A DYNAMIC AND FLEXIBLE PROTOTYPE-INSTANCE OBJECT AND CONSTRAINT SYSTEM IN C++

Richard McDaniel and Brad A. Myers

Human-Computer Interaction Institute
School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213
{richm, bam}@cs.cmu.edu
http://www.cs.cmu.edu/Web/Groups/amulet/amulet-home.html

ABSTRACT

In order to support rapid prototyping and efficient construction of user interface software, the Amulet user interface development environment uses a prototype-instance object model integrated with a constraint solver. The important innovations in the Amulet object and constraint systems are the automatic management of a part-owner hierarchy in addition to the prototype instance hierarchy, the support for multiple constraint solvers at the same time, control over slot inheritance, flexible demons, and a convenient integration of the models with C++ without requiring a pre-processor.

KEYWORDS: Object-Oriented Programming, User Interface Development Environments, Constraints, Toolkits, Amulet.

1. Introduction

Amulet is a new user interface development environment in C++ which supports the development of user interface software. As part of Amulet, we have developed a dynamic and flexible *prototype-instance object system* embedded in C++. In a prototype-instance object system, there is no distinction between classes and instances: every object can serve as a prototype for other objects, and any values (called ‘‘slots’’ in Amulet) that are not declared locally in an object are inherited from the prototype. Another important feature of Amulet's object system is that there is no distinction between methods and data: any object can override an inherited method as easily as inherited data. The object system is *dynamic* in that slots in objects can be created, set, and destroyed at run time, and the types of values in slots can also change (for example, the value in a slot can change from being a string to being a float).

The Amulet object system also supports a part-owner hierarchy, by which objects can be grouped together. A common use for part-owner in Amulet is to group graphical objects into a collection or aggregate. For instance, the graphics in a window are added as *parts* of the window.

Amulet also provides *constraints* which are relations that are declared once and then maintained by the system. Constraints in Amulet can be arbitrary C++ code. Any slot of any object can contain a constraint, rather than a regular value. In addition to writing constraints, programmers can write their own constraint *solvers* and use them concurrently with the constraint solvers that Amulet provides by default. This can be a potential boon to constraint researchers who have heretofore been expected to write their own constraint systems from scratch or modify the source code of existing constraint solvers.

The Amulet object and constraint models are based on the designs of the Garnet system [10] which has proven useful for quickly prototyping, designing, and implementing user interface software. For Amulet, we took the opportunity to fix a number of problems we experienced with Garnet, and Amulet also contains a number of important innovations, including the automatic management of a part-owner hierarchy along with the prototype instance hierarchy, control over the inheritance of slots, the support for multiple constraint solvers, and a flexible demon mechanism. In addition, Amulet's default constraint solver is more flexible than other one-way systems. Finally, it is interesting how we were able to provide dynamic slot typing, a dynamic prototype-instance system, and constraints in C++ without using a pre-processor or a scripting language.

2. Amulet

The Amulet user interface development environment aims to make the design, prototyping, implementation, and evaluation of user interfaces significantly easier, while supporting flexible experimentation with new styles of interaction. Amulet will include a number of design and implementation innovations including new constraint and object models (described in this paper), new input and output models, and new forms of interactive tools. In addition, it will support the creation of innovative user interfaces that incorporate features such as speech and gesture recognition, 3-D, animations, visualizations, world-wide web (WWW) access and editing, and multiple people operating at the same time. Amulet, which stands for Automatic Manufacture of Usable and Learnable Editors and Toolkits, is currently working (see Figure 1) and has been released for alpha testers.[1] The current version contains the object and constraint system, a retained-object model for output graphics so the system automatically handles refresh, an input model based on Garnet's ''interactors'' where high-level behavior objects can be attached to graphics to handle input, built-in objects that provide UNDO, and a full set of widgets. Amulet works as a ''virtual toolkit'' since applications written using Amulet will work without change on either Unix X/11 or Microsoft Windows.

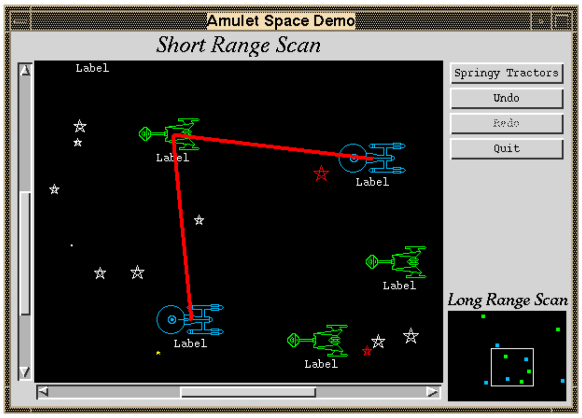

Figure 1:
A sample program created using Amulet. The ships and widgets are instances created from prototypes, and constraints are used ubiquitously to keep the lines connected to the ships, compute the widget sizes, keep the window positioned appropriately based on the scroll bars, etc.

As a research system, we have three main goals for Amulet. First, Amulet should be very flexible and effective for user interface researchers, so parts can be replaced and new technologies and widgets can be easily created and evaluated. Thus, we have made it easy to investigate new constraint solvers and to build new kinds of widgets and new kinds of interactive tools. The second goal is to be useful for students, which means that Amulet must be easy to learn. Finally, the system should provide sufficient performance, robustness and documentation so it will be useful for general user interface developers.

3. Why Build an Object System in C++?

C++ is already object-oriented; so why write an object system in it? If C++ objects could be used to provide the ease of programming that we desire for Amulet, we would have used them. Regrettably, C++ objects do not perform all the duties that we need; so we created the Amulet object system to solve the problems.

