The Basic Object System: Supporting a Spectrum From Prototypes To Hardened Code

Allen H. Dutoit† *ahd@sei.cmu.edu*
Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA

Sean Levy, Douglas Cunningham, Robert Patrick *{snl, dougc, rp2y}@cmu.edu*
Engineering Design Research Center, Carnegie Mellon University, Pittsburgh PA

Abstract

BOS is a prototype-based, object-oriented toolkit aimed at better supporting evolutionary software development. BOS attempts to support a spectrum of activities in one environment—ranging from rapid prototyping to code hardening. Features enabling rapid prototyping include a prototype-based object model, an interpreted language, run-time argument constraints, position and keyword arguments, and a user interface toolkit. BOS also provides features for code hardening such as multi-methods, multiple inheritance, external code wrapping mechanisms, and interfaces to other packages such as database management systems. BOS thus enables the end-to-end programming of software in an integrated and unified environment. BOS has been used to develop several full-size applications which have been evaluated and delivered externally.

1. Introduction

Evolutionary software development entails the rapid development of prototype components and their evolution into hardened components [Budde92] [Krogh96]. We have observed that the development of software that exhibits a long life-cycle, for example software research prototypes which are converted into commercial products, often follow an evolutionary cycle. Research prototypes start out as a proof of concept demo which is then turned into a more reliable prototype for its evaluation by target users and then optimized and refined according to new research directions, commercial ventures, or client requirements. Furthermore, the maintenance of commercial software may also be viewed as a continuation of this cycle; commercial products reach the market at an earlier stage of development, beta testing is done by potential users, and new features are added while the core technology is hardened and ported to a variety of hardware architectures.

Several recent key achievements in software engineering and object-oriented languages are consistent with this perspective [Blaschek84] [Cox91]. First, in software engineering, iterative development processes have become popular as a means to control change and mitigate risks. The spiral model describes an iteration cycle in which prototypes, requirements, design, and implementation are evolved [Boehm88]. The end of each spiral is characterized by a risk assessment phase, which is used to manage the next iteration of the prototype. Second, object-oriented languages (e.g., Smalltalk [Goldberg83] and C++ [Stroustrup91]), and later, object-oriented methods (e.g., OMT [Rumbaugh93], OOSE [Jacobson92]), became popular, partly because of their potential for developing modifiable code. From an object-oriented perspective, a software system is not viewed as an end point but as a substrate which evolves and onto which new features are grafted based on market demand.

With the above perspective of software development in mind, it is highly desirable to build environments which support evolutionary development. Such an environment would support the development of prototypes and their evolution into hardened components. Moreover, it would support both activities

† The views and conclusions contained in this document are solely those of the authors and should not be interpreted as representing official policies, either expressed or implied, of the Software Engineering Institute, Carnegie Mellon University, the U.S. Air Force, the Department of Defense, or the U.S. Government.

simultaneously on the same substrate, as new components are prototyped and integrated into a mature system. It would also accommodate multiple languages and multiple developers, thus maximizing the reuse of knowledge and reducing the need for discarding prototype code. Although environments such as Smalltalk, SELF [Ungar91a], and Tcl [Ousterhout94] address some issues associated with evolutionary development, no single environment has systematically addressed evolutionary development.

In this paper, we report on the design, implementation, and use of the Basic Object System (BOS) [Levy96a], a programming environment we envision as a first step towards supporting evolutionary development systematically. Our approach has been to investigate individual features of other languages and systems for their use during either rapid prototyping or code hardening. We then implemented and integrated those features into BOS and evaluated their synergy through use.

BOS is a dynamic, prototype-based, object-oriented environment and toolkit. It differs from other environments in that it is designed to be open, extensible, and language independent. It is packaged as a C library which enables its use from non-object-oriented environments such as C and from existing compiled object-oriented languages such as C++. Moreover, BOS provides an interface for code generators, enabling the construction of parsers supporting different syntaxes. BOS is packaged as a toolkit which includes an example of such a parser, implementing a simple syntax called stitch [Levy96b], a user interface toolkit based on Tk [Ousterhout94], and several interfaces to database management systems such as Informix, Ingres, Postgres [Stonebraker86], and Illustra.

