Using Model Dataflow Graphs to Reduce the Storage Requirements of Constraints

Richard L. Halterman and Bradley T. Vander Zanden
University of Tennessee
{halterma,bvz}@cs.utk.edu

1 Introduction

Dataflow constraints (also called one-way or spreadsheet-style constraints) allow programmers to easily 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.

Constraints are increasingly being integrated into drawing packages and interface development toolkits [18, 19, 11, 10, 14]. Unfortunately, studies of at least two of these toolkits, Garnet [18] and Amulet [19] have shown that constraints can exact a significant storage toll on programs [27]. Ultimately the execution times of programs managing a large number of constrained objects suffer since virtual memory must be accessed to meet their storage demands.

One reason constraints require so much storage is that most dataflow constraint systems explicitly represent the relationships among constraints by maintaining a *constraint graph* (see Section 2). The edges in these graphs can consume a considerable amount of storage (in Amulet, the system in which we have performed our test implementation, each constraint dependency is represented by a forward edge and a backward edge, each of which requires 12 bytes of storage). In applications that use thousands of constraints, the space consumed by dependencies can limit the number of objects that can be created.

In this paper we present a solution to the dependency problem that is based on the observation that objects that use the same constraints have the same constraint graph. Consequently, we can store a pattern of a constraint subgraph, a *model constraint 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.

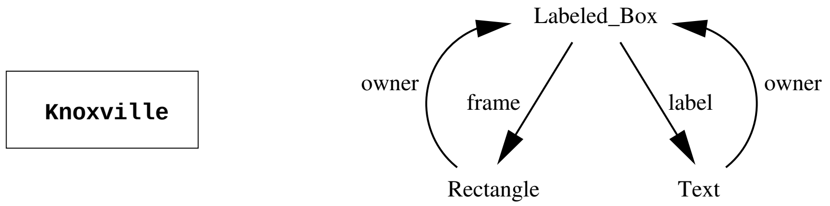
Our solution is inspired by the Reps *et. al.* idea of using supertree-subtree graphs to implicitly represent a constraint graph [20]. However, Reps dealt with restricted types of constraint graphs and restricted types of edits whereas our problem deals with arbitrary constraint graphs and arbitrary edits. Consequently, while our solution incorporates the Reps, *et. al.* idea of supertree-subtree graphs to reduce the number of explicit dependencies, we also support explicit dependencies that do not conform to our modeling scheme. A more detailed comparison between the two problems is presented in Section 6.

Our experiments show that over 50% of the explicit dependencies in most applications can be eliminated with model dependencies.

2 Background

2.1 Composite Objects

The constraints described in this paper are implemented in the context of composite objects. A *composite object* is an object made up of parts consisting of other objects[9]. For example, Figure 1 illustrates a simple composite object, a labeled box consisting of a text string enclosed within a rectangle.



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

Figure 1: A labeled box object (a) and its structural components (b)

Each component of the labeled box has `left`, `top`, `width`, and `height` *instance variables*, or *attributes*. The `label` part also has a text string and font attributes. The labeled box has named pointers to its children (`frame` and `label`) and the children have named pointers to their parent (`owner`). (See Figure 1b.) These pointers allow the labeled box to access instance variables in its parts and the parts to access instance variables in their parent and in their siblings.

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 1 should be slightly larger than the text label, and the label should be centered within the frame. Figure 2a shows the constraints that specify these relationships.

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

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

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. 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*.

Constraint Graphs. The network of variables and constraint objects and their associated relationships form a constraint graph. The vertices of a constraint graph are constraints and variables. A variable has a directed edge to a constraint if the constraint uses that variable as a parameter. A constraint has a directed edge to a variable if the constraint computes that variable. Many systems also maintain backward edges that point from constraints to their label:

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

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

--Label is centered horizontally within its owner

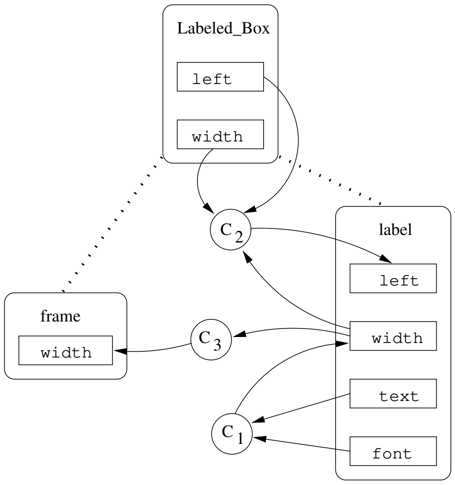
```
C2  left = self.owner.left
           + self.owner.width/2
           - self.width/2
```

frame:

