Using Dynamic Configurability to Support Object-Oriented Languages and Systems in Legion

University of Virginia Computer Science
Technical Report CS-96-19[1]
December 4, 1996

Michael J. Lewis and Andrew S. Grimshaw
{mlewis, grimshaw}@Virginia.edu
Department of Computer Science
University of Virginia
Charlottesville, Virginia

Abstract

Wide area distributed object systems will require mechanisms for creating, describing, and managing objects. The mechanisms must be scalable and must not mandate particular policies or algorithms because users will have different cost, security, performance, and functionality demands. Legion is a wide area distributed object system that supports this requirement using a first-class active class system. A Legion class is an active Legion object that manages its instances, provides their system-level support, and defines their behavior. To enable encapsulation and code reuse, and to simplify the requirements of many Legion programmers, Legion classes can be shared among Legion objects. To support instances that outlive the program that creates them, classes are persistent. To enable scalability, classes are distributed throughout the system.

Shared, persistent, active, distributed (SPAD) class systems have many benefits for wide area distributed object systems such as Legion. But supporting traditional class roles, including instantiation and inheritance, requires new and different implementation solutions. This paper describes one way to support object-oriented languages and systems in the Legion SPAD class system. Our method is based on dynamic configurability of objects and classes. We describe dynamic configurability, motivate the need for new implementation solutions for a SPAD class system, describe the implementation of our solution, and illustrate its functionality with an example.

1 Introduction

The widespread deployment of gigabit networks will effectively shrink the distance between computing resources and will enable wide area distributed and parallel computing systems. These systems will consist of many heterogeneous, distributed, and unreliable resources. Without significant software support, users will not be able to manage the complexity of this envi-

1. This work partially supported by DARPA(Navy) contract # N66001-96-C-8527, DOE grant DE-FD02-96ER25290, and DOE contract Sandia LD-9391

ronment. Meta-systems software [10]—software that resides "above" physical resources and operating systems and "below" users and applications programs—is needed. Legion [8, 12] is one example of meta-systems software. Among Legion's many components will be a run-time system, Legion-aware compilers that target this run-time system, and programming languages that present applications programmers with a high level abstraction of the system. Thus, Legion will allow its users to write programs in their favorite languages, and will transparently create, schedule, and utilize distributed objects to execute the programs.

Users of meta-systems software will require a wide range of services in many different dimensions, including security, performance, monetary cost, and functionality. No single policy or static set of policies will satisfy every user, so users should be allowed to implement their own solutions and determine their own trade-offs as much as possible. Legion supports this philosophy by providing the mechanisms for system-level services such as naming, binding, and migration, and by not mandating these services' policies or implementations. Legion requires a certain object-mandatory functional interface to all Legion objects; the implementation of the interface is left up to the object or to the class of the object. Further, Legion specifies the functional interface to a set of core system object types. Again, the implementation of these system objects will vary; only the minimum set of core member functions that each object type must support is specified by Legion. Finally (and most importantly for this paper), Legion delegates much of what is usually considered system-level responsibility to classes, which are special Legion objects. For instance, classes are responsible for creating and locating their instances, and for selecting appropriate security and object placement policies. The core system objects provide mechanisms for user-defined class objects to implement policies and algorithms that best match their requirements in terms of performance, security, cost, and functionality.

Since Legion classes are themselves Legion objects, the Legion class system has several characteristics that distinguish it from the class systems of traditional object-oriented programming languages and databases. In particular, the Legion class system is shared, persistent, active, and distributed (SPAD), characteristics whose meanings in the context of class systems are described below.

Shared: A class is shared by two programs in a traditional language only when the compiler and programmer explicitly cause the class to be local to each program, in which case each program effectively has its own copy of the class's representation. Multiple programs only

"share" a class in the sense that they see identical but separate representations of that class. The programs can remain largely autonomous, especially if they do not interact with one another. That is, changing the class in one program does not necessarily have any effect on the others that "share" it. But like many object-oriented database systems, Legion allows multiple programs to share a single *representation* of a class. Thus, when the representation changes for one object, it necessarily changes for the others. This is fundamentally different from having multiple copies of the "same" class.