We have used BOS to successfully develop and evolve several mid-to-large scale software systems, including a graphical configuration tool, a tool for capturing engineering design history, and a distributed information modeling environment.

This paper is structured as follows. Section 2 motivates BOS and states our goals when we initiated its development. Section 3 describes the core BOS programming model. Section 4 discusses issues encountered during its implementation. Section 5 describes selected components from the toolkit (i.e., stitch and user interface objects). Section 6 is a simple example of a program written in BOS. Section 7 describes the use of BOS, its observed strengths, and its drawbacks. Section 8 briefly relates BOS with two influences of this work: SELF and Tcl. Section 9 concludes this paper.

2. BOS design rationale

BOS emerged as the convergence of several research interests and pragmatic constraints present in our research group[1]. The *n*-dim group has been concerned with the study and support of collaborative engineering design in its rich socio-technical context [Subrahmanian93]. This has involved doing detailed studies of information flow between members of design teams and between departments of large organizations [Finger93], participating in collaborative design exercises with other universities [Wilkins89], and developing software tools to study and support collaborative engineering design.

n-dim views software engineering as collaborative engineering design. As a consequence, evolutionary software development is relevant to our research interests. Also, the software prototypes entailed by our research are complex and require a significant enough level of reliability and portability that supporting evolutionary development is essential to our work.

All development environments that we investigated have concentrated on supporting either rapid prototyping or the development of commercial strength software. Few support both activities well.

For example, environments such as Ada [ANSI83], C [Kernighan78], C++, and FORTRAN have been successfully used for building commercial software. These languages were designed as compiled languages, enabling compiler optimizations which produce efficient binaries. Ada and C++ provide strong type-checking, enabling the explicit definition of module interfaces, the early detection of errors, and the mitigation of miscommunication among multiple developers. Unfortunately, none of

1. The *n*-dim group at the Engineering Design Research Center, CMU, ndim-info@ndim.edrc.cmu.edu

these environments are appropriate for rapid prototyping. The compiled nature of their languages entails a long compile-test-debug cycle. The strong type-checking for Ada and C++ forces developers to commit to specific module interfaces early. Moreover, any interface change of a module (e.g., the addition of a parameter to a routine or the addition of a field to a public structure) forces the modification and recompilation of all dependent modules. When the software under development is large, this becomes an error-prone and expensive task. Finally, all of the above languages require programmers to manage memory explicitly, resulting in higher error rates due to memory management errors.

At the other extreme, environments such as Smalltalk, Tcl, CLOS [Gabriel91], and SELF support rapid prototyping well. They provide interpreted languages allowing the modification of code at runtime. They provide weak type systems, which, together with their dynamic nature, allows the deferral of classification decisions until late in the development process. Their syntax is simple and the number of language concepts is relatively small, which enables users to concentrate on design errors, rather than syntactical errors. Finally, most of them provide garbage collection, which relieves programmers from managing memory explicitly. However, all of these environments suffer scale problems; the dynamic nature of these languages often hinders performance. The lack of strong type-checking and the overall increased flexibility make it harder for multiple developers to work on a large project.

Another issue that has limited the use of environments such as Smalltalk or CLOS for commercial development is the difficulty associated with integrating legacy code. Languages such as FORTRAN and C have been used extensively for several decades. As a consequence, the number and quality of off-the-shelf components for these languages is very high and embody enormous investments of time and effort. It is often unreasonable to expect that this software be translated into a newer language solely based on the qualities and advantages of the new language.