--Frame is slightly wider than the label it encloses

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

(a) Constraint equations



(b) Constraint graph

Figure 2: Labeled box constraint equations and constraint graph

input variables. These backward edges allow dependencies to be removed if a constraint is deleted. Figure 2b shows the constraint graph for the labeled box equations in Figure 2a (backpointers are omitted).

Constraint Satisfaction. In this paper 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 variable are marked `invalid`, and 2) a *sweep* phase in which constraints are brought up-to-date by evaluating their formulas [5, 20, 12, 25]. The mark phase is our primary interest in this paper, since it uses the dataflow edges in the constraint graph to determine which constraints must be marked `invalid`.

2.3 Editing Model

Our editing model is similar to one supported by a prototype-instance model [4, 15]. An application can create instances of any object. All attributes, parts, and formulas associated with the object are inherited by the new instance. The original object is called the *prototype* of the new instance.

Until an object is instanced, the following editing operations are permitted:

- parts may be added to the object
- parts may be removed from the object
- formulas may be added to instance variables
- formulas may be removed from instance variables

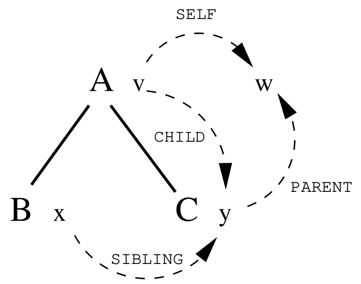
Once an object is instanced, we assume that no further editing operations are permitted on that object.

While the system we describe in this paper uses the prototype-instance model, our techniques would work equally well under the class-instance model. The task would even be easier since a class statically specifies an object and the class may not be dynamically edited.

3 The Model Dependency Paradigm

As noted in the previous section, an instance object inherits a prototype's constraints, and by extension, its constraint graph. In a conventional constraint system, storage for this constraint graph is duplicated for each new instance. In contrast, our scheme stores a model constraint graph in the prototype. The instances 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.

The relationships that currently can be expressed with model dependencies include `SELF`, `CHILD`, `PARENT`, and `SIBLING`. Figure 3 shows how these relationships are defined.



`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: Model edge relationships

A model dependency consists of either a two-tuple or a three-tuple that represents a path in the composition hierarchy from a given variable to its dependent variable. Given an instance variable v of object `A` and a model dependency edge, the actual dependent variable w can be resolved as follows:

Edge	Resolution of actual dependent variable w
(`SELF`, x)	w is the variable in object `A` named x
(`PARENT`, x)	w is the variable named x in `A`'s parent object
(`CHILD`, x, y)	w is the variable named y within a part of `A` named x
(`SIBLING`, x, y)	w is the variable named y within a part named x of `A`'s parent object

In the case of 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.

Model dependencies are automatically generated from *model parameter edges* using a process described in Section 4.2. A parameter edge specifies a path to each parameter used by a constraint. For example, the list of parameter edges for constraint C_2 in Figure 2 would be {(`PARENT, left`), (`PARENT, width`), (`SELF, width`)}. Every model dependency constraint must include a list of its parameter edges.

Although the current set of model dependencies allows a programmer to express a wide variety of relationships, it does not suffice to express any arbitrary relationship. Consequently, our implementation allows a programmer to also use explicit dependency constraints. When a variable's value is changed, the invalidation algorithm follows both the variable's model dependencies and its explicit dependencies. In this paper we will assume that a solver's mechanism for creating explicit dependencies does not have to be altered to accommodate model dependencies. This assumption holds in Amulet, the system in which we have implemented model dependencies.

4 Implementation

4.1 Division of Responsibility

The four edge relationships fall into two distinct groups. The `SELF` and `CHILD` relationships can be installed and resolved independently of any context that an owner provides, so these relationships are called *context-independent* relationships. In contrast, both `PARENT` and `SIBLING` model edges represent *context-dependent* relationships. They can be resolved correctly only if a parent object is present. In addition, 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 3. If part `B` is removed from the composite object and replaced with another part, the new part will assume the `SIBLING` relationship. Consequently, the `SIBLING` relationship is not intrinsic to part `B`, but rather to the composite object to which it belongs.

