
  new_dependency ← Model_Dependency_Edge.create(new_edge, sig)
  graph[self.slot_name].insert_edge(new_dependency)
    —Add the CHILD edge from A.B to C.v by placing the edge in A's model subtree.
  graph ← parent.model_subtree
  new_edge ← Child_Model_Edge.create(to_obj.part_name, to_slot)
  new_dependency ← Model_Dependency_Edge.create(new_edge, sig)
  graph[self.sibling_name].insert_edge(new_dependency)
    —Add the SELF edge from C.parent to C.v by placing the edge in
    —C's model subtree.
  graph ← to_obj.model_subtree
  new_edge ← Self_Model_Edge.create(to_slot)
  new_dependency ← Model_Dependency_Edge.create(new_edge, sig)
  graph[parent].insert_edge(new_dependency)
```

Fig. 13. Installation method for the `Self` and `Sibling` model parameters. The `insert_edge()` method adds a new model dependency edge to the designated graph.

```
Method Object.inherit_constraints(Object template)
    —Make the object point to its model subtree and its childrens' supertree graphs
 1  self.model_subtree ← template.model_subtree
 2  self.model_supertree ← template.model_supertree
    —Inherit the constraints
 3  for each slot s ∈ self.slots :
 4      s.constraints ← ∅
 5      for each constraint cn ∈ template.slots[s].constraints :
 6          inherited_cn ← cn.create_copy()
 7          s.constraints ← s.constraints ∪ {inherited_cn}
 8          if cn.needs_parent and self.parent = null :
 9              inherited_cn.generate_dependencies ← true
10  for each part pt ∈ self.parts :
11      pt.inherit_constraints(template.parts[pt])
```

Fig. 14. The constraint inheritance algorithm which is called when an object is created. The conditional on lines 8-9 ensures that a constraint which needs a parent in order to use model dependencies (i.e., it has SIBLING or PARENT parameters) has such a parent. Otherwise the constraint is forced to generate explicit dependencies.

```
Method Slot.invalidate()
if self.valid = true :
    self.valid ← false
    for each constraint cn ∈ self.explicit_dependencies :
        cn.invalidate()
    model_dependencies ← self.get_model_dependencies()
    for each model dependency dep ∈ model_dependencies :
        cn ← dep.traverse(self, null)
        if cn ≠ null :
            cn.invalidate()
```

```
—Retrieve the SELF and CHILD model dependencies from the model subtree
—and the PARENT and SIBLING model dependencies from the model supertree.
Method Slot.get_model_dependencies() returns Set of Model Dependency Edges
return self.object.model_subtree[self.name]
    ∪ self.object.parent.model_supertree[self.object.part_name][self.name]
```

Fig. 15. Slot invalidation algorithm for a constraint solver that incorporates model dependencies. The invalidation procedure for constraints is the same as the one found in Figure 5. `traverse()` is a custom method associated with each model dependency that uses the traversal procedure described in Section 4.4 to return the appropriate constraint. The second parameter to `traverse()` is a formula signature that allows `traverse()` to locate the appropriate constraint. Here **null** is passed as the formula signature since the model dependency edge will substitute its own `formula_signature` attribute when it delegates the call to its `edge` attribute. Figure 16 shows the `traverse` method for a `Self_Model_Edge` and a `Sibling_Model_Edge`.

```
Method Model_Dependency_Edge.traverse(Slot slt, Signature unused)
    returns Constraint
    —The second parameter is ignored. Observe that slot.invalidate() in Figure 15
    —passes null as the second parameter.
1   return self.edge.traverse(slt, self.formula_signature)
```

```
Method Self_Model_Edge.traverse(Slot slt, Signature sig) returns Constraint
1   obj ← slt.object
    —Locate the dependent slot based on the path stored in this edge
2   dependent_slot ← obj.slots[self.slot_name]
3   for each constraint cn ∈ dependent_slot.constraints :
4       if cn.formula = sig :
5           return cn
6   return null
```

```
Method Sibling_Model_Edge.traverse(Slot slt, Signature sig) returns Constraint
1   obj ← slt.object
    —Locate the dependent slot by going to the parent and then to the appropriate sibling
2   parent ← obj.parent
3   sibling ← parent.parts[self.sibling_name]
4   dependent_slot ← sibling.slots[self.slot_name]
5   for each constraint cn ∈ dependent_slot.constraints :
6       if cn.formula = sig :
7           return cn
8   return null
```

Fig. 16. Traversal methods for the `Model_Dependency_Edge` type. This figure shows the traverse methods for the `SELF` and `SIBLING` edges. The `traverse` methods for `PARENT` and `CHILD` edges are similar. The `Model_Dependency_Edge` delegates the call to the appropriate model edge and inserts the appropriate formula signature. These algorithms assume that the `parent`, `sibling`, and `dependent_slot` objects exist. The actual implemented algorithms check to see whether they get non-**null** objects before proceeding (Line 2 in `Self_Model_Edge.traverse()` and Lines 2–4 in `Sibling_Model_Edge.traverse()`). If they get a **null** object, they return **null**.

Adding and removing constraints from slots are common operations which are discussed in this section. Adding and removing parts from objects and editing a prototype are less commonly supported operations and are described in the electronic appendix that accompanies this paper.

4.5.1 *Constraint Assigned To A Slot.* When a formula is assigned to an object after it has been created, the formula will not be represented in the template's model dataflow graph. Consequently it needs to generate explicit dependencies. This can be done by setting the constraint's `generate_dependencies` flag to true.

4.5.2 *Constraint Removed From A Slot.* A constraint can be removed from a slot in two ways: 1) the constraint is replaced by an ordinary value, or 2) the constraint is replaced with a different constraint. The second case is simply constraint removal followed by constraint assignment, so only the first case needs to be considered. Constraint removal does not pose a problem to the model dependency traversal process since a dependent constraint is found by matching a formula signature stored with the model dependency edge. Although one or more model dependency edges will still point to the slot from which the constraint was removed, their formula signatures will no longer match the formula signatures of any of the slot's constraints. When these edges are followed during constraint invalidation, the edge traversal methods will return no constraint (that is, `null`) as shown in Figure 16. The invalidation procedure will then ignore this edge (in Figure 15, line 8 of the `Slot.invalidate()` method checks if the edge traversal returns a `null` constraint and does nothing if a `null` constraint is returned).

As an example, consider what happens if the formula computing `frame.width`, C_3, is removed from an instance of a labeled box. Now when `label.width` changes, the invalidation procedure will ask the model dependency edge `SIBLING, frame, width` to produce a constraint. The traversal method for this edge will follow the path to `frame.width` but will not find the proper formula signature for C_3 in `frame.width`'s list of constraints. The traversal method will therefore return `null`, and no action will be taken.

5. ADDITIONAL TYPES OF MODEL DEPENDENCIES

The basic model dependency scheme described in the previous section is not sufficient to eliminate many of the dependencies in an application (see Section 6.1 for details). This section describes a number of other common model dependency edges that were found in our analysis of Amulet applications

5.1 To All Children Edge

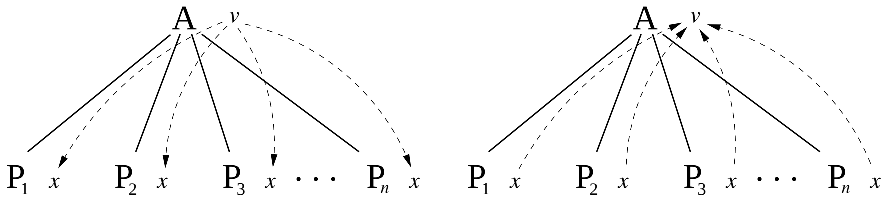
The `To All Children` edge indicates that all the children of an object depend on some slot in that object. For example, consider what happens in a *map object*, which is a special type of composite object that lays out all of its children in some type of arrangement, such as a list or table. A map object is typically given a list or array of values, which allows each of the children to be customized. For example, a map that lays out a list of labeled boxes might be given an array of labels. Each child is given an index value and formulas can use this index value to retrieve the appropriate information from the map. For example, a labeled box might use the following formula to retrieve its label from the parent:

$$box.text = self.parent.label\_array[self.index]$$

Note that the `text` slot of every labeled box will depend on its parent's `label_array` slot. Normally a `PARENT` model parameter could be used to specify this relationship and a `CHILD` dependency would be generated for each part. However, a `PARENT` model parameter requires a named part and the parts in a map are typically not named.