Qualities that are desirable in a good graphical object system include dynamic creation of objects and object types, dynamic typing, and constraints. Furthermore, the environment should be able to query objects about their structure, types, and properties at runtime. With these dynamic properties, one can easily accommodate features such as scripting languages, interactive ''inspectors'' for debugging objects, and dynamic editors (like Interface Builders) where one can edit a program while it runs. Even when using a compiler, building one's programs using constraints and other declarative features can greatly ease the programming task.

Looking at C++, we see that objects are meant to have static classes that are defined at compile time. The only means for querying the contents of a class are to read the documentation or to actually read the code. In terms of extending a class definition, it is quite impossible to add new variables or methods to a C++ class without rewriting the code and recompiling. Though one could argue that modifying the code and recompiling can be done automatically, this still does not support querying, and a better solution is to implement what one needs in the first place.

Another important motivation is to provide a constraint system integrated with the object system. Although it is possible to add constraints directly to C++ objects, most systems that do this have required the use of pre-processors or special-purpose constraint languages. In Amulet, arbitrary C++ code can be used in constraints, and no pre-processor is required.

4. The Basics - Objects, Slots, and Inheritance

Amulet defines an object as a collection of dynamically typed *slots*. Each slot in an object has a name called its *key*. A given key refers to a unique slot in a given object but can be reused in other objects. This makes objects into a name space for slot keys; each object represents a dictionary of the slots it contains. In Amulet, the keys are implemented as integers, and there is a hash table for mapping between strings and the integer keys and for generating unique keys for new slot names. However, this hash-table is only needed for debugging and when programs define new slot names (typically at load time), and not for slot access at run time, since programs use the integer keys.

The Amulet object system uses the prototype-instance model. To make a new object, one takes an existing object and makes a new instance of it. The behavior of instancing is very much like copying, except that when the programmer modifies a slot of the prototype object, all of the instances of that object which do not have a local value for that slot will reflect the same change.

Method slots are treated exactly the same as data slots--any object can inherit or override a method. In C++ terms, a method slot simply contains a pointer to a function. In a

[1]Amulet is in the public domain and is available by anonymous FTP or World-Wide-Web. To get it, FTP to `ftp.cs.cmu.edu` and retrieve /usr0/anon/project/amulet/amulet/README, or go to the WWW page listed at the top of this article.

conventional class-instance model, instances can have different data, but only sub-classes can have different methods. Thus, in cases where each instance *needs* a unique method in conventional system, a different mechanism from the regular method invocation must be used. For example, a widget might use a regular C++ method for drawing, but would have to use a different mechanism for a call-back procedure, since each instance of the widget needs a different call-back. Another advantage of treating methods the same as data in Amulet is that the actual method to use could even be computed by a constraint which returns a pointer to the appropriate function.

Having no distinction in Amulet between classes and instances, or between methods and data, means that there are fewer concepts for the programmer to learn and a consistent mechanism can be used everywhere.

Amulet's model for objects and slots is borrowed strongly from Garnet. Amulet departs from Garnet, though, in that the inheritance style of Amulet slots is not fixed. Garnet supports only one kind of inheritance. We found in Garnet that the use of slots is normally task-related and that different tasks require different kinds of inheritance. In Amulet, the object system allows each slot to be defined with one of four kinds of inheritance rules.

The default style in Amulet is *inherited*, which means that the slot value of the prototype is used unless overridden, and changes to the prototype are seen by instances that do not override the slot. Another inheritance style is *copy* which means that the instance's value is not affected by later modifications to the prototype. In order to save memory, a copied slot is not actually copied until the prototype actually does get changed. The copy style is used whenever the values of objects must be kept separate from the prototype. The third kind of inheritance style is *local* which means that the value is never available to the instances, and each instance must define their own value. The local style is useful for slots that hold some state that is particular to the object. For instance, the window object has a slot that holds the low-level X/11 or MS Windows window which is used for doing the actual drawing. Since two Amulet windows should not share the same window-manager window, this slot is defined as local. The final inheritance style is *shared*, which is similar to a ''static variable'' in C. A shared slot can never be made local in an instance. Setting the value in one instance produces the same change in all instances. Shared slots are useful for global data that all the instances need to gain access.

Since the inheritance style is associated with slots, a single object could have some slots that are inherited and represent the data that programmers are meant to modify, while other slots are local to hold the object's private data.

Garnet supported multiple inheritance, but we found it was not useful or necessary. In Amulet, we instead use the constraint mechanism to copy values around, which provides complete flexibility and control. Omitting multiple-inheritance has simplified much of Amulet's implementation leading to an easier to understand object creation procedure and better efficiency for searching for slots. It also eliminates the ambiguity and complexity for the programmer of what to do when slot names collide from multiple prototypes (super-classes).

5. Part-Owner Hierarchy

The part-owner relationship is designed to support the typical graphical grouping which programmers use to put objects into windows and assign groups of objects to move together. For example, Figure 2 shows how a graphical object is part of both a prototype-instance hierarchy and a part-owner hierarchy. The part-owner hierarchy is a simplification and generalization of Garnet's aggregate, aggregadget, aggrelist and aggregraph objects. While Garnet defined the part mechanism as part of the *graphics* system, Amulet builds the part mechanism directly into the object system as a primitive capability, related to the approach in Katie [9]. The result is that the implementation is much cleaner and simpler, and we have found many other uses for it besides graphical containment, such as attaching behaviors and commands to objects.

