

To the Graduate Council:

I am submitting herewith a dissertation written by Richard L. Halterman entitled "Reducing the Storage Requirements of Dataflow Constraints Using Model Dependencies." I have examined the final copy of this dissertation for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy, with a major in Computer Science.

Bradley T. Vander Zanden
Bradley T. Vander Zanden, Major Professor

We have read this dissertation
and recommend its acceptance:

M. L.
Wayne Claycomb
Jim L. P. R.

Accepted for the Council:

Lawminkel
Associate Vice Chancellor and
Dean of the Graduate School

REDUCING THE STORAGE REQUIREMENTS OF DATAFLOW CONSTRAINTS
USING MODEL DEPENDENCIES

A Dissertation
Presented for the
Doctor of Philosophy
Degree
The University of Tennessee, Knoxville

Richard L. Halterman
May 1999

Acknowledgments

My experience at the University of Tennessee has been extremely valuable. In particular I would like to thank Dr. Brad Vander Zanden, my advisor. He has been the ideal mentor, teacher, and friend. His insights into how to do research and how to produce a formal paper, mixed with his wise direction and unfailing patience, have contributed to a rich experience that has far exceeded my expectations for doctoral research.

I would also like to thank my Dissertation Committee members, Drs. Wayne Claycombe, Michael Langston, and James Plank for their support.

Finally, I express the deepest gratitude to my wife Janet and my daughters Jessica and Rachel. They have weathered the experience well, and their help and encouragement have made this moment possible.

Abstract

Dataflow constraints allow programmers to specify relationships among application objects in a natural, declarative manner. Most constraint solvers represent these dataflow relationships as directed edges in a dependency graph. Unfortunately, dependency 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. The solution presented here is based on the observation that objects derived from the same prototype use the same constraints and thus have the same dependency graphs. The common dependency patterns are represented in a model dependency graph that is stored in a prototype. 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.

Table of Contents

1	Introduction	1
2	Background	5
2.1	Composite Objects	5
2.2	Constraints	8
2.3	Prototype-Instance Model	16
3	The Model Dependency Paradigm	18
3.1	Types of Model Dependencies	20
3.2	Model Parameters	23
3.3	Formula Signatures	26
3.4	Limitations of Model Dependencies	28
3.5	Coexistence with Explicit Dependency Constraints	30
3.6	Editing Model	31
4	Implementation	33
4.1	Basic Data Types	35
4.2	The Model Edge Type	38
4.3	Division of Responsibility	42

4.4	Model Dependency Installation	44
4.5	Resolution of Literal Dependencies	51
4.6	Constraint Installation	55
4.7	Edits After an Instance has been Created	58
4.8	Conditional Formulas	65
4.9	Results	66
5	Extensions to the Basic Scheme	71
5.1	Mixed Dependency Formulas	72
5.2	Modeling a Prototype's Dependencies After it is Instanced	74
5.3	TO_ALL_CHILDREN Model Dependency	76
5.4	FROM_ALL_CHILDREN Model Dependency	81
5.5	Model Dependencies for Global Objects	89
5.6	Additional Extensions	92
5.7	Time Complexity of Model Dependency Resolution	95
5.8	Spurious Invalidations	98
6	Empirical Test Results	99
6.1	Performance on the Dense Dependencies Benchmark	99
6.2	Interactive Performance	104
7	Related Work	106
7.1	Dataflow Constraints in Context	106
7.2	Dataflow Constraint Optimization	109
7.3	Influencing Ideas	114
8	Conclusions	115
8.1	Future Work	115

TABLE OF CONTENTS

vi

8.2 Conclusions	119
Bibliography	121
Vita	127

List of Tables

2.1	Definition of the Slot fields and methods used by Algorithm 2.1	12
2.2	Definition of the Constraint fields and methods used by the algorithms in Algorithm 2.1	12
3.1	Model dependency edges produced from model parameter edges	24
3.2	Model dependency edge installation for the four fundamental edges	27
4.1	Definition of Object fields and methods used by the algorithms in Chapter 4	36
4.2	Definition of Slot fields and methods used by the algorithms in Chapter 4	36
4.3	Definition of Formula fields used by the algorithms in Chapter 4	37
4.4	Definition of Constraint fields and methods used by the algorithms in Chapter 4	37
4.5	Model dependency edge installation for the four fundamental edges	48
4.6	Applications used for evaluating model dependencies	67

List of Figures

2.1	A labeled box object and its structural components	6
2.2	Labeled box constraint equations and constraint graph	9
2.3	Code associated with a constraint	11
2.4	An application with multiple constraints per slot	15
3.1	An application that uses many labeled boxes	19
3.2	Model edge relationships	21
3.3	Model dependency resolution for <code>box.label.width</code>	22
3.4	Visualization of Table 3.1	25
3.5	Centering an arrow's tail within an operand node	29
3.6	A slot with both explicit and modeled dependents	31
4.1	The data structures implementing a labeled box object	39
4.2	The class hierarchy for model edges	40
4.3	Subtree vs. supertree relationships	43
4.4	Sibling relationship	43
4.5	The labeled box's model constraint graphs	45
4.6	Replacing a text part with a bitmap object	63
4.7	Application snapshots	68

4.8	Modeled results for a cross section of Amulet applications	70
5.1	The reductions of dependencies when mixed dependency formulas are used	75
5.2	The reductions of dependencies when the prototype's dependencies are modeled	77
5.3	A grid object implemented as a map object	78
5.4	A typical drop-down menu implemented as a map object	80
5.5	The TO_ALL_CHILDREN relationship	80
5.6	The reductions of dependencies including TO_ALL_CHILDREN	83
5.7	The FROM_ALL_CHILDREN relationship	85
5.8	The collection of subgraphs that constitute a model dependency graph	87
5.9	The reductions of dependencies including FROM_ALL_CHILDREN	88
5.10	A situation where all objects depend on the root of the composition hierarchy	90
5.11	The reductions of dependencies contributed by all the extensions	93
5.12	The average percentage reductions of dependencies contributed by the extensions	94
5.13	The extended class hierarchy for model edges	96
6.1	Memory requirements of objects with dense dependencies	102
6.2	Object manipulation times for the <i>Dense Dependencies</i> benchmark	103
7.1	Comparison of μ constraints to Model Amulet	112

List of Algorithms

2.1	Methods used to implement the mark-sweep algorithm	13
4.1	Installation method for the Self_Model_Edge type	47
4.2	Installation method for the Sibling_Model_Edge type	49
4.3	Model constraint graph creation algorithm	50
4.4	Invalidation algorithm for a constraint solver that incorporates model dependencies	52
4.5	Resolution method for the Model_Dependency_Edge type	53
4.6	Algorithm that permits the constraint system to install explicit dependencies	57
4.7	Modified Slot.get() that permits the constraint system to install explicit dependencies conditionally	57
4.8	Constraint inheritance algorithm	59
5.1	Algorithm that permits the constraint system to install explicit dependencies	73
5.2	The To_All_Children_Model_Edge resolve method	82
5.3	The From_All_Children install method	86
5.4	Algorithms to install and resolve Global_Model_Edge dependencies	91
5.5	Modified get_model_dependencies() method	93

Chapter 1

Introduction

Direct manipulation interfaces play an important role in modern software systems. These interfaces permit a user to interact with underlying application objects by manipulating graphical objects on the screen. For example, computer-aided design (CAD) software enables engineers to build and test designs for electronic and mechanical devices by assembling visual components on a computer screen. Multiple views of these simulated devices highlight different properties that can be used to evaluate the overall design. As another example, the enormous amount of data generated by some scientific experiments and large simulations can be overwhelming. Interactive visualization software allows researchers to analyze the data more quickly by graphically representing multiple views that display various aspects or interpretations of the data. These types of highly interactive, direct manipulation interfaces make powerful, complex software easier to use by nonprogrammers. For software developers, however, these graphical interfaces are difficult to implement because the program code required to keep multiple views consistent with the application's objects is tedious to write and maintain. As applications become more ambitious, software techniques that ease this development process become increasingly valuable.

Dataflow constraints have proven to be one such technique. A dataflow constraint (also called a one-way or a spreadsheet-style constraint) is an equation of the form $v = f(p_1, \dots, p_n)$, where the function on the

right hand side is re-evaluated each time one of its arguments changes, and the result is assigned to the left hand side variable. Dataflow constraints allow programmers to specify relationships among programming objects in a natural manner. For example, in a graphical interface they can be used to center a label within a textbox or to keep the size of a bar in a histogram consistent with a piece of data in an application. In a syntax-directed editor they can be used to specify type checking rules or to determine whether or not a variable has been declared. In object systems, dataflow constraints are attached to instance variables and used to compute the values of these variables. Constraints are especially useful in object systems since they can be written once, installed in a prototypical object, and then inherited by all instances derived from that prototype.

Without constraints, programmers must explicitly write bookkeeping code to maintain the relationships among programming objects. This process is tedious as the required code tends to be dispersed throughout the program. For example, in order to keep a label centered in a box, the box must have some code that notifies the label whenever the box is moved. Similarly the label must have some code that notifies the box if the label is destroyed, so that the box can remove the label from its notification list. The non-constraint approach is also inherently non-modular because the responsibility for maintaining a relationship is spread among several objects (for example, the box and the label). Consequently, the code is more difficult to develop and maintain. Constraints simplify this process because:

- A constraint solver assumes the tedious bookkeeping task of maintaining the relationships. None of the objects participating in the relationship needs to contain code for maintaining the relationship.
- A relationship is declared in only one place, in the object that desires the relationship. This approach is more modular because only the object that desires the relationship knows about it. The objects on which the constraint depends do not need to know about the relationship or alter their code in any way to maintain the relationship.

This declarative nature of constraints makes the development of highly interactive graphical interfaces easier for the applications programmer. As a result, constraints are increasingly being integrated into

drawing packages and interface development toolkits [29, 30, 15, 14, 19, 18, 19, 40], circuits [1], spreadsheets [27, 20], and programming environments [31].

Despite their facility and usefulness, constraints are not widely used by commercial software developers. Studies of at least two interface development toolkits, Garnet [29] and Amulet [30], have shown that constraints can exact a significant storage toll on programs [43]. Ultimately the execution times of programs managing a large number of constrained objects suffer since virtual memory must be used to meet their storage demands.

One reason constraints require so much storage is that most dataflow constraint systems explicitly represent the relationships among variables and constraints by maintaining a *dependency graph*, or *constraint graph* (see Section 2). The edges in these graphs can consume a considerable amount of storage. The solution to the dependency problem presented here is based on the observation that objects that use the same constraints have the same dependency graph. Consequently, one can store a pattern of a constraint subgraph, a *model dependency 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. Our experiments show that over 75% of the explicit dependencies in most applications can be eliminated with model dependencies (see Chapter 6).

Although a model dependency will be slower to follow than an explicit dependency, the tradeoff of time for space is a good one since typically the number of dependencies affected per object changed is small [43], and the time spent traversing dependencies is a tiny fraction of the time required to execute an operation (the redisplay time dominates the cost of executing an operation in constrained applications [14, 38]). Also, as the number of explicit dependencies becomes very large, the access to virtual memory required for their storage can significantly degrade a program's performance, thereby swinging the performance advantage to model dependencies.

The remainder of this dissertation is organized as follows. Chapter 2 presents some background material on composite objects, dataflow constraints, and the prototype-instance model. Chapter 3 introduces

the basic model dependency paradigm that addresses the dependency storage problem of dataflow constraints. Chapter 4 describes how the model dependency solution was implemented. Extensions to the basic scheme are covered in Chapter 5. Chapter 6 evaluates the effects of model dependencies and analyzes how effectively they reduce the number of explicit dependencies, and Chapter 7 reviews related research in constraints and constraint optimization. Chapter 8 discusses directions for future work motivated by our experience and concludes.

Chapter 2

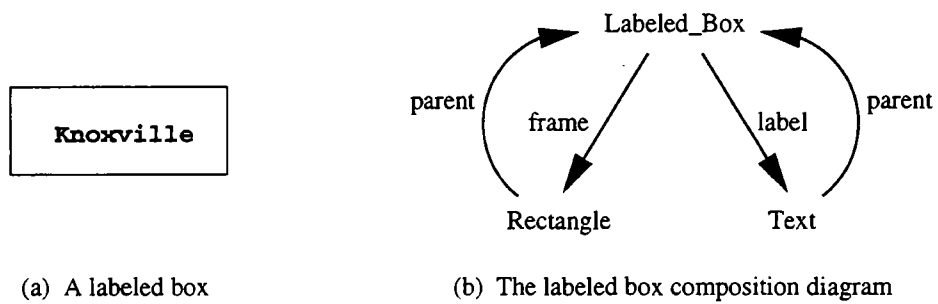
Background

This chapter describes the key concepts used in this dissertation, including composite objects, constraints, and the prototype-instance model.

2.1 Composite Objects

The constraints described in this dissertation are implemented in the context of composite objects. An *object* is a collection of related data organized into named *attributes*, also called *instance variables*. The values of an object's attributes define the object's *state*. In general an object's state is examined and/or modified via *methods*. Methods are procedures that operate on the object's instance variables.

In this dissertation two distinct kinds of objects are mentioned. Of primary interest are *application objects*, the objects created and manipulated by applications programmers. The labeled box of Figure 2.1 (a) represents a typical application object. An attribute of an application object is called a *slot*. The other kind of object is an *implementation object*. A system implementer uses lower-level implementation objects to build the infrastructure upon which higher-level application objects can be used. Implementation objects are used in the algorithms found in later sections. To better distinguish between the two kinds of objects, an attribute of an implementation object is called a *field*. The main distinction between the two kinds



(a) A labeled box

(b) The labeled box composition diagram

Figure 2.1: A labeled box object (a) and its structural components (b)

of objects is that slots of application objects can be constrained, but fields of implementation objects are simply standard data. As will be seen, a slot has a more sophisticated role than simply storing data. The term *object*, when used by itself, will refer to an application object.

A *composite object* is an application object made up of parts consisting of other application objects [12]. For example, Figure 2.1 (a) illustrates a simple composite object, a labeled box consisting of a text string enclosed within a rectangle.

Each component of the labeled box has left, top, width, and height slots. The label part also has a text string and font slots. The labeled box has named pointers to its children (frame and label) and the children have named pointers to their parent (parent). (See Figure 2.1 (b).) 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 terms *child*, *part*, and *component* are synonymous; likewise, the terms *parent* and *owner* are often used interchangeably. Parts that have a common parent are called *siblings*.

The “dot” notation is often used to reference slots and components of an application object. For example, if *box* is a labeled box object, then *box.frame* refers to its frame component, *box.left* refers to the box’s left slot, and *box.frame.left* refers to the left slot of the box’s frame part. The dot notation is also used for method calls; for example, *obj.draw()* calls the *draw()* method for object *obj*. Methods are distinguished from slots by trailing parentheses; in an actual call the parentheses may contain parameters. The same dot notation is used in the algorithms to reference fields and methods of implementation objects.

Composite objects are useful in large applications since their hierarchical structure allows programmers to build sophisticated components with much of the details of their implementation, such as which parts they are composed of, hidden from the application. Messages (that is, method calls) directed to a composite object can be delegated to subcomponents as necessary. Composite objects are therefore valuable for representing real world objects in applications such as simulations and components in graphical interfaces.

2.2 Constraints

Definition. A *one-way dataflow constraint* is a formula in which the value of the variable on the left side is determined by the value of the expression on the right side. For example, the frame of the labeled box in Figure 2.1 should be slightly larger than the text label, and the label should be centered within the frame. Figure 2.2 (a) shows the constraints that specify these relationships.

More formally, a one-way constraint is a formula of the form

$$v = C(p_0, p_1, p_2, \dots, p_n)$$

where C is an arbitrary function, p_0 through p_n are the parameters to this function, and v is the variable to which the function's result is assigned. $v, p_0, p_1, \dots, p_n$ are all slots of one or more objects. If the value of any p_i is changed during the program's execution, v 's value is automatically recomputed. Note that v has no reciprocal influence on any p_i as far as this constraint is concerned; hence, it is *one-way*. For example, if labeled box's left slot is modified, then the value of label's left slot will be automatically recomputed, but label.left has no influence on the labeled box object's left slot (hence the data flow is in one direction only).

Since only one variable may appear on the left side of the equation, these constraints are *single-output* formulas. Multi-way, multi-output constraint systems are briefly described in Section 7.1. Unless otherwise noted, the term *constraint* as used in this dissertation refers to a one-way dataflow constraint.

Dependency Graphs. The network of slots and constraint implementation objects and their associated relationships form a bipartite *dependency graph*, or *constraint graph*. The vertices of a dependency graph are constraints and slots. A slot s has a directed edge to a constraint if the constraint uses s as a parameter. This constraint is called a *dependency* of s . A constraint has a directed edge to a slot if the constraint computes that slot's value. Since only single-output constraints are being considered here each constraint vertex in the graph has exactly one outgoing edge. Many systems also maintain backward edges that point from constraints to their input slots. These backward edges allow dependencies to be removed if a constraint is deleted. Finally, the set of slots that provide input to a constraint are known as *parameters* to

label:

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

C_1 `self.width = self.compute_width(self.text,
self.font)`

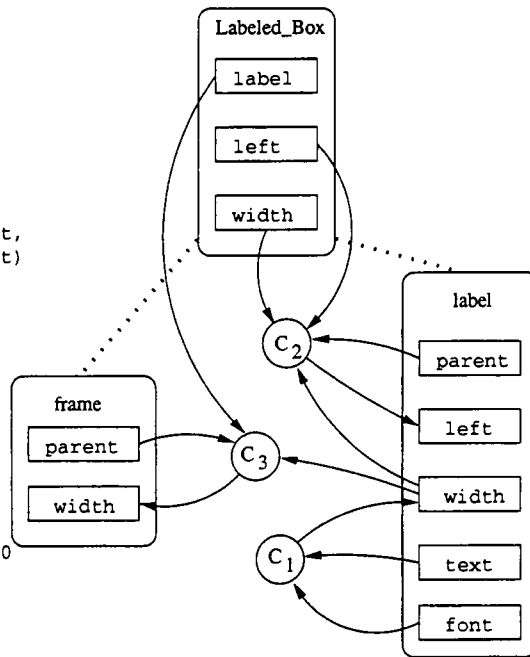
--Label is centered horizontally within its owner

C_2 `self.left = self.parent.left
+ self.parent.width/2
- self.width/2`

frame:

--Frame is slightly wider than the label it encloses

C_3 `self.width = self.parent.label.width + 10`



(a) Constraint equations

(b) Constraint graph

Figure 2.2: Labeled box constraint equations and constraint graph. Here formulas controlling the horizontal layout are shown. The corresponding vertical formulas defining the top and height slots of the various objects are omitted. Notice that constraint C_2 requires a dependency from `label.parent` since the `label.parent` slot must be accessed to read `label.parent.width`. This means that C_2 will be recomputed if the label's parent changes (that is, the label part is removed and then added to a new composite object). Similarly, `frame.parent` affects C_3 since if the frame's parent changes, the frame must recompute its width from information in its new sibling. `Labeled_Box.label` also influences C_3 since if the labeled box replaces its label child, C_3 must be notified so `frame.width` can be recomputed.

that constraint. Figure 2.2 (b) shows the constraint graph for the labeled box equations in Figure 2.2 (a) (backpointers are omitted).

Code Associated with Constraints. In order for a constraint to compute its result, executable code is associated with each constraint object. Figure 2.3 shows a function, written in C++, that might be used to compute constraint C_2 from Figure 2.2. Notice that since the constraint code is implemented as a function, the same code can be used by different constraint objects in different contexts. For example, this function can be used to center any child within its parent.

Constraint Satisfaction. In this dissertation a *mark-sweep* algorithm is used for constraint satisfaction. A mark-sweep algorithm has two phases:

1. a *mark* phase in which all constraints that depend on a changed slot are marked invalid, and
2. a *sweep* phase in which constraints are brought up to date by evaluating their formulas [8, 31, 16, 40].

Algorithm 2.1 formalizes the mark-sweep procedure. Tables 2.1 and 2.2 define the fields and methods of the Slot and Constraint objects used in the mark-sweep algorithm.

The mark phase is initiated when a slot is set by the application. This phase performs a depth-first traversal of the dependency graph, visiting all vertices reachable from the changed slot. The mark phase is of primary interest in this dissertation since it uses the dataflow edges that can potentially be modeled.

The sweep phase can be implemented in one of two ways. First, constraints can be re-evaluated as soon as the mark phase is completed.¹ Such a strategy is called *eager evaluation*. Under this approach the dependent slot is always up to date. An alternative approach defers the constraint recomputation until the value is needed. Called *lazy evaluation*, this approach is more attractive in situations where many constraints are invalidated in the mark phase but only a few of the slots determined by these constraints are of immediate use to the application. An example of this would be a large CAD diagram in which only one part of the design is visible in a window at any given time. The slots that maintain the state of the invisible

¹The terms *(re)evaluate*, *bring up to date*, *(re)compute*, and *validate* are synonymous. They all refer to the evaluation of a constraint.

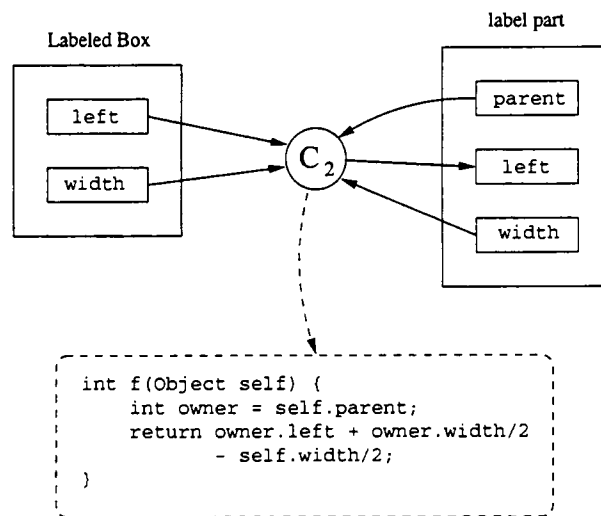


Figure 2.3: Code associated with a constraint. This figure shows a possible C++ function that could be associated with constraint C_2 from Figure 2.2. The solid edges represent the dataflow edges. The dashed edge represents a pointer to the formula function. The `self` parameter is the object containing the slot to be computed. By encapsulating the formula's code in a function and passing the object as a parameter, multiple constraints can use the same code.

Table 2.1: Definition of the Slot fields and methods used by Algorithm 2.1.

Slot Fields and Methods	
Field	Meaning
value	the value of the slot
object	the object that owns the slot
valid	true, if the slot's constraint is up to date; otherwise, false
dependents	a pointer to the set of constraints that depend on the slot
constraint	the constraint that computes the value of the slot
Method	Meaning
set()	sets the slot's value field
get()	retrieves the slot's value field
invalidate()	marks the slot invalid and each dependent constraint invalid (see Algorithm 2.1)

Table 2.2: Definition of the Constraint fields and methods used by the algorithms in Algorithm 2.1.

Constraint Fields and Methods	
Field	Meaning
valid	true, if the constraint is up to date; otherwise, false
slot	a pointer to the slot computed by the constraint
formula	a pointer to the code that computes the constraint's value
Method	Meaning
invalidate()	marks the constraint invalid and also the slot it computes invalid (see Algorithm 2.1)
evaluate()	calls the constraint formula

Methods used during the **mark** phase:

Method Slot.set(Value new_value)

```

1  self.value ← new_value
   —Prevent unnecessary invalidation if there is a cycle
2  self.valid ← false
   —All dependent constraints must be notified of the change
3  for each constraint cn ∈ self.dependents :
4      cn.invalidate()
5  self.valid ← true

```

Method Constraint.invalidate()

```

1  if self.valid = true :
2      self.valid ← false
3      self.slot.invalidate()

```

Method Slot.invalidate()

```

1  if self.valid = true :
2      self.valid ← false
3      for each constraint cn ∈ self.dependents :
4          cn.invalidate()

```

Method used with lazy evaluation during the **sweep** phase:

Method Slot.get() returns Value

```

1  if self.valid = false :
2      self.valid ← true
3      self.value ← self.constraint.evaluate(self.object)
4  return self.value

```

Method Constraint.evaluate(Object obj) returns Value

```

1  self.valid ← true
   —Pass the object to the function that computes this constraint's value
2  return self.formula(obj)

```

Algorithm 2.1: Methods used to implement the mark-sweep algorithm. The fields and methods of the Slot and Constraint implementation object types are described in Tables 2.1 and 2.2. The keyword **self** is a pointer to the object on which the method is invoked.

portion of the diagram may not need to be updated until that portion becomes visible. This strategy can be especially beneficial if the formulas are computationally expensive to re-evaluate.

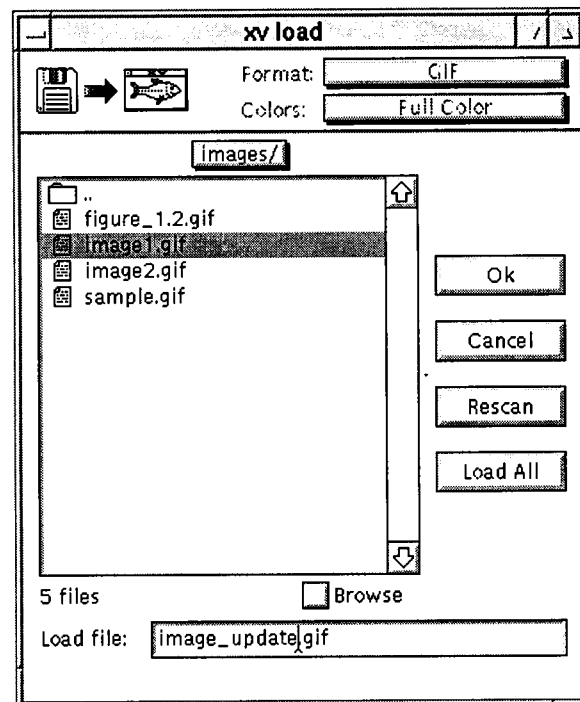
While other constraint satisfaction algorithms such as topological ordering [40] have been used for dataflow constraints, the mark-sweep algorithm is typically the easiest algorithm to implement, and within the context of graphical interfaces, the most efficient as well [17, 42].