The `TO_ALL_CHILDREN` edge solves this problem. It is similar to a CHILD dependency. However, no named part is required since the dependency is to *all* children. Hence the edge is represented as a 2-tuple (`TO_ALL_CHILDREN`, slot). Figure 17.a illustrates the `TO_ALL_CHILDREN` dependency.

The `TO_ALL_CHILDREN` dependency can be generated in multiple ways. First, a new model parameter (e.g., an `ALL_PARENT` parameter) can be defined that generates a `TO_ALL_CHILDREN` dependency. Alternatively, a toolkit-specific way might provide a better approach. For example, the Amulet toolkit [Myers et al. 1997] that we used for our test implementation provides map objects that have template objects for each type of element that can be included in the map. We modified Amulet to create a `TO_ALL_CHILDREN` edge when a `PARENT` model parameter is specified for a



(TO_ALL_CHILDREN) (FROM_ALL_CHILDREN)

Fig. 17. The TO_ALL_CHILDREN and FROM_ALL_CHILDREN relationships. In the TO_ALL_CHILDREN relationship, the x slot slot in each of A's parts $P_1 \ldots P_n$ depends on A's v slot. In the FROM_ALL_CHILDREN relationship, the dependencies are reversed and A's v slot depends on the x slot in each of its parts.

```
Method Constraint_Set.invalidate()
for each constraint cn ∈ self.set :
    cn.invalidate()

Method To_All_Children_Model_Edge.traverse(Slot slt, Signature sig) returns Constraint
— 1) slt is the slot that changed
— 2) constraint_set is of type Constraint_Set, a subclass of Constraint
constraint_set ← ∅
obj ← slt.object
for each part p ∈ obj.parts :
    —The slot name stored in this dependency edge determines the target
    —dependent slot
    dep_slot ← p.slots[self.slot_name]
    for each constraint cn ∈ dep_slot.constraints :
        if cn.signature = sig :
            constraint_set ← constraint_set ∪ {cn}
return constraint_set
```

Fig. 18. The To_All_Children_Model_Edge traverse method. The traverse() methods of all Model_Edge subclasses must return a single constraint, but the To_All_Children_Model_Edge must return a collection of constraints. To accommodate this requirement, a Constraint_Set class is created that can store a set of constraints. The Constraint_Set is a subclass of Constraint; therefore, it responds to the same messages that the Constraint class does. The Constraint_Set invalidate() method simply delegates the invalidation message to each of the constraints it stores. Also note that the traverse method gracefully handles children that do not depend on the slot. For these children no constraint gets added to the constraint set.

formula in a map template object. Hence the programmer did not have to learn a new type of model parameter.

Implementation. TO_ALL_CHILDREN edges are represented by the To_All_Children_Model_Edge class. These TO_ALL_CHILDREN edges are stored in the model subtree along with the SELF and CHILD model edges. The To_All_Children_Model_Edge.traverse() method returns a list of constraints to invalidate. It builds this list by iterating over all the children of an object and finding eligible constraints for each child in a manner similar to the basic Child_Model_Edge.traverse() method. Figure 18 formalizes the To_All_Children.traverse() method.

5.2 From All Children Model Parameter

The `From All Children` parameter indicates that a slot in a parent object depends on the same slot in each of the children objects. For example, a composite object often contains formulas that compute its width and height based on the width and height of its children. The width formula might be written as:

```
width = {
      p = self.first_part
      left = p.left
      right = p.left + p.width
      for each part p in self.parts:
          if p.left < left:
              left = p.left
          if p.left + p.width > right:
              right = p.left + p.width
      return (right - left)
}
```

The width of an object is therefore determined by the `left` and `width` slots of all its children. Note that the number of children referenced by this formula is not fixed and that the names of the children are not used. Consequently, this formula cannot use a fixed size parameter list to specify model dependencies to each of its children. What is needed is a model parameter that represents a dependency from a given slot in every child to a slot in their parent. The `FROM_ALL_CHILDREN` model parameter addresses this need. It is a 2-tuple (`FROM_ALL_CHILDREN`, slot) and generates one `PARENT` edge for each child. Figure 17.b illustrates the edges that are generated.

The `FROM_ALL_CHILDREN` parameter is represented by the `From_All_Children_-Model_Edge` class. When it is installed, a `From_All_Children_Model_Edge` produces a `Parent_Model_Edge`, but this `PARENT` edge is placed in a distinct subgraph of the model dataflow graph called the *common model supertree.* A common model supertree stores the `PARENT` dependencies that are shared by all children. The motivation for this new subgraph is that it is not sufficient to place a `PARENT` dependency edge in each of the existing model supertrees since model supertrees are associated only with named children. However, unnamed children also need to support this edge.

The `From_All_Children_Model_Edge.install()` method, shown in Figure 19, adds the appropriate PARENT edge, to the common model supertree.

The `get_model_dependencies()` method in Figure 15 also needs to be modified so that model dependencies are retrieved from the common model supertree.

5.3 Global Model Edge

Some explicit dependencies result from access to a global resource such as a graphical device context object. For example, a formula that computes the width of a text string may need information about the font size and a formula that computes the size of a dialog box might need information about the screen resolution. As another example, the color of the objects in an application might be determined by a global, user-settable property. In all of these cases, dependencies will be estab-

```
Method From_All_Children_Model_Edge.install(Object obj, Slot dependent_slot,
                                            Signature sig)
graph ← obj.common_model_supertree
new_edge ← Parent_Model_Edge.create(dependent_slot)
new_dependency ← Model_Dependency_Edge.create(new_edge, sig)
—self.slot_name is the name of the parameter slot requested by the constraint.
—Hence, the new dependency edge is stored with this slot name in the
—common model supertree.
graph.insert_edge(self.slot_name, new_dependency)
```

Fig. 19. The `From_All_Children` install method

lished from these global properties to many graphical objects in the application. If a large number of objects reference these global properties and if they are changed infrequently, then when they do change we may be willing to examine all the objects in the interface in order to discover which slots should be invalidated.

A `Global_Model_Edge` captures this notion of a dependency from a global property to graphical objects in the application. The `GLOBAL` relationship is expressed in a model parameter by specifying the triple `(GLOBAL, object, slot)`, where `object` is the global object accessed in the formula, and `slot` is the particular slot within that object that is accessed. A `Global_Model_Edge` dependency edge is created from this model parameter and stored in the global object's model subtree. When the designated slot is changed, all the slots of all the objects in the application will be examined. If the implementor of the constraint system is concerned that this examination will be too expensive, it is possible to store the specific slot in a graphical object that depends on the global slot. For example, only color slots might be examined if the global color property is changed. In our implementation we found that examining all the slots in each graphical object was acceptable.

Implementation. The `install()` and `traverse()` methods for a `Global_Model_Edge` are shown in Figure 20.

5.4 Additional Extensions

It is important to note that other toolkits might require different types of extensions. However, it seems that the situations requiring these extensions are likely to arise in other toolkits as well. subArctic, another constraint-based toolkit, provides support for formulas that depend on all the children or in which all the children depend on the parent [Hudson and Smith 1996]. Global variables are commonly used in programs and there is every reason to assume that formulas in many toolkits would want to access global variables.

The model dependency scheme also can be easily extended to support other types of dependencies that might arise in a toolkit's applications. For example, an application may make frequent use of a formula with `GRANDPARENT` parameters. A toolkit developer could handle this case by defining `GRANDPARENT` and `GRANDCHILD` subclasses that extend the `Model_Edge` class and writing `traverse` and `install` methods for these two edges.

Finally, in a prototype-instance model, the model dependency scheme can be extended so that a prototype itself uses its own model dataflow graph. In the scheme

```
Method Global_Model_Edge.install(Object obj, Slot slt, Signature sig)
    —obj is the global object accessed by the formula
    graph ← obj.model_subtree
    —A singleton GLOBAL model edge object can be used for the whole
    —application because no explicit path information needs to be stored.
    —The global_model_edge object passed to the Model_Dependency_Edge.Create()
    —method is this singleton GLOBAL edge object.
    dependency ← Model_Dependency_Edge.create(global_model_edge, sig)
    —self.slot_name is the name of the parameter slot requested by the constraint.