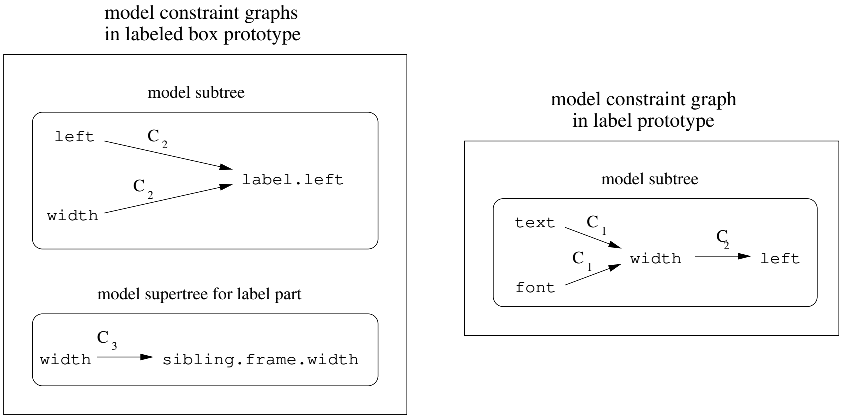


Figure 4: The labeled box's model constraint graphs. C_i refers to the corresponding constraint in Figure 3. Note that the model dependencies for a formula can be distributed across two different graphs, as is the case for label's C_2 constraint. C_2 has two references to its owner (`owner.left` and `owner.width`) and one reference to itself (`self.width`). The owner references get resolved into `CHILD` edges that are placed in the owner's model subtree. The self reference gets resolved into a `SELF` edge and is placed in the label's model subtree.

Due to the different nature of these two groups of edges, they are managed differently. The model constraint 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 its parts below. The prototype also holds individual model constraint graphs for each of its parts. These graphs store `PARENT` and `SIBLING` dependencies for variables in the parts. They are called *supertree graphs*, or just *model supertrees*, since they represent dataflow dataflow relationships to the parent object or to other children of the parent. Figure 4 shows the collection of model constraint graphs for the labeled box.

When an instance variable is modified, the object's prototype's model subtree is consulted to resolve `SELF` and `CHILD` model dependencies, and then the model supertree for that object, stored in the prototype of that object's owner, is consulted to determine if any `PARENT` and `SIBLING` dependencies exist.

In some circumstances it is necessary for model constraints to behave like explicit dependency constraints and generate explicit dependencies. A model constraint needs to establish explicit dependencies when it is explicitly added to an object by the application (in contrast to being inherited from the prototype). In this case the constraint is not represented in the prototype's model constraint graph and hence explicit dependencies must be generated.

4.2 Model Dependency Installation

A model constraint 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 prototype and its parts is examined and used to generate an appropriate set of model dependencies [1]. For each parameter edge, the model dependency is formed by simply reversing the parameter's path.

The installation process for each parameter edge type is summarized in Figure 5. For example, consider the labeled box in Figure 2. The constraint C_2 (which centers the label within its owner) is attached to the `left` variable of the `label` part. C_2 has the parameter edge list {(`PARENT, left`), (`PARENT, width`), (`SELF, width`)}. These parameter edges generate the following model dependencies:

Parameter edge	Model Dependency edge
`(PARENT, left)`	Add `(CHILD, label, left)` to the `left` vertex in the model subtree of `label`'s owner.
`(PARENT, width)`	Add `(CHILD, label, left)` to the `width` vertex in the model subtree of `label`'s owner.
`(SELF, width)`	Add `(SELF, left)` to the `width` vertex in `label`'s model subtree.

Figure 6 formalizes the algorithm used to create a model constraint graph. It is invoked the first time a prototype object is instanced. The polymorphic `param.install` method called by this algorithm creates an appropriate model dependency edge and stores it in the correct model constraint graph (either the subtree or supertree model graph). The install method is defined for each parameter edge type and implements the installation procedure outlined in Figure 5. The install method requires a reference to the object containing the constraint so that it can locate the appropriate model constraint graph. In the case of the `PARENT` and `SIBLING` parameter types, it also needs the object in order to identify which part name should be included in the model dependency's path.