To illustrate the constraint satisfaction process, consider the constraint formulas and graph of Figure 2.2. Suppose the user edits the text string within a labeled box by adding more characters to the label. Changing `label.text` invalidates constraint C_1 , which in turn invalidates `label.width` (the label's graphical width is enlarged as a result). `label.width` in turn 1) invalidates constraint C_3 which invalidates `frame.width` and 2) invalidates constraint C_2 which invalidates `label.left`. Now assume the value of `frame.width` is requested. `frame.width` would consult its constraint, see that C_3 is invalid, and call C_3 's formula. C_3 's formula requests the value of `label.width`, which is out of date. Hence, `label.width` calls C_1 's formula which brings both C_1 and `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 this execution model ensures that constraints are evaluated in topological order.

Multiple Constraints Associated with One Slot. Some constraint systems, like Amulet [30] and Rendezvous [14], permit a slot to maintain a collection of constraints, and exactly one of these constraints will determine the slot's value at any given time. In other words, a slot may be on the left side of more than one equation. As an example of the usefulness of multiple constraints, consider a graphical interface providing an open file dialog box as shown in Figure 2.4. The user specifies the file to open either by:

1. selecting the name with a pointing device from a list of files, or
2. typing the file name into an editing field at the bottom of the dialog box.

The application uses a variable named `filename` to store the name of the file to open and sets this variable based on the two kinds of user input. The constraint C_1 computes the `filename` variable based on which list item is selected; C_2 determines the `filename` based on the text typed into the edit field. If the



C_1 : application.filename = f(file_dialog.list.index)
 C_2 : application.filename = g(file_dialog.edit_field.text)

Figure 2.4: An application with multiple constraints per slot. The file to open can be selected with a pointing device from a list of files, or its name can be typed directly into an edit field at the bottom of the dialog box. Constraint C_1 determines the application's filename variable by using the index to retrieve the appropriate file from the list of files. C_2 determines filename directly from the text typed into the edit field.

user selects a list item, then C_1 will be invalidated; on the other hand, if the user types into the edit field, then C_2 will be invalidated.²

When an invalid slot's value is demanded, it polls its list of constraints to see which ones are invalid. These invalid constraints are evaluated and brought up to date. Some rule chosen by the implementer of the constraint system is then used to select which of the recomputed constraints is used to determine the value of the slot. One such rule could be to choose the first invalidated constraint.

2.3 Prototype-Instance Model

The prototype-instance model has been successfully used in a number of toolkits and languages, including ThingLab [3], SELF [7], Garnet [29], and Amulet [30]. This model facilitates rapid prototyping of graphical applications because the look and feel of all of prototype's instances can be dynamically changed by modifying the prototype. Thus a developer can experiment with multiple looks and feels without shutting down the application and recompiling.

In the prototype-instance model any object can be instanced to create a new object that inherits all the properties of the original object [5, 24]. The original object is known as the *prototype*, and the new object is an *instance* of that original object. All slots and parts associated with the prototype are inherited by the new instance. In a prototype-instance system that implements constraints, constraints are also inherited. For example, when a new instance of the labeled box in Section 2.2 is created, the slots, constraints, and parts are copied into the new instance, and a dependency graph is created for the new instance. A constraint is itself treated like a prototype; when a constraint is instanced, a new constraint object is created. The constraint prototype and its instance share the same formula, but they must be distinct objects since each must keep track of its own valid status.

The prototype-instance model differs from the class-instance model used in C++ and JavaTM. In the

²This example assumes that the filename variable is immediately updated after each interaction (that is, each item selection or each text entry). Otherwise, multiple interactions with both input techniques *before* the Ok button is pressed might result in an indeterminate value of filename.

class-instance model the *class* construct, fixed at compile time, serves as a blueprint from which instances are created at run time. Instance slots and components are normally fixed at compile time; they are statically specified by the class. In the prototype-instance model new slots and parts can be added to and removed from instances at run time. Prototypes are first-class objects that are instantiated at run time with actual data; in contrast, classes are static artifacts that cannot be manipulated at run time.

Chapter 3

The Model Dependency Paradigm

This chapter describes the basic model dependency scheme that was devised for this dissertation. To motivate it, reconsider the labeled box example from Chapter 2. Figure 3.1 shows a visual arithmetic editor that uses labeled box objects. In this editor, labeled boxes are used to represent operand vertices. One labeled box object is created and used as a prototype to generate all the operand vertices shown in the figure. As noted in Section 2.3, an instance object inherits a prototype's constraints, and by extension, its dependency graph. In a conventional constraint system, storage for this dependency graph is duplicated for each new instance. Each labeled box in the application therefore has its own constraint graph isomorphic to the graph shown in Figure 2.2 (b).

Since every labeled box shares this common graph, a more storage efficient scheme is to store a *model dependency graph* in the prototype object from which all the instances were created. The instances can then consult this graph to derive information about their own dependencies. Unlike a conventional constraint system, they do not maintain explicit dependencies. Instead, the dependencies are implicit.

In general, the model dependency scheme works as follows. Let P be a prototype object with a set of constraints. The first time an instance of P is created, the system examines P 's constraints and creates a model dependency graph. For each slot in P , this model dependency graph contains information that allows

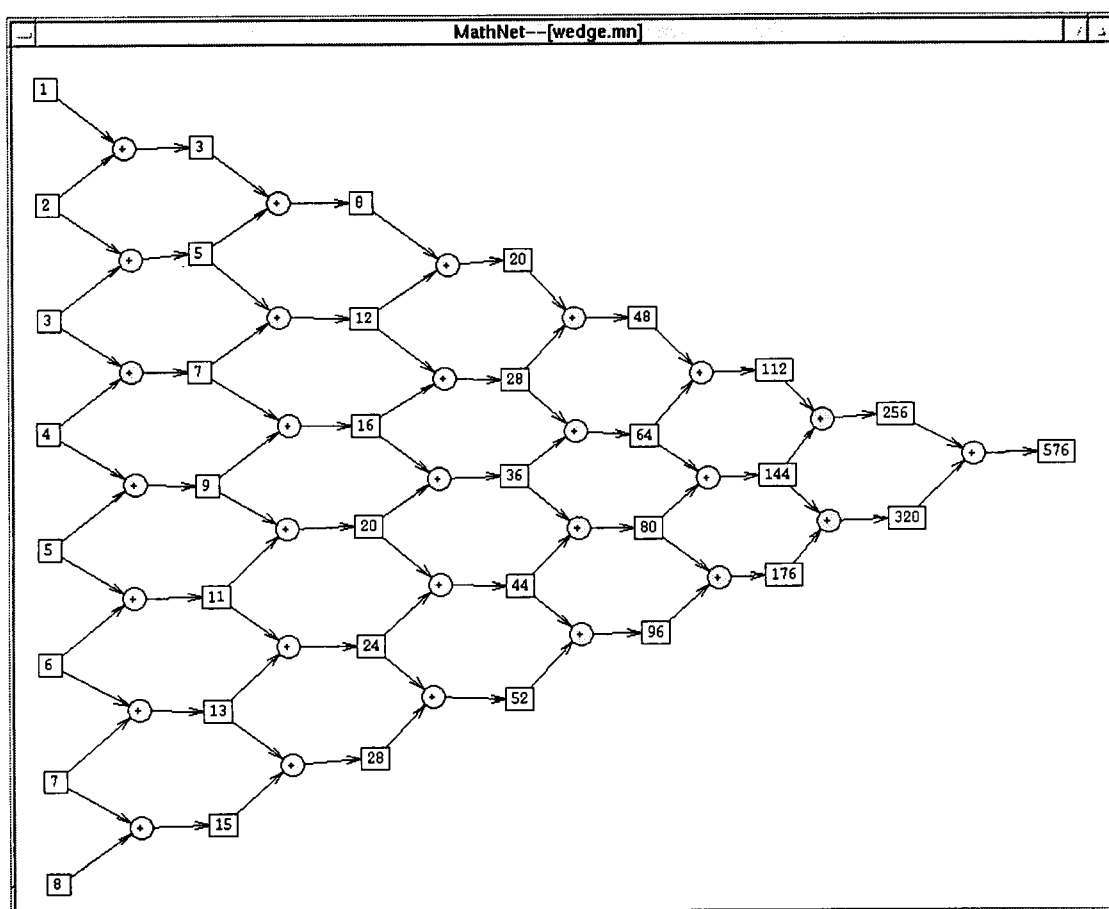


Figure 3.1: An application that uses many labeled boxes.

the constraint solver's mark algorithm to determine the constraints that depend on this slot (the details are described in Chapter 4). When a slot in an instance I is changed, the constraint algorithm consults the model dependency graph in I 's prototype (P). The model dependencies are used to find the constraints that depend on the changed slot, and these constraints are marked invalid. The slots associated with these constraints are found, and in a similar manner their model dependencies are used to invalidate additional constraints. The process continues until all constraints and slots directly and indirectly dependent on the original changed slot have been invalidated.

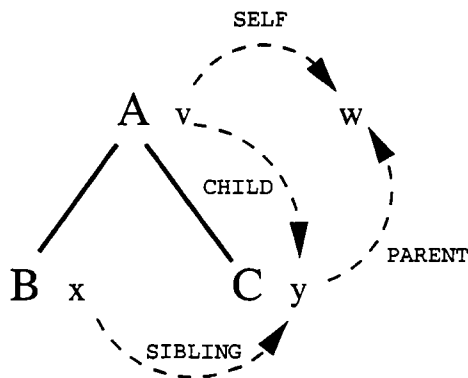
This scheme is particularly valuable in applications that generate many instances from only a few prototypes such as the application shown in Figure 3.1. In addition to the labeled boxes, the operator vertices and edge arrow structures could also store their common dependency information in their respective prototypes. It will be shown that the space savings for an application managing a large number of constrained objects can be considerable.

The technique of creating a model dependency graph and using this graph to deduce literal dependencies assumes that a *reflection* [2] mechanism is available. Reflection allows an application to iterate through the slots of an object and find the object's constraints. However, in a system that does not provide a reflection mechanism, the programmer could manually specify the set of constraints in an object, for example, by passing them as parameters to a function.

3.1 Types of Model Dependencies

We chose to allow four relationships to be expressed with model dependencies: SELF, CHILD, PARENT, and SIBLING. These four relationships were chosen after a survey of applications showed that over 50% of all constraint formulas use these four relationships exclusively. Figure 3.2 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



SELF	represents a dependency from one variable to another within the same object
PARENT	represents a dependency from a variable in a part to a variable in its owner
CHILD	represents a dependency from a variable in an object to a variable in one of its parts
SIBLING	represents a dependency between a variable in one part to a variable in another part where both parts have a common owner

Composite object A has variables v and w and parts B and C.
Object B has variable x, and object C has variable y.

Figure 3.2: Model edge relationships

edge, a dependent slot can be resolved as shown in Figure 3.2. For example, the A.v slot can follow a (SELF, w) edge to A.w or a (CHILD, C, y) edge to C.y. The (PARENT, w) edge directs C.y to A.w. The (SIBLING, C, y) edge defines a path from B.x to C.y.

To illustrate how the mark algorithm uses model dependencies, reconsider the labeled box example. label.width would have the model dependency edges

$\{(SELF, left), (SIBLING, frame, width)\}$,

which are derived from the formulas defined for constraints C_2 and C_3 . Now suppose that box is a labeled box object and that box.label.width is modified. The mark algorithm then determines the constraints to invalidate as follows (see also Figure 3.3):

1. The first model dependency edge, (SELF, left), is considered. As shown in Figure 3.2, a SELF edge references a slot in the same object as the modified slot. Since box.label contains the modified slot, the (SELF, left) edge causes box.label.left to be visited.
2. The constraint responsible for computing box.label.left, C_2 , is then invalidated, as is box.label.left.
3. The second model dependency edge, (SIBLING, frame, width), is then considered. As shown in

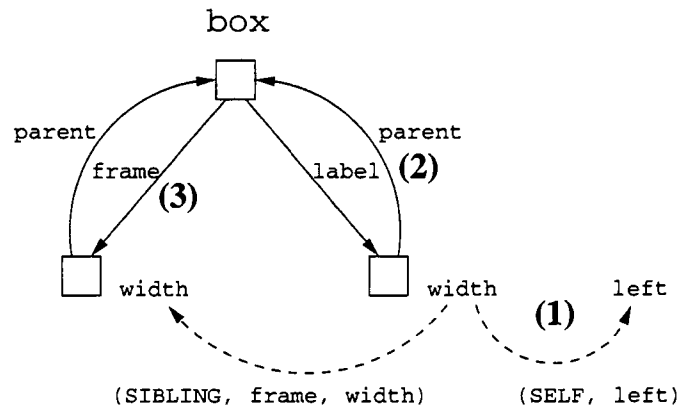


Figure 3.3: Model dependency resolution for `box.label.width`. The SELF dependency resolves to a slot in the same object (1). A SIBLING dependency is resolved by following the parent pointer (2) to the parent, in this case `box`, and then following the sibling pointer specified in the model dependency (3), in this case `frame`. Finally within `frame`, the `width` slot is located and invalidated, along with the appropriate constraint.

Figure 3.2, a SIBLING edge points to a slot in a sibling part, in this case frame. Hence `box.frame.width` is visited.

4. The constraint responsible for computing `box.frame.width`, C_3 , is then invalidated, as is `box.frame.width`.

The process of determining the actual dependent constraint from a slot and a model dependency is called *resolution*.

3.2 Model Parameters

Model dependencies are automatically generated from *model parameter edges* using a process described in Section 4.4. For each model dependency formula, the application programmer provides a list of model parameters in the form of model parameter edges. A parameter edge specifies a path to each parameter used by a constraint. For a SELF parameter edge, the model dependency is formed by simply reversing the parameter's path. For the other relationships, a parameter edge will generate more than one model dependency edge. A PARENT parameter will produce a CHILD dependency in the parent and a SELF dependency from the PARENT slot in the part itself. Both dependencies are necessary because the slot that uses the formula with the PARENT parameter must be notified if 1) the parameter slot in its parent changes, or 2) if the parent is replaced (that is, the part 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 the dependent formula must be re-evaluated.

To illustrate, consider the labeled box constraints in Figure 2.2. The constraint C_2 (which centers the label within its parent) is attached to the left slot of the label part. C_2 has the parameter edge list $\{(PARENT, left), (PARENT, width), (SELF, width)\}$. These parameter edges generate the model dependencies shown in Table 3.1 and Figure 3.4. The parameter `self.parent.left` of constraint C_2 actually represents two separate parameters: $(SELF, parent)$ and $(PARENT, left)$. The former parameter allows

Table 3.1: Model dependency edges produced from model parameter edges. Shown here are the model parameters required for constraint C_2 in Figure 2.2. Figure 3.4 pictorially illustrates the relationships between C_2 's model parameter edges and the model dependency edges they generate.

Parameter edge	Model Dependency Edge
(PARENT, left)	Add (CHILD, label, left) to the left vertex in the model dependency graph for label's parent; add (SELF, left) to the parent vertex in label's model dependency graph.
(PARENT, width)	Add (CHILD, label, left) to the width vertex in the model dependency graph for label's parent. (The parent vertex will already have the proper (SELF, left) that was generated by the (PARENT, left) parameter.)
(SELF, width)	Add (SELF, left) to the width vertex in label's model dependency graph.

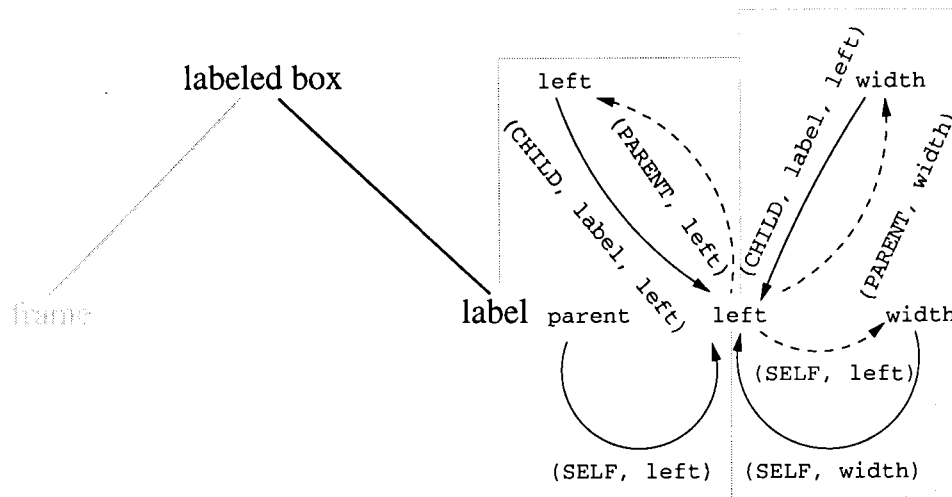


Figure 3.4: Visualization of Table 3.1 showing the model parameters and model dependencies of constraint C_2 in Figure 2.2. The dashed arcs represent model parameter edges. The solid arcs are the resulting model dependencies. The $(SELF, left)$ arc from `label.parent` is generated by both the $(PARENT, left)$ and $(PARENT, width)$ edges.

C_2 to recompute the label's left slot if the label part is removed from one labeled box and placed in another (in other words, its parent changes). The latter parameter would be insufficient to invalidate C_2 if label's parent slot changes. Hence the additional SELF dependency permits the model dependency scheme to invalidate a dependent slot properly when the parent slot changes.

A SIBLING parameter produces three model dependency edges: 1) the reciprocal SIBLING dependency, 2) a SELF dependency from the parent slot, and 3) a CHILD dependency from the *part* slot in the parent. The first dependency notifies the dependent slot when the parameter slot changes. The second dependency, the (SELF, parent) dependency notifies the dependent slot if its object is removed and placed within another composite object (that is, if the object's parent is replaced). The third dependency notifies the dependent slot if its sibling object is replaced with another object. The parent maintains slots for each of its named children. If a part is replaced, the corresponding part slot in the parent will change. Hence placing a CHILD dependency from the sibling's part slot to the dependent slot will cause the dependent slot to be notified if the sibling part is replaced.

A CHILD parameter will produce two model dependencies: 1) a reciprocal PARENT dependency and 2) a SELF dependency from the part slot in the parent. The SELF dependency ensures that the dependent slot will be notified if the child is replaced with a new part.

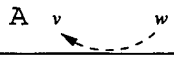
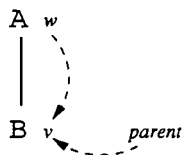
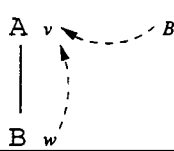
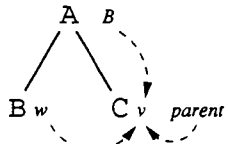
Table 3.2 outlines the installation process for each kind of model parameter.

3.3 Formula Signatures

The model dependency resolution process described in Section 3.1 first locates the dependent slot from the model dependency edge path and then invalidates the constraint that computes that slot. However, as mentioned in Section 2.2, some systems may allow multiple constraints to determine a slot. In order to determine which constraint to invalidate, each edge of a model dependency graph must store one piece of additional information that allows the proper constraint to be invalidated.

This is done by assigning each formula a unique identifier called a *formula signature*. This identifier

Table 3.2: Model dependency edge installation for the four fundamental edges. Parameter vertices are the vertices with which the model dependency edges are associated. Dependency edges are the model dependency edges generated from the parameter edge. B and C are children of A.

Variable to which formula is attached	Parameter edge	Parameter vertices	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) 

is in turn assigned to each constraint that uses that formula. The constraint's formula signature is stored with each of the model dependencies that its formula generates (that is, that are generated by its model parameter edges). When the invalidation routine reaches the appropriate dependent slot, it compares the formula signature in the model dependency edge with the formula signature in each of the slot's constraints. The constraint with the matching formula signature is invalidated.

The unique identifier can be a pointer to the formula's function or a pointer to some other unique aspect of the formula. The implementation described in Chapter 3 uses a pointer to a formula's parameter list since that is unique (two formulas with identical model parameters use separate model parameter edge lists).

3.4 Limitations of Model Dependencies

Although the current set of model dependencies including extensions to be described in Chapter 5 allows a programmer to express a wide variety of relationships, it does not suffice to express any arbitrary relationship. Composite objects as described in Section 2.1 provide natural navigation paths via their child and parent pointers. These paths can be traversed in either direction, which is crucial for model dependencies since a model dependency is generated by inverting a model parameter. Such two-way navigation paths do not exist between arbitrary objects. For example, consider the graphical arithmetic editor shown in Figure 3.1. Each operand node can use the labeled box model relationships to properly position and size its rectangular frame and text label. Similarly each operator node can lay out its circular frame and label. Each arrow edge can use model dependencies to align its arrowhead correctly with its shaft. All of these relationships are modelable since each can traverse a two-way navigational link.

Now consider the formulas that constrain the endpoints of each arrow edge. In particular, consider the formulas that center the (x,y) coordinate of an arrow's tail within an operand node. Figure 3.5 illustrates these formulas.

In this example, each arrow maintains a `from_node` slot that points to the object from which its tail should originate. (It also has a corresponding `to_node` that guides its tip.) The `from_node` slot references

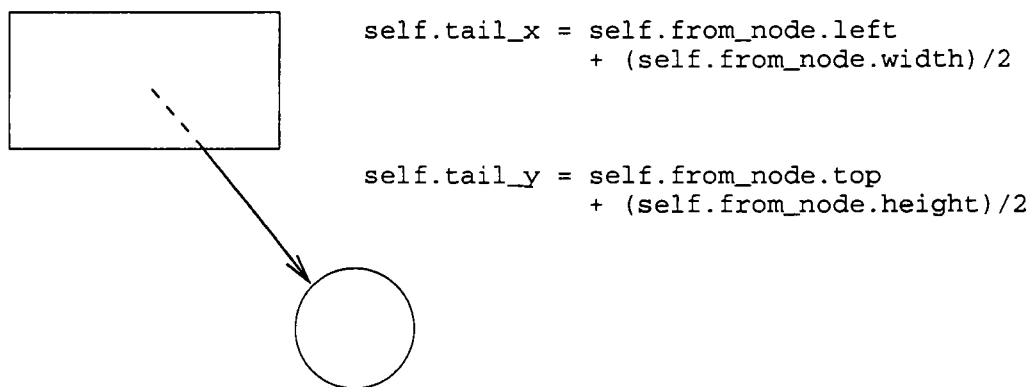


Figure 3.5: Centering an arrow's tail within an operand node. The two formulas position the (x, y) endpoint of the arrow's tail. The node is drawn on top of the arrow, so the part of the arrow underneath the node is not seen. This invisible part of the arrow's shaft is indicated with a dotted line.

an object that is not part of an arrow's composition hierarchy; this object is not a part, nor a sibling, nor a parent of the arrow object. Consequently, the operand and operator objects do not contain a reciprocal reference back to an arrow object; they are oblivious to arrows.

Since the reciprocal reference required to produce a model dependency edge from a model parameter edge is missing, a model dependency cannot be established. The missing backpointer could be artificially added to allow the dependency to be modeled, but the storage required for a backpointer is the same as the storage required for an explicit dependency. Traditional explicit dependencies provide the better solution here because they are faster—an explicit dependency provides a direct reference to the dependent constraint, whereas a model dependency must traverse the application object's structure to find the dependent slot and then examine its set of constraints to resolve the dependent constraint.

Consequently, a complete system must allow a programmer to also use explicit dependency constraints where necessary. When a slot's value is changed, the invalidation algorithm follows both the slot's model dependencies and its explicit dependencies.

Sometimes it is also prudent *not* to use model dependencies in certain situations where they otherwise would apply. Observe there would be no advantage in creating a model dependency graph for an object that is never instanced since there is no space advantage in substituting a model dependency for only one explicit dependency. In fact, due to the model dependency time penalty noted above, such a substitution would slow down the constraint system.

3.5 Coexistence with Explicit Dependency Constraints

Using the techniques described in this dissertation, model dependencies can be accommodated by an existing constraint system without altering the constraint solver's mechanism for creating explicit dependencies. Standard non-modeled constraints still set up explicit dependencies. Model dependencies may coexist with normal explicit dependencies; it is possible for one slot to manage both model and explicit dependencies simultaneously. A standard formula F_s attached to one slot V_1 and a modeled formula F_m attached to an-

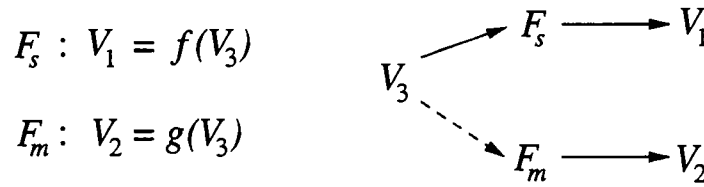


Figure 3.6: A slot with both explicit and modeled dependents. F_s is a standard formula applying function f to slot V_3 , and F_m is a modeled formula applying function g to slot V_3 . The solid arrow directed to constraint F_s represents an explicit dependency. The dotted arrow directed to constraint F_m represents a model dependency.

other slot V_2 may both reference the same slot V_3 in the right hand side of their constraint equations (see Figure 3.6). Both F_s and F_m will establish explicit dependencies for slot V_3 when the prototype object is created. When that object is instanced, the new instance object will establish explicit dependencies based on F_s 's constraint equation but will model the dependencies created by F_m 's equation. The procedure responsible for invalidating constraints must consider both a slot's explicit dependencies and model dependencies. This invalidation algorithm is described in Chapter 4.

Note that we are assuming that a given constraint must either generate all model dependencies (when a model constraint is copied into an instance) or all explicit dependencies (as standard constraints normally do). Chapter 5 relaxes this restriction, thereby allowing a constraint to generate both explicit and model dependencies.