    —Hence, the new dependency edge is added to the slot's list of model dependencies.
    graph.insert_edge(self.slot_name, dependency)
```

```
Method Global_Model_Edge.traverse(Slot slt) returns Constraint_Set
    constraint_set ← ∅
    —The variable interface is meant to convey the notion that every object in the
    —interface must be examined. The exact mechanism by which this set of objects is
    —determined is toolkit-dependent.
    for each object obj ∈ interface :
        obj.collect_global_dependencies(constraint_set, slt)
    return constraint_set
```

```
Method Object.collect_global_dependencies(Constraint_Set cn_set,
                                          Slot parameter_slot)
    for each slot s ∈ obj.slots :
        for each constraint cn ∈ s.constraints :
            for each parameter param ∈ cn.formula.model_parameters :
                if param.type = Global_Model_Edge
                    and param.slot_name = parameter_slot.name :
                    cn_set ← cn_set ∪ {cn}
    for each child p ∈ obj.parts :
        p.collect_global_dependencies(cn_set, parameter_slot)
```

Fig. 20. Algorithms to install and traverse `Global_Model_Edge` dependencies. The `install` method adds a `Global_Model_Edge` to the model subtree of the specified global object. The `traverse` method finds all the constraints that depend on the specified slot in the global object. The `traverse` method examines every constraint in every object in the graphical interface If the constraint has a global model parameter whose slotname matches the invalidated slot, then the constraint is added to the set of affected constraints (the `Constraint_Set` object to which these constraints are added is described in Figure 18). The `traverse` method assumes that all the affected objects can be found by examining an object's parts list, but it could easily be modified to find objects in another manner, for example, by scanning a list of windows and then scanning the display lists of each of the windows.

Table 7. Applications used for evaluating model dependencies.

Application/ benchmark	Description	Source
Checkers	The traditional board game	Amulet distribution
Circuit Designer	Used to build digital logic networks	Amulet distribution
Gilt	A graphical interface builder	
Labeled Box	A benchmark that creates 500 labeled boxes similar to the one described in Section 1	One of the authors
MathNet	Represents arithmetic expressions with a dataflow graph	One of the authors
Network Simulator	Simulates message passing in various network configurations	Graduate student
Dense Dependencies	A benchmark that creates 75 objects. Each object has 26 slots, 25 of which are determined by formulas. The 25 formulas establish 325 dependencies for each object.	One of the authors
Testwidgets	Test of menus, dialog boxes, scroll bars, etc.	Amulet distribution
Tree Editor	Graphically represents binary trees for visual program debugging	Graduate student
Empty	An application that does nothing but initialize the base Amulet system. This application indicates the "baseline" dependency requirements for any Amulet application.	One of the authors

described thus far, only a template object's instances use this model dataflow graph. This extension is described in the electronic appendix that accompanies this paper.

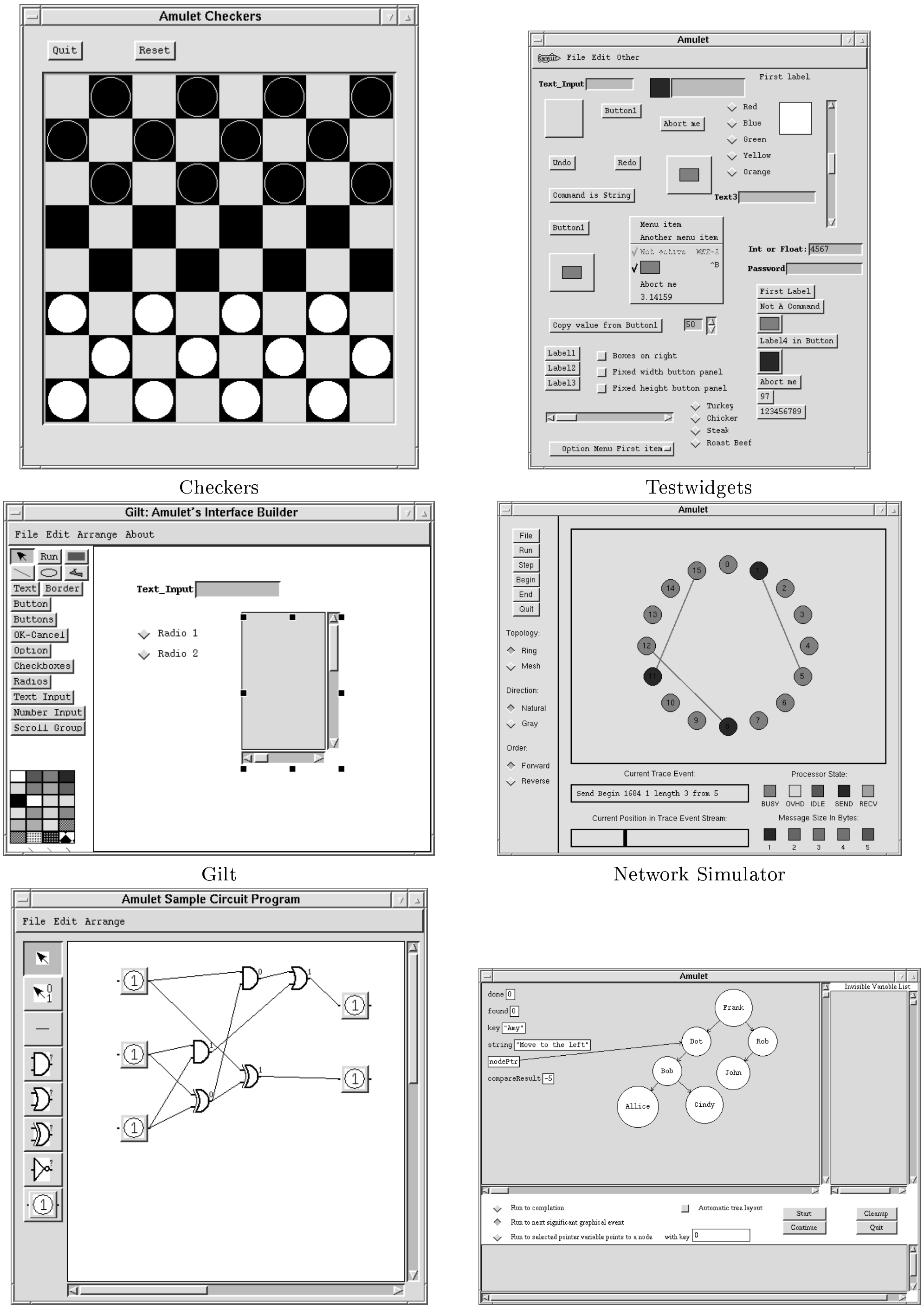
6. PERFORMANCE RESULTS

This section looks at the performance of the model dependency scheme from a number of different angles. First it looks at its success in eliminating explicit dependencies in example applications. Next it looks at the theortical time complexity of the scheme. Finally it looks at how the scheme empirically affects an application's storage and interactive performance.

6.1 Eliminating Explicit Dependencies

The model dependency scheme was evaluated by implementing it in Amulet, a graphical interface toolkit available for free from Carnegie Mellon University [Myers et al. 1997]. Amulet supports a prototype-instance object model and a constraint model similar to the ones described in Section 2. Amulet's constraint system was modified to incorporate model dependencies. Once the implementation was completed the code was instrumented so the number of model and explicit dependencies could be counted.

Henceforth, the term *Model Amulet* will refer to the modified version that supports the model dependency scheme; *Original Amulet* will refer to the original version. Model Amulet was tested on a number of existing Amulet applications and on three specialized benchmark programs. The applications included the samples found in the Original Amulet distribution and several other contributed programs. Table 7 describes the applications and benchmarks. Pictures of the graphical applications are shown in Figure 21. As indicated in Table 7, some of the applications



Checkers | Testwidgets | Gilt | Network Simulator | Circuit Designer | Tree Editor

Fig. 21. Application snapshots. MathNet can be seen on Page 3. The Labeled Box and Dense Dependencies benchmarks are non-graphical. The Empty benchmark creates the underlying base Amulet graphical objects necessary for any graphical application, but it does not create a window.