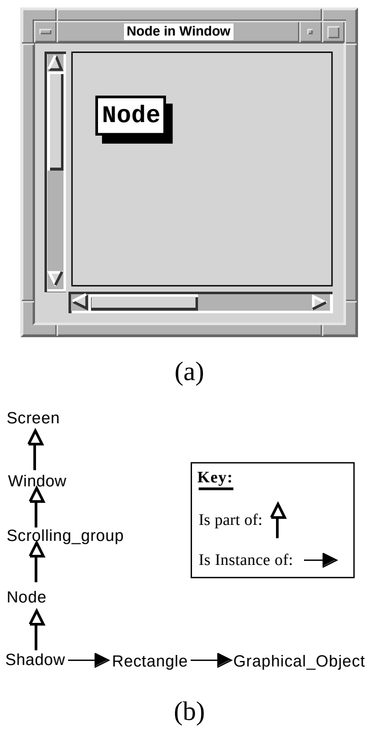

Figure 2:
The rectangle object which serves as the shadow of the Node is both in a part-owner hierarchy up to the screen, and a prototype-instance hierarchy up to Graphical_Object.

The part-owner relationship goes beyond simple grouping in two ways. First, if an owner has parts, and a copy or instance is made of the owner, new parts are automatically created for the new owner. Thus, if the programmer makes an instance or copy of the Node in Figure 2-a, instances will also be made of all the rectangle and string parts of the node. Secondly, objects use the part-owner hierarchy to name objects so the references will be correct even if copies are made.

Like inheritance, we restrict part-owner relationships to be trees. That is to say that an object is restricted to have only one owner. This restriction makes objects more efficient and easier to understand as the programmer assembles them.

In Amulet, we define two kinds of part-owner connections. The first is a bi-directional connection. A part is assigned to an owner and it is given a name in the form of a slot key. An owner can refer to its parts by using the key. Each named part should have a unique key. This is not a problem because there are usually very few parts that have a special purpose. For example, the Node in Figure 2-a may have parts called `SHADOW`, `BACKGROUND` and `LABEL`. The parts can refer to their owner by using a pre-defined slot key called `Am_OWNER`.

The other part-owner connection omits the slot key for the part. Unnamed parts are used mostly for grouping objects together, such as to hold dynamically created objects in a graphical editor. It is quite reasonable to mix named parts with unnamed parts in a single owner. Consider a scrolling window gadget which contains two scrollbars as named parts, and a collection of unnamed objects which are drawn in scrolling area. The distinction between unnamed and named parts was quite fuzzy in Garnet, and is much easier to understand and simpler in Amulet.

The semantics of adding parts to objects can be decided by either the owner or the part and is based on the types of the each. Take, for instance, when a graphical object is added to a window, a special ‘‘demon method’’ (described below) of the window is invoked which tests the type of the object. If it is a graphical object, it is added to the window’s display list so that when the window gets drawn it will recursively call the draw method on the graphical part. However, when interactor objects are added to a window, they cause different internal data structures to be set up.

Some systems, such as FormsVBT [4] have hard-wired some slots to inherit from their prototypes and others to inherit from their owners. Because the constraint mechanism is so easy to use and flexible in Amulet, we felt that it was sufficient to use constraints whenever slots should get their values from their owners rather than from their prototypes. However, we might in the future provide a means to directly link slots between owners and parts. Either the part could inherit slots from its owner or vice versa. This will likely help with both size and speed efficiency since the extra memory associated with constraints and the constraint’s evaluation of the code will not be needed.

When an object is instanced, the named parts of that object will be instanced as well. The result is that the programmer can simply make instances or copies of objects without knowing whether the object is a primitive or a complex structured group, and the system will automatically make sure that all parts are reproduced appropriately.

Amulet also provides a special form of group called an `Am_Map` that computes the parts dynamically. The `Am_Map` object uses a fancy constraint that builds a list of parts dynamically based on the contents of slots that control the activity. Typically, there will be a prototype object stored in the `Am_ITEM_PROTOTYPE` slot, and a list of strings, objects, commands, or whatever in the `Am_ITEMS` slot, and the `Am_Map` creates an instance of the item prototype for each value in the `Am_ITEMS` list. For example, programmers can build a button panel by providing a single button prototype and using the map object to replicate the button where it is needed.

Since the keys used to name parts are the same keys used to name slots, parts and slots have some commonality of function. One can fetch the value of a part identically as one fetches the value of a slot so programmers (and constraints) do not necessarily need to know whether an object is stored as a part or a slot. The main difference is that if the object is stored as a named part, instancing the owner object creates an instance of the part. If the object is stored as a slot, an instance of the owner will have a slot that points to the original object (so the same object will be shared by the prototype and the instance). For example, for an `Am_Map`, sometimes the programmer wants to be able to modify a slot in the prototype and have all the maps that use that prototype see the same change. For this case, `Am_ITEM_PROTOTYPE` can be made a regular slot so that all the instance maps will share the same object. However, sometimes the programmer will want to make modifications in instances of the map that only change the item prototype object locally. For this case, the programmer assigns `Am_ITEM_PROTOTYPE` to be a named part. Then, the item prototype object will be instanced for each instance of the map and the programmer can modify each instance individually.

6. An Extensible Constraint Solver

The Amulet object system uses constraints ubiquitously for defining relationships. The standard constraint system is a conventional, one-way propagation solver, where a slot can contain a *formula* which contains a method (in arbitrary C++ code) to calculate the value. Like Garnet, the dependencies of formulas are dynamically determined through the use of a special slot accessor that in addition to returning the slot’s value, also sets up a dependency. Also like Garnet, cycles are handled by returning the old, cached value.

An innovation in this solver is that a slot can contain multiple formulas, and the most recent one to become invalidated is used to recalculate the slot value. We find this very useful for situations where an inherited formula is necessary for the correct operation of an object, but the programmer wants an additional formula so values can flow in multiple directions. For example, a scroll bar widget might have a constraint in the exported `Am_VALUE` slot that updates the value when the user moves the indicator, but the programmer might install a constraint in the same slot to update the indicator if the user scrolls the window another way (such as using a keyboard accelerator).