3.6 Editing Model

The algorithms described by this dissertation handle editing operations in the following fashion. All edits made to an object before it is instanced are reflected in the object's model dependency graph. In particular, the following edits will be reflected:

- parts that are added to the object
- parts that are removed from the object

- formulas that are added to slots
- formulas that are removed from slots

Any further editing operations on a prototype after it has been instanced are not reflected in its model dependency graph. Thus changes to the prototype made after it is instanced that affect dependencies, such as adding, removing, and/or replacing parts and adding, removing, and/or replacing formulas, are not inherited by any of its instances. This behavior, while it may seem surprising, is consistent with the way Amulet, the system in which model dependencies were implemented, actually works. In particular, Amulet does not propagate changes to a prototype's parts or formulas to any existing instances.

Of course, any changes made to the instance *before* it is instanced will become part of *its* model dependency graph. The effects of editing an instance that uses model dependencies are covered in Section 4.7.

Chapter 4

Implementation

This chapter describes the implementation of the basic model dependency scheme, and the next chapter describes some extensions that were made to the basic scheme.

The implementation of model dependencies unfolded in several steps:

1. Experiments were performed in a very-high-level object-oriented interpreted language, Python [37], to verify the viability of the model dependency concept.
2. The decision was made to modify an existing constraint system instead of developing one from scratch. Reasons for this include:
 - An existing system already has in place such features as display management, graphical objects, and user input mechanisms. This infrastructure is beyond the scope of this research, but these features are necessary to build interesting applications that are representative of the problem domain.
 - An existing system has an established base of users and developers and consequently a variety of existing applications that can be used to test the correctness and performance of the model dependency implementation.

- An existing system is more likely to have a mature, tuned constraint system. Tests that provide favorable results for model dependencies would be more reassuring on an existing system than those on a self-built system where model dependencies might look good because the explicit dependency system is poorly implemented.
 - The model dependency paradigm can coexist with the dependency mechanisms available in standard constraint systems. Adapting an existing system supports this claim more strongly than building a complete system from scratch.
3. The final implementation platform was chosen. Amulet [30], a C++-based toolkit used for research and development of graphical interfaces, was selected since:
- it includes a dataflow constraint system like the one discussed here
 - its source code is readily available and is free
 - its architecture is designed to permit its features to be extended, and its developers encourage the system's modification by others to further the research in the areas of constraints and interactive techniques
 - it is available on the major computing platforms: Unix X, Microsoft Windows 95/NT, and Apple Macintosh, ensuring that the results of this research will be immediately accessible by a wide audience
4. Each constraint formula in the Amulet toolkit and a wide variety of its applications was examined. This investigation showed that over 50% of all the formulas exclusively used parameters that could be modeled using the SELF, CHILD, PARENT, and SIBLING relationships. This provided evidence that the model dependency scheme could be beneficial.
5. The SELF, CHILD, PARENT, and SIBLING model dependency edges were implemented in Amulet.
6. The model dependency scheme was extended in order to model more dependencies. Chapter 5 details

the nature of the extensions and the benefits they provided. This chapter focuses on the implementation of the fundamental model dependency scheme.

4.1 Basic Data Types

Since model dependencies were implemented in C++, the implementation described in this chapter has an object-oriented flavor. All the data types used in the implementation are presented as objects. The algorithms are presented as a set of methods, although Algol-like pseudocode rather than C++ is used to enhance readability. The primary classes of interest are

- **Object.** This class implements the prototype-instance application objects used by application programmers. Table 4.1 describes the `Object` fields and methods used in the algorithms described in this chapter. The actual slots of an application object are stored in the `slots` field of an `Object`. This field is an associative array where individual elements are accessed via their slot name.¹ Each object also has a parts list which can be empty.
- **Slot.** This class is used to implement a slot of an `Object`. In addition to its `value` field, a `Slot` has fields specifically used by the algorithms in this chapter. Its fields and methods are described in Table 4.2.
- **Formula.** This implementation object represents a formula defined by the application programmer. It includes a pointer to a function that computes a value. If it is a modeled formula, it will also include a pointer to the required model parameters. Its fields and methods are described in Table 4.3.
- **Constraint.** This class is used to implement a constraint object. Constraint objects are created when a formula is assigned to a slot. Table 4.4 describes the relevant fields and methods associated

¹In the actual implementation slot *keys* are used, not character string names. A macro is defined so the applications programmer can use symbolic names such as `left`, but these are converted to integer values; for example, `left` maps to 100. For this discussion it is convenient to think in terms of slot "names," as an applications developer would, keeping in mind that manipulations are more efficient with slot keys than with character strings.

Table 4.1: Definition of Object fields and methods used by the algorithms in Chapter 4. The boldface items provide support for model dependencies.

Object Fields and Methods	
Field	Meaning
parts	the set of objects that are children of this composite object; using the notation of associative arrays, if object obj has a child named p, then in the algorithms presented here obj.parts[p] is the child object named p
slots	the set of slots for this object; using the notation of associative arrays, if object obj has a slot named s, then in the algorithms presented here obj.slots[s] is the slot object named s
parent	the object's parent; this field is null if the object has no parent
part_name	the name of this object, if it is a part; this field is null if the object has no parent
prototype	the object's prototype; this field is null if the object has no prototype
model_subtree	the model dependency graph storing this object's SELF and CHILD model dependency edges
model_supertree	a set of model supertree graphs for each of the object's parts. This field is treated as an associative array; that is, model_supertree[label] would return label's model supertree
common_model_subtree	a set of model dependency edges shared by all children (see Section 5.4)
Method	Meaning
create_model_dependency_graph()	creates a model dependency graph for the object (see Algorithm 4.3)
inherit_constraints()	inherits constraints from the object's prototype (see Algorithm 4.8)

Table 4.2: Definition of Slot fields and methods used by the algorithms in Chapter 4. The boldface items provide support for model dependencies.

Slot Fields and Methods	
Field	Meaning
value	the value of the slot; the actual data of interest
object	the object to which this slot belongs
name	the slot's name
valid	true, if the slot's constraint is up to date; otherwise, false
explicit_dependencies	the set of explicit dependencies associated with the slot
constraints	the set of constraints that can be used to compute the value of this slot
Method	Meaning
invalidate()	marks the slot invalid and each dependent invalid (see Algorithm 4.4)
get_model_dependencies()	returns the slot's model dependencies (used in Algorithm 4.4)

Table 4.3: Definition of Formula fields used by the algorithms in Chapter 4. The `model_parameters` list provides support for model dependencies.

Formula Fields and Methods	
Field	Meaning
<code>function</code>	a pointer to the function that computes this formula's value
<code>model_parameters</code>	the formula's set of model parameters

Table 4.4: Definition of Constraint fields and methods used by the algorithms in Chapter 4. The boldface items provide support for model dependencies.

Constraint Fields and Methods	
Field	Meaning
<code>valid</code>	true, if the constraint is up to date; otherwise, false
<code>slot</code>	a pointer to the slot computed by this constraint
<code>modeled</code>	true, if the constraint was created from a formula with a non-empty set of model parameters; otherwise, false
<code>generate_dependencies</code>	true, if the constraint should create explicit dependencies, even if it is modeled; otherwise, false
<code>needs_parent</code>	true, if one or more of the constraint's model parameters defines a path through the parent; otherwise, false
<code>formula</code>	a pointer to the formula object that computes this constraints's value
Method	Meaning
<code>invalidate()</code>	marks the constraint invalid and the slot that the constraint computes invalid (see Algorithm 4.4)
<code>create_instance()</code>	make a copy of the constraint
<code>evaluate()</code>	calls the constraint's formula function

with Constraint objects. Note that constraint objects share formula information by pointing to the same formula object. When a constraint creates an instance of itself, it simply creates an instance of a constraint object and copies its formula object pointer to the instance.

- **Model_Edge and its subclasses.** The details of this data type are discussed in Section 4.2.
- **Model_Dependency_Graph.** The details of this data type are discussed in Section 4.3.

Figure 4.1 provides a concrete example illustrating the relationships among Objects, Slots, Constraints, and Formulas. This figure shows an abbreviated picture of the implementation of the labeled box from Figure 2.1.

4.2 The Model Edge Type

While the Constraint, Formula, Object, and Variable types are built into Amulet's constraint system and need only slight modification to accommodate model dependencies, the Model_Edge class and its subclasses are new types created specifically to implement model dependencies. Instances of Model_Edges are used to specify the model parameters of formulas and the model dependency edges in model dependency graphs. The class hierarchy for model edges is shown in Figure 4.2. 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

- generate an appropriate set of model dependency edges and add them to the appropriate model dependency graph (`install()`),
- take a slot, follow a path through a composition hierarchy and produce the constraint that depends on that slot (`resolve()`), and
- create a new Model_Edge of similar type (`create()`).

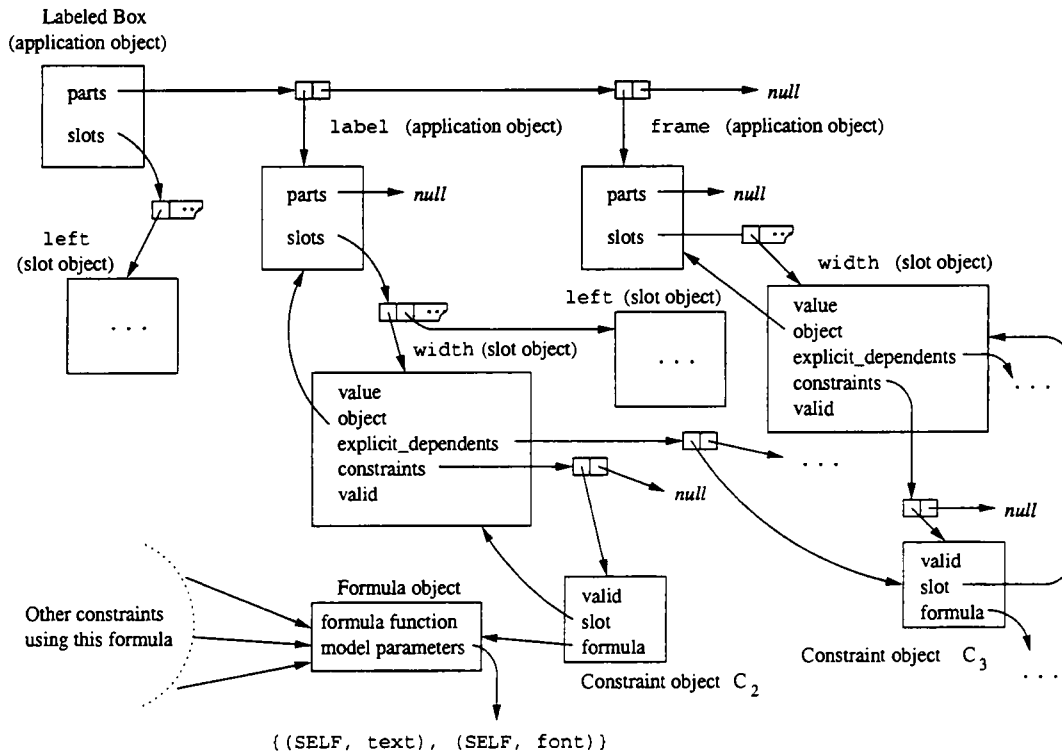


Figure 4.1: The runtime data structures implementing a portion of a labeled box application object. The root object of the composition hierarchy is represented by the box at the upper left corner of the diagram. It has two children in its parts list, a label and a frame. It also has a pointer to a list of its slots. Neither the label nor the frame child has any children of its own. Details of the label's width slot show that its value is determined by exactly one constraint, C_2 (the only constraint in its constraints list), and that it affects constraint C_3 (one of the constraints in its explicit_dependencies list). Model dependencies eliminate this explicit dependency, but it is shown here for illustrative purposes. Constraint objects are fairly lightweight and are of constant size; formula objects, which can be shared by many constraint objects, are responsible for storing common information that requires more storage, like model parameter lists.

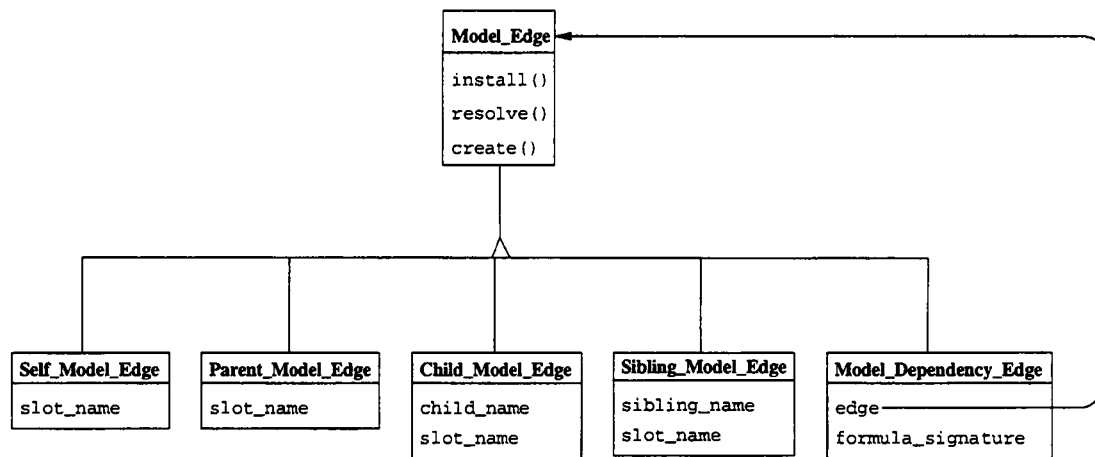


Figure 4.2: The class hierarchy for model edges. **Model_Edge** is an abstract base class that merely defines the **Model_Edge** method interface. Each subclass adds field(s) and defines the methods to specialize edge behavior. The `slot_name` field indicates the parameter slot for a parameter edge or dependent slot for a dependency edge. The `child_name` and `sibling_name` provide the required additional path information for **CHILD** and **SIBLING** edges. Observe that a **Model_Dependency_Edge** is a **Model_Edge** with the addition of a **Model_Edge** reference (`edge`) and a `formula_signature` field. The **Model_Dependency_Edge** methods delegate their work to the **MODEL_EDGE** pointed to by the `edge` field; therefore any subclass of **Model_Edge** can also be used as a model dependency.

The behavior of the `install()` methods for each subclass is described in Section 4.4. The `resolve()` methods are considered in Section 4.5. The `create()` method is used by the `install()` method to create model edges.

The subclasses `Self_Model_Edge`, `Child_Model_Edge`, `Parent_Model_Edge`, and `Sibling_Model_Edge`, specify a model parameter for a modeled constraint formula. All model parameter edges create an appropriate `Model_Dependency_Edge` when their `install()` method is invoked.

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 to a slot. Initially, it appeared that the `Model_Edge` subclasses could be used for both model parameters and model dependencies. However, as noted in Section 3.3, a model dependency requires a formula signature in addition to a path. Since the formula signature is an additional piece of information, each type of model dependency edge could subclass the model edge that describes its path. For example, a `Child_Model_Dependency_Edge` might subclass `Child_Model_Edge`. Unfortunately, this solution requires that two new subclasses be defined each time a new model edge type is added to the system (the model edge class and the dependency edge that subclasses this model edge).

The solution that was chosen instead defines a single `Model_Dependency_Edge` class and makes it a subclass of `Model_Edge`. 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. (Since the pointer to a formula's list of parameter edges is unique, it is used as the formula signature.) The `resolve()` message is delegated to the model edge. The `install()` message could also be delegated to the model edge, but it is actually defined to do nothing because the `install()` method is only meant to be used with model parameter edges.

This solution has two elegant characteristics:

1. The inheritance relationship allows a `Model_Dependency_Edge` to behave like a `Model_Edge`.
2. The composition relationship (each `Model_Dependency_Edge` has a pointer to a `Model_Edge`) sim-

plifies the `Model_Edge` class hierarchy because there is no need to define a separate dependency edge class each time a new model edge type is added to the system.

As will be seen later in this dissertation, the hierarchy is extensible. If the toolkit developer wishes to implement a `GRANDPARENT` model edge, for example, all that need be done is subclass `Model_Edge`, add any appropriate fields, and customize the new type's `install()` and `resolve()` methods.

4.3 Division of Responsibility

The four basic edge relationships fall into two distinct groups. The `SELF` and `CHILD` relationships can be installed and resolved independently of a parent, so these relationships are called *parent-independent* relationships. In contrast, both `PARENT` and `SIBLING` model edges can be resolved correctly only if a parent object is present, so these relationships are called *parent-dependent* relationships. As Figure 4.3 illustrates, parent-independent relationships represent *subtree* relationships, and parent-dependent relationships represent *supertree* relationships.

Note that the `PARENT` and `SIBLING` relationships are intrinsic not to the object but rather to the composite object to which they belong. For example, consider the `SIBLING` relationship from `B.x` to `C.y` in Figure 4.4 (for example, `C.y = C.parent.B.x`). If part `B` is removed from the composite object and replaced with another part named `B`, the new part will assume the `SIBLING` relationship (that is, `C.y` will now depend on the new part's `x` slot). Consequently, the `SIBLING` relationship is not intrinsic to part `B` but rather to the composite object to which it belongs.

Due to the different nature of these two groups of edges, they are managed differently. An object's `SELF` and `CHILD` model dependencies are stored in its prototype while its `PARENT` and `SIBLING` model dependencies are stored in its parent's prototype. In other words, the portion of the model dependency graph stored in the prototype of an object manages only `SELF` and `CHILD` model dependencies. It is known as the *model subtree graph*, or just *model subtree*, since it represents dataflow relationships within the object itself or to its parts below. The prototype also holds individual model constraint graphs for each

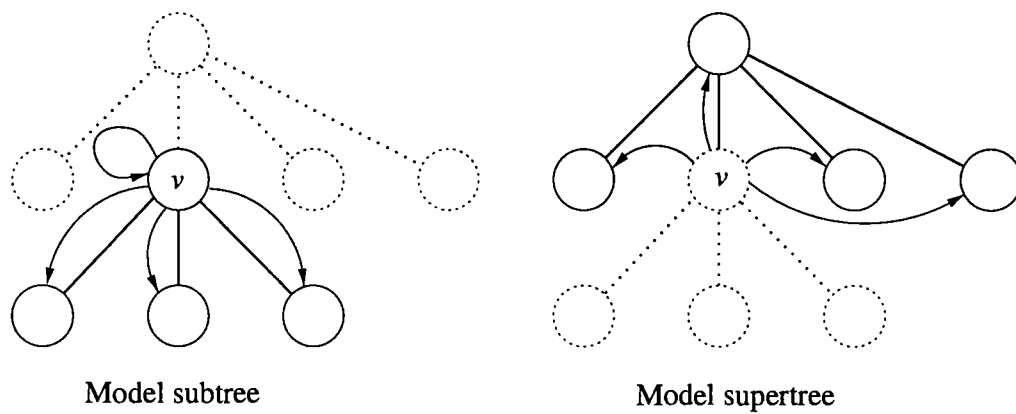


Figure 4.3: Subtree vs. supertree relationships. SELF and CHILD relationships refer to slots in object v itself or its children and hence are called **subtree** relationships. PARENT and SIBLING relationships refer to slots in object v 's parent or siblings and hence are called **supertree** relationships. The path for a subtree edge does not pass through the parent while the path for a supertree edge does.

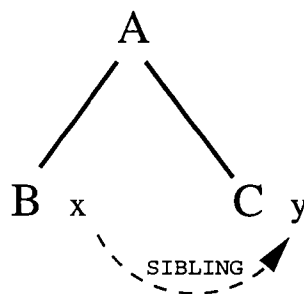


Figure 4.4: Sibling relationship

of its parts. These graphs store PARENT and SIBLING dependencies for slots in the parts. They are called *model supertree graphs*, or just *model supertrees*, since they represent dataflow relationships to the parent object or to other children of the parent. Figure 4.5 shows the collection of model constraint graphs for the labeled box object.

When a slot is modified, the prototype's model subtree is consulted to resolve SELF and CHILD model dependencies, and then the model supertree for that object, stored in the parent's prototype, is consulted to determine if any PARENT and SIBLING dependencies exist.

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 4.4 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.

Observe that this distributed approach also handles the case where two instance objects with the same prototype are added to different composite objects (that is, composite objects with different prototypes). Since the two parts have identical structure their SELF and CHILD dependencies will be identical, and hence they can share their model subtree graph. The PARENT and SIBLING dependencies for each instance are different since they belong to parents with different types, so they cannot share a model supertree graph. This is not problematic because the parent's prototype stores the model supertree graphs.

Examples further illustrating the benefits of the modular nature of this distributed model dependency graph scheme can be found in Section 4.7.

4.4 Model Dependency Installation

As noted at the beginning of this chapter, a model dependency graph is created dynamically for a prototype object the first time the prototype is instanced. The parameter edge list for each model constraint in the

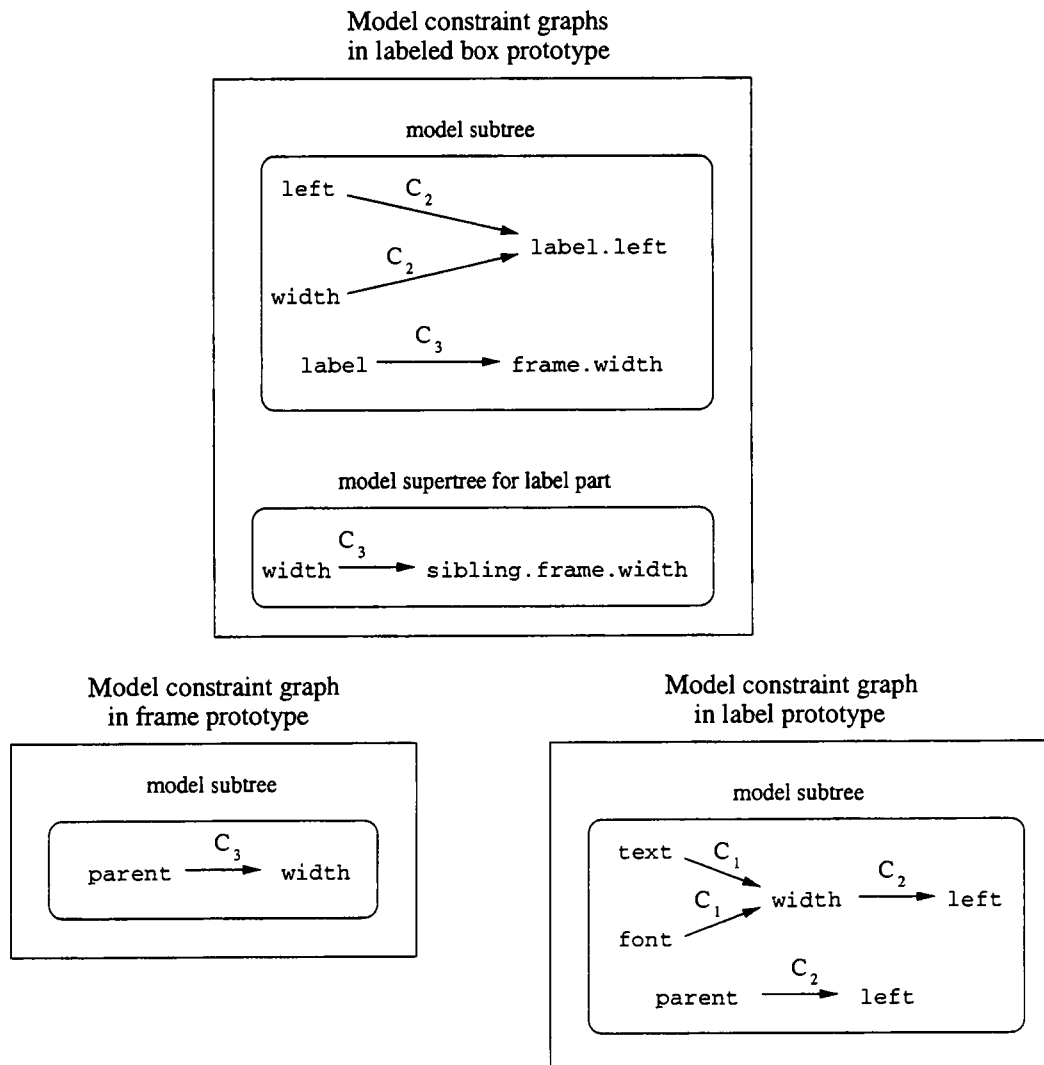


Figure 4.5: The labeled box's model constraint graphs. C_i refers to the formula signature associated with the corresponding constraint in Figure 2.2. Note that the model dependencies for a formula can be distributed across two different graphs, as is the case for label's C_2 constraint. C_2 has two references to its parent (`parent.left` and `parent.width`) and two references to itself (`self.width` and `self.parent`). The parent references get resolved into CHILD edges that are placed in the parent's model subtree. The self references get resolved into SELF edges and are placed in the label's model subtree.

prototype and its parts is examined and used to generate an appropriate set of model dependencies as described in Section 3.2. Table 4.5 shows the model dependency edge that should be generated by the `install()` method of each `Model_Edge` subclass. (Table 4.5 is a refinement of Table 3.2; it indicates whether the model dependency edge is stored in the model subtree or the model supertree.) Algorithms 4.1 and 4.2 show the `install()` method for two subclasses, `Self_Model_Edge` and `Sibling_Model_Edge`. The `install` method requires a reference to the object containing the constraint and the slot determined by the constraint so that it can locate the appropriate model dependency graph and invert the parameter edge.

Each model dependency edge is allocated from a registry to avoid duplicating edges. In particular, the `create()` method for each `Model_Edge` subclass uses the registry when it is called by the `install()` method. For example, the `(SELF, left)` edge above may be represented in the model dependency graphs of many prototypes or appear multiple times in the same model dependency graph. The registry will allocate this edge only once the first time it is needed, and all subsequent requests for an edge like this one return a pointer to the original `(SELF, left)` edge.