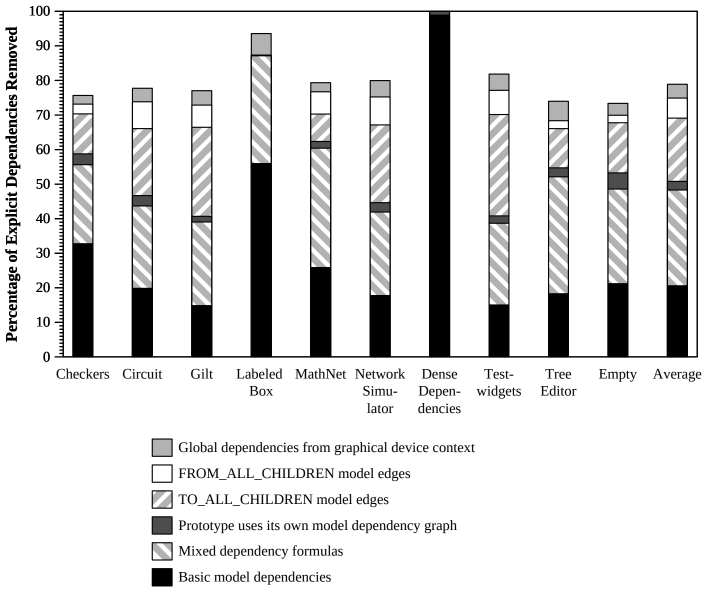


Fig. 22. The percentage of explicit dependencies removed by the various facets of the model dependency scheme. Basic model dependencies refers to a scheme in which explicit dependencies can be eliminated only if a formula exclusively references slots in a `SELF`, `CHILD`, `PARENT`, or `SIBLING` object. Mixed dependencies refers to a scheme in which the exclusivity provision is relaxed so that a formula can reference additional slots by generating explicit dependencies. The remaining schemes show the additional explicit dependencies that are eliminated when formulas can use global model dependencies, from_all_children parameters, and to_all_children dependencies and when prototypes can use their own model dataflow graphs. Average represents the average reduction for actual applications (excluding the Labeled Box, Dense Dependencies, and Empty applications).

were crafted by the designers of Amulet itself, some were written by graduate students as class projects, and some were created by the authors. They thus represent a range of application programmer expertise, from those expert at using constraints to those just learning how to use constraints. Existing applications can be compiled and run under Model Amulet without any alteration. However, the point of Model Amulet is to save storage by using model dependency formulas, so the formulas in these test programs were modified to declare model parameters.

The numbers of explicit dependencies generated by both the standard version and modeled version were then compared. Figure 22 summarizes the results of the experiments. As Figure 22 reveals, the results with a scheme that uses only the basic model dependency model were not encouraging (the basic scheme does not allow a formula to have both explicit and model dependencies). While over 50% of all formulas exclusively use the four model relationships, on average these formulas only generate approximately 20% of the dependencies in an application. This poorer-than-expected result prompted the extensions described in Sections 3.4 and 5.

Collectively the extensions increased the average number of modelable dependencies from 20% to 77.9% in the benchmark Amulet applications (excluding the special benchmarks). Mixed dependencies and children handling dependencies eliminated the most explicit dependencies. Global dependencies and having a prototype use its own model dataflow graph eliminated modest numbers of explicit dependencies. Overall, the results show that the model dependency scheme performs quite well.

6.2 Time Complexity of Model Dependency Traversal

This section examines the time required to traverse a model dependency and invalidate the constraint to which it points. Constraints pointed to by explicit dependencies can be invalidated in $O(1)$ time because the mark phase follows a pointer directly to the constraint. As the following analysis shows, a constraint pointed to by a typical model dependency edge can also be invalidated in $O(1)$ time.

The time complexity for model dependencies has two components:

(1) locating the model dependencies in the model dataflow graphs, and

(2) traversing each model dependency to locate the constraint to be invalidated.

Locating Model Dependencies. A slot's model dependencies can be found in three different model graphs: 1) the model subtree, 2) the model supertree, and 3) the common model supertree. In each model graph, the slot and its list of model dependencies must be located. Typically the graphs would be implemented as hashtables keyed on a slot name, so finding the slot and its model dependencies should require on average $O(1)$ time.

Model Dependency Traversal. Traversing a model dependency and invalidating the constraint to which it points involves three steps: 1) locating the dependent part by traversing the composite object's part hierarchy, 2) locating the dependent slot, and 3) locating the constraint on the dependent slot's constraint list.

The length of the traversal is determined by the types of model dependency edges. The four neighborhood edges and the `FROM_ALL_CHILDREN` edge require at most two hops in the object hierarchy. For example, a `SIBLING` edge goes to the parent and then an appropriate child. Hence these edges require $O(1)$ time to locate the dependent part. The `TO_ALL_CHILDREN` edge visits each of an object's children. However, if explicit dependencies were used instead, there would have to be one explicit dependency for each child. In other words, if there are m children, the `TO_ALL_CHILDREN` edge corresponds to m explicit dependencies. Therefore, a `TO_ALL_CHILDREN` edge can locate a dependent child in $O(1)$ amortized time (m children/m explicit dependencies = 1). The time to locate the dependent children for the `GLOBAL` model edges depends on the number of objects in the system. If there are n objects in the system then $O(n)$ objects must be searched. In the worst case only one object will depend on the global object and so only one explicit dependency will be eliminated. This worst case leads to $O(n)$ complexity for a `GLOBAL` model edge. However, in practice most objects typically have at least one dependency on a global object so the average case running time is $O(1)$.

Once the dependent part is located the dependent slot must be found. In almost any toolkit the number of slots within an object is usually bounded by a fairly large constant such as 30–40. Either a hashtable or linked list can be used to locate a slot, so the time is $O(1)$.

Finally, the time to locate a constraint on a slot's constraint list depends on the number of constraints on the list. In most existing toolkits, including Garnet [Myers et al. 1990], Amulet [Myers et al. 1997], Eval/vite [Hudson], subArctic [Hudson and Smith 1996], and Rendezvous [Hill 1993], the list's size is typically bounded by a small constant (1 or 2 in most cases), so the time to locate a constraint is on average $O(1)$.

With the exception of the worst case for global model edges, each of the three phases involved in model dependency traversal requires $O(1)$ time, so the traversal time for each dependency is $O(1)$. Consequently the overall cost to invalidate one model dependency is $O(1)$, the same as the cost to invalidate an explicit dependency.

This time complexity means that while model dependencies will be more expensive than explicit dependencies, the speed difference should be limited to a small constant factor. As Section 6.3.2 indicates, this small speed penalty is quite acceptable for most applications.

6.3 Empirical Performance

Section 6.1 showed that the model dependency scheme can significantly reduce the number of explicit dependencies in an application. The first part of this section shows how this reduction might benefit storage and performance. The second part addresses interactive performance issues when virtual memory is not an issue and shows that while model dependencies may require more traversal time than explicit dependencies, the slower traversal time does not affect perceived interactive performance.

6.3.1 *Performance on the Dense Dependencies Benchmark.* The *Dense Dependencies* benchmark is a program that was devised to simulate an application whose storage is dominated by constraints and their dependencies rather than by the object system[2]. Each Dense Dependencies object consists of 26 slots, 25 of which are determined by formulas. If the slots are named $s_0, s_1, \ldots, s_{25}$, then a formula F_i in slot s_i calculates the sum $s_i = \sum_{j=0}^{i-1} s_j$ (slot s_0 does not contain a formula). Hence the 25 formulas generate a total of $1 + 2 + \ldots + 25 = 325$ dependencies. The formulas all involve the `SELF` relationship; consequently, Model Amulet can eliminate the explicit dependencies.

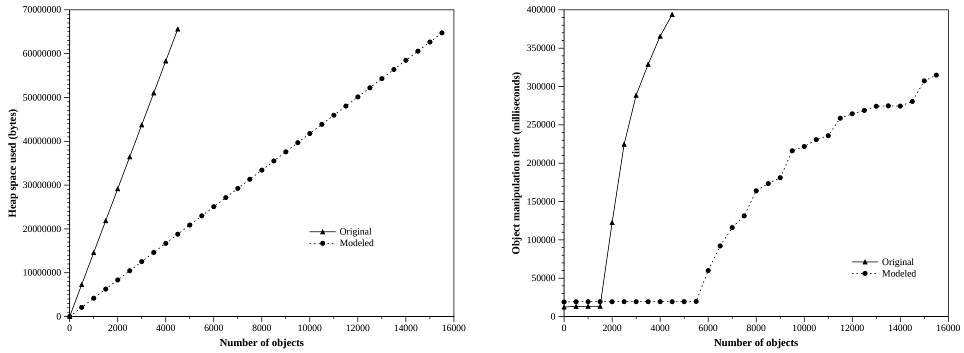
The benchmark performs the following activities:

(1) A specified number of Dense Dependencies objects are created.