The slot accessor for constraints is a form of Get that takes an extra parameter: the constraint context. The constraint context is passed to the constraint method and represents the state of the constraint's evaluation. The `GV` macro expands to `Get` with the extra constraint context parameter, `cc`. A constraint context is defined as a C++ virtual class, and can do whatever the constraint solver needs. The formula constraint solver stores a pointer to the formula which is used to set up a dependency between the formula and the slot being retrieved. The `cc` for formulas also holds the current state of the ‘‘depends-on’’ list which is used to prevent stale dependencies. Having a `cc` parameter allows Amulet code for constraint solving to be completely reentrant because the `cc` holds the variables that we had to make global in Garnet. To use `GV` in other C++ procedures, one can simply pass the `cc` as a parameter.

A major way Amulet differs from Garnet is that Amulet formulas are re-evaluated whenever the programmer accesses the value of any slot. This means that *all* slots are always up to date whenever any slot is accessed. We made Amulet ‘‘eager’’ because our experience with Garnet, which used lazy evaluation, showed that there were very few constraints that were correctly never evaluated, and Garnet users were often annoyed that constraints were not evaluated if they were not accessed. Usually, this was because they wanted to put side-effects into the formulas. Therefore, in Amulet, we also allow arbitrary side-effects in the formula procedure, including setting other slots and creating and destroying objects. Formulas can have the effect of multiple-outputs by simply setting some slots, and returning a value for its ‘‘main’’ slot. It is even possible to have a side effect of creating or destroying objects in a constraint. One use for this is the `Am_Map` constraint discussed above. Another example is if the programmer wants multiple views, one can use constraints to create and delete the objects in the views based on the main ‘‘model’’ objects.

Side effects are permissible in Amulet constraints because we store any new constraints that need to be evaluated in the demon queue (discussed in a later section). Likewise, when slots and objects are destroyed, any demon that has been queued for them will be removed. Unlike Rendezvous [7], we have no need to delay side effects because their effects on the constraint solver is delayed by the demon queue. New constraints get stored at the end of the queue to be seen later and demons for deleted constraints get removed. In terms of dependencies, the order that constraints are stored in the demon queue is not crucial because the formula constraint solver uses a lazy evaluation strategy to cause dependent constraints to be evaluated when they are called by the formula. The slot caches the result of the formula so it will not be re-evaluated unless necessary.

For side-effects, the order of constraint evaluation could be made important. Amulet only guarantees that slots will be valid and side-effects complete when slots are accessed from outside of constraints. Inside constraints, no guarantees are made because we cannot predict which constraints will produce side-effects or where the effects will be. Formulas are permitted to rely on side-effects with the caveat that one cannot make divergent cycles. For instance, a map object may have a width formula that depends on the objects created by another formula. It is possible through side-effects to set up a situation where a loop of formulas will constantly invalidate each other. Presently, formula design with side-effects must be handled carefully in Amulet because we cannot detect these kinds of infinite loops. In practical code, though, these situations do not arise and would be easy to code differently if they did.

Amulet formulas use a new technique for keeping dependencies from going stale. Most constraint solvers do not have this problem because they do not even support ‘‘pointer variables’’ [14] in constraints. Some of those that do support these do not clean up unused dependencies when a formula changes from using one set of slots to another. Consider the following formula:

```
Am_Define_Formula (int, width_of_parts)
{
  Am_Value_List parts =
      self.GV (Am_GRAPHICAL_PARTS);
  int width = 0;
  for (parts.Start (); !parts.Last ();
       parts.Next ())
    //get the current part, type-cast to an object,
    //get that object's width and type-case that
    //to an integer, and add to the current width
    width += (int)Am_Object(
                parts.Get()).GV(Am_WIDTH);
  return width;
}
```

The formula reads a list of objects and builds a dependency on each. When the list of objects changes (a very common occurrence), the dependencies of the formula change. We want to make sure that stale dependencies are removed from the formula. Since the order in the list of dependencies rarely changes, we can use that order to our advantage. Amulet saves the list of dependencies for a formula in the order that they were generated. When the formula is re-evaluated, the dependencies are followed in the list. If a dependency stays the same (has the same object and slot key), then the slot can be retrieved directly from the list without searching the object. This can save a good bit of time from formula calculation. If the dependency changes, the old dependency is removed and the list replaces the list item with the new dependency. This algorithm has the merits that it never requires any lists to be searched, but simply traverses a single list as the formula is evaluated. Since, the dependencies are always kept up to date, efficiency is further improved because formulas never get false invalidations from the unused slots. Another benefit is that our debugging tools can show the programmer the correct dependency information. One drawback is that a dependency that is requested multiple times is stored multiple times. This problem occurs rarely in practice, however, because programmers will use temporary variables to hold values of dependencies that are used multiple times.

Pointer variables can be troublesome when the object stored as a pointer is not initialized. When a programmer references a part that does not exist, the system will return a ‘‘null’’ object. Trying to fetch a slot from the null object would normally cause a runtime error. In a formula, though, the programmer often wants write dependencies as a long chain of pointer referencing that terminates with the final GV. For example, `GV_Owner().GV(OBJ_OVER) .GV(LEFT)`. Cluttering the formula with checks to see if parts are initialized is annoying and error-prone. Often, all ‘‘real’’ uses of the formula will have valid values, and the only time it will be invalid is when the formula is defined in a prototype object which may never be used except for creating other objects.