Persistent: Legion objects, and therefore Legion classes, will also have different persistence characteristics from classes in traditional programming languages, in which a class's representation only needs to persist for the lifetime of a program that uses the class. When a traditional class is changed, programs that use the class are typically recompiled to reflect that change. Thus, the new version of the class effectively replaces the old version in the new object code or executable. This model cannot easily be supported in the Legion environment because classes live outside—and are therefore independent from—the object code of the programs that use them. For instance, as the manager of its instances, a class is potentially responsible for migrating, securing, and locating them. Therefore, a class object cannot simply be overwritten or replaced indiscriminately every time it changes; doing so could create "orphan" Legion objects. Databases that support schema evolution encounter a similar problem, but can deal with the problem more easily because (1) the Database Management System (DBMS) can serve as a centralized entity with global knowledge of the contents of the system, and (2) database classes are not active entities represented by processes.

Active: Legion classes can have at least one thread of control, can accept member function invocations, and can call the member functions of other Legion objects. In traditional programming languages and database systems, classes are passive. They are used by compilers to give appropriate characteristics to instances, but this is done via tables and other static constructs that can be generated directly from the description of the class in the source language. Classes are objects in a number of other programming languages and systems. However, no language or system of which we are aware associates a separate, independent, address-space-disjoint thread of control with each class, as Legion does.

Distributed: The representation of a Legion class will not necessarily be "local" to all of its uses; the class could be running as a Legion object on a different machine from the one on

which it is being used. In contrast, the programmers and compilers of traditional languages must make the class's representation local to its use by explicitly including the class's description and implementation in an included header file, or by linking object files or library code into the program's executable. Thus, by the time the program is ready to run, a class's representation is local to the code that uses the class.

These four characteristics challenge Legion-targeting compilers to perform some of their fundamental responsibilities in new ways. The process of building compilers that target Legion-like systems will lead to many interesting and challenging research problems. This paper introduces one mechanism, *dynamic configurability*, for supporting class creation, class alteration, instantiation, and inheritance in SPAD class systems. Section 2 introduces the concept of dynamic configurability and describes our implementation. Section 3 explains the need for new implementation solutions for object-orientation in a SPAD class system, and describes how we use dynamic configurability to support object-orientation in Legion. Section 4 presents an illustrative example that includes initial performance results, and Section 5 summarizes the paper.

2 Dynamic configurability

A *dynamically configurable object* (DCO) is an object whose implementation can change incrementally after the object is instantiated and running.[2] An incremental change can take one of several different forms. For instance, the implementation of a member function in the object's public interface can be replaced with a different implementation. This would allow, for example, an optimized implementation to be plugged into an object "on-the-fly," without requiring recompilation of the DCO itself or of the other objects that use it. In general, a DCO is made up of *implementation components*, each of which contains the implementation of one or more *interface elements*. An interface element can be either (1) a function in the public interface of the object, (2) a private function that is not in the public interface of the object but that can be called from within the object, or (3) an instance variable along with the full set of operations that is allowed on that variable within the object. Thus, an incremental change to a DCO is the addition, alteration, or removal of one or more of the interface elements of that DCO.

2. In this paper, unless otherwise stated, the term "object" refers to an object in the Legion sense: an entity that can be represented by an active process with its own address space and thread of control.

The implementation of a DCO has two characteristics that allow it to support dynamic configurability through incremental change; (1) it exports a core member function named *incorporate()* which can be used to add, remove, and alter interface element implementations, and (2) it contains a *dynamic interface element mapper* (DIEM) which maps interface elements to locations within the running image of the object. Other objects can call a DCO's incorporate() member function, which dynamically incorporates new implementations into the DCO's running image, and which alters the DCO's DIEM to reflect an implementation change. The DIEM manages the DCO's changing interface. All references to a DCO's interface elements–both references from within the object and references made by other objects via remote member function calls–"go through" the DIEM, which maps them to the correct addresses within the DCO's running image. The anatomy of a DCO is depicted in Figure 1.