(2) In a loop executed 10,000 times, one object is chosen at random. Slot s_0 is modified and all the formulas that depend on s_0 are brought up-to-date. Since s_0 is referenced by every formula in the object, the modification is guaranteed to affect all the formulas in that object.

Memory. The amount of raw heap storage required by the benchmark was measured directly by examining the memory consumed from the heap (all the graphical objects used by Amulet applications are allocated dynamically from the heap). Figure 23.a summarizes the benchmark's behavior under both Original and Model

[2] Amulet's objects are very heavyweight objects since Amulet supports a large number of features beyond those provided by most constraint systems. The size of these objects makes it difficult to measure the effect of explicit dependency reduction because the object size overwhelms the size of the constraints and the dependencies. The Dense Dependencies benchmark neutralizes this problem and thus provides an indication of the effect that model dependencies might have on the space and time resources of a more streamlined toolkit.



(a) (b)

Fig. 23. Memory requirements of objects (a) and cumulative manipulation times of objects (b) for the *Dense Dependencies* benchmark. The original version of Amulet begins to use a significant amount of virtual memory at around 1,500 objects. The modeled version can manage about 5,500 objects before virtual memory is required. Once virtual memory is required, the performance degrades at a slower rate than in the original version. (On the system tested, the virtual memory limit for a process is 64 Mbytes.) All applications were tested on a lightly loaded 200 MHz Pentium–based machine with 32 Mbytes of RAM and 512 Kbyte L2 cache running BSD/OS 3.0 and Amulet V3.0 with no debugging and `-O3` optimization.

Amulet. The operating system used for testing restricts each running process to 64 Mbytes of virtual memory. Original Amulet is limited to about 4,500 objects before virtual memory is exhausted. Model Amulet can handle about 15,500 objects; thus, the modeled version can accommodate about 3.4 times more Dense Dependencies objects.

In Amulet, each explicit dependency requires 32 bytes of storage. Analysis of the heap space used by both toolkits reveals that the storage difference between the two versions is exactly (32 bytes) × (the number of objects) × (the number of dependencies per object). This computation can be generalized to give the expected storage savings in any system:

$$s = c \times d \times \alpha$$

where s is the expected storage savings, c is the storage cost of an explicit dependency, d is the number of dependencies in the system, and α is the fraction of dependencies that can be modeled. For typical Amulet applications, the results from Section 22 indicate that $\alpha = .78$ is a reasonable estimate.

Time. Figure 23.b compares the cumulative time it takes to perform the experiment. The modeled version is consistently 1.42 times slower than the standard version up to about 1,500 objects. Above 1,500 objects the standard version begins using virtual memory, and performance drops off dramatically. The modeled version can manipulate up to 5,500 objects until virtual memory is accessed. Model Amulet can therefore represent about 3.7 times more Dense Dependencies objects in core memory than Original Amulet. After the point of virtual memory access the performance degradation is more gradual in the modeled version. When both versions are managing as many objects as virtual memory permits (4,500 for the standard version versus 15,500 for the modeled version), the modeled version is about 1.3 times faster.

In sum, model dependencies achieve significant space savings and performance improvement in this benchmark, permitting a larger number of application objects to be managed with better performance than with explicit dependencies.

6.3.2 *Interactive Performance.* None of the interactive applications that were available for testing manipulate a large number of objects. One reason that existing applications do not manage a large number of constrained objects may be because their designs are limited by the large storage requirements of each Amulet graphical object in addition to the storage requirements of a large number of explicit dependencies. Since the number of objects was small, none of the interactive applications required any virtual memory.

Fortunately, interactive applications are not significantly affected by the extra time required to resolve model dependencies. To the unaided eye, the versions using model dependencies do not exhibit any perceptible loss of interactive performance compared to their explicit dependency counterparts. The lack of any noticeable performance difference between the modeled and unmodeled versions of Amulet is not surprising when one considers the following two points:

(1) Previous studies have shown that redisplay time dominates the cost of all other operations in a graphical application [Hill 1993; Vander Zanden 1994]. One study of a set of Amulet applications revealed that the time spent traversing and invalidating dependencies constituted roughly 1% of the time consumed by an application [Vander Zanden et al.]. Thus increasing this time by even 42% (the amount of extra time required by model dependencies in the Dense Dependencies benchmark) has no noticeable affect on the application's performance.

(2) In interactive applications, the total number of slots that depend on a changed slot is generally small, and is typically less than 20 [Vander Zanden and Venckus 1996].

The significance of the second point is that the Dense Dependencies benchmark represents a worst case scenario for many user interface interactions. Each manipulation in the benchmark affects 325 dependencies and forces the recomputation of 25 constraints (each of which uses an average of 12 individual arithmetic operations for its computation). This is more re-evaluation work than is required by many interactions in a typical application.

A second experiment was performed in which a graphical (but still non-interactive) version of the Labeled Box benchmark was manipulated by moving and resizing (by changing the label text) a large number of labeled boxes in a window. This type of interaction is representative of the interactions in a real interface and, unlike the Dense Dependencies benchmark, it requires a portion of the display to be drawn. In this experiment, the overall performance of the modeled application was slower than the overall performance of the original version by a factor of only 1.002, and the number of boxes was not so large as to require either version to use virtual memory.

7. RELATED WORK

This section considers related work from a variety of different angles. It first shows how dataflow constraints fit within the field of constraint solving. It then describes some of the past research that has been done on optimizing the storage costs of dataflow constraints. Finally it discusses the related idea of lightweight glyphs. Section 1 described how model dataflow graphs in syntax-directed editors also affected the development of the model dependency scheme.

7.1 Dataflow Constraints in Context

Constraint solvers can typically be classified as either domain-specific or domain-independent. A domain-specific solver can only satisfy constraints that are specified over a restricted domain, such as real numbers, integers, or Booleans. A domain-independent solver can satisfy constraints that are specified over an arbitrary domain. A linear equation solver is an example of a domain-specific solver [Golub and Van Loan 1989]. A dataflow constraint solver is an example of a domain-independent solver. Domain specific solvers can satisfy a greater range of constraints within their domain because the constraint solver can use algorithms that exploit information about that domain. For example, a linear equation solver can use its knowledge of linear algebra to satisfy linear equality constraints. In contrast, a domain-independent solver cannot satisfy these constraints because it lacks the requisite domain-specific knowledge. Nonetheless, domain-independent constraints have found greater use in graphical interfaces because of their versatility, their simplicity, and their ability to express most of the relationships that arise in graphical interfaces.

7.1.1 *Multi-way Constraints.* Multi-way dataflow constraint systems have also been developed for graphical interfaces, including ThingLab [Borning 1981; Freeman-Benson et al. 1990; Sannella et al. 1993], Kaleidoscope [Freeman-Benson 1990] and MultiGarnet [Sannella and Borning 1992; Sannella 1994; Vander Zanden 1996]. Multi-way constraints are more powerful than one-way constraints. since they can be solved for any variable in the constraint, either on the left side or the ride side. This increased power, while valuable, also makes multi-way constraints less predictable than one-way constraints and harder to manage by the programmer. For example, suppose there is a constraint $right = left + width$. What should the constraint solver do if `right` is changed? Changing `left` will cause the object to move and changing `width` will cause the object to change size. Both changes provide plausible interpretations for how the change to `right` should be handled. *Stay* constraints have been introduced to allow programmers to specify which variables should remain unchanged [Borning et al. 1987]. Stay constraints can be used in concert with *constraint hierarchies*, which allow a programmer to prioritize constraints so that if some constraints cannot be satisfied, preference is given to satisfying the higher priority constraints [Borning et al. 1987]. In the above example, the programmer could introduce stay constraints for all three variables, but give a higher priority to satisfying the stay constraints for `right` and `left`. Hence when `right` is changed, the constraint solver would resize the object by changing the object's `width`. However, sometimes the user expects the object to move instead of being resized so even stay constraints are not a complete solution to the problem. Rosener proposed a number of ways to make constraint hierarchies and stay constraints more flexible [Rosener 1994] but the fact remains that multi-way constraints are more difficult to manage and more unpredictable than one-way constraints, and their perceived benefits are often not commensurate with these drawbacks.

