The Design and Implementation of Amulet's ORE System

Bradley T. Vander Zanden*
University of Tennessee
bvz@cs.utk.edu

November 23, 1998

Abstract

This essay is primarily concerned with the design and implementation of Amulet's ORE object system. The ORE chapter in the Amulet reference manual provides a good, high-level introduction to the ORE system. This essay assumes that the reader has already read the ORE chapter in the Amulet reference manual. However, a very quick description of the ORE system is given ?rst, followed by a detailed description of its implementation.

1 ORE Design

An ORE object consists of a set of named ?elds called *slots*. A slot may contain either a piece of data or a pointer to a method. A slot's value may be computed by one or more constraints. When a slot's value is requested, these constraints are automatically evaluated before the slot's value is returned.

The ORE system supports a prototype-instance object model. This means that an object, called an instance, can be created from any other object, called a prototype. The programmer may add new slots to the instance and may change the

*A disclaimer–The author is not the original developer of the ORE system. Brad Myers and Richard McDaniel at Carnegie Mellon University did the bulk of the development work. However, the author has extensive experience with the code in the Amulet ORE system via his own development of Amulet applications. This document is the author's attempt to codify his understanding of Amulet's ORE system so that interested programmers can gain an initial understanding of the code in the ORE system.

values of any slots that are inherited from the prototype. The instance inherits default values from the prototype for any slots that the programmer does not override.

The ORE system also supports a composite object model. A composite object is an object that is composed of simpler objects, either primitive objects such as Am_Rectangle or Am_Text, or other composite objects. The parts of a composite object can be named so that constraints can easily reference the parts of a composite object and so that operations can easily retrieve the parts of a composite object. Each part also has a pointer to its parent so that constraints in the part can easily access the parent's slots. When an instance of a composite object is created, ORE recursively creates instances of all of the composite object's parts so that the entire composite object is instanced.

Finally the ORE system supports a demon model. This means that a programmer can attach procedures to a slot that should be executed either when 1) the slot changes value, or 2) the slot is invalidated because a constraint that computes its value is invalidated. When a slot changes value or is invalidated, the demon procedures associated with the slot are added to a queue, called a demon queue. At various points during an application's execution, typically during a `Get` but at other times as well, the demon procedures are dequeued and executed. The demon procedures take one argument, which is the slot that caused them to be queued.

In summary, the ?ve primary components of the object system are: 1) objects, 2) constraints, 3) the prototype-instance inheritance mechanism, 4) the composite object mechanism, and 5) demons. The design and implementation of each of these subsystems are explained in greater detail in the sections that follow. The object, constraint, and prototype-instance components each have their individual sections. The composite object and demon mechanisms are explained in the object section. This description of the composite object and demon mechanisms, in conjunction with the design information given in the ORE manual, should be enough to get a high-level overview of how composite objects and demons are designed and implemented.

2 Terminology

In the ensuing discussion, it may be easy to confuse the Amulet objects discussed in the Amulet manual with the objects that the ORE module uses to implement these Amulet objects. The external interface provided to programmers is a prototype-instance model. However, internally Amulet uses C++'s class-instance model to implement Amulet objects. We will use the terms "Amulet object", prototype object, and instance object to refer to the prototype-instance objects that Amulet programmers manipulate. We will use the names of the C++ classes that ORE uses to

implement these prototype-instance objects to refer to the implementation objects.

3 Objects

The main implementation classes for objects are `Am_Object` and `Am_Object_Data`. ORE also implements the `Am_Object_Advanced` class, which is primarily a class that gives the Amulet programmer access to advanced features.

When the user creates an instance of an Amulet object, Amulet returns an Am_Object. However, an Am_Object is simply a lightweight container class. Each Am_Object contains a single instance variable, called `data` that points to an object of type Am_Object_Data. An Am_Object_Data object contains all the information (or data) about the newly created instance. So an Am_Object is the object that the programmer manipulates and an Am_Object_Data is the object that Amulet manipulates.