The need for a simple object system and development environment emerged from the above considerations. Given that our group had an urgent need and could not embark on a full scale development of a programming environment, we decided to develop an environment which provides or allows for:[2]

- flexible typing and execution of incomplete programs
- strong typing
- interfaces to external systems
- powerful datatypes and generic I/O facilities
- object-oriented components
- swift development cycles
- separation of functional and interactive components

3. BOS programming model

The first design goal of BOS was to provide a flexible model enabling programmers to create objects, modify their structure and behavior, and change their inheritance *without* restarting applications. This enables programmers to prototype and fine-tune components quickly. We accomplished this by adopting and refining the SELF[3] prototype-based object model.

The second design goal was to enable programmers to harden components and reimplement them in C or C++ to address performance issues. In addition, this mechanism allows the wrapping of legacy code. We accomplished this by adapting and improving on the code wrapping features of Tcl.

The final task was to reconcile the conflicts between the two goals. This was accomplished by modifying the features of either or both Tcl and SELF as incorporated into BOS.

3.1. Object model overview

Objects and slots. The object model implemented by BOS is heavily inspired by the SELF prototype-based object model. Most constructs are represented as *objects*; each object is a collection of named *slots*. State and behavior are accessed solely via messages.

2. These requirements are a subset of the requirements for an evolutionary development environment according to R. Budde et. al. ([Budde92], pp. 148-50)
3. In this paper, we refer to SELF 2.0 described in [Ungar91a].

In BOS, a slot has several attributes, including a *name,* a *value,* a *type,* and a *priority.*[4] Slot names and values are defined as in SELF. Unlike SELF, BOS has slot types which constrain the kind of object which can be assigned to the slot (see Section 3.2). Finally, the priority of a slot is an integer which specifies the manner in which the slot should answer message sends. If the priority is zero, the slot is considered a value slot, and will merely return its value upon receiving a message. If the priority is negative, the slot is a method, which evaluates its content upon receiving a message. If the priority is positive, the slot denotes an inheritance relationship between the object in which the slot is defined and the object to which the slot refers. The use of priorities by the message dispatcher is described in Section 3.3.

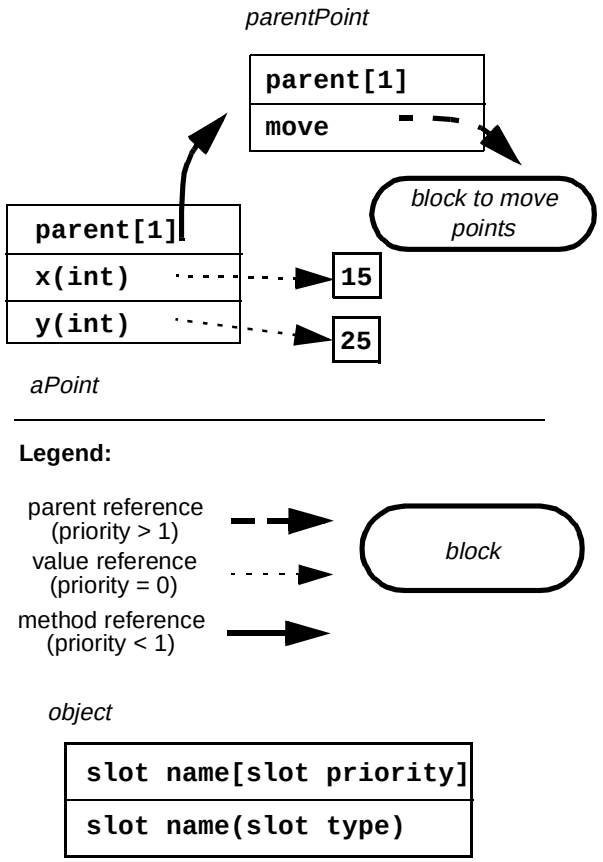

Figure 1. An example of a prototype and its associated parent object

Figure 1 illustrates the three kinds of slots using a point example. A point object, *aPoint,* contains two value slots maintaining its state (i.e., its coordinates). *aPoint* also contains a parent slot referring to *parentPoint* which provides the behavior shared by all points. Finally, the **`move`** method slot in *parentPoint* refers to a block object containing the code to move a point.