7.1.2 *Constraint Logic Programming.* A final related area is constraint logic programming [Jaffar et al. 1992]. When applied to logic programming, constraints provide a generalization of the resolution principle used in standard logic programming languages such as Prolog [Roussel 1975]. One advantage of constraint logic programming is that it can mix constraint solvers from different domains, thus achieving better coverage than a single domain-specific solver. Constraint logic programming has never caught on in the graphical interfaces community because it lacks a notion of state; thus, an incremental change in an application requires a complete re-execution of the program as opposed to an incremental update of the program's state.

7.2 Dataflow Constraint Optimization

Researchers have investigated various approaches to dataflow constraint storage optimization. These include constant propagation, constraint plans that save storage by eliminating some constraints entirely, and μconstraints which reduce the cost of a constraint by encoding its dependencies in a compact manner.

7.2.1 *Constant Propagation. Constant propagation* allows constraints whose parameters are all constants to be evaluated once and replaced with the constant result [Maloney et al. 1989; Myers et al. 1994]. Constant propagation works well in a system where many slots have fixed values. Unfortunately one non-constant slot in a formula will prevent it from being eliminated. Consequently, constant propagation has little effect in a dynamic system where few slots have a fixed value. For example, a study of Amulet applications suggest that constant propagation can eliminate 15-25% of constraints [Vander Zanden and Venckus 1996]. This reduction is a good start, but other optimizations, such as the model dependency optimizations described in this paper, must be used to obtain even more storage savings.

7.2.2 *Constraint Plans.* Freeman-Benson eliminates entirely the storage required for constraint objects and their dependencies by compiling a given constraint graph or subgraph into a *plan* [Freeman-Benson 1989]. A plan is created by "unwrapping" individual constraint methods and using the constraint graph to sequence the code segments that make up these methods into one large procedure. The parameters to the procedure are the slots in the original graph whose values may be modified. The constraint plan approach is ideal for dataflow graphs with a static structure. In systems like Amulet with pointer variables, dynamic edits, and arbitrary dependency edges, the required analysis may not be possible and dynamic recompilation of the modules will almost certainly be required even if the analysis can be performed.

7.2.3 *μconstraints.* Hudson and Smith reduce the physical storage required for dependencies in certain common graphical layout relationships using a concept they call μconstraints [Hudson and Smith 1996]. Each constraint consists of as little as 17 bits, which is enough to encode an operation code, a small set of predefined dependencies to other objects, and a small set of predefined slots that can be accessed in these objects, such as `left` or `width`. Actual dependency edges in the constraint network are dynamically inferred as needed by consulting μconstraint encodings and the composite object's physical structure.

μconstraints share two important concepts with the model dependency paradigm:

(1) Dependencies that can be encoded are restricted to the direct composite object relations involving `SELF`, `PARENT`, `CHILD`, and `SIBLING`.

(2) A programmer may use standard heavyweight constraints for dependencies that cannot be represented in this encoding scheme.

However, μconstraints suffer from the same types of shortcomings that the original model dependency scheme suffered from, including an inability to mix model and explicit dependencies in a constraint and an inability to deal with references to global resources. μconstraints are also more restricted than model dependencies in that μconstraints by design support only a few types of formulas while model dependencies allow a formula to express arbitrary code.

A comparison of μconstraints with model dependencies on the benchmark Amulet applications revealed that μconstraints could model 38% of the formulas but that these formulas represented only 9% of the dependencies in an application [Vander Zanden and Halterman 1999]. These findings are consistent with the basic model dependency scheme, which was able to model 55% of the formulas but only about 20% of the dependencies. It is also probable that the μconstraint scheme could be improved with toolkit-tuning, just like the model dependency scheme.

7.2.4 *Lightweight Glyphs.* Calder and Linton used lightweight *glyphs* to implement structured graphical objects [Calder and Linton 1990; Linton et al. 1989] (glyphs are often called *flyweight* objects [Gamma et al. 1995]). Glyphs do not store all the information that specifies their appearance but depend on graphical context information passed into their drawing methods. In a similar manner model dependencies refrain from storing all the dependency information needed by the slots of an object, but instead require the object to access additional dependency information from its template.

8. CONCLUSIONS

This section explores some directions for extending our research, followed by a final assessment of the benefits of model dependencies.

8.1 Future Work

The research in this paper was primarily oriented toward developing the model dependencies scheme. The resulting scheme is effective but it is not as application programmer friendly as it might be. The programmer interface could be improved in a couple of ways:

(1) Application programmer interface. Currently an applications programmer must supply a list of model parameters to a formula that uses model dependencies. While it is not uncommon for constraint systems to require programmers to supply the parameters or dependencies that are to become part of

the dataflow graph, Amulet frees the programmer from this burden and automatically constructs the dependencies during the formula's execution. This shortcoming could be overcome by removing the requirement for a parameter list specification and inferring model dependencies.

(2) Adding a `CUSTOM` model edge. New model edge relationships can be added to the model dependency scheme by subclassing the `Model_Edge` class (see Section 4.1). However, some implementation knowledge of the model dependency scheme is required to add these edges, so it is unlikely that anyone other than a toolkit developer would add such edges. It would be nice if there was a simpler mechanism that would allow an application programmer to specify custom model edges in a formula. The difficulty with specifying an arbitrary model edge is that it is not always obvious to the constraint solver how to invert the edge to create a dependency. One approach might allow a programmer to specify both the model parameter edge and the corresponding model dependency edge. In essence, the programmer would supply the information needed by the `install()` method. Since the syntax for specifying a model dependency edge is the same as the syntax for specifying a model parameter edge (the formula signature can be filled in by the system), the programmer would not need to learn any additional syntax.

8.2 Conclusions

This paper has introduced a model dependency scheme that can reduce the number of explicit dependencies in a constraint graph significantly. For example, a sample implementation in the Amulet toolkit reduced the number of explicit dependencies by almost 80%.

The significance of the techniques developed in this paper is that they can reduce the storage requirements of programs that manage a large number of constrained objects. The reduction can allow applications to create more objects before being forced into virtual memory. When an application is forced into virtual memory its interactive performance typically degrades to unacceptable levels. Therefore, making applications small enough to be stored in RAM memory is an important goal.

The techniques developed in this paper accomplish their storage savings without a significant cost in performance. Although explicit dependency programs are slightly faster when the number of constrained objects is small, model dependency programs can be significantly faster when the number of objects force non-modeled applications to use virtual memory to provide the storage required by explicit dependencies. Additionally, even when explicit dependency applications are faster than model dependency applications, there is no perceptible difference in interactive performance.

Model dependencies thus provide a useful new mechanism for improving the storage efficiency of one-way, dataflow constraint systems.

ACKNOWLEDGMENTS

The authors wish to thank Brad Myers and the anonymous referees for their helpful comments.

REFERENCES

Alpert, S. R. 1993. Graceful interaction with graphical constraints. *IEEE Computer Graphics and Applications 13,* 2 (March), 82–91.

Barth, P. 1986. An object-oriented approach to graphical interfaces. *ACM Transactions on Graphics 5,* 2 (Apr.), 142–172.

Borning, A. 1981. The programming language aspects of ThingLab, a constraint-oriented simulation laboratory. *ACM Transactions on Programming Languages and Systems 3,* 4 (Oct.), 353–387.

Borning, A., Duisberg, R., Freeman-Benson, B., Kramer, A., and Woolf, M. 1987. Constraint hierarchies. In *OOPSLA'87 Conference Proceedings.* 48–60.

Calder, P. R. and Linton, M. A. 1990. Glyphs: Flyweight objects for user interfaces. In *ACM SIGGRAPH Symposium on User Interface Software and Technology, Proceedings UIST '90.* Snowbird, Utah.

Demers, A., Reps, T., , and Teitelbaum, T. 1981. Incremental evaluation for attribute grammars with application to syntax-directed editors. In *Proceedings of the Principles of Programming Languages Conference.* Williamsburg, VA, 105–116.

Freeman-Benson, B. N. 1989. A module mechanism for constraints in Smalltalk. *Sigplan Notices 24,* 9 (Oct.). ACM Conference on Object-Oriented Programming; Systems Languages and Applications; OOPSLA '89.

Freeman-Benson, B. N. 1990. Kaleidoscope: Mixing objects, constraints, and imperative programming. In *OOPSLA/ECOOP'90 Conference Proceedings.* 77–88.

Freeman-Benson, B. N., Maloney, J., and Borning, A. 1990. An incremental constraint solver. *Communications of the ACM 33,* 1 (Jan.).