An Am_Object may be thought of as a bridge object between the programmer and the ORE implementation. Whenever a programmer invokes a method on an Am_Object, the method manipulates the Am_Object's Am_Object_Data. Multiple Am_Objects may point to the same Am_Object_Data, and they are considered identical if they do so. It is fair to say that the Am_Object_Data class is considered by ORE to be the "real" Amulet object and that the Am_Object class is just a shell or container.

The Am_Object_Data class keeps the following information about an Amulet object:

1. Prototype: A pointer to the object's prototype. The pointer is a pointer to the prototype's Am_Object_Data object.

2. Owner: A slot object (slot objects are described in Section 3.1) that points to the object's owner. The slot object actually points to the owner's Am_Object_Data object.

3. Parts: A list of the object's parts, both graphical and non-graphical. The list points to the parts' Am_Object_Data objects. The list is implemented using a ?rst_part/next_part style. The ?rst_part ?eld points to the ?rst part in the parts list. The next_part ?eld points to the next part in the *owner's* parts list.

4. Instances: A newly created instance will not have any instances, but if the new instance is ever used as a prototype, then the Am_Object_Data object will use a list to keep track of these instances. The list points to the instances' Am_Object_Data objects. The list is implemented using a ?rst_instance/next_instance style. The ?rst_instance ?eld points to the ?rst instance in the instances list.

The next_instance ?eld points to the next instance in the *prototypes's* instance list.

5. Demons: The Am_Object_Data class has several ?elds that contain information about an Amulet object's demons, including:

   - demon_set: A pointer to the "set" of demon procedures associated with this object. The ORE manual has more information about demon sets.
   - demon_queue: The demon queue on which the demons associated with this object should be queued for execution. The ORE manual has more information about demon queues.
   - demons_active: This ?eld serves two purposes:
     (a) the left-most bit indicates whether the demons for this object are active or inactive. This bit is set by the Invoke_Demons method, which is described by the ORE manual.
     (b) the remaining bits indicate whether a per-object demon has already been queued. The ?rst slot to cause a per-object demon to be enqueued sets the appropriate bit of the demons_active ?eld to true, so that subsequent slots will not enqueue this demon again.
   - default_bits: This ?eld indicates which demons are enabled by default when an instance is created. The bits in this ?eld correspond to bits in the demon set. If a bit is true, then the demon corresponding to this bit will be enabled in *every* slot in the newly created instance.
   - bits_mask (a.k.a. demon_mask): The bits mask is used to control whether the presence of an enabled demon in a prototype slot will force ORE to make a temporary slot in every instance. The ORE manual describes this ?eld (under the name demon_mask) in more detail.

6. slots: As described in Section 3.1, a slot is represented internally by an Am_Slot_Data object. Pointers to an object's local slots are stored in a dynamic array called *data*. We will refer to this array as a *slot array*. The slot array dynamically grows and shrinks as slots are added to or deleted from the Amulet object. A slot is stored locally if it meets one of the following three conditions: 1) its value has been explicitly set by the programmer, 2) it is inherited but has constraints, or 3) it is inherited but has at least one enabled demon that is contained in the bits_mask. If a slot does not meet one of these three conditions, then it is stored in one of the object's ancestors in the prototype-instance hierarchy.

The ORE implementation classi?es slots into one of three types: 1) owner slot, 2) part slot, and 3) data/method slot. For the sake of ef?ciency, the owner slot is explicitly stored in an Am_Object_Data's owner ?eld rather than in the slot array. *Pointers* to part slots and data/method slots are stored in the slot array. However, part slots are created differently than data/method slots. Every Am_Object_Data has a ?eld called `part_slot`. This ?eld is a slot that is created when the Am_Object_Data object is created. When a part is added to an owner using `Add_Part`, a pointer to the part's part_slot is added to the owner's slot array. When the part is removed from the owner, the pointer to the part_slot is removed from the owner's slot array but the part_slot is not destroyed. In contrast, when a data/method slot is added to an Amulet object, a slot object is created for that slot and a pointer to it is added to the appropriate slot array. When the slot is deleted from the Amulet object, the slot is destroyed and its memory is returned to the memory manager. From an implementation point of view, the difference between a part slot and a data/method slot is that a part slot is created when an Amulet object is created, whereas a data/method slot is not created until it is actually needed.