Algorithm 4.3 formalizes the procedure used to create a model dependency graph. It is invoked the first time a prototype object is instantiated. As Algorithm 4.3 shows, the creation of an object's model dependency graph requires an examination of every slot in the object and, recursively, every slot of all its descendent parts. This slot examination requires that each constraint used to compute that slot's value also be examined. Finally, if the constraint is a modeled constraint, each of its model parameters must generate a model dependency edge. This comprehensive self-examination makes the model dependency graph creation process the most time consuming facet of the model dependency implementation. The time to perform the triply-nested loops for each object, however, is limited by the following observations:

Innermost loop: In practice most formulas rarely have parameter lists longer than five edges.

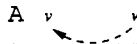
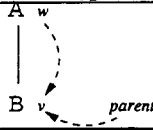
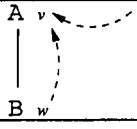
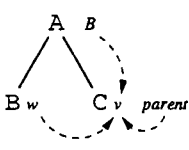
If the model dependency graph is implemented as a hashtable keyed on a slot's name, each edge's `install()` method call requires on average only a small amount of constant time to execute (Algorithms 4.1 and 4.2 are representative of the `install()` methods).

Method Self_Model_Edge.install(Object obj, Slot slt, Signature sig)

```
1 graph ← obj.model_subtree
2 new_edge ← Self_Model_Edge.create(slt)
3 new_dependency ← Model_Dependency_Edge.create(new_edge, sig)
  — slot_name is the parameter vertex on which slt depends.
  — Hence the new (SELF, slt) dependency
  — edge should be added to slot_name's list in the model dependency graph.
4 graph.insert_edge(self.slot_name, new_dependency)
```

Algorithm 4.1: Installation method for the Self_Model_Edge type. The insert_edge() method for a model dependency graph adds a new model dependency edge to the set of edges for a given vertex.

Table 4.5: Model dependency edge installation for the four fundamental edges. This is a refinement of Table 3.2; it distinguishes between the model subtree and model supertrees as targets for the parameter vertices.

Variable to which formula is attached	Parameter edge	Parameter vertices	Dependency edges
A.v	(SELF, w)	A.w	(SELF, v) $\in$ A.model_subtree 
B.v	(PARENT, w)	A.w B.parent	(CHILD, B, v) $\in$ A.model_subtree (SELF, v) $\in$ B.model_subtree 
A.v	(CHILD, B, w)	B.w A.B	(PARENT, v) $\in$ B.model_supertree (SELF, v) $\in$ A.model_subtree 
C.v	(SIBLING, B, w)	B.w C.parent A.B	(SIBLING, C, v) $\in$ B.model_supertree (SELF, v) $\in$ C.model_subtree (CHILD, C, v) $\in$ A.model_subtree 

Method `Sibling_Model_Edge.install(Object obj, Slot slt, Signature sig)`

- To better explain the process, let object A have children
- B and C, and let C.v depend on B.w. (See Table 4.5.)
- So the formula is computing C.v and the model parameter to
- install (that is, the parameter edge for which this method
- was called) is (SIBLING, B, w).
- Thus, this edge's fields correspond to `sibling_name = B` and `slot_name = w`.
- `obj.parent = A`, `obj.part_name = C`, and `slt = v`.
- Three model dependency edges must be installed for this one
- parameter edge.

```

1  parent ← obj.parent
   —If the object has no parent, then the installation cannot continue.
2  if parent ≠ null :
   —First, add the SIBLING model dependency from one part to another: from B.w to C.v
   —This causes C.v to be recomputed if its sibling's slot changes
   —It creates the edge (SIBLING, C, v) for vertex w in B's model supertree
3  graph ← parent.model_supertree[self.sibling_name]
4  new_edge ← Sibling_Model_Edge.create(obj.part_name, slt)
5  new_dependency ← Model_Dependency_Edge.create(new_edge, sig)
6  graph.insert_edge(self.slot_name, new_dependency)
   —Next, the part slot in the parent must have a CHILD model dependency to its
   —child's dependent slot: from A.B to C.v
   —This causes C.v to be recomputed if it gets a new sibling
   —It creates the edge (CHILD, C, v) for vertex B in the A's model subtree
7  graph ← parent.model_subtree
8  new_edge ← Child_Model_Edge.create(obj.part_name, slt)
9  new_dependency ← Model_Dependency_Edge.create(new_edge, sig)
10 graph.insert_edge(self.sibling_name, new_dependency)
   —Finally, a SELF dependency must be added to the parent slot's dependency list:
   —from C.parent to C.v
   —This causes C.v to be recomputed if it gets a new parent
   —It creates the edge (SELF, v) for vertex parent in the C's model subtree
11 graph ← obj.model_subtree
12 new_edge ← Self_Model_Edge.create(slt)
13 new_dependency ← Model_Dependency_Edge.create(new_edge, sig)
14 graph.insert_edge(parent, new_dependency)

```

Algorithm 4.2: Installation method for the `Sibling_Model_Edge` type. The `sibling_name` field selects the appropriate model supertree; `slot_name` is the parameter vertex on which `slt` depends, so the dependency goes from `slot_name` to `slt`.

```
Method Object.create_model_dependency_graph()  
1  for each slot  $s \in \text{self.slots}$  :  
2      for each constraint  $cn \in s.constraints$  :  
3          if  $cn.modeled = \text{true}$  :  
4              for each parameter  $param \in cn.formula.model\_parameters$  :  
5                   $param.install(\text{self}, s, cn.formula.model\_parameters)$   
6  for each child  $pt \in \text{self.parts}$  :  
7       $pt.create\_model\_dependencies\_graph()$ 
```

Algorithm 4.3: Model constraint graph creation algorithm. Observe that the pointer to the model parameter list serves as the formula signature.

Middle loop: Generally the number of constraints associated with a slot is small, usually only zero or one.

Thus in practice the two inner loops are bounded by constants, and so the expense of the model dependency graph creation method is mostly dependent on the number of slots in the composition hierarchy.

Fortunately, a model dependency graph is created only when an object is instantiated for the first time. In many common applications like the one shown in Figure 3.1 where a prototypical object is used to mass produce program elements, the relatively expensive method detracts very little from the program's overall performance.

4.5 Resolution of Literal Dependencies

Once the model dependencies are installed, an instance can use them to derive explicit dependencies. When a slot is changed, the resolution algorithm consults the model subtree and supertree graphs containing that slot's name and uses the stored model paths to traverse the object's composition hierarchy to find the dependent slots.

As Section 2.2 mentioned, there may be multiple constraints that determine a slot. When the dependent slot is located, its set of determining constraints is searched to find the one with a formula signature that matches the formula signature stored with the model dependency edge. If a matching constraint is found, it is invalidated.

Algorithm 4.4 formalizes the invalidation process. The polymorphic `resolve()` method is customized for each model edge so that the correct path from an invalidated slot to a dependent slot is followed. Algorithm 4.5 shows how a `Self_Model_Edge` and a `Sibling_Model_Edge` are resolved into a dependent constraint.

To illustrate the invalidation and resolution algorithms, let `box` be a labeled box and let `box.label` be its `label` part. Suppose the label's width slot is invalidated (this could occur, for example, as a result of editing the label's text slot). The following actions will be performed:

Method Constraint.invalidate()

```

1  if self.valid = true :
2      self.valid ← false
3      self.slot.invalidate()

```

Method Slot.invalidate()

```

1  if self.valid = true :
2      self.valid ← false
3      for each constraint cn ∈ self.explicit_dependencies :
4          cn.invalidate()
5      model_dependencies ← self.get_model_dependencies()
6      for each model dependency dep ∈ model_dependencies :
7          cn ← dep.resolve(self, null)
8          if cn ≠ null :
9              cn.invalidate()

```

Method Slot.get_model_dependencies() **returns** Set of Model Dependency Edges

—Retrieve the *SELF* and *CHILD* model dependencies from the prototype's model subtree and the *PARENT* and *SIBLING* model dependencies from the model supertree of the parent's prototype.

```

1  return self.object.prototype.model_subtree[self.name]
   ∪ self.object.parent.prototype.model_supertree[self.object.part_name][self.name]

```

Algorithm 4.4: Invalidation algorithm for a constraint solver that incorporates model dependencies. `resolve()` is a method associated with a model dependency that uses the resolution procedure described in Section 4.5 to return the appropriate constraint. The second parameter to `resolve()` is a formula signature that allows `resolve()` 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. (See Algorithm 4.5).

Method `Model_Dependency_Edge.resolve(Slot slt, Signature unused)`
returns Constraint
—The second parameter is ignored. Observe that `slot.invalidate()` in Algorithm 4.4
—passes null as the second parameter.
1 **return** `self.edge.resolve(slt, self.formula_signature)`

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

Method `Sibling_Model_Edge.resolve(Slot slt, Signature sig)`
returns Constraint
1 `obj` $\leftarrow$ `slt.object`
—Locate dependent slot based on the path stored in this edge
2 `parent` $\leftarrow$ `obj.parent`
3 `sibling` $\leftarrow$ `parent.parts[self.sibling_name]`
4 `dep_slot` $\leftarrow$ `sibling.slots[self.slot_name]`
5 **for** each constraint `cn` $\in$ `dep_slot.constraints` :
6 **if** `cn.signature = sig` :
7 **return** `cn`
8 **return** null

Algorithm 4.5: Resolution method for the `Model_Dependency_Edge` type. `Self_Model_Edge.resolve()` and `Sibling_Model_Edge.resolve()` are shown to illustrate the `resolve()` methods used by the model edges. 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 `dep_slot` objects exist. The actual implemented algorithms check to see whether they get non-null objects before proceeding (Line 2 in `Self_Model_Edge.resolve()` and Lines 2–4 in `Sibling_Model_Edge.resolve()`). If they get a null object, they return null.

1. If any explicit dependencies of `box.label.width` exist, they will be invalidated in the usual manner as described in Section 2.2. (Algorithm 4.4, `Slot.invalidate()` method, Lines 3 and 4.)
2. The subtree model dependencies are located in the model subtree for `box.label` (this graph is found in the prototype object for `label`; see Figure 4.5). The model edge entry (`SELF`, `left`) is found along with the formula signature of constraint C_2 . This model dependency is added to the set of model dependencies that must be resolved. (Algorithm 4.4, `Slot.get_model_dependencies()` method, Line 1.)
3. The supertree model dependencies are located in the model supertree for `box.label` (this graph is found in the prototype object for `label`'s parent; see Figure 4.5). The model edge entry (`SIBLING`, `frame`, `width`) is found along with the formula signature of constraint C_3 . This model dependency is added to the set of model dependencies that must be resolved. (Algorithm 4.4, `Slot.get_model_dependencies()` method, the second part of the statement in Line 1.)
4. Each edge in the set of model dependencies is resolved to a literal constraint. (Algorithm 4.4, `Slot.invalidate()` method, Lines 7–9.)
 - (a) The model edge (`SELF`, `left`) is resolved. The edge's path directs the algorithm to the `box.label.left` slot. (Algorithm 4.5, `Self_Model_Edge.resolve()` method, Line 2.)
 - (b) Constraint C_2 is located in `box.label.left`'s list of constraints and C_2 's signature matches the one stored with the dependency edge. Therefore, the constraint is returned to `Slot.invalidate()` to be invalidated. (Algorithm 4.5, `Self_Model_Edge.resolve()` method, Lines 3–5.)
 - (c) The model edge (`SIBLING`, `frame`, `width`) is resolved. The edge's path directs the algorithm to the `box.frame.width` slot. (Algorithm 4.5, `Sibling_Model_Edge.resolve()` method, Lines 2–4.)
 - (d) Constraint C_3 is located in `box.frame.width`'s list of constraints and C_3 's signature matches

the one stored with the dependency edge. Therefore, the constraint is returned to `Slot.invalidate()` to be invalidated. (Algorithm 4.5, `Sibling_Model_Edge.resolve()` method, Lines 5–7.)

4.6 Constraint Installation

Thus far this chapter has described how model dependency edges can be generated from model constraints and how model dependency edges can be resolved once they have been installed. This section and the next section mark a transition to another aspect of the model dependency scheme, namely how model dependency constraints get installed in objects and how model dependency constraints are handled when the application edits an object by adding/removing constraints or adding/removing parts.

There are two ways that a constraint can be assigned to a slot:

1. The constraint is directly assigned to a slot by code in the application. This is constraint *assignment*.
2. The constraint is inherited from a prototype. This is constraint *inheritance*.

Constraint Assignment. Constraint assignment is mainly used for redefining the behavior of some slots of an object immediately after it has been instanced. When a formula is assigned to an object after it has been created, the formula will not be modeled in the prototype's model dependency graph. Consequently it needs to generate explicit dependencies. Section 4.7 describes other situations in which model constraints must 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 (see Table 4.4).

Setting this flag forces the constraint to generate explicit dependencies.

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.

Amulet normally establishes explicit dependencies by the process described in Algorithm 4.6. Before a constraint is evaluated it pushes itself on to a global constraint stack. Whenever a slot is accessed via a `get()` call, the stack is checked to see if it is non-empty. If a constraint appears on the top of the stack, then this slot's value must have been requested by that constraint; therefore, this slot can add the top constraint to its list of dependents. This procedure was modified to accommodate model constraints that conditionally generate model dependencies. If the `generate_dependencies` flag is true when the constraint's formula calls `get()` for a slot, then `Slot.get()` establishes the explicit dependency as in Algorithm 4.6. Otherwise, `Slot.get()` will not add the constraint to its list of dependents. The modified `Slot.get()` method can be found in Algorithm 4.7.

Constraint Inheritance. Constraint inheritance can be divided into two cases: 1) the constraint to be inherited will use only `SELF` and `CHILD` model parameters and 2) the constraint to be inherited uses at least one `PARENT` or `SIBLING` model parameter. In case 1 the constraint only generates dependencies involving subtree objects. Since the model dependency graphs associated with these objects are guaranteed to be in place, the new constraint can refrain from generating explicit dependencies.

Case 2 is a little more subtle. If the new instance has a parent, then the `PARENT` and `SIBLING` model dependencies generated by the constraint can be found in the parent prototype's model dependency graphs. Thus, the instance's inherited constraint should not generate explicit dependencies. On the other hand, if the instance has no parent, then explicit dependencies must be used since the model supertree graphs responsible for holding the required `PARENT` and `SIBLING` model dependencies do not exist. In Figure 4.4, if part C is instanced *before* it is added to the composite object A, then no model supertree for part C exists to place the `(SIBLING, C, y)` model edge. Therefore, an inherited constraint must generate explicit dependencies if it uses any `PARENT` or `SIBLING` model parameters and its object has no parent. Consequently when an object is created it must be told whether or not it has a parent. A new object will have a parent if it is being created as a part of a composite object and it is not the root object. The object creation method

Method Slot.get() returns Value

```

1  if self.valid = false :
    —If the constraint stack is non-empty, establish a dependency to the top formula
    —since it has requested this slot's value
2  if constraint_stack.empty() = false :
3      self.dependents ← self.dependents ∪ constraint_stack.top()
    —Next, get the slot's value as usual
4  self.valid ← true
5  self.value ← self.constraint.evaluate(self.object)
6  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 will know that this constraint requested its value
1  constraint_stack.push(self)
2  self.valid ← true
    —Pass the object to the function that computes this constraint's value
3  result ← self.formula.function(obj)
    —Remove this constraint from the stack
4  constraint_stack.pop()
5  return result

```

Algorithm 4.6: Algorithm that permits the constraint system to install explicit dependencies. constraint_stack is a global stack of constraints. Its push() method pushes an item on to the top of the stack. Its pop() method removes an item from the top, while its top() method simply returns the top element of the stack without removing it.

Method Slot.get() returns Value

```

1  if self.valid = false :
    —If the constraint stack is non-empty and its top item expects explicit dependencies,
    —establish a dependency to the top formula
    —since it has requested this slot's value
2  if constraint_stack.empty() = false
    and constraint_stack.top().generate_dependencies = true :
3      self.dependents ← self.dependents ∪ constraint_stack.top()
    —Next, get the slot's value as usual
4  self.valid ← true
5  self.value ← self.constraint.evaluate(self.object)
6  return self.value

```

Algorithm 4.7: Modified Slot.get() that permits the constraint system to install explicit dependencies conditionally for modeled constraints

can supply this information to the constraint creation method via a parameter. Algorithm 4.8 formalizes the constraint inheritance algorithm.

4.7 Edits After an Instance has been Created

As seen in Section 4.6, when a new model constraint is assigned to a slot, that constraint must generate explicit dependencies. This constraint assignment is one example of editing an object after it has been instanced. The model dependency scheme must be resilient in the face of editing operations. The following edit operations are commonly used in prototype-instance 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
4. a part is removed from a composite object

In Case 1 it is necessary for model constraints to behave like explicit dependency constraints. To handle Case 3 the model dependency graph is augmented to reflect any model dependencies the new part may require. In Cases 2 and 4 the resolution mechanism must behave gracefully when faced with missing constraints.

Constraint assigned to a slot. This situation is covered in Section 4.6. A model constraint assigned to a slot behaves like a standard constraint and generates explicit dependencies.

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. In either case the original constraint is no longer used to compute that slot's value. 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 resolution process since a dependent constraint is found

```
Method Object.inherit_constraints(Object prototype, Boolean has_parent)
1  for each slot s ∈ prototype.slots :
2      s.constraints ← ∅
3      for each constraint cn ∈ s.constraints :
4          inherited_cn ← cn.create_instance()
5          if cn.modeled = true :
6              if cn.needs_parent and has_parent = false :
7                  inherited_cn.generate_dependencies ← true
8              else
9                  inherited_cn.generate_dependencies ← false
10             s.constraints ← s.constraints ∪ inherited_cn
11  for each part pt ∈ prototype.parts :
12      pt.inherit_constraints(pt, true)
```

Algorithm 4.8: Constraint inheritance algorithm. `create_instance` is a method of the `Constraint` class that creates a copy of a given constraint. `has_parent` is `false` in the initial call in the recursive chain because the root object does not have a parent. During the recursion, `has_parent` is `true` because the remaining objects are parts which do have parents.

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 resolution methods will return no constraint (that is, null) as shown in Algorithm 4.5. The invalidation procedure will then ignore this edge (in Algorithm 4.4, line 8 of the `Slot.invalidate()` method checks if the edge resolution returns a null constraint and does nothing if a null constraint is returned).

In the labeled box example, consider what happens if the formula computing `frame.width`, C_3 , is removed. This would occur if the user or programmer fixed the width after the box object was created. Now when `label.width` changes, the model dependency resolution will follow the path to `frame.width` but will not find the proper formula signature for C_3 in the list of constraints for that slot since the constraint has been removed. The resolution method will therefore return null, and no action will be taken with regards to `frame.width`, which is the correct behavior.

Part added to a composite object. When a part `p` is added to object `obj` after `obj` has been instanced, 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 not be represented in the model dependency graph. To see why, suppose the root of the prototype labeled box composite object is created from some appropriate prototype (for example, from a composite object prototype). Next, the frame component is created and added as a part to the labeled box prototype. If at this point the labeled box is instanced, then the label part will not be present; nonetheless, a model dependency graph will be created for the labeled box.

It is important to notice in this example that since the formula associated with `frame.width` has a model edge parameter (SIBLING, label, width), a (SIBLING, frame, width) model dependency edge can be created and placed in a model supertree graph named label, managed by frame's parent even though the label part does not yet exist. If the label part is subsequently added to a labeled box instance, the model

dependency in label's model supertree is already present and can be used to derive a literal dependency to frame.width. Note that the responsibility for the creation of the model dependency is that of the dependent object, not the missing part.

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 dependency graph creation method (Algorithm 4.3) 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 either 1) constraint C_2 establishes explicit dependencies or 2) C_2 's model dependencies are added to the model dependency graph.

The first approach requires the new part's constraints to generate explicit dependencies; this approach is analogous to Case 1 above where a new constraint is assigned to a slot. Thus, the model dependency graph for the parent can remain unchanged, and the unrepresented model dependencies will be covered by explicit dependencies. This approach unfortunately has a subtle problem that can lead to incorrect behavior. Suppose part p is added to obj after obj has been instanced and, therefore, its model dependency graph has already been created. The object obj will behave correctly since any model dependencies needed on behalf of p 's constraints will be covered instead by explicit dependencies. If, however, after p has been added obj is again instanced to create obj' , obj 's model dependency graph is not updated to reflect the new part p . The object creation algorithm, however, assumes that all the necessary model dependencies for all the components of obj are in place and directs all model constraints within obj' not to establish explicit dependencies. Thus, $obj'.p$ will contain slots that are not properly computed and kept up to date.

The solution is to add the missing model dependencies to the existing model dependency graph when the new part is added. The new part can then use the newly installed model dependencies instead of explicit dependencies, and any future instances will behave correctly since the augmented model dependency graph will cover any needed model dependencies. This solution means that any instances created *before* the new part was added will appear to be "missing" a part with respect to the augmented model dependency

graph. As shown in the discussion of missing parts below, model dependencies to parts that do not exist are properly handled by the constraint solver.

Adding a part to a composite object will not affect the existing model supertree for that part since none of that part's constraints can use model parameters that would produce model dependencies in that part's model supertree. The part's model supertree stores only PARENT and SIBLING model dependencies; the PARENT dependencies result from CHILD parameters in constraints used by its parent, and SIBLING dependencies result from SIBLING parameters in constraints used by its sibling(s). Adding a part may affect its parent's model subtree or the model supertree for one or more of its siblings, however. The routine that adds a new part to an object must also ensure that all the model dependencies required for that part are added to the model dependency graph.

While the model dependencies scheme must correctly handle this situation of part addition after instantiating, it rarely occurs in practice. Generally a prototype is constructed as a template from which a collection of "working objects" is created. When extended functionality is needed (for example, new parts are necessary), the original prototype is usually instanced to form a new object from which the new working objects are created. Of the set of Amulet applications tested in Section 4.9, none added a part to an object after that object was instanced.

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 standard constraint system takes care to remove any dangling explicit dependencies, but dangling model dependencies cannot be removed since other instances use the same model dependency graph. These other instances may not have had their corresponding part removed, so the model dependencies to this part must still be retained. As in the case with constraint removal, removing a part does not disturb the model dependency resolution 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.

To illustrate, consider the labeled box with the `Next` text label in Figure 4.6. Assume it has been created from a labeled box prototype so the model dependency graph of Figure 4.5 is already in place. Now, remove the label part. Moving this box, thereby adjusting its left slot, should affect the left slot of its label part, according to its model dependency graph. However, resolution will find no label part, and so no constraint will be invalidated, which is the correct behavior.

Now suppose we replace the text label in Figure 4.6 with a bitmap object, the right pointing arrow. (Observe that this exchange is simply removing one part and then adding another.) The bitmap part assumes the label part name. A bitmap object has slots keeping track of its position, size, and an array of pixel color values that constitute its image. Notably, it has a width slot. If the bitmap is resized and its width changes, its sibling frame will have its width slot recomputed since the model supertree for a part named label still exists in label's parent. This behavior is correct. Observe once more that the label part is not responsible for creating or managing the dependency to its frame sibling. That dependency information is kept in the parent where relationships among siblings can be more easily managed.

Continuing the bitmap-for-text part exchange, the old text label part had a formula for left that depended on its parent's left and width; therefore, the labeled box's model subtree has two (`CHILD`, `label`, `left`) edges, one associated with its left vertex and the other with its width vertex. Assume the new bitmap label has no such formula associated with its left slot. If the labeled box's width value changes, the resolution algorithm will locate the new label's left slot, find no attached constraint (specifically no attached constraint with the expected formula signature), and therefore do nothing. This behavior is correct.



Figure 4.6: Replacing a text part with a bitmap object.

Finally, suppose the old `label` part that was removed from the labeled box to make room for the bitmap object is now added to a new composite object. (As described above, its supertree-dependent formulas must now establish explicit dependencies, but any subtree dependencies are still valid.) If it is given the same part name (`label`), then there is a small chance that it is replacing another part that was also named `label` and the old `label` part used exactly the same formula to compute exactly the same slot as `label` does. 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 should occur rarely in practice.

Discussion. In sum, the model dependency resolution scheme is resilient in the face of edits. If the dependent slot cannot be found, then the composite object's structure has changed (for example, the part has been removed or replaced with another part) and no constraints are invalidated. Note that this strategy allows the model dependency scheme to work even when an individual instance alters its structure.

The formula signatures also permit the model dependency scheme to work properly when model constraints are removed from slots in instances. If the resolution process finds a dependent slot and that slot does not contain a constraint with a signature matching the one stored in the model dependency graph, then the constraint must have been removed. In this case no invalidation is performed and the results agree with non-modeled behavior. (However this model dependency look-up does take some time that would not be spent if explicit dependencies were used instead.)

Finally, it should be noted that this resiliency in the face of edits is only required by the prototype-instance model. A class-instance model statically specifies an object, and the class may not be dynamically edited. Hence, the model dependency scheme works equally well in a class-instance model.

4.8 Conditional Formulas

Formulas involving conditional access of slots can introduce dependencies under the model dependencies scheme that would not appear if only explicit dependencies were used. To see how, consider the following formula:

```
self.width = if self.parent.visible then self.parent.width
              else 2
```

Here the width slot of an object is computed from its parent's width slot, but only if its parent is visible. Consider the consequences of assigning this formula to the width slot of an object `obj`. If explicit dependencies are being used and `obj`'s parent is visible when this formula is executed, then dependencies will be formed from `obj.parent.visible` to `obj.width` and from `obj.parent.width` to `obj.width`. If `obj`'s parent is not visible when this formula is executed, then only the first dependency will be created.

If the formula is modeled, both right hand slots must be specified as model parameters since:

1. model dependencies are created from model parameter specifications supplied with the model formula definition (see Section 3.2) and
2. it is impossible to determine at compile time which branch of the conditional will be true; in fact, the active branch may vary during the program's execution.

Compared to a system using only explicit dependencies, a model dependency system with conditional modeled formulas will likely have "spurious dependencies" (that is, dependencies that should be inactive). The system will work correctly, but some constraints may be evaluated unnecessarily. With the above formula applied to `obj`, if `obj.parent.visible` is false, then the constraint will be recomputed if `obj.parent.width` changes. Since such an explicit dependency would not exist, the constraint would not be re-evaluated in a non-modeled system.