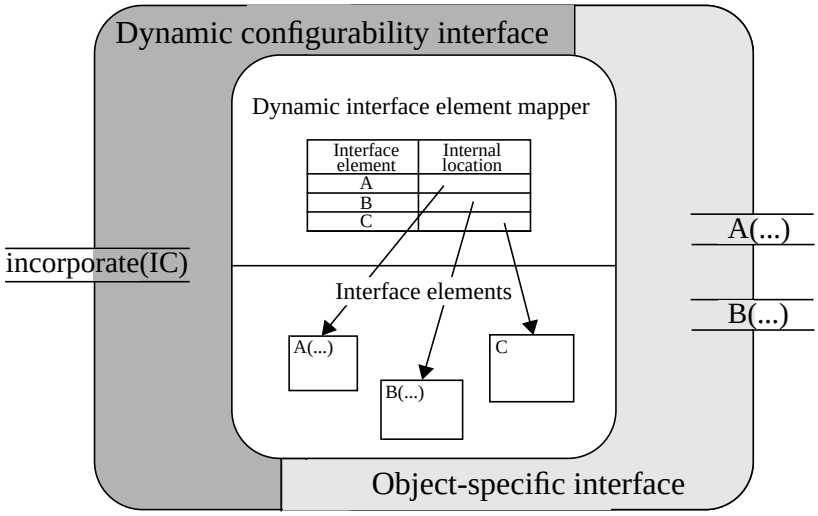

FIGURE 1. The anatomy of a dynamically configurable object

Removing the implementation of an interface element requires removing the old implementation from the running DCO and removing the corresponding entry from the DIEM. Adding a component involves mapping its implementation into the DCO and updating the DIEM to "point to" the new implementation. Finally, changing a component's implementation is a matter of removing the old implementation and adding the new one. These operations are implemented by the incorporate() member function, which takes an implementation component as an argument and returns an integer that indicates the success of the operation. The implementation component parameter contains a list of interface elements, their implementations in the form of object code, and the location of the interface elements within the implementation. If an interface element in the

implementation component parameter is not already present within the DCO, the element and its implementation are added to the object. If the interface element is already present, the current version is replaced by the incoming version. An existing interface element can be removed by passing an interface element that has the same name, but that contains an empty implementation.

The creator (or creators) of a DCO builds the DCO in two separate steps.[3] First, the creator runs a *seed* DCO, which contains an empty DIEM and which exports only the incorporate() member function. Next, the creator configures the DCO by calling incorporate() with new implementation components. This augments the functionality of the object, causes the DIEM to be filled in appropriately, and adds functions to the object's interface. Thus, Figure 1 depicts an object that started as a seed DCO, and whose incorporate() function was called with implementation components that included public member function interface elements A and B, and data interface element C.

2.1 Implementation issues

To implement dynamic configurability efficiently as described above, a DCO must meet two fundamental technical requirements. First, it must be able to incorporate interface element implementations by dynamically augmenting and shrinking its own code segment, and by loading and calling new object code at run-time. Second, it must be able to alter dynamically the public interface that it exports. We address each of these issues in separate sections below.

2.1.1 Dynamic code loading

Several programming languages contain mechanisms that allow programs to load, access, and run new executable code in a process that is already running. Examples include Java [1] class loaders and Kali-scheme dynamic address spaces [5]. Modern operating systems, such as Windows via COM/OLE [4], also enable this functionality. Most modern Unix implementations enable dynamic code loading by providing direct user access to the operating system's dynamic linking facilities. The dlopen() function dynamically loads an object file into the process that calls the function. Internal symbols within the object file can be located using the dlsym() function, which returns the address of a symbol that is passed as a parameter. This address can then be cast

3. The term "creator" is intentionally vague. DCO's can be created by programmers, compilers, classes, other Legion objects, or some combination of all four. Sections 3 and 4 include a more concrete presentation of the roles that different Legion object types will typically play in the process of creating DCO's.

to the appropriate type (e.g. a function pointer), and can be used to call or otherwise access the code in the object file. Our implementation is built in C++ and uses the dynamic linking routines to run on top of Unix.

2.1.2 Dynamic public interface configuration

An object in a distributed system typically accepts some set of public member functions that is fixed at the time the object is created. Changing the interface that an object exports requires access to the code that dispatches incoming messages to the object's user-defined functions. The Legion run-time library [7], which our implementation uses, allows this level of access. The Legion library is designed to be linked into Legion objects to allow them to communicate with one another and to comply with the Legion object model [12]. Since Legion is intended to support many programming models, different levels of security and functionality, etc., the library was designed and implemented to allow for considerable flexibility and extensibility. In particular, objects that use the Legion library can alter their public interface at run time. The details of how to do this is beyond the scope of this paper, but can be found in [7].