3.1 Slots

Slots in Amulet are implemented using two classes called `Am_Slot` and `Am_Slot_Data`. Like `Am_Object`, `Am_Slot` is a container class. It has a single ?eld called `data` that points to an object of type `Am_Slot_Data`. An Am_Slot_Data contains all the information about a slot. Whenever the programmer wishes to manipulate a slot, ORE gives the programmer an Am_Slot object that has a pointer to the appropriate Am_Slot_Data object. Internally ORE always manipulates the Am_Slot_Data object.

An `Am_Slot_Data` object maintains the following information about a slot:

1. The slot's current value: Am_Slot_Data is a subclass of Am_Value, so a slot object inherits both a value ?eld and a type ?eld from Am_Value.
2. The slot's key: This ?eld is an integer key that uniquely identi?es the slot.
3. The object to which the slot belongs: This ?eld is a pointer to an Am_Object_Data object.
4. Constraint information: A slot maintains three types of constraint information:
   - ² constraints: `constraints` is a list of constraints that have been attached to the slot. Any of these constraints can compute the slot's value.

- invalid_constraints: `invalid_constraints` is a list of the slot's constraints that have been invalidated and need to be re-evaluated.
- dependencies: `dependencies` is a list of constraints that depend on this slot. Each constraint on this list has retrieved the value of this slot using a `Get` call. When the slot's value is changed or invalidated, all the constraints on this list will be invalidated.

5. Demon information: A slot maintains information about which demons are enabled for it and which of these demons are currently queued. The demon_set of the object to which this slot belongs determines the possible set of demons that may be attached to this slot. The enabled_demons ?eld determines which of these demons are actually attached to this slot. The enabled_demons ?eld is a bit ?eld. A 1 bit means that the demon in the corresponding position in the demon_set is enabled for this slot.

   The queued_demons ?eld is a bit ?eld that keeps track of which of the slot's demons are currently queued on the demon queue. A 1 bit means that the demon in the corresponding position in the demon_set is queued for this slot.

6. Type checking information: This ?eld contains an index into an array of type checking functions that is contained in the slot owner's demon_set. When the slot's value is set, the setting routine checks whether the type_check index is non-zero, and if so, it uses type_check to extract the appropriate type checking function from the type_check array and uses the function to type check the new value.

7. Inheritance information: This ?eld contains the inheritance rule for this slot. The ORE manual describes inheritance rules for slots in greater detail.

3.2 Am_Object_Advanced

As noted earlier, the `Am_Object_Advanced` class is primarily a class that is intended to give Amulet programmers access to more advanced features in Amulet. Am_Object_Advanced is a subclass of Am_Object and it does not add any additional instance variables. It does add a number of additional public methods that allow the user to access and set some of the individual ?elds in an Am_Object_Data object. Since Am_Object_Advanced does not add any additional instance variables to an Am_Object, an Amulet programmer typically coerces an Am_Object to an Am_Object_Advanced and then invokes the desired methods.

Method Name	Parameters	Description
Get	None	Executes the constraint's code and returns the computed value
Invalidated	Slot	Noti?es the constraint that the value of this slot is no longer valid. Any constraint that indirectly depends on a changed slot will have this method called.
Changed	Slot	Noti?es the constraint that the value of this slot has changed. Any constraint that directly depends on a changed slot will have this method called.
Constraint_Added	Slot	Noti?es the constraint that it has been assigned to this slot.
Constraint_Removed	Slot	Noti?es the constraint that it has been removed from this slot.
Dependency_Added	Slot	Noti?es the constraint that it now depends on this slot.
Dependency_Removed	Slot	Noti?es the constraint that it no longer depends on this slot.

Table 1: Amulet's API for a constraint object. Several less important methods are omitted.

4 Constraints

The Amulet constraint system is designed to handle multiple, cooperating constraint solvers. This section provides an overview of Amulet's constraint API and then provides a more nuts-and-bolts description of the various classes that implement the constraint system.