Section 5.8 indicates that the number of spurious invalidations in most applications is small, so this unavoidable effect is tolerable.

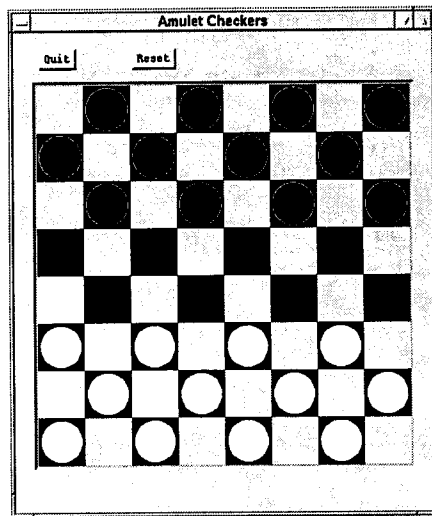
4.9 Results

The basic model dependency scheme described in this chapter was implemented in Amulet. Once the implementation was completed the code was instrumented so the number of model and explicit dependencies could be counted. The main criterion for evaluating the merit of model dependencies is the number of explicit dependencies saved. Amulet's object system is quite heavyweight; it supports a large number of features beyond those provided by most constraint systems. This makes it an ideal research tool in general, but it makes judging the success of model dependencies difficult when raw heap space is measured. Counting dependencies instead of raw memory provides a more accurate picture of how well model dependencies work. As an added benefit the results are truly platform independent; a particular operating system's memory management strategies will not flavor the results. There was a tradeoff involved in doing the implementation in Amulet. Most constraint solvers are implemented in far more streamlined systems where the results would have led to significant reductions in heap space. Unfortunately, the source code for these systems is typically not publicly available nor is it meant to be extended by other researchers. Amulet by design meets both of these criteria.

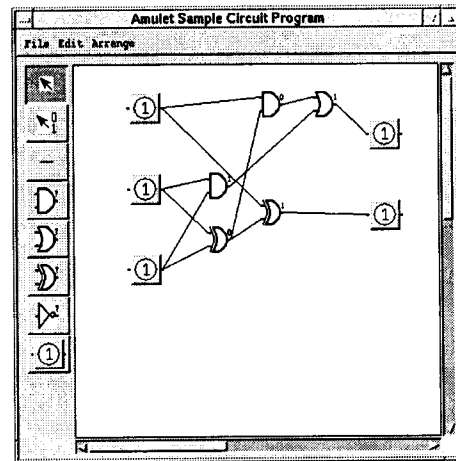
Henceforth, the term *Basic Model Amulet* will refer to the modified version that supports the basic model dependency scheme described in this chapter; *Original Amulet* will refer to the original version. Basic Model Amulet was tested on a number of existing Amulet applications and on three specialized benchmark programs. The applications included the samples found in the Original Amulet distribution and several other contributed programs. Table 4.6 describes the applications and benchmarks. Pictures of the graphical applications are shown in Figure 4.7. As indicated, some of the applications were crafted by the designers of Amulet itself, some were written by graduate students as class projects, and some were created by the author. They thus represent a range of application programmer expertise, from those expert at using constraints to those just learning how to use constraints. No alterations are required to get existing applications to compile and run under Basic Model Amulet. However, to maximize the advantages of Basic Model Amulet the formulas in these test programs were modified to take advantage of modeled constraints.

Table 4.6: Applications used for evaluating model dependencies.

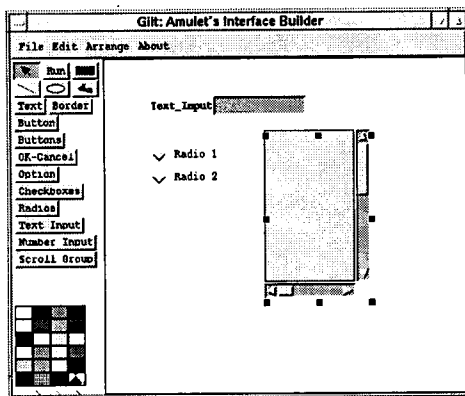
Application/ benchmark	Description	Source
Checkers	The traditional board game	Amulet distribu- tion
Circuit Designer	Used to build digital logic networks	Amulet distribu- tion
Gilt	An interface builder that permits the construction of a graphical user interface by direct manipulation	Amulet distribu- tion
MathNet	Represents arithmetic expressions with a dataflow graph	The author
Network Simula- tor	Simulates message passing in various network configurations	Graduate student
Testwidgets	Extensive test of menus, dialog boxes, scroll bars, etc.	Amulet distribu- tion
Tree Editor	Graphically represents binary trees for visual program debugging	Graduate student
Labeled Box	A benchmark that creates 500 labeled boxes similar to the one described in Chapter 1	The author
Dense Dependenc- ies	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	The author
Empty	An application that does nothing but initialize the base Amulet system. No new objects are added to the system. This application indicates the "baseline" dependency requirements for any Amulet application.	The author



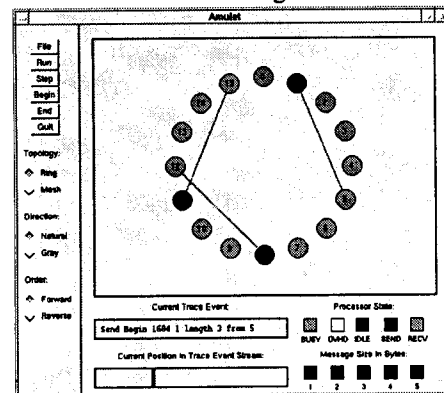
Checkers



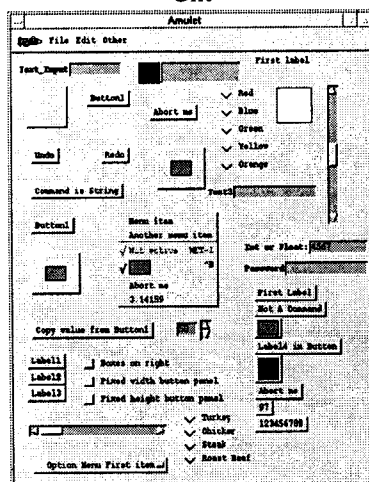
Circuit Designer



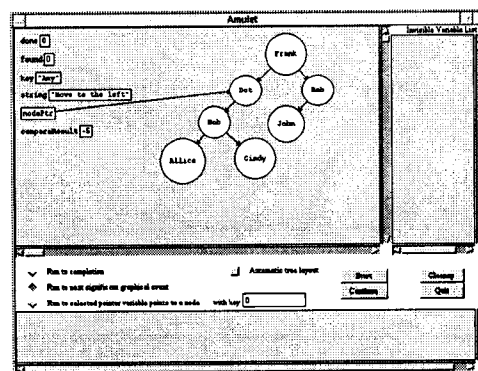
Gilt



Network Simulator



Testwidgets



Tree Editor

Figure 4.7: Application snapshots. MathNet can be seen on Page 19. 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.

To ensure that Basic Model Amulet does not alter the behavior of an application, the behavior of these modified versions was compared to the original programs running under Original Amulet. In all cases the modeled versions behaved identically to the original versions.

The numbers of explicit dependencies generated by both the standard version and modeled version were then compared. Figure 4.8 summarizes the results of the experiments.

As Figure 4.8 reveals, the results were not encouraging. 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 a more in-depth analysis of the nature of the formulas and the dependencies they produce. While the basic model dependency scheme presented in this chapter *does* work (explicit dependencies are eliminated), it has some limitations that greatly reduce its effectiveness. Chapter 5 describes how the basic scheme was extended to overcome some of these limitations thereby resulting in a system that was able to eliminate a large percentage of explicit dependencies in the tested applications.

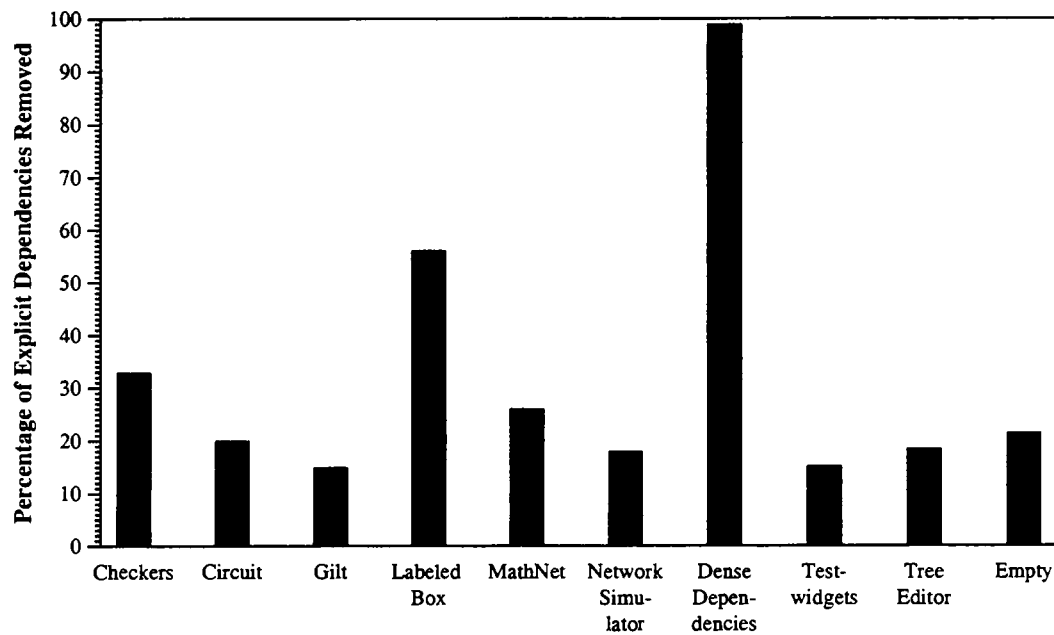


Figure 4.8: Modeled results for a cross section of Amulet applications. The height of the bars reflect the percentage of explicit dependencies removed. The average for actual applications (that is, excluding the benchmark Labeled Box, Dense Dependencies, and Empty applications) is 20.6%. The average for all ten applications and benchmarks is 32.0%.

Chapter 5

Extensions to the Basic Scheme

The scheme described in Chapters 3 and 4 provides the fundamental framework for the model dependency concept. Unfortunately, as noted at the end of the last chapter, experiments revealed that while over 50% of all formulas exclusively use the four fundamental relationships, on average these formulas only generate approximately 20% of the dependencies in an application. It became apparent that a small percentage of formulas were generating a large number of non-modelable dependencies. An analysis of the dependencies generated in Amulet applications revealed a number of common situations that the basic scheme was missing:

- **Mixed dependency formulas.** Many formulas could not be modeled because only one or two of their parameters could not be modeled. Allowing formulas to be “partially” modeled would substantially increase the number of model dependencies.
- **Prototypes using model dependencies.** The basic scheme assumes that an instance uses model dependencies stored in its prototype’s model dependency graph. The prototype itself should also be able to use these same model dependencies thereby reducing the number of explicit dependencies it maintains.

- **Formulas with missing part names.** A large number of dependencies are generated by unnamed children. The CHILD model dependency in the basic scheme requires the name of the child to locate the dependent slot. Most often the child is unnamed because the dependency is to *all* the children of an object. A model relationship that captures this notion of “to all the children” would be beneficial.
- **Formulas with a varying number of parameters.** Some formulas depend on information *from* all their children. These formulas are so general that the names and number of children are not specified. The model dependency would be a PARENT edge in each child’s model supertree. However, the formula is too general for the basic scheme to produce the appropriate model supertrees; in fact, the formula should work properly even as parts are added or removed *after* the model dependency graph has been created. A model relationship that captures this notion of “from all the children” would be beneficial.
- **Formulas depending on shared global resources.** Some explicit dependencies result from access to a global resource such as the device context of a window in a graphical interface. If a large number of objects reference this resource and it is changed infrequently, we may be willing to examine all the objects in the interface in order to discover which slots should be invalidated. In other words, a model relationship that captures this notion of “to all objects” would be nice.

This chapter describes how the model dependency scheme was extended to accommodate these five cases.

5.1 Mixed Dependency Formulas

The basic model dependency scheme allows a formula to be modeled only if all of its parameters can be modeled. This restriction eliminates every formula with at least one parameter that cannot be modeled. This significantly reduces the effectiveness of the basic scheme because it is quite common for a formula to have a number of modelable parameters and only one or two non-modelable parameters. For example, consider the following simple formula:

```
self.left = self.parent.left + self.next.left
```

The first parameter `self.parent.left` can be modeled, but `self.next` refers to an object that is possibly not part of the composite object hierarchy. Consequently the formula cannot be modeled even though three of its four parameters are modelable (`self.parent`, `self.parent.left`, and `self.next` are all modelable). In other words, the formula will generate four explicit dependencies just because one cannot be modeled.

Ideally, if labeled box B_1 uses the above formula to compute its left slot and labeled box B_2 is the object referenced by $B_1.next$, then an explicit dependency should be established from $B_2.left$ to $B_1.left$, but the $B_1.parent.left \rightarrow B_1.left$, $B_1.parent \rightarrow B_1.left$, and $B_1.next \rightarrow B_1.left$ dependencies should be modeled. A finer-grained approach was therefore adopted that permits formulas to establish mixed dependencies.

Implementation. Mixed dependencies are implemented by requiring each slot reference within a formula to indicate if the constraint system should establish an explicit dependency on the slot's behalf. Algorithm 5.1 illustrates this scheme. The original Amulet constraint system already passed a flag to a slot that suppressed the creation of an explicit dependency, so this implementation did not have to add an extra parameter for slot access.

Method `Slot.get(Boolean use_model)` returns Value

```

1  if self.valid = false :
    —If a dependency should be established or the top of 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
2  if constraint_stack.empty() = false :
3      cn = constraint_stack.top()
4      if use_model = false or cn.generate_dependencies = true :
5          self.dependents ← self.dependents ∪ constraint_stack.top()
    —Next, get the slot's value as usual
6  self.valid ← true
7  self.value ← self.constraint.evaluate(self.object)
8  return self.value
```

Algorithm 5.1: Algorithm that permits the constraint system to install explicit dependencies. A Boolean parameter is passed to `get()` that indicates whether or not an explicit dependency should be created.

Results. Permitting a constraint to establish a combination of model and explicit dependencies increased the average number of removed explicit dependencies from 19.7% to 47.4%. This represents 27.7% more dependencies removed than the basic scheme. Figure 5.1 shows the results for the ten applications and benchmarks tested.

5.2 Modeling a Prototype's Dependencies After it is Instanced

One straightforward extension was motivated by 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 dependency graph to derive its own model dependency information; it can use the model dependency graph created for its instances. The algorithm that creates the model dependency graph G for P 's instances (shown in Algorithm 4.3) uses the model parameters of P 's formulas to direct the construction; thus, G provides a more accurate representation of P 's dependencies than the model dependency graph stored in P 's prototype (which P must use until G is created). Therefore, any dependencies that can be modeled for P 's instances can also be modeled for P itself. The net result is that P may be able to model some dependencies that it previously could not when it used its prototype's model dependency graph.

Implementation. When an object obj is instanced for the first time, a model dependency graph G for obj is created as before. Two additional steps are performed during this object creation process:

1. obj 's model dependency graph pointer is redirected from its prototype's model dependency graph to the newly created G , and
2. all explicit dependencies that are attached 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 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.

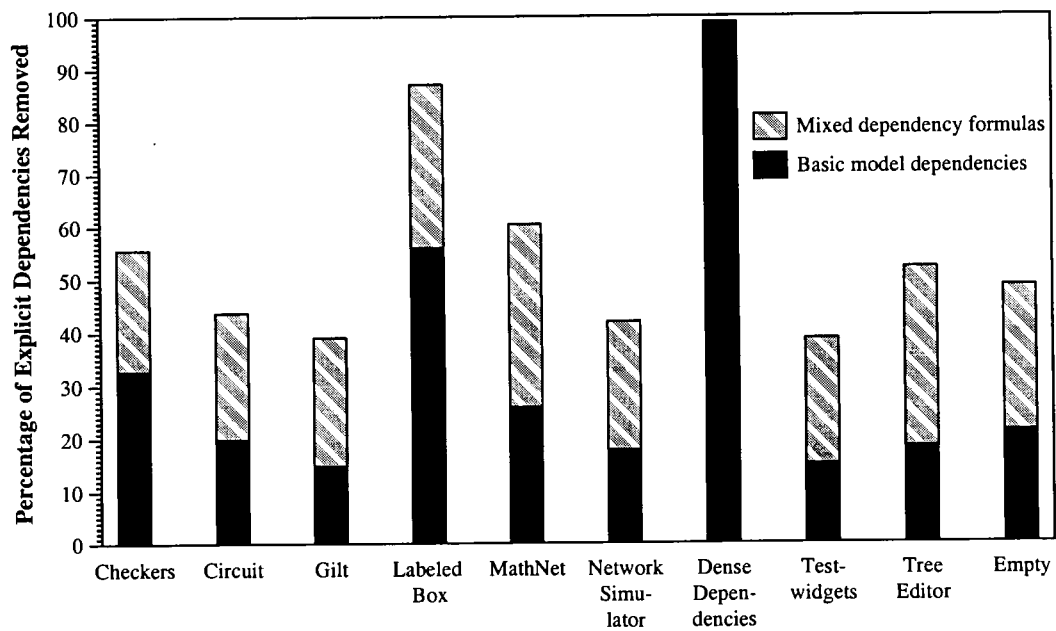


Figure 5.1: The reductions of dependencies when mixed dependency formulas are used. The average for actual applications (that is, excluding the benchmark Labeled Box, Dense Dependencies, and Empty applications) is 47.4%. The average for all ten applications and benchmarks is 56.6%.

Results. This change was not difficult to implement, and it increased the number of removed explicit dependencies by about 2.4%, from 47.4% to 49.8%. Figure 5.2 shows the results for the ten applications and benchmarks tested.

5.3 TO_ALL_CHILDREN Model Dependency

Amulet supports *map objects* which are special kinds of composite objects that generate multiple graphical parts from a single prototype object. Map objects are used when all the children of an object are similar enough to be created from one prototype. One example might be a grid object in which all of its components are rectangles (see Figure 5.3). Each rectangle has the same color, line_style, height, and width, but each rectangle has its own unique position. This position is given as a (left, top) offset from the parent grid's (left, top) location. All the rectangles together form the grid surface of the grid object. The common characteristics can be collected into a grid element prototype, known as the *item prototype*, and the individual position information can be stored in an array. To create the parts of the grid, an instance of the item prototype is created for each element in the position array. Each grid element has an index slot that indicates its order in the list of grid parts. Each grid element uses this index to look up the element's position in the position array; specifically, each grid element can use formulas to specify its position relative to the grid's top left corner:

```
self.left = self.parent.left + self.parent.position_array[self.index].left
self.top  = self.parent.top  + self.parent.position_array[self.index].top
```

Now if the grid object is moved, each grid element component will correctly reposition its (left, top) coordinates to maintain the visual appearance of the grid's surface.

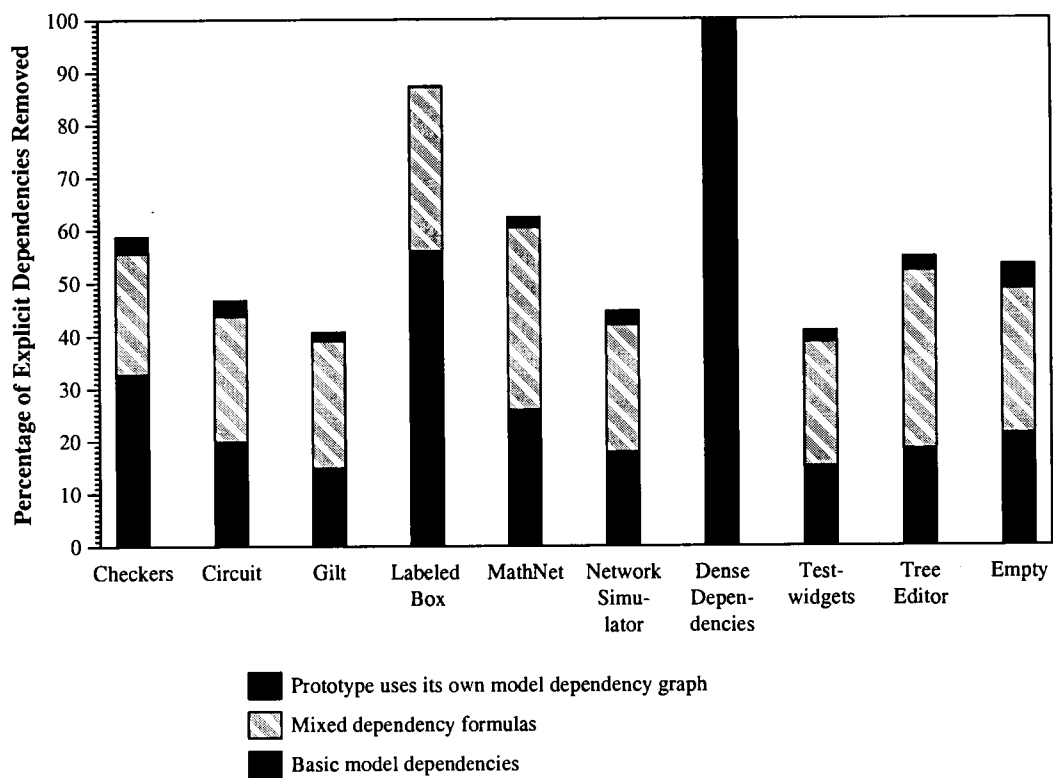


Figure 5.2: The reductions of dependencies when the prototype's dependencies are modeled. The average for actual applications (excluding the benchmark Labeled Box, Dense Dependencies, and Empty applications) is 49.8%. The average for all ten applications and benchmarks is 58.9%.

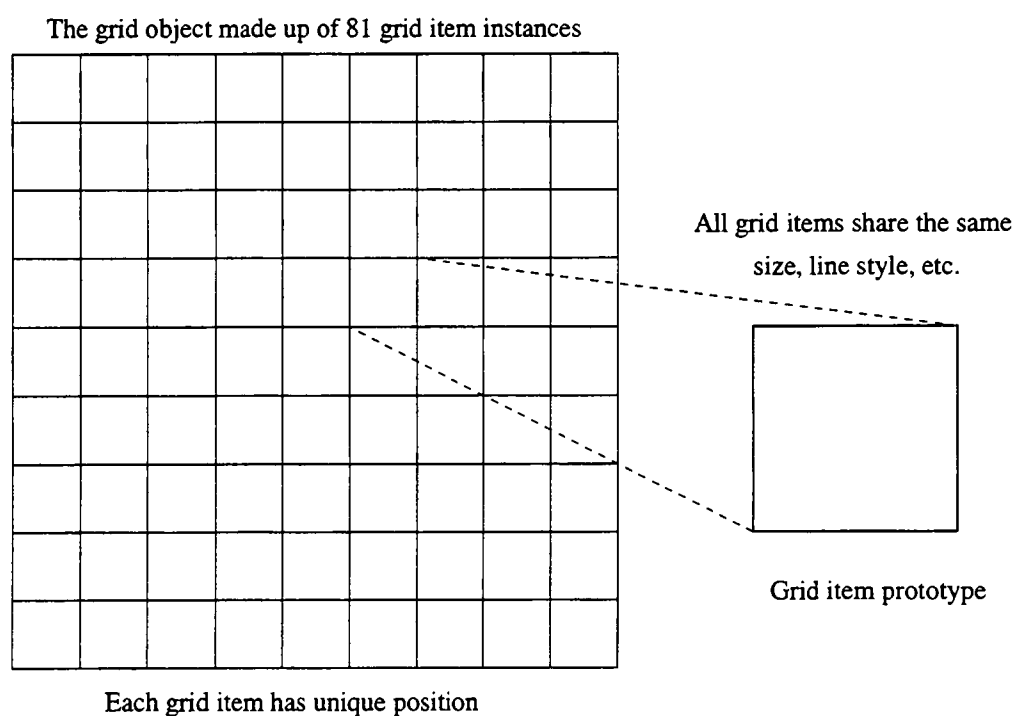


Figure 5.3: A grid object implemented as a map object. Each grid element has a unique value for its left and top slots, but it inherits all other slots like width, height, and line_style from its prototype. The grid elements are generated from a list of (left, top) pairs. This list is stored in the composite grid object's position_array slot; this array is passed as a parameter to the grid object's creation routine.

Map objects are used extensively in graphical applications. For example, the selectable menu items in the drop-down menu in Figure 5.4 are map components; the text labels are provided in an accessory data structure when the menu is created.

The map dependency relationship closely corresponds to the basic CHILD relationship. Unfortunately, the children of a map object are generated automatically from the item prototype and are *unnamed*. A CHILD model dependency edge is a 3-tuple that includes the part's name. This name is necessary so that the resolution algorithm can locate the dependent slot and its constraint. Since maps are widely used in graphical applications, the basic scheme's inability to handle unnamed parts forces a large number of potential model dependencies to be explicitly represented.

What is needed is a model edge that represents a dependency from a slot in a parent to a particular slot in *all* of its children. The TO_ALL_CHILDREN relationship was developed to address this need. It is similar to the CHILD dependency. It is represented as a 2-tuple (TO_ALL_CHILDREN, slot). Unlike the CHILD model edge which is a 3-tuple, no named part is required since the dependency is to *all* children of the object. Figure 5.5 illustrates the TO_ALL_CHILDREN dependency. Observe also that only one model dependency edge is needed to model multiple CHILD dependencies within the same object; thus, the TO_ALL_CHILDREN edge saves a little model dependency storage in addition to the larger explicit dependency storage savings.

A TO_ALL_CHILDREN edge is created when a PARENT model parameter is specified for a formula in a map item prototype object. Thus a PARENT model parameter may produce either a CHILD model dependency or a TO_ALL_CHILDREN model dependency depending on the context of the object in which it is specified. If a PARENT model parameter is specified within a named part, a CHILD model dependency edge is created; if it is specified within a map item prototype, a TO_ALL_CHILDREN model dependency is created.