Normally, we would use exception handling to remedy this problem, as was done in Rendezvous [7]. When an error is detected in a formula, the system would cause a throw to force the call state out of the function code. The C++ compilers used by our group, however, do not implement exception handling. This forced us to work around the problem in another way. In Amulet, we intend to use the constraint context to allow programmers to write formulas with broken links. When `GV` is called on a null object, the `cc` is marked as ‘‘uninitialized’’ and a special uninitialized slot is returned to the function. The uninitialized slot will return zero as its value for any type requested so the formula is unlikely to fail due to receiving an unusable type. When the formula completes, the system will put the special uninitialized type into the slot and ignore the value returned from the function. Likewise, when a formula retrieves an uninitialized slot value from an object, it too will be marked as uninitialized, so there will be no runtime error. However, as soon as the programmer tries to use an uninitialized slot outside of a formula, a runtime error does occur, since this usually signifies a programmer bug. The error message will describe precisely which pointer is broken so that programmers will be more able to correct the bug quickly.

The most important innovation in the Amulet constraint system is that the programmer can use multiple constraint *solvers* at the same time, even in the same slot. Many researchers have investigated new kinds of solvers, like multi-way [12] or incremental [5] and each of these has useful applications. Therefore, we wanted to allow a programmer to use the appropriate solver for different aspects of a design. Also, researchers want to investigate new solver technologies without having to build the other parts of a system. Amulet is the first system with an architecture that supports the integration of multiple solvers.

This is provided through a meta-level that deals with slots and the kinds of messages that slots can produce and receive. Every slot can contain two lists of constraints: the set of constraints that depend on the value of the slot, and the set of constraints on which the slot depends. Note that a single slot can depend on multiple constraints. The constraint lists are not maintained in any particular order. A single constraint can be stored in multiple slots and even multiply in a single slot. A constraint can even be stored in both the depended upon list and the dependency list. For example, an incremental constraint solver like Bramble [5] might want to use a single constraint object for a whole collection of slots.

A slot has three basic messages that can be sent to a constraint. One is for when the value of the slot changes. The second message is ‘‘invalidated’’ which is used by constraint solvers to mark slots that need to be recalculated. For example, the invalidation message is sent by formula constraints to all the dependent slots. The slot will respond by invalidating its dependent constraints which recursively repeats the invalidation process for any dependent formula constraints and so on. Other constraint solvers may choose to withhold propagating an invalidation signal until it knows whether the value of the slot is going to change or not.

The third message is used when a value is requested from an invalid slot. The slot sends the ‘‘get’’ message to the constraint, and the constraint is expected to generate a response. A constraint always has the option of refusing to return a value, in which case the slot sends the get message to a different constraint. If no constraints return a value, then the slot will quietly keep its original value and consider itself valid.

The main item of contention in this scheme is what order will constraints be sent the get message? The policy implemented for Amulet is straight-forward. The last constraint invalidated is the first constraint to be queried. Since practical constraints used for graphics do not usually compete, this policy has proven adequate. As we develop more constraint solvers, we will continue to investigate this issue.

Permitting a constraint to disregard a get message prevents a certain kind of circularity. For instance, in a multi-output constraint, it is possible for a get message to be called on its other output slots before it finishes computing the response for the get message on the slot which was originally invalidated. Assuming the constraint does not want to recurse or use a co-routine calling style, the constraint has the ability to disregard the get messages on its other slots, finish computing the value for the first message and later re-invalidate the slots that it missed earlier, this time being able to produce the desired response.

The other messages that an object sends to constraints are for bookkeeping purposes. The slot can tell the constraint that it has been added or removed. Likewise, it can tell the constraint that the slot has been instanced or copied so that a new constraint will be needed for the new slot.

As a first order test of Amulet’s multiple constraint solver’s ability, we implemented a second constraint solver called a ‘‘web.’’ The web constraint is a multi-output, multi-way solver that performs planning on demand. Web constraints are built somewhat like formulas except that instead of returning a result through the return value, the web calls the setting equivalent of GV called SV. SV is Set with a constraint context parameter just like GV is Get with a context. The web can generate dependencies using GV and set up output slots using SV. The same slot can be referenced both by SV and GV in the same web which allows for multi-way constraints. The number of either kind of dependency is not fixed and constraints can change dependencies freely to different slots as the needs of the constraint changes. This is in direct contrast to the formula constraint which is directly tied to the value of a specific slot.

The web constraint also has some other advanced features over formulas. A web will keep track of which dependencies are invalid and will verify that the slot's value has changed before re-evaluating. The web also tracks the order that dependencies change. When a dependencies invalidate, they are stored in a list in the order that the invalidations occur. This list is accessible in the constraint's method. Being able to tell the history of slot modification is useful because it allows the constraint to apply consistent semantics across multiple slots. This is why we use web constraints to implement the slots of the line and polygon objects. The line object has two sets of input slots. One set is point based and has slots called `X1`, `Y1`, `X2`, and `Y2` which are used when the user creates a new line by rubber-banding. The other set is rectangle-based and has slots called `LEFT`, `TOP`, `WIDTH`, and `HEIGHT` which are set when the line is moved without changing its orientation. However, if the the slots `X1`, `TOP`, and `WIDTH` were set in some random order, our normal one-way formula mechanism would not allow us to get the semantics correct. With the web constraint history mechanism we can handle this case with relative ease.

The reason we can use web and formula constraints together in the same system is that neither webs nor formulas try to exert full control over the value of a slot. If two constraints on the same slot both try to maintain a slot at a different value, then the system could cycle and never know when to stop. Just having a cycle is permissible in Amulet because the constraint solver will only use the value of one of the constraints to set the value of a slot. It is when the constraint watches the slot value and forces it to become a certain value (by sending the invalidate message and responding to the get message) that trouble can happen. A ‘‘good neighbor’’ policy is currently required for Amulet constraints though we are experimenting with ideas that can relax this requirement.