3 Supporting object-orientation

This section describes the application of dynamic configurability to supporting object-oriented languages and systems on top of the Legion SPAD class system. We address class creation, class alteration, instantiation, and inheritance. We first describe why a new implementation solution for these old features is necessary, and then discuss how we employ dynamic configurability for one implementation.

3.1 Why a new solution is necessary

This section is designed simply to introduce some of the problems associated with applying common object-orientation implementation solutions to a SPAD class system, particularly Legion. We do not claim to address all existing implementation solutions, nor any particular system in detail. We also do not attempt to introduce all possible alternative solutions. The primary purpose of this section is to motivate the need for our solution, which is presented in Section 3.2. We do this by showing that common existing implementation solutions can't be translated directly to a SPAD class system, and by exposing the shortcomings of some straightforward extensions to the existing solutions.

3.1.1 Creating and altering class objects

Consider the process of creating a new class in an object-oriented programming language. Normally, a compiler simply creates a new representation of the class every time the program is compiled. This is safe to do because the class's representation is stored in the object code of the program, which is either being created for the first time or is completely replacing an earlier version. This is not true in a SPAD class system. The representation of a Legion class lives in an active class object outside of the object code of the compiled program. This means that the creator of a Legion class–presumably a Legion-targeting compiler–must at least cause the executable program for a new class object to be run. As soon as the class is created, it can be used by other objects in the system. Therefore, if a class is changed and recompiled, its representation can be safely replaced by the new version only if no other objects are using the class. Otherwise, the system must enact some policy for replacing the class. The policy could be to deny support for objects that are using the old version, to update all objects that use the class, or some approach in between.

The problems that Legion class alteration introduces are similar to those that are associated with schema evolution of database systems. Some database systems support schema evolution by creating separate views of the database for different users; schema updates only alter certain users' views. This strategy effectively sidesteps the problem of having a single representation for all users. Other systems update all instances and references to changed classes in order to make the system consistent [14]. This solution might be desirable for certain Legion classes, but it is not nearly as easy to implement, particularly because a database contains a logically centralized and authoritative entity (the DBMS) that has access to all parts of the system. The size and autonomous nature of Legion precludes any such entity, so a different solution is necessary.

3.1.2 Creating instances

In traditional object-oriented programming languages, a single copy of a class's member functions is created when the class is created. Each time the class is instantiated, only a new copy of the data is created, not a new copy of the functions. When the member functions are invoked, they operate on the appropriate object data (in C++, a pointer to the appropriate data is passed via the "this" variable). Because Legion classes are distributed, not local to the code that uses them, a strict translation of this implementation solution in a SPAD class system would be inefficient and

inappropriate. If the member functions associated with Legion classes were to reside within the classes themselves, then invoking a method on an object would result in calling a member function on that object's class. This would require that all of an object's data be accessible to the class when the member function were invoked. None of the three most straightforward approaches–(1) having the data of all instances reside within the class, (2) having the class query each instance for its data when a method is invoked, or (3) having the instance pass the class its data with a method invocation–scales well with the number of instances of a class.

A more sensible solution would be to have each instance contain its own copy of the member functions it exports. Since Legion classes are responsible for determining the characteristics of their instances (as are all classes in object-oriented languages and systems) and for creating new instances (a unique feature of Legion), the Legion object model prescribes that class objects logically maintain the executables for their instances. Different executables will exist, for example, for different machine architecture types, performance and security characteristics, etc. To create a new instance, the class runs an executable that implements the new instance. This is an appropriate solution, even with monolithic executables that have static code segments (i.e., "non-dynamically-configurable" objects), for many applications.

However, when classes must be able to change, the solution becomes less adequate. Changing a class definition would entail replacing the executables that represent the instances. If instances are up and running at the time of the class change, they become out of date. This causes problems for a class that needs to answer queries about its instances' interfaces, because different instances could potentially have different interfaces. Out of date instance executables also cause problems for a class that migrates an instance to a new host by having the instance checkpoint itself, then running the instance's executable on the new host, and finally having the instance restore its own state from the checkpoint.[4] Supporting this mechanism without any other changes would at least require that all instances be able to read the checkpoint formats of all supported prior versions of the class.