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, Reading, Massachusetts.

Golub, G. H. and Van Loan, C. F. 1989. *Matrix Computations, 2nd Ed.* The Johns Hopkins University Press, Baltimore, MD.

Hill, R. D. 1993. The *RENDEZVOUS* constraint maintenance system. In *ACM SIGGRAPH Symposium on User Interface Software and Technology, Proceedings UIST '93.* Atlanta, Georgia.

Hoover, R. 1992. Alphonse: Incremental computation as a programming abstraction. *Sigplan Notices 27,* 7 (July), 261–272. ACM SIGPLAN'92 Conference on Programming Language Design and Implementation.

Hudson, S. and King, R. 1988. Semantic feedback in the Higgens UIMS. *IEEE Transactions on Software Engineering 14,* 8 (Aug), 1188–1206.

Hudson, S. E. Eval/*vite* user's guide (v1.0). Tech. rep., College of Computing Georgia Institute of Technology, Atlanta, Georgia.

Hudson, S. E. 1991. Incremental attribute evaluation: A flexible algorithm for lazy update. *ACM Transactions on Programming Languages and Systems 13,* 3 (July).

Hudson, S. E. 1994. User interface specification using an enhanced spreadsheet model. *ACM Transaction on Graphics 13,* 3 (July), 209–239.

Hudson, S. E. and Mohamed, S. P. 1990. Interactive specification of flexible user interface displays. *ACM Transactions on Information Systems 8,* 3 (July), 269–288.

Hudson, S. E. and Smith, I. 1996. Ultra-lightweight constraints. In *ACM SIGGRAPH Symposium on User Interface Software and Technology, Proceedings UIST '96.* Seattle, Washington.

Jaffar, J., Michaylov, S., Stuckey, P. J., and Yap, R. H. C. 1992. The CLP($\mathcal{R}$) language and system. *ACM Transactions on Programming Languages and Systems 14,* 3 (July).

Linton, M. A., Vlissides, J. M., and Calder, P. R. 1989. Composing user interfaces with InterViews. *IEEE Computer 22,* 2 (Feb.).

Maloney, J., Borning, A., and Freeman-Benson, B. 1989. Constraint technology for user-interface construction in ThingLabII. *Sigplan Notices 24,* 10 (Oct.). ACM Conference on Object-Oriented Programming Systems Languages and Applications; OOPSLA '89.

Myers, B. A., Giuse, D. A., Dannenberg, R. B., Vander Zanden, B., Kosbie, D. S., Pervin, E., Mickish, A., and Marchal, P. 1990. Garnet: Comprehensive support for graphical highly interactive user interfaces. *IEEE Computer 23,* 11 (Nov.).

Myers, B. A., Giuse, D. A., Mickish, A., and Kosbie, D. 1994. Making structured graphics and constraints practical for large-scale applications. Tech. Rep. CMU-CS-94-150, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania. May.

Myers, B. A., McDaniel, R. G., Miller, R. C., Ferrency, A., Faulring, A., Kyle, B. D., Mickish, A., Klimovitski, A., and Doane, P. 1997. The Amulet environment: New models for effective user interface software development. *IEEE Transactions on Software Engineering 23,* 6 (June).

Reps, T., Teitelbaum, T., and Demers, A. 1983. Incremental context-dependent analysis for language-based editors. *ACM Transactions on Programming Languages and Systems 5,* 3 (July). Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages, January, 1982.

Rosener, W. J. 1994. Integrating multi-way and structural constraints into spreadsheet programming. Ph.D. thesis, Department of Computer Science, University of Tennessee.

Roussel, P. 1975. *PROLOG: Manuel de Reference et d'Utilisation.* Group d'Intelligence Artificielle, Université de Marseilles.

Sannella, M. 1994. Skyblue: A multi-way local propagation constraint solver for user interface construction. In *ACM SIGGRAPH Symposium on User Interface Software and Technology.* Proceedings UIST'94, Marina del Rey, CA, 137–146.

Sannella, M. and Borning, A. 1992. Multi-Garnet: Integrating multi-way constraints with Garnet. Tech. Rep. 92-07-01, Department of Computer Science and Engineering, University of Washington. Sept.

Sannella, M., Maloney, J., Freeman-Benson, B., and Borning, A. 1993. Multi-way versus one-way constraints in user interfaces: Experiences with the DeltaBlue algorithm. *Software Practice and Experience 23,* 5, 529–566.

Vander Zanden, B. 1996. An incremental algorithm for satisfying hierarchies of multi-way, dataflow constraints. *ACM Transactions on Programming Languages and Systems 18,* 1 (January), 30–72.

Vander Zanden, B., Myers, B. A., Giuse, D., and Szekely, P. 1991. The importance of pointer variables in constraint models. In *ACM SIGGRAPH Symposium on User Interface Software and Technology.* Proceedings UIST'91, Hilton Head, SC, 155–164.

Vander Zanden, B., Myers, B. A., Giuse, D. A., and Szekely, P. 1994. Integrating pointer variables into one-way constraint models. *ACM Transactions on Computer Human Interaction 1,* 2 (June), 161–213.

Vander Zanden, B. T. 1994. Optimizing toolkit-generated graphical interfaces. In *ACM SIGGRAPH Symposium on User Interface Software and Technology.* Proceedings UIST'94, Marina del Rey, California, 157–166.

Vander Zanden, B. T. and Halterman, R. 1999. Reducing the storage requirements of constraint dataflow graphs. To appear in Proceedings of the 12th ACM SIGGRAPH Symposium on User Interface Software and Technology.

Vander Zanden, B. T., Myers, B. A., Szekely, P., Giuse, D. A., McDaniel, R., Miller, R., Halterman, R., and Kosbie, D. Lessons learned about one-way, dataflow constraints in the Garnet and Amulet graphical toolkits. Submitted for publication.

Vander Zanden, B. T. and Venckus, S. A. 1996. An empirical study of constraint usage in graphical applications. In *ACM SIGGRAPH Symposium on User Interface Software and Technology, Proceedings UIST '96.* Seattle, Washington.

Appendix

An appendix to this paper is available in electronic form (PostScript™). Any of the following methods may be used to obtain it; or see the inside back cover of a current issue for up-to-date instructions.

—By anonymous ftp from **acm.org**, file **[pubs.journals.tochi.append]p0001.ps**

—Send electronic mail to **mailserve@acm.org** containing the line
send [anonymous.pubs.journals.tochi.append]p0001.ps

—By *Gopher* from **acm.org**

—By anonymous ftp from **ftp.cs.princeton.edu**, file **pub/tochi/append/p0001.ps**

—Hardcopy from *Article Express*, for a fee: phone 800-238-3458, fax 201-216-8526, or write P.O. Box 1801, Hoboken NJ 07030; and request ACM-TOCHI-APPENDIX-0001.

APPENDIX

THIS DOCUMENT IS THE APPENDIX TO THE FOLLOWING PAPER:

Using Model Dataflow Graphs to Reduce the Storage Requirements of Constraints

BRADLEY T. VANDER ZANDEN and RICHARD HALTERMAN
University of Tennessee

ACM Transactions on Computer-Human Interaction

A. STRUCTURAL EDITS TO COMPOSITE OBJECTS

This appendix describes how the following editing operations are handled by the model dependency scheme:

(1) a part is added to a composite object, and

(2) a part is removed from a composite object.

A.1 Part Added To A Composite Object

When a part `p` is added to an object `obj` after `obj` has been created, there is no guarantee that all the model dependencies needed by `p`'s model constraints will be present. Specifically, any of `p`'s formulas with `PARENT` or `SIBLING` model parameters will probably not be represented in `obj's` model dataflow graph. For example, suppose that the template for a labeled box object is only partially specified. In particular, assume that the `frame` component is present but the label component is not (perhaps the programmer wants the flexibility of being able to use either a text string or an icon as the label).

The `CHILD` model dependencies from the parent's `left` and `width` slots to `label.left` will not be present in the labeled box's model subtree since the model dataflow graph creation method (Figure 12) will not examine the C_2 constraint in the missing label. If the `label` part is subsequently added, the C_2 constraint (which has two `PARENT` parameters) computing `label.left` will not be invalidated by the labeled box's `left` and `width` slots unless the constraint C_2 establishes explicit dependencies.