There is an interesting relationship between constraints and inheritance. Since the constraints in Amulet can contain variables [14], the same constraint expression may evaluate to a different value in different objects. Therefore, when a constraint is inherited, we must copy the constraint expression and re-evaluate it in the context of the instance. Often the inherited constraints are required for the correct operation of the object. For example, the value slot of most widgets contain formulas that compute the appropriate output based on the user's actions. Other times, constraints act as defaults that can be overridden. For example, the default width of a button might be a constraint depending on the label. If the programmers explicitly sets the width of the button, then the constraint should be removed. In Garnet, this was often troublesome, so Amulet lets the programmer declare whether a constraint should be replaced when a slot is set.

7. Connecting Objects to the Rest of the Environment - Demons

Although constraints can be used for many tasks, Amulet also provides a flexible ‘‘demon’’ mechanism whereby methods can be associated with specific operations on objects. For example, the window's add-part demon was mentioned earlier. Commonly, the demons perform low-level, internal maintenance tasks, but they are also available for use by advanced Amulet programmers. Although Garnet had a demon mechanism internally, it was quite inflexible and not available to programmers.

Demons can either be associated with an object or a slot. Object demons are available for creation, destruction and adding of parts to objects. We did not provide a demon for every kind of operation that can be done to an object or slot for efficiency. For instance, we do not provide a demon when a slot's value is fetched. Slot access is the most commonly used operation in the system, and adding a single bit check to this operation can slow the whole system down by 10% or more depending on how it is done. Having a user's demon called on every slot fetch could slow the system down by an order of magnitude.

Slot demons are used to control how an object responds to data manipulation. Slot demons can be configured to respond to one of two messages: invalidation or value-changed. An invalidation demon can be used in lieu of a constraint because invalidation is the same message that is used to trigger the constraint solver. It is more common to have a demon respond to the value-changed method, however. For example, value-changed demons are used in Amulet to handle redrawing of the changed graphical objects. A demon is attached to each slot that affects the object's appearance. This demon queues the object for later redrawing.

Amulet does not cause a demon to be invoked immediately. An Amulet application usually consists of bursts of activity where a great many objects are suddenly created and slots are set. In order to prevent demons from being called more often than needed, demons are put onto a queue to be invoked when the queue is read. If a slot changes multiple times, the demon is only enqueued once. Furthermore, a demon can be marked so it is enqueued only once per object, even if multiple slots in that object change. For example, the redrawing demon is only enqueued once for an object even if both the left and top change. Invoking the demon queue occurs in Amulet during the event loop processing and when slot values are queried. We are searching for techniques to even further reduce the number of demons that get called and the number of times the queue is invoked.

Amulet uses the slot demon mechanism to make slots eager. A system-defined validation demon is automatically assigned to every object. The validation demon gets enqueued on the invalidation message, and it works on a per slot basis. The validation demon is a simple, one-line procedure that calls the Validate method on its slot. Since the demon queue is invoked on every Get call, the system acts as though it were eager. To turn off eagerness in a given slot, one can simply turn off the demon.

8. Implementation Issues

8.1 Making an Object System Programmable in C++

One of the major hurdles in designing an object system is making the code easy to read and write. Most object systems have created a whole new language. In Amulet, we have instead extended an existing, popular language, C++, so that Amulet will be easier to adopt and so it will integrate well with other code. Fortunately, C++ provides features that we were able to use to simplify the syntax of Amulet and allow the programmer to use normal C operations for primitive types like integers and strings. To do this, we make strong use of the C++ features of overloading and casting operators. For setting a slot, we have a single C++ method `Set` which has different, overloaded choices for the various built-in types. Thus, the programmer can simply write:

```
obj.Set(Am_LEFT, 33);
obj.Set(Am_TEXT, "my string");
obj.Set(Am_OTHER_OBJ, obj2);
obj.Set(Am_DRAW, my_draw_function);
```

C++ does not allow functions to be discriminated based on the return type, but we were able to get around this by returning a special slot object, which then has type conversion routines into the various primitive types, so you can write code like:

```
int i = obj.Get(Am_LEFT);
Am_String s = obj.Get(Am_TEXT);
Am_Object obj2 = obj.Get(Am_OTHER_OBJ);
Am_Value v;
obj.Get(slot, v);
```

In the last line we use the special `Am_Value` type to retrieve the value of a slot. The `Am_Value` type is essentially a union type that permits programmers to dynamically access and set the type and value.

We also permit the same kind of dynamic typing in the constraint system. The common situation is for the programmer to know what type a formula will always return. For example, a formula stored in the left slot of a graphical object will always return an integer because the slot cannot accept anything else. But for some slots, the type returned can vary from time to time. To allow this, we use the special `Am_Value` type as a return parameter as one choice for building formulas.

One of our chief concerns was the amount of excess syntax needed in C++ expressions. The main syntax burden comes when one declares a new object because at that point, the programmer changes the value of a great many slots all at once. In Common Lisp, we were able to define complicated macros which could interpret a potentially very long, variable length parameter list. In C++, we could have used a variable length parameter list as well but the C++ varArgs mechanism does not provide any type-checking. We decided to use simple methods that use the return value to pass back the object being set. This makes it possible to chain a long list of slot setting and part adding calls into what amounts to a single line of C++ code:

```
//Define a formula called my_int_formula that
// returns an int
Am_Define_Formula(int, my_int_formula) {
  return self.GV_Part(OTHER_OBJ)
             .GV(Am_WIDTH) - 10;
}

Am_Object obj = Am_Rectangle.Create()
 .Set(Am_LEFT, 33) //note no semi-colon
 .Set(Am_TEXT, "my string")
 .Add_Part(OTHER_OBJ, MyProto.Create())
 .Set(Am_WIDTH,
   Am_Formula::Create(my_int_formula));
```

One of the supported types is `void*` which allows a pointer to anything to be stored into a slot, but we wanted to have a more type-safe way to store application-specific data structures. Therefore, we provide a C++ class called `Am_Wrapper` type to ‘‘wrap’’ arbitrary C++ objects. In addition to type-checking, wrappers also help with memory management. Amulet uses wrappers internally to implement graphical styles, fonts, point-lists, etc.