4.1 Basic Design

As shown in Table 1, Amulet's constraint API is oriented toward constraint solvers that work with data?ow graphs. Conceptually, each constraint and each slot can be thought of as a vertex in the data?ow graph. There is a directed edge from a slot to a constraint if the constraint's formula references the slot. There is a directed edge from a constraint to a slot if the constraint determines the value of that slot.

In practice, the data?ow graph is implemented as follows. Each slot has a ?eld

called `dependencies`. The dependencies ?eld points to a list of constraints that reference the slot in their formulas. Hence the dependencies list represents the set of directed edges from a slot to constraints that depend on this slot. Amulet does not require a constraint to keep a pointer to the slot which it sets, but it does assume that the constraint will be able to notify the slot if the constraint's value changes or might possibly change.

When a slot is changed, Amulet expects that all the slots and constraints that depend either directly or indirectly on the slot will be noti?ed and marked out-of-date. A constraint that depends directly on a changed slot is noti?ed of the change via the **Changed** method. A constraint that depends indirectly on a changed slot is noti?ed of the change via the **Invalidated** method. The idea is that a constraint which depends on a changed slot will have to be re-evaluated, but a constraint that indirectly depends on a changed slot may not have to be re-evaluated. The two methods allow a constraint to respond in the appropriate fashion.

When a constraint is marked out-of-date, either because it depends directly or indirectly on a changed slot, Amulet ?nds the slot that the constraint sets, and it adds the constraint to this slot's list of invalid constraints. It also adds a Validate demon for the slot to an Am_Demon_Queue. When the queue is processed, the validate demon will be called, the slot's invalid constraints will be executed, and the slot's value will be brought up-to-date.

Constraint Solving Amulet assumes that each constraint will have a piece of code that can be executed to satisfy that constraint. The code you provide the `Am_Define_Formula` macro is an example of such a piece of code. This code is invoked by calling the constraint's **Get** method. When the Validate demon mentioned in the previous section is invoked, it goes through a slot's list of invalid constraints and calls their Get method. The Get method executes the constraint's code and returns the newly computed value. If there are multiple invalid constraints, the last constraint to be executed sets the slot (the last constraint to be executed is the ?rst constraint that was invalidated).

Constraint Activation/Deactivation In Amulet a constraint is activated when it is assigned to a slot and it is deactivated when it is removed from a slot. Amulet calls a constraint's `Constraint_Added` and `Constraint_Removed` methods to allow the constraint to take any bookkeeping actions required to activate or deactivate itself.

Adding/Removing Dependencies Amulet expects a constraint solver to add a dependency from a slot to a constraint if the constraint accesses that slot. The con-

straint solver does this by calling a slot's `Add_Dependency` method (Add_Dependency is one of the methods de?ned for the Am_Slot class). The `Add_Dependency` method adds the constraint to the slot's dependency list and calls the constraint's `Dependency_Added` method to let the constraint know that it now depends on this slot.

Similarly, when the constraint no longer needs to access the slot, Amulet expects the constraint solver to remove the dependency between the slot and the constraint. The constraint solver does this by calling a slot's `Remove_Dependency` method (Remove_Dependency is one of the methods de?ned for the Am_Slot class). The `Remove_Dependency` method removes the constraint from the slot's dependency list and calls the constraint's `Dependency_Removed` method to let the constraint know that it no longer depends on this slot.

4.2 Constraint Implementation

The main implementation classes for constraints are `Am_Constraint` and `Am_Constraint_Conte` Both of these classes are abstract base classes. In order to create a new constraint solver, one must subclass each of these two classes. Am_Constraint provides the API described in the previous section. Each subclass of Am_Constraint must implement each of Am_Constraint's methods.

Am_Constraint_Context primarily serves to establish which constraint solver is active when a Get method is executed. When a constraint starts executing, it it supposed to call a macro called Am_PUSH_CC and pass it a constraint context object. The type of the object should be the constraint context de?ned for this particular type of constraint. Am_PUSH_CC pushes the constraint context onto a stack of constraint contexts. Similarly, once the constraint has ?nished its execution, it should call Am_POP_CC so that the constraint context object gets popped off the stack.