One approach to supporting instantiation in the face of class changes includes building version capability into the class object. The class would associate a version number with every instance, and would maintain different executables for different versions of the class. Migration

4. Legion uses exactly this approach for object migration.

and interface queries would be handled differently for different instances, with knowledge of the version number of the instance that is being supported. Again, this type of class may be appropriate for many applications, but it doesn't necessarily scale well when the class changes frequently. Using dynamic configurability to support object-orientation (described in Section 3.2) enables a simpler solution that scales well.

3.1.3 Inheritance

Object-oriented languages and systems support a wide variety of inheritance semantics and policies. Interface inheritance allows a subclass to inherit just the interface of its superclass, while implementation inheritance causes the superclass's implementation of member functions and data to be incorporated into the implementations of the subclass's instances. Neither form of inheritance is easy to implement in a distributed class system, because the representation of a class must be obtained before the class can be used for inheritance. This leads to the problem of how to represent the interface or implementation. Object-oriented databases can assume that all objects are built using a single language; thus, the class interface and its instances' implementation can be represented in that language. Distributed object systems, like CORBA, often simplify matters by not supporting implementation inheritance, and by imposing a single interface definition language (IDL). Dynamic configurability allows Legion to support implementation inheritance in a SPAD class system without mandating a particular programming language with which to define and build classes.

3.2 The new solution

In this section, we describe the characteristics and implementation of one style of Legion class, which is based on dynamic configurability. We do not claim to present the only possible way of implementing object-orientation in SPAD class systems. The style we present may not be appropriate for some applications, and thus it is not required in Legion. Legion explicitly allows other implementations to coexist with the one presented in this paper. We describe a new class type, and then show how it is created and altered, and how it supports instantiation and inheritance.

3.2.1 Dynamically configurable classes

A *dynamically configurable class* (DCC) is a class object whose instances are DCO's, and which can be used as a superclass in a SPAD class system.[5] Instead of maintaining a set of mono-

lithic static executables for its instances, a DCC maintains an *implementation component store* (ICS) that contains sets of implementation components. Each set comprises one full implementation for the DCO's instances. Just as normal Legion classes could maintain different monolithic executables for different architecture types and performance or security trade-offs, DCC's can maintain different sets of implementation components for the same purposes. For simplicity, we will assume that an ICS maintains implementation components that comprise a single implementation. The anatomy of a DCC is depicted in Figure 2. The meaning of the member functions in the dynamic configurability interface will be explained as appropriate in the sections below.

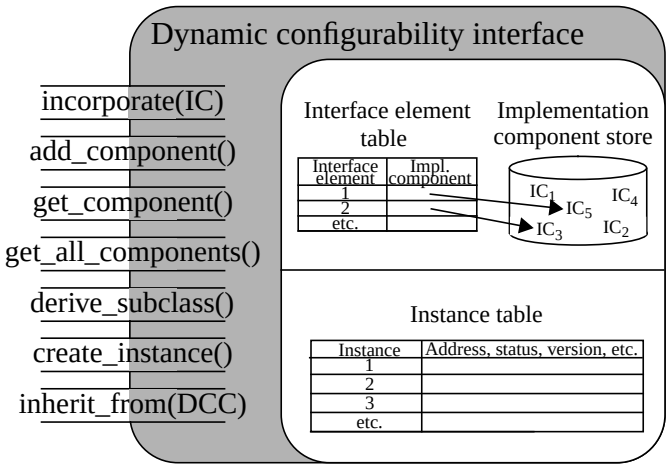

FIGURE 2. The anatomy of a dynamically configurable class

3.2.2 Class creation

A compiler or programmer builds a new DCC by first running a *seed* DCC that initially contains an empty ICS. The compiler or programmer then configures the DCC appropriately by calling the member function named *add_component()* to augment, alter, or shrink the ICS that the DCC maintains. When add_component() is called, the implementation component parameter is added to the ICS. The DCC maintains an interface element table that maps each interface element to the implementation component in the ICS that contains that element's implementation. For each interface element in the implementation component parameter, the DCC determines if the interface element is in the table. If not, it adds the element and remembers that its implementation

5. The term "dynamically configurable class" is *not* intended to imply that the class object itself can necessarily be configured on-the-fly with different interface elements for its own implementation. A DCC is so named because its *instances* are dynamically configurable.

is in the incoming component. If the element is already in the table, its implementation component is set to the new component.