Note that, due to the prototype nature of BOS, there are no restrictions on the kind of slots which can be defined on objects. In the example above, the **`move`** method could have been defined directly on *aPoint.* This is useful for defining the behavior of one-of-kind objects (e.g., the behavior specific to the **`true`** object is defined on the **`true`** object).

Blocks and methods. As in SELF and Smalltalk, closures are represented as blocks. Formal arguments are represented as slots in the block object. The slot name corresponds to the formal argument name, its type corresponds to the argument type, and its value corresponds to the argument default value. The code segment of the block is represented as internal state. A block is activated by sending the **`value`** message to the block with zero or more actual arguments. Upon activation of the block, BOS clones the block to create an *activation* object, initializes the activation's slots with the actual arguments, and executes its code segment. When the execution completes, the result of the last expression in the activation is returned as a result and the activation is garbage collected. Expressions may return any number of results (including no results at all). A common method which can return multiple results is the **`stitch`** method (provided by all collection objects such as bags, sets, and vectors) which returns all elements of a collection. Figure 2 illustrates the use of multiple

```
// create three vectors and add elements
// to them
v1: (prototypes vector clone add: 1).
v2: (prototypes vector clone add: 3 4).
v3: (prototypes vector clone add: 7 8).

// add elements to v1 using multiple
// results
v1 add: 2 (v2 stitch) 5 6 (v3 stitch).

// v1 is now <1, 2, 3, 4, 5, 6, 7>
```

Figure 2. Multiple results for vector concatenation

4. After version 2.0, SELF abandoned priorities [RSmith95].

results for concatenating vectors. Note that the usefulness of multiple results is a consequence of their combined use with variadic methods (i.e., in this example, the **`add`** method on vectors may take an arbitrary number of arguments).

BOS blocks differ from SELF blocks in several ways. First, method and argument names are orthogonal; the message selector in BOS only includes the message name. Although BOS uses keyword arguments for resolving ambiguities during message dispatching, arguments may be passed without keywords; that is, arguments can be passed by position only. Considering only slot names for initial message selection enables BOS to differentiate between typing errors, ambiguities, and undefined messages (see Sections 3.3 and 3.4). The second difference is that slots of a block object are classified into two categories: *required* and *optional* arguments. Required argument slots are arguments that must be specified in a message send. Optional argument slots may also be used as local variables in the block. The need for optional arguments was motivated in the domain of user interfaces. From a performance point of view, it is desirable to specify a large number of attributes upon creation of a window or other graphical objects. From a usability point of view, we found it unreasonable to require the user to specify all possible attributes of a window. The existence of optional arguments provides a trade-off by allowing the development of methods with a large number of options with default values.

BOS blocks provide a wrapping mechanism similar to Tcl commands. The code segment of a BOS block is represented either by executable byte code or a C function. Unlike Tcl and SELF, BOS supports argument constraint checking (see Section 3.2) and provides the programmer with stronger assumptions about the actual arguments passed to the block. This significantly reduces the amount of code the programmer must write to check the consistency of arguments. Also, unlike Tcl which represents everything as strings, BOS represents objects and values as C structures (see Section 4.1). This reduces the overhead entailed by converting arguments to and from wrapped code and across blocks. Figure 3 depicts a method referring to a block with two arguments.