Fundamentally, every wrapper contains a reference counter. When a slot is copied, due to inheritance or other means, the value in the slot is sent a note-reference message to indicate that it is being referenced in some other place. Likewise there is the release message that will be sent when the slot is destroyed. To handle reference counting outside the context of a slot, wrapper objects are given special constructors, destructors, and assignment operators. When the programmer modifies the value of the wrapper object, the object will automatically create a unique copy of itself by calling the make-unique message. In this way, modifying the data of the wrapper will not erroneously change the value that is still stored in an object’s slot or in other slots that share the wrapper. For advanced programmers, there are also techniques for destructively modifying wrapper data directly within slots.

Making a new wrapper type is relatively easy because Amulet defines two macros that the programmer can use. The programmer essentially writes his object as a normal C++ class, and adds the Amulet macros at the appropriate places to make it into a wrapper. The macros add a dynamic type id-tag to the class so that it can be distinguished from other wrapper classes, and define standard methods that provide convenient services and make wrapper classes consistent.

8.2 Efficiency

Since Amulet has just been released, we would like to be able to better understand the use patterns of Amulet before we spend much effort on improving the efficiency. However, having created the earlier Garnet system with a similar design, we have some notion of where the important concerns are. Therefore, Amulet is reasonably efficient, but not as good as its going to be after a couple more iterations. The current version runs acceptably on 486 PCs and Sun SPARCStation 2s.

8.2.1 Size

A serious problem with Garnet is the size of objects, so in Amulet we have taken pains to keep the size small. The largest contribution to memory size in Amulet is the slots. Slots hold the data of objects so naturally the brunt of the memory cost goes into the structures that hold the memory of the application.

When an object inherits slots from its prototype, Amulet tries to continue using the prototype's slots and not allocate memory for the instance. (Garnet always copied all slots down to optimize speed of access.) Only when the instance has a different slot value do we want to allocate local memory. This practice has resulted in a significant savings in memory size. A typical Amulet object has about 30 slots defined in its prototype. Most of these slots are for default values like methods and the visible slot. On average, about 10 slots are made local. Around four of the local slots for graphical objects contain Amulet's state structures holding information like the bounding box and the window in which the object is contained. Another two or three of the slots are made local by constraints. The other local slots are parameters that the programmer sets directly. The largest objects are the widgets with around 40 to 50 potential slots, but only about 20 are ever made local, mostly due to slots that contain constraints that determine the widget's layout and color. Had we copied down all slots from the prototypes, the number of slots allocated would easily double or triple.

In principle, the size of a slot should be no larger than the data it contains, which is usually one word. However, other data is needed for each slot, including the slot key, the slot type, constraints and demons in the slot, and dependency pointers for constraints that use this slot. Accounting for just the slot key, the value, and some state bits, the realistic minimum size of an Amulet slot is three words. When the slot contains dependency pointers and constraints, the size needed increases to about seven words. Judging from some of our early applications, we see that about half of all local slots do contain constraints. This works out to about 10 to 20 slots with constraints for widgets and only two or three slots with constraints for primitive graphical objects. In the future, we will make two different sizes for slots, which would further reduce memory use by about one quarter.

Another idea worth exploring is using ''virtual objects'' [3]. Virtual objects replace lists of objects that share many common features with a single prototype object that acts like many objects at once. This is related to glyphs [1] from InterViews.

8.2.2 Speed

We expected that moving to C++ from CommonLisp would increase our speed performance by a large factor. In fact, we have seen a factor of three improvement in speed even though we spent years optimizing the performance of Garnet and we have not yet worked on Amulet's.

Naturally, we want the system to perform even better, though. Presently, we use a basic sequential search to look up slots, so switching to hash tables for slot lookup and caching prior results are expected to be big wins.

9. Future Work

In addition to the performance enhancements, there are a number of interesting additions we plan to make to the object and constraint systems in Amulet. We expect to add a preprocessor in order to improve the syntax. For example, it might allow references like `obj.SLOT` instead of `obj.Get(SLOT)` and allow the expressions of formulas to be defined in-line without requiring an extra function name to be declared, possibly like:

```
obj.LEFT = {{return owner.LEFT + 10}};
```

We also plan to add a number of new and existing constraint solvers to Amulet, including a multi-way solver based on SkyBlue [12], a new multi-way solver by Brad Vander Zanden [15], and an incremental solver based on Bramble [5].

There are also a large number of additions that will be made to other parts of Amulet, including new widgets, interaction techniques, and interactive tools.

10. Related Work

Of course, Amulet is most like Garnet. The differences between Amulet and Garnet's objects [11] and constraints [14] were discussed above. The primary other research project investigating the prototype-instance model is the SELF [2] group. There are many differences between the SELF and Amulet models, however. SELF is its own language, so it does not have to integrate with an existing language. SELF uses a pure copy-down semantics, so after an instance is created, changes to the prototype are not reflected in the instances. Finally, SELF does not support constraints.