3.2.3 Instantiation

The state and contents of the ICS determine the characteristics of the DCC's instances. The user of a DCC can create a new instance of that DCC by calling its *create_instance()* public member function. DCC class objects will create instances by first running a seed DCO, and then configuring the DCO appropriately by repeatedly calling the incorporate() member function with the appropriate implementation components as parameters, as described in Section 2. This causes the instantiated object to be configured appropriately.

3.2.4 Inheritance

A new subclass DCC can created by deriving from an existing superclass DCC. This is done by calling the *derive_subclass()* member function on the superclass object. In response, the superclass DCC will schedule and execute a subclass DCC that initially maintains no IC's, but that exports the core DCC interface. The superclass DCC will then transfer an appropriate set of IC's to the subclass DCC. This reflects the first step in a typical inheritance process–the subclass inherits the characteristics of its superclass.

Once the subclass is created to reflect exactly the characteristics of its superclass, these characteristics can be changed based on the inheritance semantics that are being implemented. This will typically be the job of the Legion-targeting compiler. To add, remove, or alter interface elements, the compiler creates IC's based on the Legion program that defines the subclass, and sends these IC's to the subclass (as parameters to add_component() calls) to effect the appropriate change. Each compiler can manage the inheritance process appropriately for the semantics of the programming language it supports. For example, C++ uses the "public," "private," and "protected" keywords to determine which member functions get put in the public and private interface of the derived subclass. C++ also contains a "scope" operator that allows a base class's member functions to be accessible through the derived class. To support C++-like semantics, it would be up to the compiler, possibly in concert with the superclass, to give the appropriate IC's to the subclass, and to name the interface elements appropriately so that the right set are accessible according to C++ inheritance rules.

3.2.5 Multiple inheritance

Multiple inheritance allows a subclass to have more than one superclass. With DCC's, the subclass will have one primary superclass, and the inheritance process will begin just as it does with single inheritance; that is, a subclass will be created by its primary superclass and will be initially configured using only the IC's that it inherits from the primary superclass. With multiple inheritance, the extensions and alterations to the initial version of the subclass can come from other superclasses, not just compilers. Thus, to manage multiple inheritance, a Legion-targeting compiler would obtain the appropriate IC's from all superclasses using the *get_component()* DCC member function. It would use these IC's, and possibly some IC's that it creates itself, to configure the subclass according to the inheritance rules of the language the compiler supports, and according to the content of the program it is compiling. A DCC exports the *inherit_from()* member function, which takes the name of another DCC as a parameter, to enable one class to inherit all the components of another class. Inherit_from() is implemented simply by calling *get_all_components()* on the superclass (which returns all of the implementation components), and then adding each component to the local ICS.

4 Illustrative example

This section provides a simple example that is intended to better illustrate the functionality of dynamically configurable objects and classes. The example is not intended to show the full power of dynamic configurability, nor is it designed to make particularly appropriate use of the Legion wide area distributed object system, which we have just begun building. It is also not intended to address all of the complications that dynamic configurability introduces. In fact, the example was chosen for its simplicity and its ability to illustrate in a straightforward manner the concepts described above. The example has been implemented in C++ on top of the new Legion run-time library [7]. The example runs over the small prototype Legion system running at the University of Virginia–hardly wide area, but more than large enough to support the example. Dynamically configurable objects and classes have been implemented over Legion exactly as described in Sections 2 and 3.2. However, we have not yet added compiler support to allow programmers to specify object-oriented operations (e.g. class creation, instantiation, inheritance, etc.) within a programming language. Therefore, we have hand-translated each of these operations specifically for this particular example application. Below, we'll use the acronym HGCC, for "hand

generated compiler code," for the part of our implementation that will typically be carried out by a compiler.