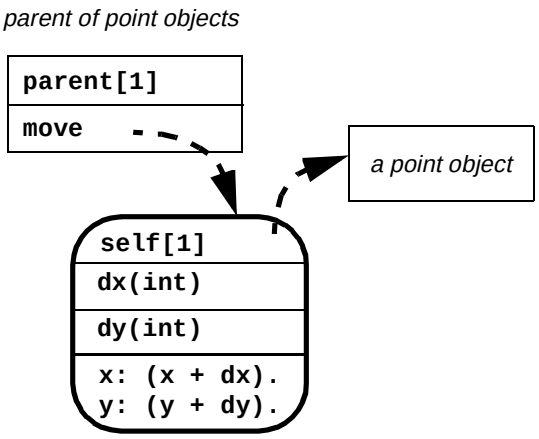

Figure 3. An example of a block object

Note that methods and blocks in BOS are orthogonal concepts. A method slot can refer to a non-block object, in which case it behaves as a value slot. Similarly, a value slot can refer to a block, in which case it returns the block (without evaluating it) when answering a message. Table 1 summarizes the relationship between slot behavior, slot type, and slot content. This clear distinction between methods and blocks leads to a simple model and a robust implementation. Methods implementing control structures (e.g., **`ifTrue`**, **`do`**, and **`while`**) are common uses of value slots referring to blocks. For example, for **`do`** methods, the block implementing the iteration code is passed as an argument and then stored in a value slot of the activation.

Table 1: Slot behavior

Slot type	Content	Behavior
value or parent slot	block	return content
	non-block	
method slot	block	evaluate block
	non-block	send value message to content

Inheritance. As in SELF, inheritance is represented as parent slots with priorities. Objects inherit state and behavior from their ancestors when answering messages (see Section 3.3). In addition, all objects, by definition, inherit from **`traits object`**, the BOS root object which provides methods applicable

to all objects. As in SELF, all BOS objects inherit from themselves. The rationale for this design decision is discussed in the following section, describing the BOS type system.

3.2. Slot and argument constraints

Slot types. The slot type attribute is used by BOS to enforce constraints on the kinds of objects which can be referred to by a given slot. To be consistent with the prototype-based nature of BOS, types are represented as objects. Moreover, any object (including immutable objects such as "1") can be used as a type. In BOS, an object **A** is said to conform to a type **T** if **A** inherits from **T**. Given that all objects inherit from **traits object**, the programmer may delay any classification decisions by initially setting slot types to **traits object**. Also, the object **nil** conforms to all types. This allows the programmer to use **nil** as a default value for initializing optional arguments and slot values in prototype objects. Finally, self-inheritance (e.g. given that **A** inherits from **A**, **A** conforms to type **A**) renders the type system consistent with the prototype-based nature of BOS and allows the programmer to use slot types to constrain arguments to be one-of-a-kind objects. This symmetry between the type system and prototype object model makes BOS distinctive. Figure 4 illustrates the use of this feature with a simple implementation of the **and** and **or** messages for **true** and **false**.

```
true _DefineSlots: (|
    and = [aBool(true)||| true].
    and = [aBool(false)||| false].
    or = [aBool(bool)||| true].
|).

false _DefineSlots: (|
    and = [aBool(bool)||| false].
    or = [aBool(true)||| true].
    or = [aBool(false)||| false].
|).
```

Figure 4. Using BOS types for one-of-a-kind objects.

While slot types in BOS are intuitively similar to variable types in C and C++, several critical differences should be noted. C++ provides static type-checking, while BOS checks argument constraints at run-time. This is desirable during prototyping when types and inheritance structures are still changing. This also enables consistency checking of actual arguments against formals even in the presence of polymorphism.

Note that the BOS type system is biased towards supporting prototyping rather than code hardening by checking argument constraints at run-time. On the one hand, this trade-off allows code to be modified and reloaded at run-time. On the other hand, type conformance in BOS requires the execution of the application under test, while C++ type-checking guarantees type conformance before the application is ever executed.