There are many research systems which support constraints. The EVAL/vite system [8] integrates constraints with C++ by using a preprocessor and a special sublanguage for the constraints. Like Amulet, EVAL/vite is a ''one-way'' solver. CONSTRAINT [13] and its follow-on algorithm QuickPlan [15] are ''multi-way'' solvers that allow constraints to be expressed over arbitrary types (e.g., integers, strings, colors, etc). MultiGarnet [12], which is an implementation of the SkyBlue multi-way solver, was integrated with Garnet. This was the first attempt at combining two constraint solvers in a single system, and inspired

Amulet's goal for making it easy to integrate multiple solvers. Rendezvous [6] was designed to help create multi-user applications in Lisp. Like Amulet, Rendezvous allows multiple one-way constraints to be attached to a variable. However, Rendezvous requires that variables be explicitly declared and uses a different implementation algorithm. Also, Rendezvous requires that all side-effects from constraints be deferred.

11. Conclusion

Amulet contains an effective and efficient implementation of the prototype-instance object model and constraint solving integrated into a powerful user interface development environment. These features have contributed to the ease of use of previous systems, so we are excited that they can be conveniently provided in C++ without the need for a pre-processor. The innovations reported here which solve problems shown by experience with previous systems, along with future enhancements, will help make Amulet an attractive environment for researchers, students and user interface developers.

Acknowledgements

Andy Mickish, Alan Ferrency, Alex Klimovitski, Chun K. So, and Amy McGovern helped develop Amulet. For help with this paper, we would like to thank Brad Vander Zanden, Alan Ferrency and David Kosbie. For major contributions to the stale dependency algorithm, we would like to thank David Kosbie and Dario Giuse who illustrated the basic problems, and implemented a similar algorithm in Garnet. This research was sponsored by NCCOSC under Contract No. N66001-94-C-6037, Arpa Order No. B326. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of NCCOSC or the U.S. Government.

References

1. Paul R. Calder and Mark A. Linton. Glyphs: Flyweight Objects for User Interfaces. ACM SIGGRAPH Symposium on User Interface Software and Technology, Proceedings UIST'90, Snowbird, Utah, Oct., 1990, pp. 92-101.

2. Craig Chambers, David Ungar, and Elgin Lee. "An Efficient Implementation of SELF, a Dynamically-Typed Object-Oriented Language Based on Prototypes". *Sigplan Notices 24*, 10 (Oct. 1989), 49-70. ACM Conference on Object-Oriented Programming; Systems Languages and Applications; OOPSLA'89.

3. Brad A. Myers, Dario A. Giuse, Andrew Mickish, and David S. Kosbie. Making Structured Graphics and Constraints Practical for Large-Scale Applications. Tech. Rept. CMU-CS-94-150, Carnegie Mellon University Computer Science Department, May, 1994. Also appears as CMU-HCII-94-100.

4. Gideon Avrahami and Kenneth P. Brooks and Marc H. Brown. A Two-View Approach To Constructing User Interfaces. Computer Graphics, Proceedings SIGGRAPH'89, Boston, MA, July, 1989, pp. 137-146.

5. Michael Gleicher. A Graphics Toolkit Based on Differential Constraints. ACM SIGGRAPH Symposium on User Interface Software and Technology, Proceedings UIST'93, Atlanta, GA, Nov., 1993, pp. 109-120.

6. Ralph D. Hill. The Rendezvous Constraint Maintenance System. ACM SIGGRAPH Symposium on User Interface Software and Technology, Proceedings UIST'93, Atlanta, GA, Nov., 1993, pp. 225-234.

7. Ralph D. Hill, Tom Brinck, Steven L. Rohall, John F. Patterson, and Wayne Wilner. "The Rendezvous Architecture and Language for Constructing Multiuser Applications". *ACM Transactions on Computer-Human Interaction 1*, 2 (June 1994), 81-125.

8. Scott E. Hudson. A System for Efficient and Flexible One-Way Constraint Evaluation in C++. Tech. Rept. 93-15, Graphics Visualizaton and Usability Center, College of Computing, Georgia Institute of Technology, April, 1993.

9. David Kosbie. *Hierarchical Event Histories*. Ph.D. Th., Computer Science Department, Carnegie Mellon University, 1995. In progress.

10. Brad A. Myers, Dario A. Giuse, Roger B. Dannenberg, Brad Vander Zanden, David S. Kosbie, Edward Pervin, Andrew Mickish, and Philippe Marchal. "Garnet: Comprehensive Support for Graphical, Highly-Interactive User Interfaces". *IEEE Computer 23*, 11 (Nov. 1990), 71-85.

11. Brad A. Myers, Dario Giuse, and Brad Vander Zanden. "Declarative Programming in a Prototype-Instance System: Object-Oriented Programming Without Writing Methods". *Sigplan Notices 27*, 10 (Oct. 1992), 184-200. ACM Conference on Object-Oriented Programming; Systems Languages and Applications; OOPSLA'92.

12. Michael Sannella. SkyBlue: A Multi-Way Local Propagation Constraint Solver for User Interface Construction. ACM SIGGRAPH Symposium on User Interface Software and Technology, Proceedings UIST'94, Marina del Rey, CA, Nov., 1994, pp. 137-146.

13. Brad T. Vander Zanden. *Incremental Constraint Satisfaction and Its Application to Graphical Interfaces*. Ph.D. Th., Cornell University, Ithaca, NY, 1988.

14. Brad Vander Zanden, Brad A. Myers, Dario Giuse and Pedro Szekely. "Integrating Pointer Variables into One-Way Constraint Models". *ACM Transactions on Computer Human Interaction 1* (June 1994), 161-213.

15. Brad Vander Zanden. An Incremental Algorithm for Satisfying Hierarchies of Multi-way, Dataflow Constraints. Tech. Rept. CS-95-282, Computer Science Department, University of Tennessee, March, 1995.