Implementation. TO_ALL_CHILDREN edges are represented by the `To_All_Children_Model_Edge` class, a subclass of `Model_Edge`.

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.resolve()` method returns a list of constraints to invalidate.

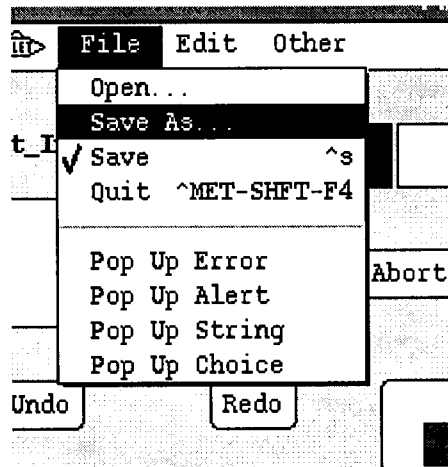


Figure 5.4: A typical drop-down menu implemented as a map object. The labels “Open...,” “Save As...,” etc. are provided in an accessory data structure when the menu is created. The label’s position within the data structure determines its position within the drop-down menu.

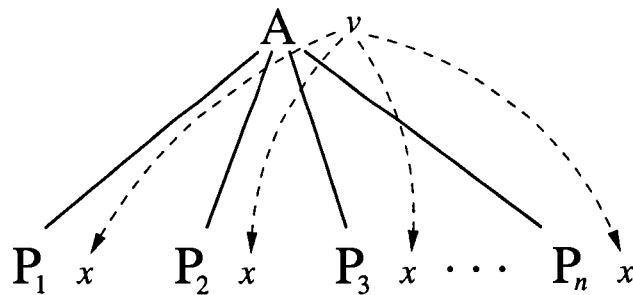


Figure 5.5: The TO_ALL_CHILDREN relationship. The v slot of object A influences the x slot in all of its parts $P_1 \dots P_n$.

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.resolve()` method. Algorithm 5.2 formalizes the `To_All_Children.resolve()` method.

Potential for Duplicate Model Dependencies. Sometimes the item prototype is stored as a named child within a map object. This leads to duplicate model dependencies since any CHILD dependencies within the item prototype will be resolved twice: once for the item prototype when it is considered as a named child, and again when that same item prototype object is visited by the TO_ALL_CHILDREN resolution routine. Fortunately the duplication has no effect on program behavior since it simply causes the constraint system to try to invalidate the same constraint twice. The second time the constraint system tries to invalidate the constraint, it will see that the constraint has already been invalidated, and it will do nothing. The extra effort spent on the second invalidation is negligible.

Results. The TO_ALL_CHILDREN relationship significantly decreases the number of explicit dependencies. The total savings increases from 49.8% to 68.1%, a gain of 18.3%. Figure 5.6 shows the results for the ten applications and benchmarks tested.

5.4 FROM_ALL_CHILDREN Model Dependency

The previous section handled the situation where all the children depend on a slot in the parent. Conversely, some situations demand that a slot be computed from the value of a particular slot found in all of its children. For example, if a composite object is added to a window or its appearance is modified, the display manager must (re)draw it. Typically, the display manager will compute the smallest rectangle that encloses all the parts that comprise the object and only redraw that portion of the screen. To assist the display manager in this regard the composite object often maintains formulas in its width and height slots that automatically compute its width and height. For example, the width formula might be written as:

Method Constraint_Set.invalidate()

```
1  for each constraint cn ∈ self.set :
2      cn.invalidate()
```

Method To_All_Children_Model_Edge.resolve(Slot parameter_slot, Signature sig)

returns Constraint

—The parameter_slot parameter is the slot that changed

—constraint_set is of type Constraint_Set, a subclass of Constraint

```
1  constraint_set.set ← ∅
2  obj ← parameter_slot.object
3  for each part p ∈ obj.parts :
    —The slot name stored in this dependency edge determines the target dependent slot
4      dep_slot ← p.slots[self.slot_name]
5      for each constraint cn ∈ dep_slot.constraints :
6          if cn.signature = sig:
7              constraint_set.set ← constraint_set.set ∪ {cn}
8  return constraint_set
```

Algorithm 5.2: The To_All_Children_Model_Edge resolve method. The resolve() methods of all Model_Edge subclasses must return a single constraint, but the To_All_Children_Model_Edge must actually return a collection of constraints. To reconcile this difference, the Constraint_Set class is introduced. The Constraint_Set is a subclass of Constraint; therefore, it responds to the same messages that the Constraint class does. It has a set field that points to a set of constraint pointers. The Constraint_Set invalidate() method simply delegates the invalidation message to each element of its set field. Note that this design permits an element of a Constraint_Set object's set field to itself be a pointer to a Constraint_Object allowing for nested constraint sets. This Constraint_Set class is also used in Section 5.5. Also note that the resolve method gracefully handles children that do not depend on the slot. For these children no constraint gets added to the constraint set.

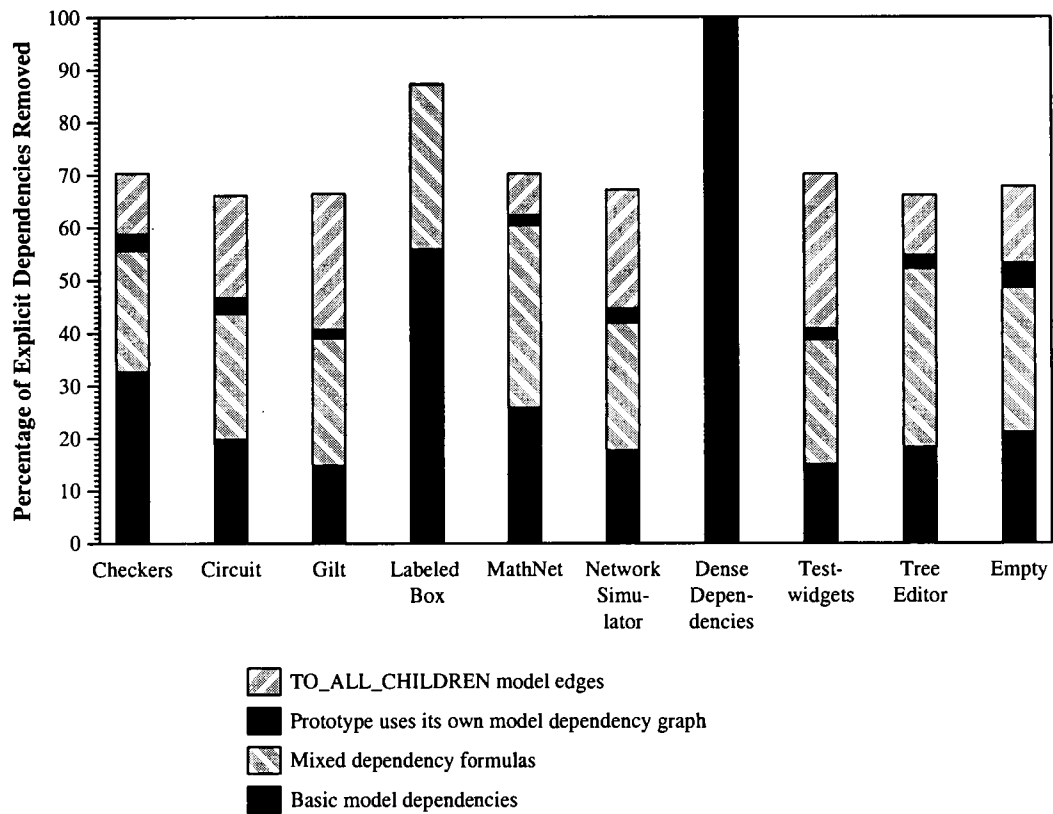


Figure 5.6: The reductions of dependencies including TO_ALL_CHILDREN. The average for actual applications (excluding the benchmark Labeled Box, Dense Dependencies, and Empty applications) is 68.1%. The average for all ten applications and benchmarks is 73.2%.

```
width = {
    p = self.first_part
    left = p.left
    right = p.left + p.width
    for each part p of self:
        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. Amulet uses a formula similar to this one for determining the width of all its graphical objects.

At first glance it appears that the formula should be able to use CHILD model parameters to specify the dependence on each child's left and width slots. However, the CHILD model parameter must be explicitly declared for each child and must include the child's part name. For a number of reasons, these restrictions preclude the width formula from using CHILD model parameters in their existing form:

1. Automatic child generation: If the object's children are automatically generated, as happens for example in a map object, the width formula will reference a variable number of parts. However, the basic model dependency scheme requires a fixed size parameter list.
2. Unnamed parts: The width formula may reference one or more unnamed parts but the CHILD model parameter requires a part name.
3. Use in different objects: The width formula is meant to be used in different composite objects. Even if each object had a fixed set of named parts, the width formula could not specify the correct set of CHILD parameters for each object since the names and number of children vary from object to object. Hence a programmer would be forced to write a specialized width formula for each new type of object.

What is needed is a model edge that represents a dependency from a given slot in every child to a slot in their parent. The `FROM_ALL_CHILDREN` model parameter was developed to address this need. It is a 2-tuple `(FROM_ALL_CHILDREN, slot)` and generates one `PARENT` edge for each child. Figure 5.7 illustrates the edges that are generated.

Implementation. The `FROM_ALL_CHILDREN` relationship is represented by the `From_All_Children_Model_Edge` class, a subclass of `Model_Edge`. A `From_All_Children_Model_Edge` is used as a model parameter to a formula that, like the above width formula, must examine a slot in all of its children. 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 dependency graph called the *common model supertree*. A common model supertree stores the `PARENT` dependencies that are shared by all children. Two relevant points motivated the introduction of a new type of model dependency subgraph:

- It is not sufficient to place a `PARENT` dependency edge in each of the existing model supertrees since model subtrees are associated only with named children present when the parent's model dependency graph was created. The `PARENT` model dependencies in the common model supertree would apply to all children of the composite object, named and unnamed.
- Even in situations where all the children are named and are represented by their own model supertrees,

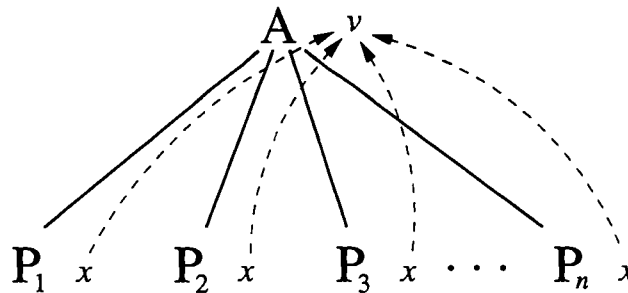


Figure 5.7: An example of the `PARENT` model dependency edges generated by the `FROM_ALL_CHILDREN` relationship. The formula that determines slot v in object A uses the `FROM_ALL_CHILDREN` parameter to indicate that it references the x slot in all of A 's parts $P_1 \dots P_n$. The constraint system in turn creates `PARENT` model dependency edges from the x slot in each part to slot v .

placing the PARENT model dependencies common to all children in one graph avoids duplicating the edge. Like the TO_ALL_CHILDREN edge described in Section 5.3 and as shown in Figure 5.7, one PARENT edge stored in the common model supertree represents multiple model dependency edges within the same object and therefore saves model dependency storage in addition to explicit dependency storage.

Figure 5.8 shows the complete collection of subgraphs that make up a model dependency graph. A model dependency graph may have at most one model subtree, at most one common model supertree, and as many child-specific model supertrees as are necessary.

To install an edge in the common model supertree, a `From_All_Children_Model_Edge` parameter edge must be specified in a formula. The system cannot infer one by examining each child and deducing common model PARENT dependencies. The `From_All_Children_Model_Edge.install()` method, shown in Algorithm 5.3, adds the proper edge, creating the common model supertree if necessary.

Results. The FROM_ALL_CHILDREN relationship provides a modest decrease in explicit dependencies, saving an additional 5.8%. The cumulative savings is therefore boosted to 73.9%. Figure 5.9 shows the results for the ten applications and benchmarks tested.

```
Method From_All_Children_Model_Edge.install(Object obj,
                                           Slot dependent_slot, Signature sig)
1  graph ← obj.parent.prototype.common_model_supertree
2  new_edge ← Parent_Model_Edge.create(dependent_slot)
3  new_dependency ← Model_Dependency_Edge.create(new_edge, sig)
   —Note that the first argument to the next method call is this parameter edge's slot name
   —which is the source of the new dependency edge. The new dependency edge is stored
   —with this slot name in the common model supertree.
4  graph.insert_edge(self.slot_name, new_dependency)
```

Algorithm 5.3: The From_All_Children install method

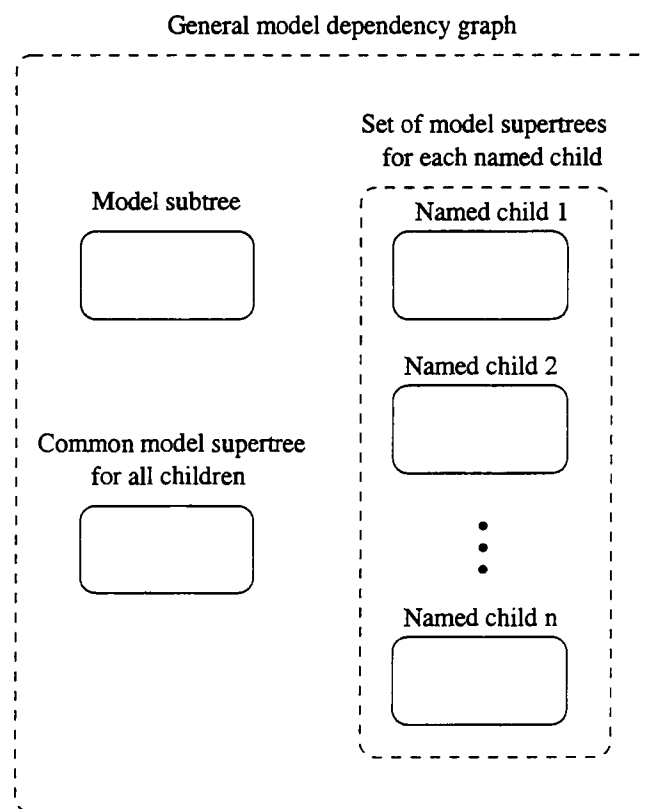


Figure 5.8: The collection of subgraphs that constitute a prototype's model dependency graph.

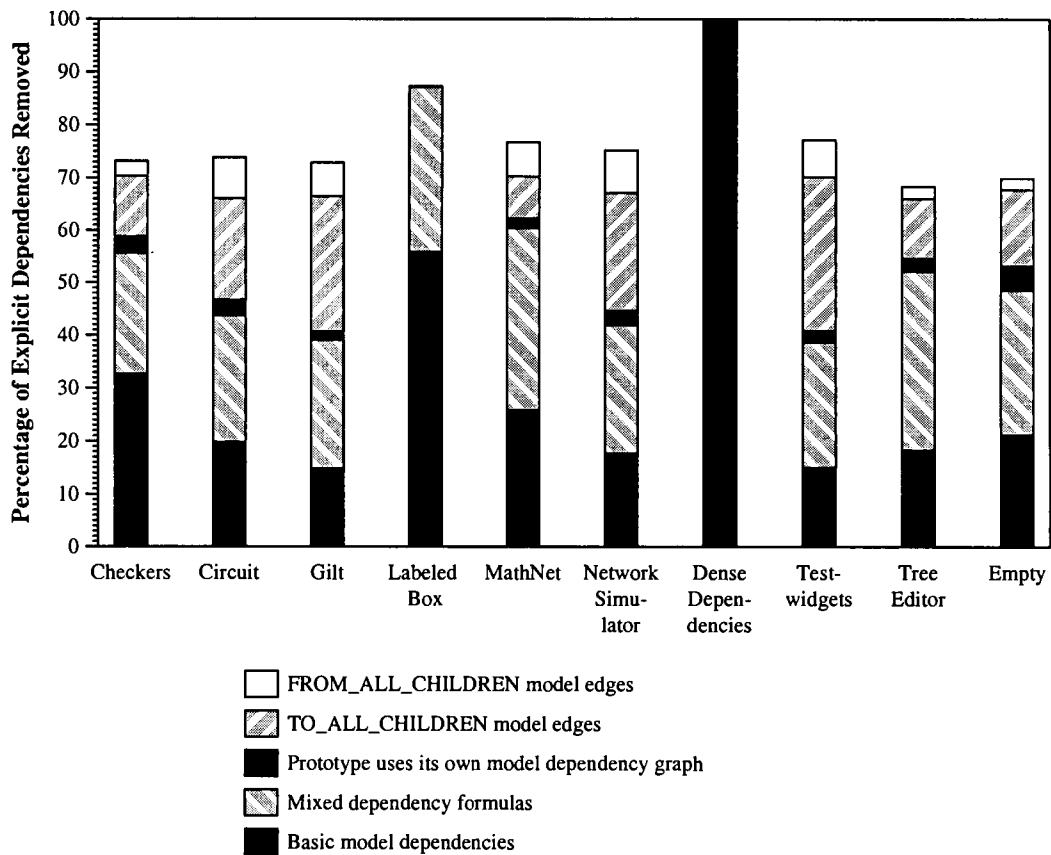


Figure 5.9: The reductions of dependencies including FROM_ALL_CHILDREN. The average for actual applications (excluding the benchmark Labeled Box, Dense Dependencies, and Empty applications) is 73.9%. The average for all ten applications and benchmarks is 77.5%.

5.5 Model Dependencies for Global Objects

The extensions to model dependencies that have been described thus far in this chapter exploit the local part hierarchy of a composite object. The last extension to be described in this chapter takes advantage of the fact that many formulas may reference the same global object stored at the root of a composite object. For example, computing some properties of a graphical object may require access to lower-level system information involving the graphical device context of the screen. Such information includes screen resolution, number of available colors, and font size, and may be used to compute the width and height of an object, the way an image is displayed, and the way that a text object is displayed, respectively. This means that a dependency must be established from the device context to selected slots in almost every graphical object in the application. Figure 5.10 shows how all objects might depend on a slot, such as the device context slot in the root object in the hierarchy.

Since all graphical objects are ultimately part of some window which ultimately is part of the screen, the composite object notion is still viable. In this case, if the device context slot of the screen is changed, every window and graphical object in the application can be notified by recursively visiting all components of the interface. This process is expensive, but in practice device context changes are rare so the tradeoff of the elimination of a large number of explicit dependencies for a rarely executed, relatively time-consuming process is a good one.

Implementation. A new `Global_Model_Edge` class is subclassed from `Model_Edge` to implement model dependencies for global objects. The GLOBAL relationship is expressed in a model parameter by specifying a 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 is stored in the model subtree of the specified global object. The `install()` is shown in Algorithm 5.4. Only one GLOBAL model dependency edge needs to be created since a GLOBAL model dependency edge does not need to store any state information. The edge (GLOBAL) stored at vertex v in graph G means that changing slot v in an object obj that has model dependency graph G may

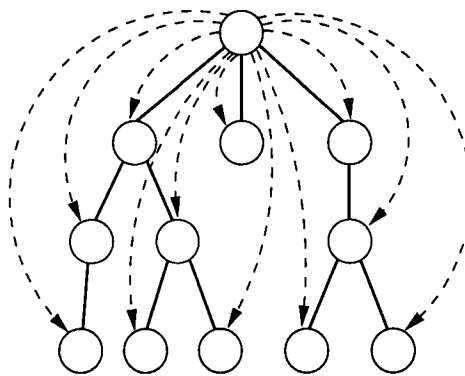


Figure 5.10: A situation where all objects depend on a slot in the root of the composition hierarchy. The dashed edges indicate the dependency dataflow. Each edge originates from the same slot in the root object and ends at the same slot in each object.

Method Global_Model_Edge.install(Object obj, Slot slt, Signature sig)

—The obj parameter is the global object accessed by the formula

- 1 graph $\leftarrow$ obj.model_subtree
 —Note that only one GLOBAL model edge object is required since no
 —other information needs to be stored except the fact that
 —the edge is a GLOBAL dependency.
 —The global_model_edge object passed to the
 —Model_Dependency_Edge.Create() method is this single GLOBAL edge object
- 2 new_dependency $\leftarrow$ Model_Dependency_Edge.create(global_model_edge, sig)
 —Note that the first argument to the next method call is this parameter edge's slot name
 —which is the source of the new dependency edge. Hence the new GLOBAL dependency
 —edge should be added to slot_name's list in the model dependency graph.
- 3 graph.insert_edge(self.slot_name, new_dependency)

Method Global_Model_Edge.resolve(Slot parameter_slot)

returns Constraint_Set

- 1 constraint_set.set $\leftarrow \emptyset$
 —Determine the object in which the slot resides
- 2 obj $\leftarrow$ parameter_slot.object
- 3 obj.collect_global_dependencies(constraint_set, parameter_slot)
- 4 **return** constraint_set

Method Object.collect_global_dependencies(Constraint_Set cn_set,
 Slot parameter_slot)

- 1 **for** each child p $\in$ obj.parts :
- 2 **for** each slot s $\in$ p.slots :
- 3 **for** each constraint cn $\in$ s.constraints :
- 4 **for** each model parameter param $\in$ cn.formula.model_parameters :
- 5 **if** param.type = Global_Model_Edge
 and param.slot_name = parameter_slot.name :
- 6 cn_set.set $\leftarrow$ cn_set.set $\cup$ {cn}
- 7 p.collect_global_dependencies(cn_set, parameter_slot)

Algorithm 5.4: Algorithms to install and resolve Global_Model_Edge dependencies. Every descendent of the root of the composite object is visited. Each slot of every object is examined. All the constraints computing those slots are examined. If the constraint is modeled and therefore has model parameters, each parameter is checked to see if 1) it is a Global_Model_Edge parameter, and 2) its slot matches the slot passed to the Global_Model_Edge.resolve() method. If both criteria are met, then the constraint is added to the set of constraints that must be invalidated. The Constraint_Set class, a subclass of Constraint, is described in Algorithm 5.2. An initially empty Constraint_Set object is passed to collect_global_dependencies() by reference so the method may modify its contents.

affect any slot in all objects below *obj* in the composition hierarchy. Any slot that has a formula which references the slot in this global object will be affected, but most generally will not. Algorithm 5.4 shows the `resolve()` method for the `Global_Model_Edge` class. Before the `resolve()` method adds a constraint to the set of constraints to be invalidated it must first examine the constraint's list of model parameters (stored in the model formula responsible for this constraint). If a GLOBAL model parameter containing slot *v* is found, then the constraint is added to the list of constraints to invalidate.

Results. The GLOBAL relationship provides a modest decrease in explicit dependencies, saving an additional 4.0%. The cumulative savings including all the implemented extensions for typical Amulet applications (excluding the special benchmarks) is 77.9%. The results for the tested applications and benchmarks appear in Figure 5.11. Figure 5.12 illustrates the savings obtained by each of the extensions.

Approximately 78% of the dependencies were consistently eliminated by the model dependency scheme for the applications tested. As the primary criterion for judging the success of the research, explicit dependency removal results show that the model dependency scheme presented in this dissertation performs quite well. Henceforth the term *Model Amulet* will refer to the Basic Model Amulet system augmented with the extensions presented in this chapter.

It is important to note that the extensions presented in this chapter require only one change to the mark algorithm of Algorithm 4.4. The `get_model_dependencies()` method of Algorithm 4.4 needs to be modified as shown in Algorithm 5.5 so that model dependencies are also retrieved from the common model supertree of the parent's prototype.

5.6 Additional Extensions

A systems-level programmer may find other avenues for creating custom model dependencies that extend the currently supplied collection. For example, an application may make frequent use of a formula with GRANDPARENT parameters. While not directly supported by the current implementation, this relationship still can be handled within the context of composite objects. In particular, the `Model_Edge` hierarchy, shown

Method `Slot.get_model_dependencies()` **returns** Set of Model Dependency Edges

- Retrieve the *SELF* and *CHILD* model dependencies from the prototype's model
- subtree, the *PARENT* and *SIBLING* model dependencies from the model supertree of the parent's prototype, and the *PARENT* dependencies due to *FROM_ALL_CHILDREN* parameters
- from the common model supertree of the parent's prototype.