4.3 Resolution of Literal Dependencies

Once the model dependencies are installed, an instance can use them to derive explicit dependencies. When a variable is changed, the resolution algorithm consults the model constraint graph vertex representing that variable and uses

[1] In this paper we are assuming that we can use a *reflection* [2] mechanism to iterate through the instance variables of an object and find the object's constraints. However, in a system that does not provide a reflection mechanism, the programmer can manually specify the set of constraints in an object, for example, by passing them as parameters to a function.

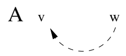
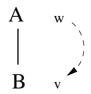
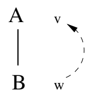
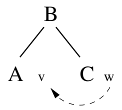
Variable to which formula is attached	Parameter edge	Installed vertex	Dependency edge	
v is in object A	(SELF, w)	Vertex w in A's model subtree	(SELF, v)	
v is in a part named B of object A	(PARENT, w)	Vertex w in A's model subtree	(CHILD, B, v)	
v is in object A	(CHILD, B, w)	Vertex w in B's model supertree maintained by A	(PARENT, v)	
v is in a part named A of object B	(SIBLING, C, w)	Vertex w in C's model supertree maintained by B	(SIBLING, A, v)	

Figure 5: Model dependency edge installation. Installed vertex is the vertex with which the model dependency edge is associated. Dependency edge is the model dependency edge generated from the parameter edge.

the stored model path to traverse the object's structure to find the dependent variable. The constraint that determines that variable's value can then be invalidated. If the dependent variable 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 objects are invalidated. Note that this strategy allows the model dependency scheme to work even when an individual instance alters its structure. Figure 7 formalizes the resolution algorithm.

To illustrate the resolution algorithm, let `box` be a labeled box and suppose its `width` attribute is invalidated (this could occur, for example, as a result of editing the label's text attribute). If any explicit dependencies of `box.label.width` exist, they will be invalidated in the usual manner as described in Section 2.2. Next, the context-independent model dependencies are located in the model subtree for `box.label` (this graph is found in the prototype object for `label`; recall Figure 4). The model edge entry `(SELF, left)` is found. The edge directs the algorithm to the `box.label.left` variable. The copy of constraint C_2 being used to determine `box.label.left`'s value is then invalidated. Finally, the context-dependent model dependencies are located by consulting the model supertree for `label`, which is found in `box`'s prototype object. There the edge `(SIBLING, frame, width)` is found. In a manner similar to the resolution of the `SELF` dependency, the edge's target dependent variable `box.frame.width` is located, and the appropriate constraint is invalidated.

5 Empirical Test Results

We implemented our model dependency scheme in Amulet [19], a C++ based toolkit used for research and development of graphical interfaces that includes a dataflow constraint system like the one discussed here. We additionally instrumented the Amulet support code so that the number of model and explicit dependencies could be counted. The criteria for evaluating the merit of model dependencies is the number of explicit dependencies saved. Amulet's object

`Constraint:`	
`invalid`	true, if the constraint has been invalidated; otherwise, false
`out_var`	the variable computed by this constraint
`modeled`	true, if the constraint is capable of using model dependencies; otherwise, false
`model_parameters`	the set of model parameters supplied to the formula upon which this constraint is based
`Object:`	
`parts`	the set of objects that are parts of this composite object
`variables`	the set of instance variables of this object
`Variable:`	
`object`	the object in which this variable resides
`invalid`	true, if the variable has been invalidated; otherwise, false
`explicit_dependencies`	the set of explicit dependencies associated with the variable
`constraint`	the constraint that can be used to compute the value of this variable

```
Method  Object.create_model_dependency_graph()
for each variable var ∈ self.variables :
    cn ← var.constraint
    if cn.modeled = true :
        for each parameter param ∈ cn.model_parameters :
            param.install(self)
for each part pt ∈ self.parts :
    pt.create_model_dependencies_graph()
```