When the Get method is invoked on an Am_Object, the Get method eventually gets passed to the current constraint context. The current constraint context is responsible for retrieving the value of the requested slot, as well as performing any other necessary actions, such as creating a dependency from the slot to the requesting constraint.

The abstract base class for a constraint context does not specify any instance variables that must be de?ned for a constraint context. It is the responsibility of each constraint solver to determine what information should be stored in a constraint context.

As an example, consider the one-way formula constraints that are provided by the Amulet implementation. These one-way formulas are implemented using the classes `Formula_Constraint`, which is a subclass of Am_Constraint,

and `Formula_Context`, which is a subclass of Am_Constraint_Context. Formula_Constraint uses a helper class called `Am_Formula` which contains the code for a single constraint. When a Formula_Constraint is to be executed, its Get method is called. The Get method creates and pushes a Formula_Context object onto the context stack (this stack is actually a virtual stack whose implementation is not important for this essay), calls the function code in its Am_Formula object, pops the Formula_Context object off the stack, and returns the computed value.

Among other things, a Formula_Context object stores the constraint that created it and the slot whose value this constraint computes. The Get method for a Formula_Context object ?rst checks whether the slot being requested is on the list of slots that the constraint depends on (a Formula_Constraint maintains the list of slots it depends on–it adds to this list when `Dependency_Added` is called and it subtracts from this list when `Dependency_Removed` is called. If not, the Get method creates a dependency from the slot to the constraint. It then returns the value of the requested slot.

In sum, the implementor of a constraint solver has a great deal of ?exibility in implementing a new constraint solver. The primary requirement is that the constraint solver be data?ow-oriented, and that it implement the API for the Am_Constraint and Am_Constraint_Context classes. The types of data it stores, any additional classes it de?nes, and any additional methods it de?nes are completely left up to the constraint implementor.

5 Prototype-Instance Model

ORE has different rules for the inheritance of parts versus the inheritance of values and constraints. When an instance of an object is ?rst created, parts and values/constraints are treated similarly. The instance inherits all parts from the prototype that have not been labeled Am_NO_INHERIT. The instance also inherits all of the prototype's slots and all of the constraints on these slots. Initially the only slots that are copied down to the new instance are 1) slots containing constraints and 2) slots that contain demons that should be executed even though the slot is inherited (see Section 3 for more details). Later, when the programmer explicitly sets slots that have not been copied down, copies of the slots being set are made and added to the instance's slot array.

The difference in the handling of parts and value/constraints comes after an object has been created. The following list describes how inheritance is handled after an object has been created. Before presenting it, a couple things must be made clear. First, we assume that the inheritance rule for each slot is Am_INHERIT. In other words, we assume that a slot is inherited from its prototype unless it has been

explicitly set. Second, when a slot inherits a constraint from its prototype slot, the slot is considered to be inherited until one of the slots that the constraint depends on is marked no longer inherited. Until that time the slot is still considered to be inherited. After one of the constraint's dependent slots is marked not inherited, the slot is also considered to be no longer inherited.

The list of rules for inheritance after an object has been created can now be enumerated as follows:

- Adding a part to the prototype: If a part is added to the prototype, the new part will *not* be added to the prototype's existing instances. Any new instances will received the added part, but any pre-existing instances do not receive the added part.

- Removing a part from the prototype: If a part is removed from the prototype, the removed part will *not* be removed from the prototype's existing instances. Any new instances will not receive the removed part, but any pre-existing instances will retain the part.

- Adding a slot to the prototype: If a slot is added to the prototype, then that slot will be inherited by each existing instance unless the slot has already been added to that instance. If a constraint is stored with the slot when it is added to the prototype, then the constraint is propagated to any instances that have not overridden that slot. In order to propagate the constraint, a slot object is created in each instance that receives the constraint.