4.1 ArraySorter example

Suppose a programmer were to build a simple Legion object that sorts an integer array. The programmer first describes the object's characteristics by defining its class, which we'll call ArraySorter. Typically (but not necessarily), this will be done within an object-oriented programming language that has a Legion-targeting compiler. The compiler's role is to use the class definition to create a DCC whose instances will be array sorters. To do so, the compiler must first create implementation components that reflect the appropriate characteristics of an ArraySorter, and then must run a seed DCC and configure it with the implementation components.

In our implementation, the programmer directly specifies what the implementation components are by putting each one in a separate file. The interface elements for this application are (1) a *create_array()* function which creates a new array of a specified size within the object, (2) a *randomize()* function which fills the array with random integers, (3) an integer data element named *array_size*, (4) an integer array data element named *int_array*, (5) a *sort()* function which uses bubble sort, and finally (6) a *show()* function which returns the contents of the array. Each interface element is contained in its own implementation component, except for the two data elements, which share a single implementation component. The public functions all need access to the data interface elements; therefore, the access must go through the DIEM. Typically, it will be the job of a compiler to detect references to interface elements and to translate them into references that go through the DIEM. For now, this translation is done by hand; we'll call the resulting code "translated component code." The HGCC uses a regular C++ compiler to compile the translated component code into separate object files, each of which then contain external references to a DIEM. The references eventually resolve to the DIEM in the DCO into which the component is incorporated. The HGCC then creates implementation components by adding appropriate information about internal symbol names and interface element names. For now, the HGCC is hard-coded to know exactly what the internal symbol names are for this particular application, but clearly this is information that a real compiler would have. Next, the HGCC runs a seed DCC, and calls add_component() once for each implementation component. This configures the class to allow its instances to be array sorters.

Once the class is up and running with an appropriate ICS, the class can be instantiated by calling the create_instance() member function on the DCC. This process could be triggered by a Legion-targeting compiler when it compiles a program that includes code to instantiate a Legion class. It could also be triggered by direct calls from other objects. Our implementation includes a tool that uses a function provided by the Legion library to create an instance of a named Legion class object. The instantiation process is carried out exactly as described in Section 2; that is, the class runs a seed DCO and configures it with the implementation components in the ICS by repeatedly calling incorporate() on the DCO instance.

Now suppose that the programmer is unhappy with the performance of his array sorters, and wishes to replace the bubble sort implementation of the sort() function with a quicksort implementation. The first step is to create an implementation component that includes the quicksort function, and that has the same interface element name (e.g. "sort") as the old bubble sort implementation. This implementation component is then sent by the compiler to the ArraySorter DCC, which adds it to its ICS, replacing the old bubble sort implementation. At this point, all future instances of ArraySorter will contain the quicksort implementation, because the quicksort implementation component will get passed to the DCO instances when create_instance() is called. A DCC's *update_instances()* member function can be called to make the current instances reflect the change. To implement update_instances(), the DCC simply calls incorporate() on each of its instances, passing all implementation components that have changed since the instance was created. In response, each DCO instance will replace the bubble sort implementation with the quicksort implementation, and all subsequent calls to the instance's sort() function will be implemented with a quicksort.

The scenario above is admittedly relatively simple–only the implementation (not the interface) of a single public member function is changed, and the change does not affect any state the object may have. Other more complicated changes could (1) remove functions from the public interface, thereby leaving other objects and programs out of date, (2) change the interface to a data or function interface element that other interface elements within the object require, thereby "breaking" those elements, or (3) change the implementation of a data component, which could cause an object to drop its state. Our implementation is general mechanism, and does not yet attempt to address these important and difficult problems. Ideally, a class or compiler would place some restrictions on allowable operations, such as requiring that all versions of a class be succes-

sive supersets of the original version, or ensuring that changes don't adversely affect existing public member functions. We will address these problems with future work.

Getting back to the ArraySorter example, suppose that the programmer wishes to augment the functionality of array sorters by giving them the ability to generate statistics about the contained array, e.g. *average()*, *max()*, *min()*, etc. In our implementation, we put the statistics functions in their own class, called Stats, and have the ArraySorter class inherit from the Stats class. The DCC Stats class is built in much the same way that the ArraySorter is built; that is, the HGCC creates appropriate implementation components, runs a seed DCC, and configures it to be a Stats class. To make the ArraySorter class inherit from the Stats class, ArraySorter's inherit_from() function is called with the name of the Stats class as a parameter. This causes the ArraySorter DCC to get all the implementation components of the Stats class, and to put them in its ICS. Thus, subsequent versions of ArraySorter will automatically contain statistics functions, and the update_instances() function can be used to configure existing instances to contain statistics functions.