Fortunately, any model constraints that reference `PARENT` or `SIBLING` parameters and that are in an object with no parent will already be marked as having to generate explicit dependencies. The reason is that either the object was 1) created without a parent, in which case the algorithm in Figure 14 has marked these constraints, or 2) removed from another object, in which case the removal process marked these constraints (part removal is discussed next). Consequently no special action needs to be taken when a part is added to a composite object.

A couple of additional points need to be made about part addition. First, if the new part is given the same part name as a previous part that was removed, then there is a chance that one of the slots in the new part has the same formula as a similarly named slot in the previous part (e.g., both `width` slots might be computed by the same formula). In this unlikely event, duplicate dependencies, explicit and model, will map from the same slot to the same constraint. This causes a small duplication of effort since the duplicate edges will both invalidate the slot when one is sufficient to do the job. However, the behavior of the application will be unaffected. This duplication rarely occurs in practice.

Second, although model dependencies from other parts to the added part are missing, model dependencies from the added part to the pre-existing parts will be present. For example, any model dependencies that point from the label part to either the parent part or the frame part will be created and placed in the labeled box's model supertree. The reason is that the responsibility for the creation of the model dependency is that of the dependent object, not the missing part. For example, the formula for `frame.width` has a model edge parameter `(SIBLING, label, width)`. This parameter generates a `(SIBLING, frame, width)` model dependency edge and stores it in `label`'s model supertree graph, which is managed by `frame`'s parent. When a `label` part is subsequently added to a labeled box instance, the model dependency in `label`'s model supertree is already present and will properly invalidate `frame.width` whenever `label`'s left changes.

A.2 Part Removed From A Composite Object

When a part is removed from a composite object, any model dependencies that point from parent or sibling slots to slots within this removed part are now "dangling." The constraint system removes any dangling explicit dependencies, but dangling model dependencies cannot be removed since other instances use the same model dataflow graph. These other instances may not have had their corresponding part removed, so the model dependencies to this part must still be retained in the template. As in the case with constraint removal, removing a part does not break the model dependency traversal process because if a dependent slot cannot be located by following the path stored in the model dependency edge, then no dependent constraint is returned. Hence a small amount of unnecessary work is performed to discover that the dependent slot no longer exists, but no behavioral damage is done.

The constraints in the removed part that were pointed to by the model dependencies must now be marked as generating explicit dependencies since they will need to be re-evaluated if and when the part is added to another composite object (recall that an added part expects these constraints to generate explicit dependencies). A model constraint can be forced to generate explicit dependencies instead of model dependencies as follows:

(1) Invalidate the model constraint and set the constraint's `generate_dependencies` flag to true.

(2) The next time the constraint is evaluated, it will notice that its flag for generating explicit dependencies has been set and it will generate explicit dependencies, just like a standard constraint.

The constraints that need to be marked are the ones which reference PARENT or SIBLING parameters. These constraints will have their `needs_parent` field set to true, so the part removal algorithm simply checks this field in each constraint and performs the above procedure on the constraint if the field is true.

B. HANDLING PROTOTYPES

This appendix describes additional considerations that must be taken into account if a prototype-instance model is used with a model dependency scheme.

B.1 Edits to Prototypes

In a prototype-instance system, special actions must be taken if the prototype itself, rather than one of its instances is edited. In this case, the prototype's model dataflow graph may also have to be adjusted to reflect the edit. For example, when a part is added to a prototype, the prototype's model dataflow graph should be augmented to reflect any model dependencies the new part may require.

The adjustments that are made to the model dataflow graph when a prototype in a prototype-instance object system is edited can be summarized as follows:

(1) A constraint is assigned to a slot: In this case, the dataflow graph can be updated by calling the `install` method on each of the constraint's parameters. However, recall that if one of the constraint's parameters references a parent, then it needs the parent in order to invert itself and generate a model dependency edge. Hence the constraint should not install itself in the model dataflow graph if one of its parameters requires a parent and its parent is missing. Lines 3-6 in Figure 12 perform the necessary check and installation.

(2) A constraint is removed from a slot: In this case, the dataflow graph can be updated by calling a `remove` method on each of the constraint's parameters. However, as in the previous case, if one of the constraint's parameters references a parent and the parent is missing, then the constraint's model dependencies are not present in the model dataflow graph and the graph does not have to be updated. Lines 3-6 in Figure 12 can perform the necessary check and removal provided that the call to `install` is replaced with a call to `remove`. The `remove` method is similar to the `install` method in that it inverts the parameter in order to find the model dependency edges created by the parameter. It then goes to the appropriate model dataflow graphs, locates the model dependency edges, and removes them. A sample `remove` method for a Sibling parameter is shown in Figure 24.

(3) A part is added to a prototype. The part should already have a model subtree that contains model dependencies from constraints that have only SELF and CHILD parameters. In this case only constraints referencing SIBLING and PARENT parameters have to update the model dataflow graph (these constraints may also reference SELF and CHILD parameters). The part addition method can accomplish this update by scanning through the part's constraints

```
—The remove method removes the model dependency edges generated by a SIBLING
—parameter (see Table 6). The method takes as arguments the object and the
—slot whose constraint requested the parameter. For example, assume
—that the parameter edge is (SIBLING, B, w) and that C.v requested
—the parameter (as in Table 6). C and v will be passed in as
—the template object and the to_slot respectively. The remove method will remove
—three model dependency edges: 1) a (SIBLING, C, v) edge from B.w to C.v, 2) a
—(CHILD, C, v) edge from A.B to C.v, and 3) a (SELF, v) edge from C.parent to C.v.
Method Sibling_Model_Edge.remove(Object template, Slot to_slot, Signature sig)
    —Get C's parent, A
    parent ← template.parent
    —First, remove the (SIBLING, C, v) edge that goes from vertex B.w to vertex C.v.
    —This edge is found in B's model supertree. self.sibling_name
    —and self.slot_name refer to fields in the sibling
    —parameter edge, which contain the values B and w in this example.
    graph ← parent.model_supertree[self.sibling_name]
    graph[self.slot_name].delete_sibling_edge(template.part_name, to_slot, sig)
    —Next, delete the (CHILD, C, v) edge that goes from vertex A.B to vertex C.v.
    —This edge is found in A's model subtree.
    graph ← parent.model_subtree
    graph[self.sibling_name].delete_child_edge(template.part_name, to_slot, sig)
    —Finally, delete the edge (SELF, v) that goes from vertex C.parent to C.v.
    —This edge is found in C's model subtree.
    graph ← template.model_subtree
    graph[parent].delete_self_edge(to_slot, sig)
```

Fig. 24. Removal method for the `Sibling_Model_Edge` type. The `delete_sibling_edge`, `delete_child_edge`, and `delete_self_edge` methods search for an edge of the specified type and formula signature and delete this edge if it is found.

and for each one whose `needs_parent` field is set, calling each of the parameters' `install` methods.

(4) A part is removed from a prototype. As noted earlier, when a part is removed from a composite object, any model dependencies that point from parent or sibling slots to slots within this removed part are now "dangling." If the part is removed from a prototype, then the corresponding part will be removed from all the instances as well. Hence these dangling dependencies are no longer needed and should be removed from the parent's model supertree graphs. Since these dependencies are created by PARENT or SIBLING parameters, the part removal algorithm can simply scan the removed part's constraints and for each constraint whose `needs_parent` field is marked true, call each parameter's `remove` method. The part's model subtree graph is still valid so it does not have to be modified. For this reason the part removal algorithm ignores constraints whose `needs_parent` field is marked false.

B.2 Modeling a Prototype's Dependencies After it is Instanced

The prototype-instance model allows one additional, useful extension to the model dependency scheme. This extension is based on the observation that once a prototype P has been instanced and a model dependency graph for its instances has been created, P no longer needs to use its prototype's model dataflow graph to derive its own model dependency information; it can use the model dataflow graph created for its instances.

Implementation. When an object `obj` is instanced for the first time, a model dependency graph G for `obj` is created and two additional steps are performed:

(1) `obj`'s model dataflow graph pointer is redirected from its template's model dataflow graph to the newly created graph G, and

(2) all explicit dependencies pointing to model constraints within `obj` are removed, since many of them are now modeled by G. The constraints are then invalidated so they will be re-evaluated and will re-establish explicit dependencies where necessary.

Note that this effort is expended at most once for each object, when that object is instanced for the first time.