  If the slot being added to the prototype does not contain a constraint but does contain a demon that should be run even if the slot is inherited, a temporary slot will *not* be created in the instances. Hence the demons will not actually run. This appears to be a bug in the current ORE system.

- Removing a slot from the prototype: When a slot is removed from the prototype, the prototype now inherits the slot from its prototype (due to a bug in the current ORE implementation, this does not always happen–if nothing used to inherit from the prototype, then the prototype will initially inherit the slot's value, but if the slot's value changes, the prototype will not get the new value).

  For the sake of clarity, we will call the slot that the prototype now inherits `new_prototype_slot`. If new_prototype_slot contains a constraint, this constraint will *not* be inherited by the prototype, nor will it be inherited by any of the prototype's existing instances. Since the prototype does not inherit new_prototype_slot's constraint, neither do any instances of the prototype created after the slot is removed. So neither the existing instances nor

instances created after the slot is removed will receive new_prototype_slot's constraint. This appears to be a bug in ORE.

If new_prototype_slot has a demon that should be run if the slot is inherited, then a temporary slot will be created in the prototype object so that the prototype will execute the demon if new_prototype_slot's value is changed.

The existing instances of prototype will also inherit from new_prototype_slot if they previously inherited from the prototype *and* if they do not contain a constraint that was inherited from the prototype. If they contain a constraint that was inherited from the prototype, this constraint is not removed. Whether they inherit from new_prototype_slot then depends on whether or not the constraint depends on inherited values.

- Setting a slot with a value: When a value, as opposed to a constraint, is stored in a slot, the following actions happen:

  1. All instances that inherit this slot are noti?ed of the changed value. If an instance inherits the slot but maintains a temporary slot object for the slot the value will be copied into the temporary slot.
  2. If the slot previously inherited its value, its temporary slot object, if one exists, is converted to a permanent slot object. If a temporary slot object does not exist, then a new slot object is allocated for the slot and stored in the slot array of the object to which the slot belongs.
  3. If the slot previously inherited its value, then it noti?es all of its dependencies, via something Amulet calls a slot_event, that it is no longer inherited. This noti?cation allows constraints that previously were marked inherited to notify their slots that they are no longer inherited and so on.

- Setting a slot with a constraint: When a constraint is stored in a slot, that constraint is propagated to every instance that inherits from the slot. This can result in confusing behavior if the prototype previously had a different constraint. The reason is that any instances that have the pre-existing constraint but are still marked as inheriting from the prototype will receive the new constraint, while any instances that have the pre-existing constraint but are not marked as inheriting from the prototype will not receive the new constraint. Whether or not an instance with the pre-existing constraint is marked inherited depends solely on whether that constraint depends yet on a non-inherited slot. As an additional source of confusion, any instances created after the constraint is added will receive the new constraint. So one can

be left with a situation where some of the pre-existing instances have the old constraint, some of the pre-existing instances have the new constraint, and all of the instances created after the constraint is added will contain the new constraint.

- Destroying the prototype: If the prototype is destroyed, all the instances will also be destroyed.

5.1 Slot Access

The previous section described the rules for object and slot inheritance. This section briefly describes how inheritance operates when a slot is accessed. When a slot's value is requested from an object, ORE first searches the object's slot array. If the slot array does not contain the slot, ORE then searches the prototype's slot array, then the slot array of the prototype's prototype and so on until the slot is either located or the root of the prototype instance hierarchy is reached. If no slot can be found, an error value is returned.

5.2 Implementation

ORE implements Amulet's prototype-instance model using four fields in an Am_Object_Data: prototype, first_instance, next_instance, and default_rule. The first three of these fields are described in Section 3. The last of these fields stores the default inheritance rule for the slots in that object. The ORE manual describes the different types of inheritance rules that may be used for slots. When a slot is created, the default rule is copied into the `rule` field of the slot's Am_Slot_Data object. If the inheritance rule has been changed in the slot's prototype, then the `rule` field is subsequently assigned the value of the `rule` field in the slot's prototype. The `rule` field is the only field related to the prototype-instance model that is stored in an Am_Slot_Data object.