```

1  return self.object.prototype.model_subtree[self.name]
   ∪ self.object.parent.prototype.model_supertree[self.object.part_name][self.name]
   ∪ self.object.parent.prototype.common_model_supertree[self.name]

```

Algorithm 5.5: Modified `get_model_dependencies()` method. This minor modification to the method found in Algorithm 4.4 allows the mark algorithm to properly handle the common model supertree dependencies used by the *FROM_ALL_CHILDREN* relationship.

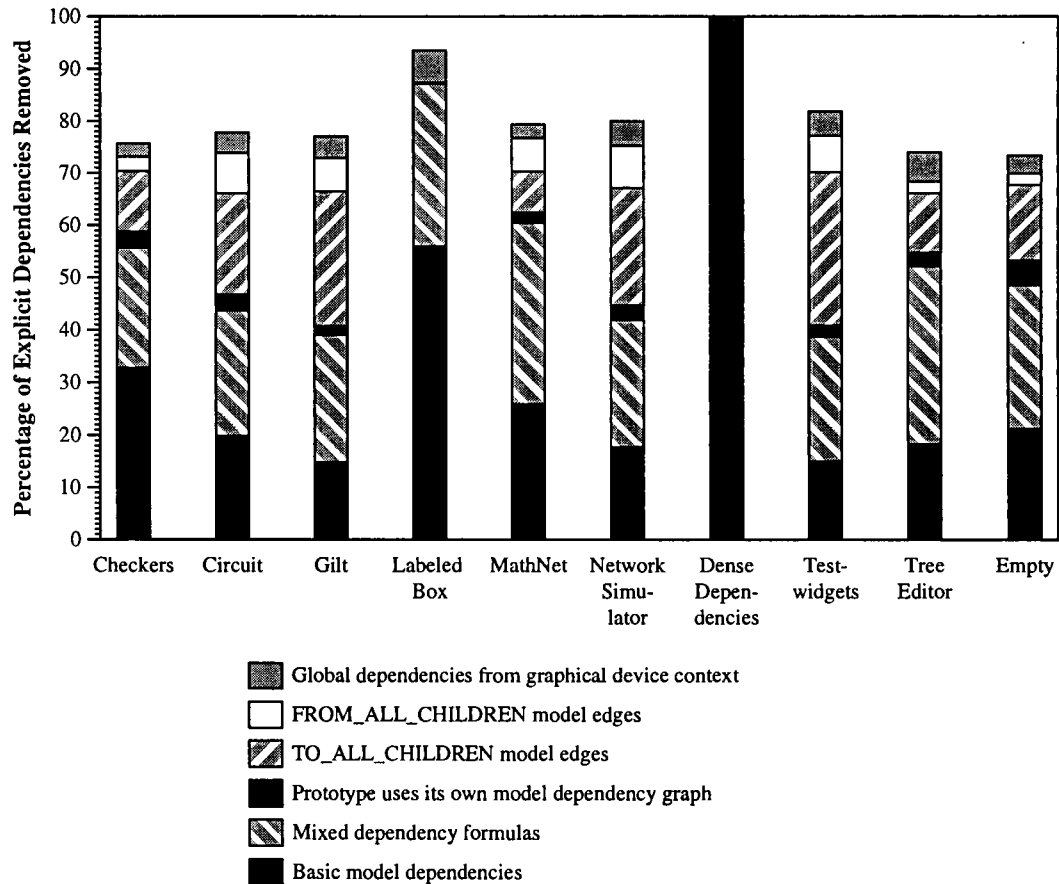


Figure 5.11: The reductions of dependencies contributed by all the extensions. The average for actual applications (excluding the Labeled Box, Dense Dependencies, and Empty applications) is 77.9%. The average for all ten applications and benchmarks is 81.2%.

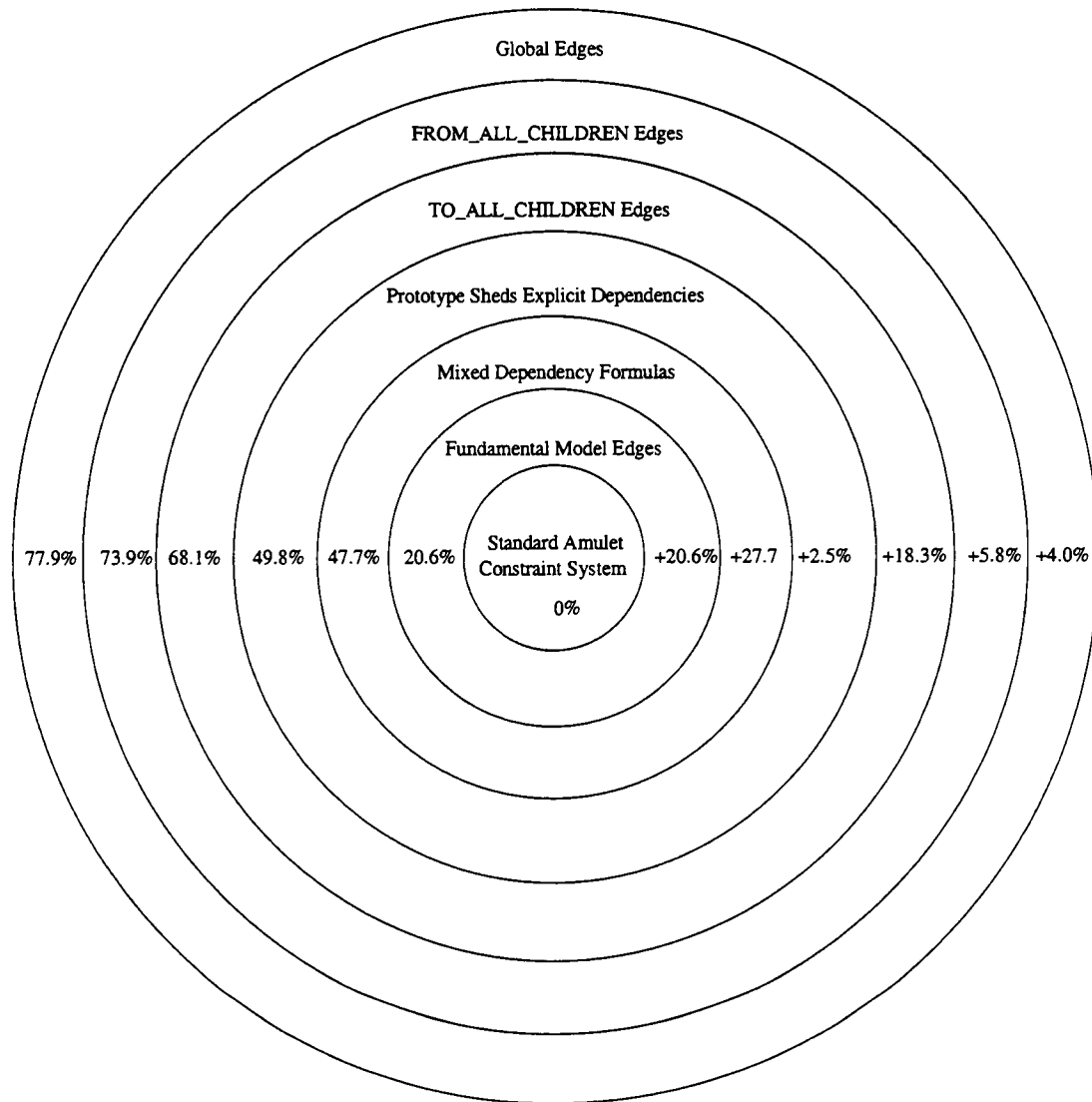


Figure 5.12: The average percentage reductions of dependencies contributed by the extensions. Both the incremental and cumulative effects are shown. The percentages are based on an analysis of actual Amulet applications and exclude the benchmarks.

with the current extensions in Figure 5.13, can be extended by a toolkit developer with an understanding of how model dependencies work. The toolkit developer would define an appropriate edge class and then write custom `resolve()` and `install()` methods for the desired edge. In general, the typical applications-level programmer would not create new types of model dependencies.

5.7 Time Complexity of Model Dependency Resolution

This section examines the time required to invalidate a model dependency. Explicit dependencies can be invalidated in $O(1)$ time because the mark phase follows a pointer directly to the constraint to be invalidated. As the following analysis shows, the 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 dependency graphs , and
2. resolving each model dependency into a 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. The model subtree and common model supertree require $O(1)$ time to locate since explicit fields point to these graphs in the object's prototype and in the parent's prototype. The model supertree has to be retrieved from the set of model supertrees in the parent's prototype. If the set is implemented as a hashtable, the retrieval will on average require $O(1)$ time. Since an object's named parts are normally bounded by a small constant, a list is often a better choice for this set. In this case, the access is still $O(1)$. 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. Overall, locating model dependencies should require $O(1)$ time.

Model Dependency Resolution. Resolving a model dependency into a constraint involves three steps: 1) locating the dependent part by traversing the composite object's part hierarchy, 2) locating the dependent

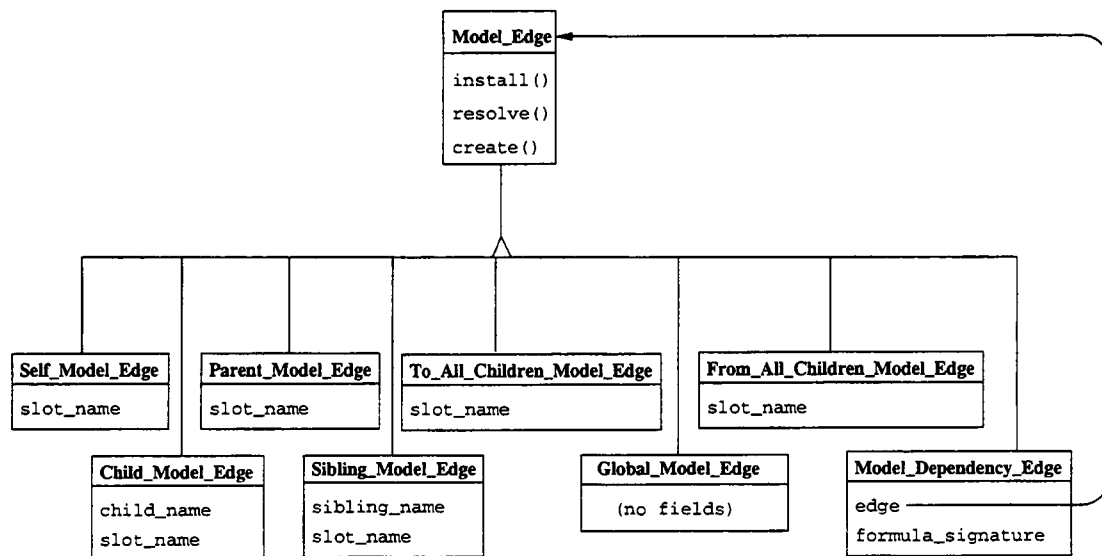


Figure 5.13: The extended class hierarchy for model edges including the `To_All_Children_Model_Edge`, `From_All_Children_Model_Edge`, and `Global_Model_Edge` classes. Notice that the `Global_Model_Edge` class uses no fields, so a single `Global_Model_Edge` object can be created and used wherever a GLOBAL edge is needed.

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 basic edges and the FROM_ALL_CHILDREN relation all 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. TO_ALL_CHILDREN 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 part 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. The number of explicit dependencies replaced by this model dependency could be as low as zero or as high as $O(n)$, so the time cannot be amortized. Hence, the time to locate all the dependent children is $O(n)$. Fortunately, a global object containing a slot such as a device context is almost never modified (indeed, it is almost unheard of), so the time complexity is acceptable. Also in practice a GLOBAL model edge eliminates $O(n)$ explicit dependencies since each object in the interface typically has at least one dependency on a global object. Hence in practice a GLOBAL model edge can locate a dependent part in $O(1)$ amortized time.

Once the dependent part is located the dependent slot must be found. 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. The list's size is almost always 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 global model edges each of the three phases of dependency resolution requires $O(1)$ time, so the resolution 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 by a small constant factor. As Section 6.2 indicates, this small speed penalty is quite acceptable for most applications. One exception to this analysis is the global model edge which in the worst case requires $O(n)$ time to invalidate, where n is the number of objects in the interface. In practice the edge requires $O(1)$ amortized time to invalidate constraints and hence is only slower than explicit dependencies by a constant.

5.8 Spurious Invalidations

As mentioned in Section 4.8, conditional formulas can result in spurious invalidations where a constraint is invalidated by following a model dependency that, due to the current state of the application, would not exist as an explicit dependency in a non-modeled system. Analysis of the dependencies that lead to spurious invalidations reveals that on average about 5.5% of the total dependencies lead to spurious invalidations. In other words, if a model dependency was replaced by an explicit dependency edge in each instance that it models, 5.5% of the resulting edges would be unnecessary or, in our terminology, spurious. The spurious invalidations lead to extra formula evaluations. Profiling has shown that on average the percent increase in formula evaluations for modeled applications is 6.5%. In actual applications this time overhead for spurious invalidations and formula evaluations is not significant. Since a dependency responsible for a spurious invalidation is modeled and not explicit, it does not consume any space. The model dependency edge must be present in the model dependency graph since the state of the application may change causing the dependency's constraint invalidation to be essential rather than spurious.

Chapter 6

Empirical Test Results

While the efficacy of model dependencies in terms of explicit dependency reduction is demonstrated in Chapter 5, this chapter considers other criteria for success. Section 6.1 uses a specialized benchmark to reduce the overwhelming resource burden imposed on Amulet graphical objects (see Section 4.9) and measures the influence of model dependencies on system space and time resources. Section 6.2 addresses interactive performance issues.

6.1 Performance on the Dense Dependencies Benchmark

As noted in Section 4.9, measuring raw heap space for most Amulet applications does not necessarily give an accurate picture of the benefit of model dependencies. Graphical objects under Amulet consume an enormous amount of space aside from space required by formulas and explicit dependencies. In particular, slot objects consume a significant amount of space over and above that required by constraints. Since each graphical object stores an average of 30–40 slots, the storage for slots dominates the storage for constraints and their corresponding explicit dependencies in the tested applications.

The *Dense Dependencies* benchmark. Benchmarks can be devised that reduce the overwhelming storage burden incurred by the total Amulet system. These benchmarks can be used to further validate the

model dependency scheme by directly measuring the space and time performance of model dependencies. One such benchmark is the *Dense Dependencies* program. It defines a Dense Dependencies object that is derived directly from the root object in Amulet's prototype-instance hierarchy. This simple object does not incur as much overhead as Amulet's graphical objects because the only slots stored in the object are those used by the benchmark.

Each Dense Dependencies object consists of 26 slots, 25 of which are determined by formulas. If the slots are named $s_0, s_1, \dots, s_{25}$, then a formula F_i in slot s_i references slots $s_0 \dots s_{i-1}$ (slot s_0 does not contain a formula). Hence all 25 formulas generate a total of $1 + 2 + \dots + 25 = 325$ dependencies. The large number of dependencies per object compensate for Amulet's heavyweight base object system. The benchmark performs the following activities:

1. The prototype Dense Dependencies object is defined and the formulas are assigned to their slots. The formulas all involve the *SELF* relationship; consequently, Model Amulet can eliminate the explicit dependencies.
2. A specified number of Dense Dependencies objects are created from the prototype.
3. In a loop executed 10,000 times, one object is chosen at random from the collection of objects. A fixed slot, s_0 , is modified. s_0 is guaranteed to affect all the formulas in that object.

This benchmark is useful for two reasons:

- It is non-interactive. It is not subject to the inconsistent pace of a human user.
- As the number of objects grows, the Amulet heavyweight object overhead is dwarfed by the storage cost of the explicit dependencies. The raw heap measurements for this benchmark therefore give a more accurate picture of 1) memory required for explicit dependencies versus memory required for model dependencies and 2) the performance cost of spilling into virtual memory.

The Dense Dependencies benchmark therefore represents a fairly pure test of model dependencies' effect on the system's space and time resources.

Space. 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 6.1 summarizes the benchmark's behavior under both Original and Model 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 constraint dependency consists of a forward and backward edge, each of which requires 12 bytes of storage. Since each edge is allocated separately, the heap manager tacks on an extra four bytes of bookkeeping code for each edge. The true cost of an Amulet explicit dependency is therefore 32 bytes. Analysis of the heap space results reveals that the difference in storage between the two versions is exactly $(32 \text{ bytes}) \times (\text{the number of objects}) \times (\text{the number of dependencies per object})$. This simple comparison is possible since all dependencies in the benchmark are modeled. In any given system the expected storage savings from using model dependencies can be computed as follows:

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

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 4.8 indicate that $\alpha = .78$ is a reasonable estimate.

Time. Figure 6.2 compares the cumulative time it takes to perform the experiment in which a loop is executed 10,000 times. Each iteration invalidates slot s_0 and then all dependent formulas are re-evaluated. The modeled version is consistently 1.42 times slower than the standard version up to about 1,500 objects. This slowdown is understandable since an explicit dependency can invalidate a dependent constraint through a direct reference while a model dependency must be resolved from the prototype's model dependency graph given the actual object's structure. Section 6.2 reveals that this apparently large time discrepancy between

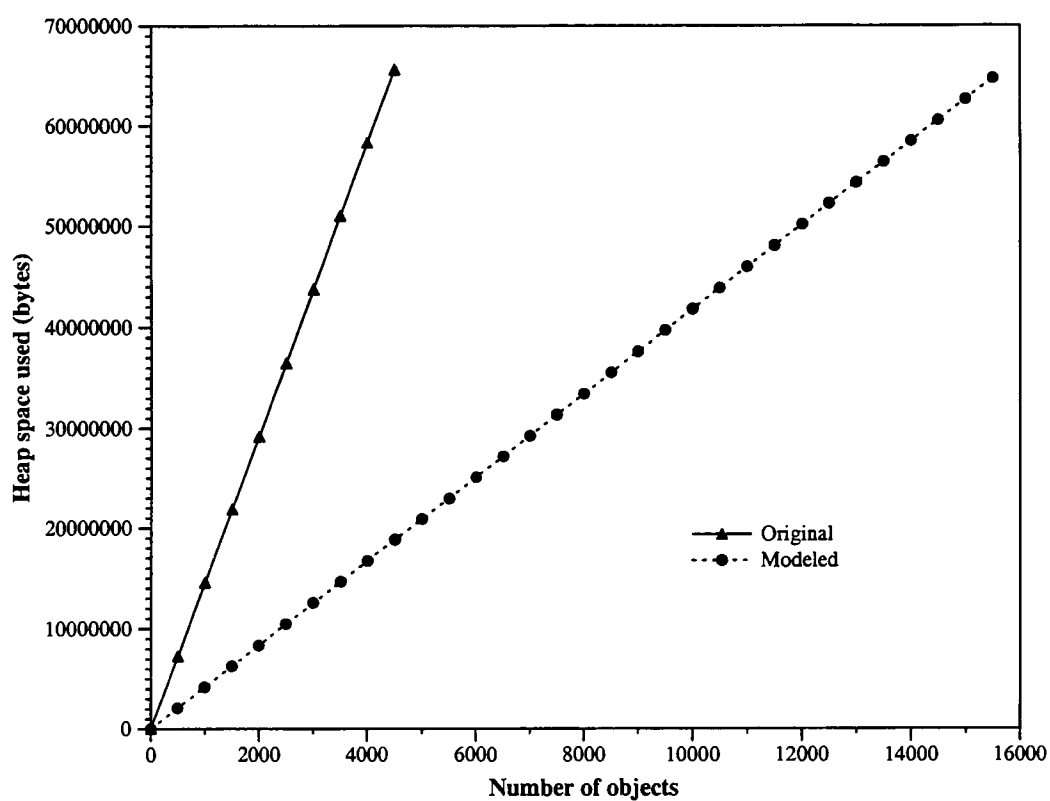


Figure 6.1: Memory requirements of objects with dense dependencies

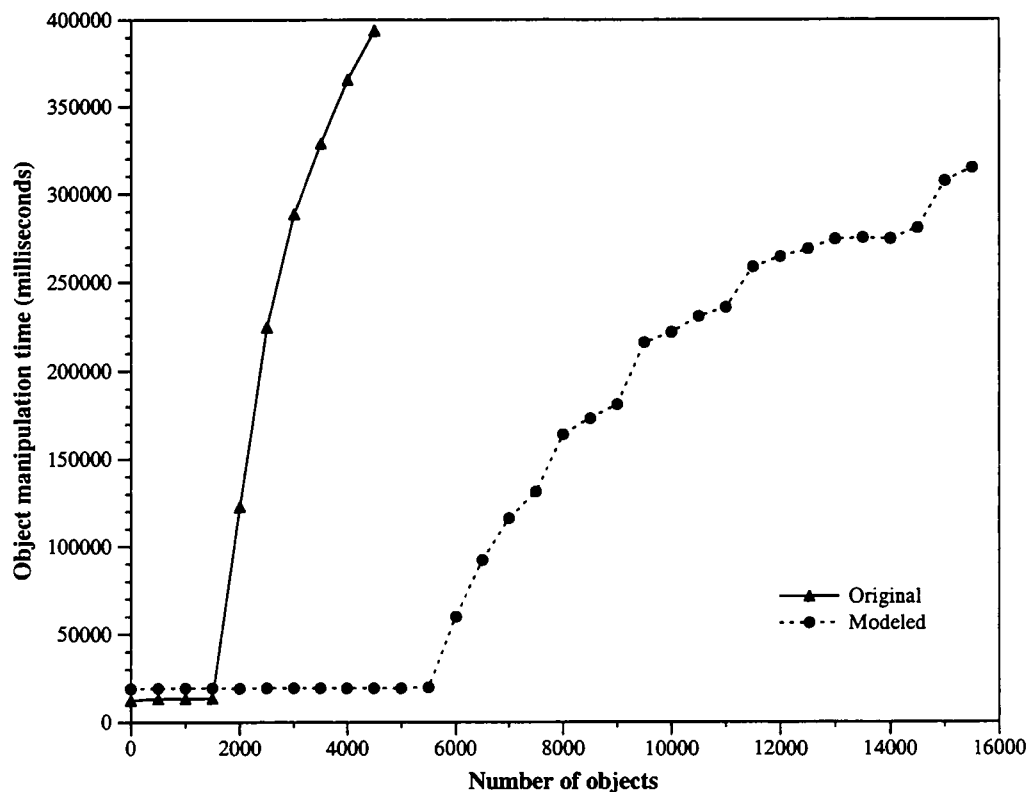


Figure 6.2: Cumulative object manipulation times for the *Dense Dependencies* benchmark. The original version of Amulet begins to use a significant amount of virtual memory at around 1,500 objects. The modeled version can manage about 5,500 objects before virtual memory is required. Once virtual memory is used in the modeled version, as the number of objects increases, 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.

the two versions is not in practice as detrimental as it seems (see Section 6.2).

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 can 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 vs. 15,500 for the modeled version), the modeled version is about 1.3 times faster.

Although the Dense Dependencies benchmark is a somewhat contrived benchmark, it is more demonstrative of what might be accomplished with model dependencies in many lighter weight constraint-based systems. As shown by Figures 6.1 and 6.2, 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.

6.2 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. As the Dense Dependencies benchmark showed, the performance penalty for model dependency resolution for programs using only a small amount of memory is about 42%. Model dependencies would be impractical for general use if they significantly degraded the performance of small interactive applications.

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 percep-

tible loss of interactive performance compared to their explicit dependency counterparts. In fact, during a three minute interactive session with the Testwidgets application the amount of extra time required specifically for model dependency creation and resolution amounted to approximately 0.8 seconds. This result is less surprising considering the following two points:

1. Previous studies have shown that redisplay time dominates the cost of all other operations in a graphical application [14, 41]. Thus, the time spent traversing dependencies is a tiny fraction of the time required to execute an operation.
2. In interactive applications, typically the total number of slots that depend on a changed slot is small [43].

Point 1 says that in normal constraint systems constraint satisfaction time is insignificant compared to updating the display. This suggests that slowing the constraint satisfaction time by a factor of 1.42 as indicated by the Dense Dependencies benchmark may have little overall effect. The second point says that dependency trees are usually short and narrow. The Dense Dependencies benchmark executes 10,000 object manipulations and each manipulation affects 325 dependencies and forces the recomputation of 25 constraints (each of which uses an average of 12 individual arithmetic operations for its computation). The re-evaluation work required for each interactive event in a typical application will generally be far less, so while the Dense Dependencies benchmark is a good measure of the raw performance of model versus explicit dependencies, it presents a worst-case scenario compared to typical graphical applications. When a graphical (but still non-interactive) version of the Labeled Box benchmark was tested, which moved and resized (by changing the label text) a large number of labeled boxes in a window, the modeled version was slower than the original version by a factor of only 1.002. (The number of boxes was not so large as to require either version to use virtual memory.) This time penalty of 0.2% is not perceptible to the human user.

Chapter 7

Related Work

This chapter considers related work from a variety of different angles. It first shows how dataflow constraints fit within the constraint solving world. It then describes some of the past research that has been done on optimizing the storage costs of dataflow constraints. Finally it discusses two related ideas, model dependency graphs in syntax-directed language editors and flyweight patterns, that influenced and motivated the development of the model dependency scheme described in this dissertation.

7.1 Dataflow Constraints in Context

All constraint systems share the notion of programming as a declarative task. Relations among a set of programming objects are specified and the constraint solver finds a solution that satisfies these relations [23]. 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 [13]. 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 user can give a linear equation solver the pair of constraints

$$\begin{array}{rcl} x & + & y = 10 \\ x & - & 2y = 1 \end{array}$$

and the linear equation solver can use its knowledge of linear algebra to satisfy these 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.

Multi-way, multi-output constraints. Multi-way, multi-output dataflow constraint systems have also been developed for graphical interfaces, including ThingLab [3, 11, 36], Kaleidoscope [10] and Multi-Garnet [35, 34, 39]. Multi-way constraints are more powerful than one-way constraints. Their primary advantage is that they allow information to flow both ways in an equation, either from the left side to the right side or from the right side to the left side. For example, they cleanly handle circular loops, such as expressing the two-way relationship

$$a = b$$

As another example, they can express the relationship

$$\text{right} = \text{left} + \text{width}$$

and handle changes to right by changing either left or width. However, circular loops are rarely problematic in one-way constraint systems and generally the programmer wants information to flow one way. For example, position information in a composite object tends to flow from the root to the parts and the programmer is not interested in a reciprocal relationship in which position information flows from a part to the root.

Multi-way constraints are also less predictable than one-way constraints and harder to manage by the programmer. First, multi-way constraints can lead to overconstrained systems. In this case one or more constraints must be relaxed (i.e., not satisfied). Constraint hierarchies have been introduced to allow pro-

grammers to prioritize constraints so that less important constraints are relaxed before more important constraints [4]. However, most graphical applications are actually underconstrained which creates a different problem, how to choose which variables should change and which variables should stay the same when edits occur. For example, in the equation

$$\text{right} = \text{left} + \text{width}$$

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 [4]. In the above example, the programmer could introduce stay constraints for all three variables, but give a higher priority to 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 [32] 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.

Constraint Logic Programming. A final related area is constraint logic programming [21]. When applied to logic programming, constraints provide a generalization of the resolution principle used in standard logic programming languages such as Prolog [33]. In fact, standard logic programming techniques are a proper subset of constraint logic programming techniques. Particularly, unification is simply a special case of constraint solving. 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 μ constraints which reduce the cost of a constraint by encoding its dependencies in a compact manner.

Constant propagation. *Constant propagation* allows some constraints to be removed from the system completely [26, 28]. If all the parameters of a constraint are constant, the constraint can be evaluated and replaced with the constant result. The elimination of this constraint may permit the removal of other constraints as well. Since constraints often have multiple dependencies, the savings in a fairly static system where many slots have fixed values may be significant. Unfortunately one non-constant slot in a formula will prevent it from being eliminated, just as one non-modelable parameter prevents a formula from being modeled in the basic model dependency scheme presented in Chapter 4. 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 [43]. This reduction is a good start, but other optimizations, such as the model dependency optimizations described in this dissertation, must be used to obtain even more storage savings.

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* [9]. A plan consists of a single Smalltalk module and is created by analyzing the dependency graph dataflow, “unwrapping” individual constraint methods, and sequencing the code segments that make up these methods into one large procedure that becomes a method in the module. The parameters to that procedure are the slots in the original graph whose values may be modified. The constraint plan approach is ideal for dependency 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.

μ constraints. Hudson and Smith reduce the physical storage required for dependencies in certain com-

mon graphical layout relationships using a concept they call μ constraints [19]. Each constraint consists of as little as 17 bits, which is enough to encode

1. an operation code (e.g., addition, subtraction),
2. a small set of predefined dependencies to other objects such as parent or first_child, and
3. a small predefined set of potential parameter variables, such as left or width.

Actual dependency edges in the constraint network are dynamically inferred as needed by consulting μ constraint encodings and the composite object's physical structure. Heavyweight constraints that establish explicit dependencies are used where μ constraints are insufficient to specify the desired relationships completely.

μ constraints share two important concepts with the model dependency paradigm:

1. Relations 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.

Compared to model dependencies, however, μ constraints have some serious restrictions:

1. μ constraints can only be attached to a small set of layout slots. Model dependency constraints can be attached to any slot.
2. The formulas may have at most one supplied slot parameter and at most one implied slot parameter depending on the formula selected. Model dependency formulas allow an arbitrary number of model parameters.
3. A μ constraint dependency can only point to a restricted set of layout slots. A model dependency can point to any slot.

4. Named children and siblings are not supported. The child and sibling relations are limited to `FIRST_CHILD`, `LAST_CHILD`, `MAXIMUM_CHILD`, `MINIMUM_CHILD`, `PREVIOUS_SIBLING`, and `NEXT_SIBLING`. Model dependencies are more general.
5. The formulas are limited to six different predefined layout operations. Model dependency formulas consist of arbitrary code.
6. Mixed dependencies are not supported by μ constraints.
7. Once an encoding has been established, it does not appear that μ constraints can be as easily extended to handle additional dependency relations because the number of bits allocated for dependency relations is fixed. For example, if three bits are allocated for dependency relations, only eight relations can be modeled. To increase the representable dependency relations the number of bits must be increased, which requires the decoding software to be rewritten. With model dependencies, additional model edges can be created by a systems programmer by subclassing the `Model_Edge` type.

An analysis was undertaken to compare the dependency reducing capability of μ constraints to that of model dependencies. Figure 7.1 summarizes the results.

Figure 7.1 shows that μ constraints are not very successful at reducing dependencies in actual applications. The small percentage of dependencies eliminated by μ constraints mirrors the results with the basic model dependency scheme of Chapter 4: the basic four relationships do not represent many dependencies in real applications. While model dependencies eliminated almost 20% of the dependencies, the results are even worse for μ constraints since the number of encodable slots and types of formula functions supported are a subset of those supported by model dependencies. Most practical applications, especially those that use widgets, use many formulas that rely on non-layout slots such as `visible` or `fill_style`. At the beginning of the model dependency research, a variety of Amulet applications and the Amulet system code was examined. Under the basic model dependency scheme, excluding all the extensions in Chapter 5, we found that about 55% of the formulas can be modeled. With μ constraints, the number of modelable formu-

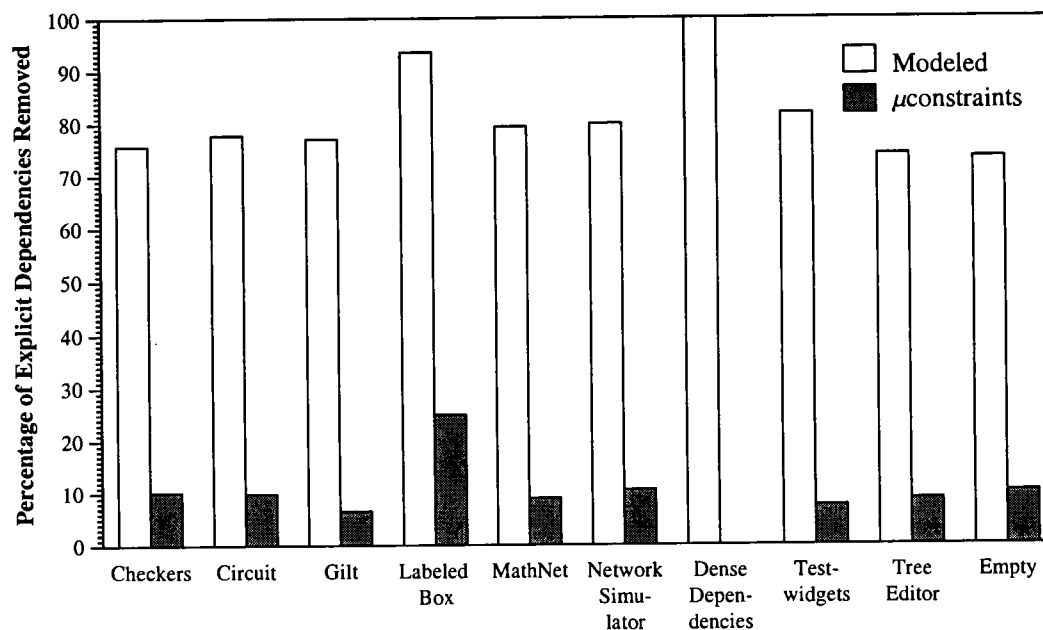