Figure 6: Model constraint graph creation algorithm. `install` is a method that creates a dependency edge from a parameter edge given an object. All edge subclasses (`SELF`, `PARENT`, `CHILD`, and `SIBLING`) customize the `install` method to meet their needs.

```
Method  Constraint.invalidate()
if self.invalid = false :
    self.invalid ← true
    self.out_var.invalidate()
```

```
Method  Variable.invalidate()
if self.invalid = false :
    self.invalid ← true
    for each constraint cn ∈ self.explicit_dependencies :
        cn.invalidate()
    model_dependencies ← get_model_dependencies_from_child_graph(self)
                         ∪ get_model_dependencies_from_owner_graph(self)
    for each model dependency dep ∈ model_dependencies :
        cn ← dep.get_constraint(self.object)
        if cn :
            cn.invalidate()
```

Figure 7: Invalidation algorithm for a constraint solver that incorporates model dependencies. `get_model_dependencies_from_subtree` retrieves the `SELF` and `CHILD` model dependencies from the variable's prototype object. `get_model_dependencies_from_supertree` retrieves the `PARENT` and `SIBLING` model dependencies from the prototype of the variable's owner. `get_constraint` is a method associated with a model dependency that uses the resolution procedure described in Section 4.3 to return the appropriate constraint.

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 used provides a more accurate picture of how well model dependencies work. [2] As an added benefit the results are truly platform independent; a particular

[2]There was a tradeoff involved in doing our implementation in Amulet. Most constraint solvers are implemented in far more streamlined systems where our results would have led to significant reductions in heap space. Unfortunately, the source code for these systems is typically not publically

Checkers	The traditional board game
Circuit Designer	Used to build digital logic networks
Gilt	An interface builder that permits the construction of a graphical user interface by direct manipulation
Labeled Box	Objects are similar to the one described in Section 1
MathNet	Represents arithmetic expressions by a dataflow graph
Network Simulator	Simulates message passing in various network configurations
Self Chain	Each object uses 25 formulas to control 26 attributes establishing 325 dependencies
Testwidgets	Extensive test of menus, dialog boxes, scroll bars, etc.
Tree Editor	Graphically represents binary trees for program debugging
Tribox	Objects consist of three concentric rectangles governed by 12 constraints

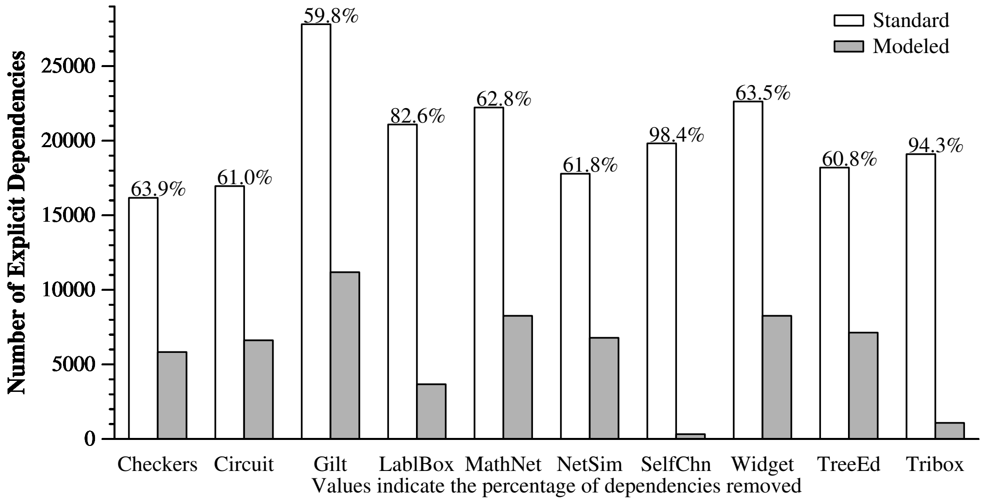


Figure 8: Standard vs. modeled results for a cross section of Amulet applications. Reduction of explicit dependencies by 60% or more was common, while some specialized benchmarks achieved a greater than 90% reduction in the number of physical dependencies.

operating system's memory management strategies will not flavor the results.

We tested our modification to the Amulet constraint system on a number of existing Amulet applications and on three specialized benchmark programs. The applications included the samples found in the Amulet distribution and several other contributed programs. No alterations are required to get existing applications to compile and run under modeled Amulet. However, we also modified, where possible, the formulas in these test programs to take advantage of modeled constraints. In all cases the modeled versions behaved identically to the original versions. Figure 8 summarizes the results of our experiments.