5 Summary

In this paper, we have identified and described SPAD class systems, of which Legion is an example. We have described why supporting traditional class roles, including instantiation and inheritance, in SPAD class systems requires new and different implementation solutions from those that are used to support traditional languages and databases. We also introduced and described dynamic configurability of classes and objects, explained how dynamic configurability can be used to support object-oriented languages and systems in Legion, and illustrated the functionality with a simple array sorter example. Our implementation is built in C++ and uses the new Legion run-time library to run over the current Unix-based version of Legion.

References

[1] K. Arnold, J. Gosling, The Java Programming Language Addison-Wesley, 1996.

[2] A. Black, N. Hutchinson, E. Jul, and H. Levy, ``Distribution and Abstract Types in Emerald," University of Washington, TR 85-08-05, August 1985.

[3] A. D. Birrell and B. J. Nelson, Implementing Remote Procedure Calls," *ACM Transactions on Computer Systems*, Volume 2, Number 1, February 1994.

[4] K. Brockschmidt, Inside OLE, Microsoft Press, Reading, Washington, 1995.

[5] H. Cejtin, S. Jagannathan, and R. Kelsey, "Higher-Order Distributed Objects," *ACM Transactions on Programming Languages and Systems*, Volume 17, Number 5, September 1995.

[6] R. Chin and S. Chanson, “Distributed Object-Based Programming Systems,” *ACM Computing Surveys*, Volume 23, Number 1, March 1991.

[7] A. J. Ferrari, M. J. Lewis, C. L. Viles, A. Nguyen-Tuong, A. S. Grimshaw, “Implementation of the Legion Library,” University of Virginia Computer Science Technical Report CS-96-16, November 1996.

[8] A. S. Grimshaw, W. A. Wulf, “Legion—A View from 50,000 Feet,” *Proceedings of the Fifth IEEE Symposium on High Performance Distributed Computing*, IEEE Computer Society Press, Syracuse, NY, August 6-9, 1996.

[9] A. S. Grimshaw, “Easy to Use Object-Oriented Parallel Programming with Mentat,” *IEEE Computer*, May 1993.

[10] A. S. Grimshaw, J. B.Weissman, E. A. West, and E. Loyot, “Meta-systems: An Approach Combining Parallel Processing And Heterogeneous Distributed Computing Systems,” *Journal of Parallel and Distributed Computing*, Volume 21, Number 3, June 1994.

[11] Adele Goldberg, Smalltalk-80: The Language and its Implementation, Addison-Wesley, Reading, Massachusetts, 1983.

[12] M. Lewis and A. S. Grimshaw, “The Core Legion Object Model,” *Proceedings of the Fifth IEEE Symposium on High Performance Distributed Computing*, IEEE Computer Society Press, Syracuse, NY, August 6-9, 1996.

[13] J.R. Nicol, C.T. Wilkes, and F.A. Manola, “Object-Orientation in Heterogeneous Distributed Systems”, *IEEE Computer*, Volume 26, Number 6, June 1993.

[14] Object Design, Inc. ObjectStore: C++ API User Guide Release 4, June 1995.

[15] J. Palsberg, C. Xiao, K. Lieberherr, “Efficient Implementation of Adaptive Software,” *ACM Transactions on Programming Languages and Systems*, Volume 17, Number 2, March 1995.

[16] J. W. Stamos and D. K. Gifford, “Implementing Remote Evaluation,” IEEE Transactions on Software Engineering, Volume 16, Number 7, July 1990.

[17] B. Stroustrup, The C++ Programming Language, 2nd Edition, Addison-Wesley Publishing Company, Reading, Massachusetts, 1991.

[18] M. van Steen, P. Homburg, L. van Doorn, A.S. Tanenbaum, and W. de Jonge. “Towards Object-based Wide Area Distributed Systems”. In L.-F. Carbrera and M. Theimer, (eds.), *Proceedings International Workshop on Object Orientation in Operating Systems*, pp. 224-227, Lund, Sweden, August 1995.