Figure 7.1: Comparison of μ constraints to Model Amulet. μ constraints were able to eliminate only a few of the dependencies used by the tested applications. The average percentage of explicit dependencies removed by μ constraints for “typical” applications (that is, Labeled Box, Dense Dependencies, and Empty excluded) is 8.9%. The average for all ten applications and benchmarks is 9.7%. Applications that use widgets extensively fare especially poorly because widgets use formulas for a wide variety of computations, many of which are not layout related. μ constraints failed to model any Dense Dependencies dependencies since none of the Dense Dependencies objects’ formulas use layout slots, hence the missing bar.

las is about 38.5%; therefore, it is not surprising that μ constraints on average model fewer than half of the dependencies modeled by the *basic* model dependency scheme.

It is interesting to compare the work required to resolve a μ constraint dependency compared to a model dependency. The relation encoded in a μ constraint is actually a parameter path. Search is involved to determine the actual dependencies. For example, consider object *obj* whose layout slot width is modified. The constraint solver must poll every potential dependent slot within *obj* itself, *obj*'s parent, all of *obj*'s children, and *obj*'s next and previous siblings. Each of these potentially dependent slots is examined to see if it has a μ constraint that encodes a path back to *obj*.width. Fortunately, the number of potentially dependent slots is small per object, but the number of objects to consider may be large if *obj* has many parts. Recalling the analysis of Section 5.7 a model dependency can be resolved in constant time with no search involved. In contrast, some search of the composite object structure is required to resolve a μ constraint. The potential cost of resolving all the dependencies for a μ constraint is $O(n)$, where n is the number of parts in an object. Because the modified slot may have no dependencies, this cost cannot be amortized. The number of parts of an object is normally bounded by a small constant, so in practice the search should take constant time.

Given a set of dependencies that can be represented by both μ constraints and model dependencies, it seems that the model dependency approach will typically be faster. The cost of resolving μ constraint dependencies is fixed in advance because the same slots will always be searched. In contrast, the cost of resolving model dependencies depends on the number of model dependencies. A survey of Amulet showed that most slots have two or fewer dependencies [43]. Hence, the model dependencies approach will typically visit fewer slots. Both model dependencies and μ constraints must traverse the object hierarchy to resolve dependencies, so the cost of visiting a dependent slot is roughly comparable. On average, therefore, the model dependency approach should be faster.

7.3 Influencing Ideas

Model Dependency Graphs. The model dependency graph concept was applied to attribute grammars by Reps, *et. al.* [31] in their work with syntax-directed programming editors. The text of a computer program can be represented by an attributed derivation tree for the grammar of the given programming language. User editing operations correspond to modifications to the attributed abstract syntax tree; for example, adding a keyword might result in expanding a non-terminal in a production rule. Attribute information flows among the vertices of the tree in the same way data flows among slots in a constraint network. Model graphs are used to represent the dependencies among variables in the vertices of a subtree. The model constraint graphs used in this dissertation, however, out of necessity have fewer restrictions; a model constraint graph's structure is not limited by the productions of a formal language. The restricted constraint graphs that arise from the attribute grammars that generate these trees can be analyzed statically and this information plus the model graphs obviate the need to store any explicit dependencies at run time. The problem addressed in this dissertation deals with more general constraint graphs where the relationships among the vertices can be arbitrary; any object can be dependent upon any other object. As mentioned earlier in this dissertation, this generality renders a pure model dependency scheme largely ineffective.

Flyweight Design Pattern. Calder and Linton used lightweight *glyphs* to implement structured graphical objects [6]. These glyphs were implemented in an extension to the InterViews toolkit [25]. 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 prototype. This technique of not storing information that can be reconstructed by other means when needed is formalized in the *flyweight* design pattern [12].

Chapter 8

Conclusions

This chapter explores some directions for extending our research, followed by a final assessment of the benefits of model dependencies and their contribution to the body of knowledge in computer science.

8.1 Future Work

The research described in this dissertation can be extended in a number of different directions to try to produce greater versatility for the model dependency paradigm, an improved programmer interface, and increased storage savings.

Greater Versatility. As mentioned in Section 3.6, a model dependency graph is not altered once a prototype has been instantiated, even if the prototype is subsequently modified (adding a part to a prototype is the one exception to this rule). One useful extension to this research would allow the prototype's model graph to be updated if the prototype is edited. Unlike the existing Amulet model, the edits would be inherited by all of its instances (currently instances created before the edit do not inherit the edits, but instances created after the edits do). Since the edits would be inherited, the model dependency graphs of any descendent instances in the prototype-instance hierarchy would also have to be updated.

Applications 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 constraint graph, Amulet frees the programmer from this burden and automatically constructs the dependencies during the formula's execution. In other words, when a slot is requested by a formula, Amulet's constraint solver can automatically create an explicit dependency from that slot to the formula's constraint (see Algorithm 4.6). Hence, compared with the original version of Amulet, the current application programmer interface to model dependencies has the following shortcomings:

- Model formulas require the programmer to provide a model parameter list specification, and
- The programmer is responsible for making sure the number of model parameter edges and the specified parameter paths agree with the code in the formula. Any inconsistencies result in incorrect run time behavior that may be difficult to diagnose and fix.

These shortcomings could be overcome by removing the requirement for a parameter list specification and inferring model dependencies. The constraint solver's mechanism for explicit dependency formation would then have to be extended to handle the creation of model dependencies. If the path between the formula and the requested variable could be modeled, then an explicit dependency would not be established. Instead, a model edge would be added to the model dependency graph if it is not already there. There are two problems that must be overcome to make this approach feasible:

1. Different executions of the same formula may produce different sets of model dependencies. For example, if the formula has a conditional, then one branch may produce one set of model dependencies and another branch may produce a different set of dependencies. One solution to this problem might be to take the union of the different sets of model dependencies and another solution might be to take the intersection of the different sets of model dependencies. The union approach might produce spurious constraint invalidations but would result in a greater storage savings. The intersection approach would avoid spurious constraint invalidations but would require a greater number of

explicit dependencies and hence incur a greater storage cost. Experimental work would be required to determine which approach produces the best results.

2. Since the constraint system cannot know when all the possible branches of a formula's code have been executed, each formula execution will force the constraint solver to check for new model dependencies. These checks could significantly slow down the execution of a formula. Algorithmic approaches that minimize the time spent creating model dependencies would need to be investigated.

Another possibility for eliminating the explicitly provided model parameter list is to build a preprocessor that analyzes a formula's code and infers the model parameters. The formulas output by this translator would look like the current model formulas (that is, they would have the required model parameter lists). The translator's output would then be passed on to the C++ compiler as usual. If such a translator could be built, applications would not suffer the penalty of runtime model parameter determination; model parameters would be deduced at compile time by the preprocessor. Such an implementation would run as fast as the current Model Amulet.

This solution is not trivial to design and implement since the translator would have to perform significant dataflow analysis¹ to deduce the parameter slots. Since formulas can contain arbitrary code, a function called from the formula (and functions called from those functions, etc.) must be traced to see if any slots are accessed within those functions. The preprocessor must therefore have the full analysis capabilities of a complete compiler for the target language. In general, a complete analysis is impossible, since such static analysis is undecidable [22].

One possibly useful compromise could generate "easily" detected parameters such as those used explicitly within the formula's function body. Any parameters accessed via function calls within the formula or through pointer aliases would go undetected and therefore lead to explicit dependencies.

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 4.2). However, some implementation knowledge

¹Here the term *dataflow analysis* refers to the static analysis performed by optimizing compilers.

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.

Additional storage savings under Amulet. Model dependencies may enable some additional storage optimizations in Amulet:

- A number of slots that could be inherited are currently stored locally in an object because they need to represent explicit dependencies. The local storage for these slots can be eliminated using model dependencies, since the explicit dependencies can be generated by examining the prototype. A preliminary profile of Amulet suggests that about 6% of slots could be eliminated in this fashion.
- Many slots maintain a pointer for explicit dependencies. An alternative approach is to store explicit dependencies in a hashtable or binary tree keyed by slot name. When a slot is modified, the slot's dependencies can be retrieved from the hashtable or tree. For added efficiency, a flag could be stored with each slot indicating whether it has explicit dependencies. In the original Amulet, so many slots have explicit dependencies that this scheme might not save space because of the additional storage overhead of the search structure. However, model dependencies might reduce the number of slots with explicit dependencies to the point where this alternative approach might save storage.

Incremental model dependency graph construction. Currently each prototype maintains a complete model dependency graph; that is, any dependency information required by an object can be obtained from one graph. Storage could be further reduced if each prototype's model dependency graph stored only

the differences from its prototype's model dependency graph. For example, if object `Box` has a model dependency graph with 100 model dependency edges and its instance `Labeled_Box` adds one edge, then `Labeled_Box`'s graph would store one edge instead of 101. The disadvantage to this approach is that a potentially long prototype-instance chain must be searched to locate all model dependency edges. If the graph changes involve more than simple edge additions (for example, if edges are deleted in a subprototype graph), the process may become complicated. The model dependency premise is that many duplicate instances will be created from one prototype, so the tradeoff of graph duplication for fast access is probably good, especially since graph edges are allocated from a registry to avoid duplicating edges (see Section 4.4).

8.2 Conclusions

This dissertation has introduced a number of techniques that allow model dependencies to reduce the number of explicit dependencies in a constraint graph significantly. One of the principle findings in this dissertation is that a scheme based only on representing model dependencies among children and their parents looks appealing in theory but is not useful in practice. In particular, while over 50% of the formulas defined by users require only `SELF`, `PARENT`, `CHILD`, and `SIBLING` dependencies, these formulas create only about 20% of the dependencies in an actual application.

Further investigation revealed the characteristics of formulas that were creating the largest number of non-modelable dependencies:

1. Formulas that had a significant number of modelable dependencies but one or two non-modelable dependencies.
2. Formulas that create `PARENT` or `CHILD` dependencies to unnamed parts or variable lists of parts.
3. Formulas that depended on a global system resource, such as a device context.

The model dependency paradigm was thus augmented to allow mixed dependencies in formulas, to allow a single modeled edge to describe a list of edges, and to allow custom edges that reference a specific

global resource. With these extensions, almost 80% of the dependencies in our benchmark applications can be modeled.

The significance of the techniques developed in this dissertation 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 dissertation 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. Such improvements in storage efficiency may encourage the incorporation of constraints into the design and evolution of mainstream programming languages, much in the same way garbage collection and threads have become an integral runtime feature of some modern languages.

The ultimate goal in constraint optimization research is to devise mechanisms that greatly reduce or even eliminate the penalties normally imposed by the use of constraints in computer programs. Programmers should have the freedom to build constrained applications with the assurance that the resulting executables will be nearly as space and time efficient as equivalent systems that do not use constraints. The result will be a net gain in programmer productivity since constraint-based programs can be developed much more quickly and in a more modular fashion than the standard approach. The research described in this dissertation provides another step towards this goal of making constraints practical for mainstream software development.

Bibliography

Bibliography

- [1] ALPERN, B., HOOVER, R., ROSEN, B. K., SWEENEY, P. F., AND ZADECK, F. K. Incremental evaluation of computational circuits. In *ACM SIGACT-SIAM'89 Conference on Discrete Algorithms* (Jan. 1990), pp. 32–42.
- [2] ARNOLD, K., AND GOSLING, J. *The Java Programming Language*. Addison-Wesley, Reading, Massachusetts, 1996.
- [3] BORNING, A. The programming language aspects of ThingLab, a constraint-oriented simulation laboratory. *ACM Transactions on Programming Languages and Systems* 3, 4 (Oct. 1981), 353–387.
- [4] BORNING, A., DUISBERG, R., FREEMAN-BENSON, B., KRAMER, A., AND WOOLF, M. Constraint hierarchies. In *OOPSLA'87 Conference Proceedings* (1987), pp. 48–60.
- [5] BORNING, A. H. Classes versus prototypes in object-oriented languages. In *Proceedings of the ACM/IEEE Fall Joint Computer Conference* (Nov. 1986).
- [6] CALDER, P. R., AND LINTON, M. A. Glyphs: Flyweight objects for user interfaces. In *ACM SIGGRAPH Symposium on User Interface Software and Technology, Proceedings UIST '90* (Snowbird, Utah, Oct. 1990).
- [7] CHAMBERS, C., AND UNGAR, D. Making pure object-oriented languages practical. In *OOPSLA '91 Conference Proceedings* (Oct. 1991).

- [8] DEMERS, A., REPS, T., , AND TEITELBAUM, T. Incremental evaluation for attribute grammars with application to syntax-directed editors. In *Proceedings of the Principles of Programming Languages Conference* (Williamsburg, VA, Jan. 1981), pp. 105–116.
- [9] FREEMAN-BENSON, B. N. A module mechanism for constraints in Smalltalk. *Sigplan Notices* 24, 9 (Oct. 1989). ACM Conference on Object-Oriented Programming; Systems Languages and Applications; OOPSLA '89.
- [10] FREEMAN-BENSON, B. N. Kaleidoscope: Mixing objects, constraints, and imperative programming. In *OOPSLA/ECOOP'90 Conference Proceedings* (1990), pp. 77–88.
- [11] FREEMAN-BENSON, B. N., MALONEY, J., AND BORNING, A. An incremental constraint solver. *Communications of the ACM* 33, 1 (Jan. 1990).
- [12] GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Massachusetts, 1995.
- [13] GOLUB, G. H., AND VAN LOAN, C. F. *Matrix Computations, 2nd Ed.* The Johns Hopkins University Press, Baltimore, MD, 1989.
- [14] HILL, R. D. The RENDEZVOUS constraint maintenance system. In *ACM SIGGRAPH Symposium on User Interface Software and Technology, Proceedings UIST '93* (Atlanta, Georgia, Nov. 1993).
- [15] HUDSON, S. E. Eval/vite user's guide (v1.0). Tech. rep., College of Computing Georgia Institute of Technology, Atlanta, Georgia.
- [16] HUDSON, S. E. Incremental attribute evaluation: A flexible algorithm for lazy update. *ACM TOPLAS* 13, 3 (July 1991), 315–341.
- [17] HUDSON, S. E. Incremental attribute evaluation: A flexible algorithm for lazy update. *ACM Transactions on Programming Languages and Systems* 13, 3 (July 1991).

- [18] HUDSON, S. E. User interface specification using an enhanced spreadsheet model. *ACM Transaction on Graphics* 13, 3 (July 1994), 209–239.
- [19] HUDSON, S. E., AND SMITH, I. Ultra-lightweight constraints. In *ACM SIGGRAPH Symposium on User Interface Software and Technology, Proceedings UIST '96* (Seattle, Washington, Oct. 1996).
- [20] IBM CORPORATION. Lotus 1-2-3. 1998.
- [21] JAFFAR, J., MICHAYLOV, S., STUCKEY, P. J., AND YAP, R. H. C. The CLP($\mathcal{R}$) language and system. *ACM Transactions on Programming Languages and Systems* 14, 3 (July 1992).
- [22] LANDI, W. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems* 1, 4 (Dec. 1992), 323–337.
- [23] LELER, W. *Constraint Programming Languages: Their Specification and Generation*. Addison-Wesley, Reading, Massachusetts, 1988.
- [24] LIEBERMAN, H. Using prototypical objects to implement shared behavior in object oriented systems. *Sigplan Notices* 21, 11 (Nov. 1986), 214–223. ACM Conference on Object-Oriented Programming; Systems Languages and Applications; OOPSLA'86.
- [25] LINTON, M. A., VLISSIDES, J. M., AND CALDER, P. R. Composing user interfaces with InterViews. *IEEE Computer* 22, 2 (Feb. 1989).
- [26] MALONEY, J., BORNING, A., AND FREEMAN-BENSON, B. Constraint technology for user-interface construction in ThingLabII. *Sigplan Notices* 24, 10 (Oct. 1989). ACM Conference on Object-Oriented Programming Systems Languages and Applications; OOPSLA '89.
- [27] MICROSOFT CORPORATION. Microsoft Excel. 1998.
- [28] MYERS, B. A., GIUSE, D. A., MICKISH, A., AND KOSBIE, D. 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 1994.

- [29] MYERS, B. A., GUISE, D. A., DANNENBERG, R. B., VANDER ZANDEN, B., KOSBIE, D. S., PERVIN, E., MICKISH, A., AND MARCHAL, P. Garnet: Comprehensive support for graphical highly interactive user interfaces. *IEEE Computer* 23, 11 (Nov. 1990).
- [30] MYERS, B. A., MCDANIEL, R. G., MILLER, R. C., FERRENCY, A., FAULRING, A., KYLE, B. D., MICKISH, A., KLIMOVITSKI, A., AND DOANE, P. The Amulet environment: New models for effective user interface software development. *IEEE Transactions on Software Engineering* 23, 6 (June 1997).
- [31] REPS, T., TEITELBAUM, T., AND DEMERS, A. Incremental context-dependent analysis for language-based editors. *ACM Transactions on Programming Languages and Systems* 5, 3 (July 1983). Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages, January, 1982.
- [32] ROSENER, W. J. Integrating multi-way and structural constraints into spreadsheet programming. Ph.D. thesis, Department of Computer Science, University of Tennessee, 1994.
- [33] ROUSSEL, P. *PROLOG: Manuel de Reference et d'Utilisation*. Group d'Intelligence Artificielle, Université de Marseilles, 1975.
- [34] SANNELLA, M. Skyblue: A multi-way local propagation constraint solver for user interface construction. In *ACM SIGGRAPH Symposium on User Interface Software and Technology* (Marina del Rey, CA, Nov. 1994), Proceedings UIST'94, pp. 137–146.
- [35] SANNELLA, M., AND BORNING, A. Multi-Garnet: Integrating multi-way constraints with Garnet. Tech. Rep. 92-07-01, Department of Computer Science and Engineering, University of Washington, Sept. 1992.
- [36] SANNELLA, M., MALONEY, J., FREEMAN-BENSON, B., AND BORNING, A. Multi-way versus one-way constraints in user interfaces: Experiences with the DeltaBlue algorithm. *Software Practice and Experience* 23, 5 (1993), 529–566.

- [37] VAN ROSSUM, G. *Python Reference Manual*. Reston, Virginia, 1996.
- [38] VANDER ZANDEN, B. Optimizing toolkit-generated graphical interfaces. In *ACM SIGGRAPH Symposium on User Interface Software and Technology* (Marina del Rey, CA, Nov. 1994), Proceedings UIST'94, pp. 157–166.
- [39] VANDER ZANDEN, B. An incremental algorithm for satisfying hierarchies of multi-way, dataflow constraints. *ACM Transactions on Programming Languages and Systems* 18, 1 (January 1996), 30–72.
- [40] VANDER ZANDEN, B., MYERS, B. A., GIUSE, D. A., AND SZEKELY, P. Integrating pointer variables into one-way constraint models. *ACM Transactions on Computer Human Interaction* 1, 2 (June 1994), 161–213.
- [41] VANDER ZANDEN, B. T. Optimizing toolkit-generated graphical interfaces. In *ACM SIGGRAPH Symposium on User Interface Software and Technology* (Marina del Rey, California, Nov. 1994), Proceedings UIST'94, pp. 157–166.
- [42] VANDER ZANDEN, B. T., MYERS, B. A., SZEKELY, P., GIUSE, D., MCDANIEL, R., MILLER, R., HALTERMAN, R., AND KOSBIE, D. Lessons learned about one-way, dataflow constraints in the garnet and amulet graphical toolkits. To be published.
- [43] VANDER ZANDEN, B. T., AND VENCKUS, S. A. An empirical study of constraint usage in graphical applications. In *ACM SIGGRAPH Symposium on User Interface Software and Technology, Proceedings UIST '96* (Seattle, Washington, Oct. 1996).

Vita

Richard Halterman was born in Lakeland, Florida on May 4, 1958. He attended schools in the Polk County public school system where he graduated with high honors from Lakeland High School in June, 1976. He received a Bachelor of Science degree in Chemistry from Florida Southern College, Lakeland, Florida in May, 1979. For six years he taught mathematics and science classes for grades 7–9 at the Fort Lauderdale-Plantation Seventh-day Adventist School in Plantation, Florida. During that time he worked on a Master of Science in Mathematics at Florida Atlantic University in Boca Raton, Florida, which he completed in December, 1987. Since July, 1987 he has been teaching in the computer science department (now School of Computing) at Southern Adventist University, Collegedale, Tennessee. In August of 1994, he entered The University of Tennessee, Knoxville to pursue the Doctor of Philosophy Degree in computer science which was received in May, 1999.