We have not yet tuned the performance of our model dependency implementation (the standard Amulet implementation is highly optimized). Nonetheless, our measurements show that the interactive performance of an application using model dependencies is only about 20% slower than one using explicit dependencies. It is notable that to the unaided eye, the versions using model dependencies do not exhibit any perceptible loss of interactive performance compared to their explicit dependency counterparts. This result was expected because previous studies have shown that redisplay time dominates the cost of all other operations in a graphical application [10, 26].

available nor is it meant to be extended by other researchers. Amulet by design meets both of these latter criteria.

6 Related Work

As mentioned in Section 1, the model dataflow graph concept was originally applied to attribute grammars by Reps, *et. al.* [20] in their work with syntax-directed programming editors. Model graphs are used to represent the dependencies among attributes in the vertices of an attributed, abstract syntax tree. 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 explicitly store any explicit dependencies at runtime. Our problem deals with more general constraint graphs where the relationships among the vertices can be arbitrary; any object can be dependent upon any other object.

One-way constraints have been applied to other application areas including circuits [1], graphical interfaces [10, 11, 13, 14, 18, 25, 19] and spreadsheets. Multi-way, multi-output dataflow constraint systems have also been developed for graphical interfaces, including ThingLab [3, 8, 23], Kaleidoscope [7] and MultiGarnet [22, 21, 24]. Multi-way constraints are more powerful than one-way constraints, but one-way constraints suffice for many applications and in general are more easily managed by programmers.

Researchers have adopted various approaches to storage optimization for dataflow constraints. *Constant propagation* allows some constraints to be removed from the system completely [16, 17]. 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 may be significant. Unfortunately, constant propagation has little effect in a dynamic system where few variables attain a fixed value.

Hudson and Smith greatly reduce the physical storage required for dependencies in certain common graphical layout relationships using a concept they call μconstraints [14]. Each constraint consists of only four bytes, 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 predefined set of potential parameter variables, such as `left` or `width`. Literal dependency edges in the constraint network are dynamically inferred as needed by consulting μconstraint encodings and the composite object's physical structure. Like our approach, a programmer may use standard heavyweight constraints for dependencies that cannot be represented in this encoding scheme. Our approach differs from μconstraints in that it does not store any dependency information in an object that can be derived from its prototype. Also, our scheme does not limit the representable relations to a small set of predetermined parameters or operations. Like μconstraints we currently limit the set of model dependencies, but we plan to add a `CUSTOM` relationship that allows a programmer to provide arbitrary code to extend the set of model dependency edges. It does not appear that μconstraints can be easily extended to handle arbitrary dependency edges.

Freeman-Benson eliminates entirely the storage required for constraint objects and their dependencies by compiling a given constraint network or subnetwork into a *plan* [6]. A plan consists of a single Smalltalk module and is created by analyzing the constraint network dataflow, "unwrapping" individual constraint methods, and sequencing the code that make up these methods into one large procedure that becomes a method in the module. The constraint plan approach is ideal for constraint networks with a static structure. In systems with pointer variables, dynamic edits, and arbitrary dependency edges, the required analysis may not be possible and dynamic recompilation of the modules will almost certainly be required even if the analysis can be performed.

7 Conclusions

Model dependencies can significantly reduce the number of explicit dependencies in a constraint graph. This reduction can in turn reduce the storage requirements of programs that manage a large number of constrained objects. As shown in Section 8, this savings does not come at a significant cost in 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.

References

[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. H. Classes versus prototypes in object-oriented languages. In *Proceedings of the ACM/IEEE Fall Joint Computer Conference* (Nov. 1986).

[5] 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.

[6] 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.

[7] FREEMAN-BENSON, B. N. Kaleidoscope: Mixing objects, constraints, and imperative programming. In *OOPSLA/ECOOP'90 Conference Proceedings* (1990), pp. 77–88.

[8] FREEMAN-BENSON, B. N., MALONEY, J., AND BORNING, A. An incremental constraint solver. *Communications of the ACM 33*, 1 (Jan. 1990).

[9] GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Massachusetts, 1995.

[10] 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).

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

[12] HUDSON, S. E. Incremental attribute evaluation: A flexible algorithm for lazy update. *ACM TOPLAS 13*, 3 (July 1991), 315–341.

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

[14] 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).

[15] 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.

[16] 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.

[17] 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.

[18] 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).

[19] 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).

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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).