Slot type signatures. The type signature of a method slot containing a block is defined as the sequence of slot types of its required arguments. For example, if a method **M** takes three required arguments, **A**, **B**, and **C**, of slot type $\mathbf{T_a}$, $\mathbf{T_b}$, and $\mathbf{T_c}$, the type signature of **M** is said to be $\mathbf{<T_a, T_b, T_c>}$. The type signature of a value slot is the empty sequence—i.e., a value slot is treated as a method taking one optional argument (see Figure 5). Slot type signatures are used by BOS during message dispatching to match methods by using actual arguments.

```
// 3 required
// Type signature: <int, float, bool>
m = [a(int).b(float).c(bool)||| ...]

// 2 required & 1 optional
// Type signature: <int, float>
m = [a(int). b(float) | c(bool)|| ...]

// value slot
// Type signature: <>
m = true.
```

Figure 5. Examples of type signatures.

3.3. Message dispatching

BOS is an asymmetric, multiple-dispatching message system. It is a multiple-dispatching system in the sense that receiver and actual arguments are used to select methods. It is asymmetric in the sense that multi-methods are located in the inheritance graph of the receiver (unlike CLOS, which provides symmetric multi-methods).

From the programmer's point of view, message

dispatching occurs for every message send. In other words, any change in the state of the system is visible as early as the next message send. For example, the programmer may change the inheritance hierarchy, change the type signature of a method, or add or remove slots between any two message sends.

Message dispatching in BOS comprises two steps: message lookup and disambiguation. Message lookup selects candidate methods by matching the message selector against slot names. Then, disambiguation filters out any candidate method whose formal arguments are incompatible with the actuals.

Message lookup. In the presence of a single inheritance structure (i.e., when every object in the system has exactly one parent slot), the BOS message lookup algorithm is identical to the one in SELF. The lookup starts with the receiver to which the message was directed. If the receiver does not have a slot matching the name of the message, the lookup continues with the parent of the receiver. If the parent does not define a matching slot, the lookup continues with the parent of the parent, recursively. The lookup returns when either one or more matching slots are found or the inheritance graph is unsuccessfully traversed. If the lookup is unsuccessful, the **`messageNotUnderstood`** error is raised.

Multiple inheritance. In the presence of multiple parent slots within an object, the lookup algorithm uses parent priorities to direct the search. Given an object with multiple parents, only parents with the highest priority are initially searched. If matching slots are found, the lookup returns without examining parents of lower priority. If the search is unsuccessful, the search is repeated with parents of the next highest priority. If all parents are searched unsuccessfully, the lookup raises the **`messageNotUnderstood`** error. In other words, priorities are used to impose a lexical ordering on all possible paths from the receiver to matching slots. In this manner, the lookup algorithm is simply reduced to a depth first, branch and bound search algorithm.

The semantics of priorities are similar to that of SELF 2.0. BOS provides priorities as a mechanism for the programmer to resolve ambiguities due to multiple inheritance. Note that parent slot priorities need not be unique within an object and thus provide only a partial ordering of the objects. Setting equal parent priorities enables the programmer to use type signatures for selecting messages instead of parent priorities.

Multi-methods. BOS allows multiple slots in the same object to have the same name, as long as their slot type signatures are unique. Consequently, the message lookup can return multiple candidate methods. Then, the actual arguments are used to filter candidate methods. First, the number of required arguments is used to filter out any methods whose number of required arguments is greater than the number of actual arguments. Then, any actual arguments passed by keyword are used to filter out methods whose formal argument names are incompatible. Finally, the actual arguments are checked against the formal argument slot types for type conformance; any methods whose type signature does not conform to the actual arguments are discarded. If this filtering step discards all candidate methods, BOS raises the **`argumentMismatch`** error. If more than one method is left, the **`ambiguousMessage`** error is raised. Otherwise (i.e., exactly one method is left), the message is bound and the message sending algorithm proceeds to the argument binding step described in Section 3.4. Note that the consistency of actual and formal arguments is checked even when the message lookup returns a single method. Another example of multi-methods is presented in Figure 4.

Note that BOS resolves multi-methods at *run-time* as opposed to C++ which provides static overloading. Although the semantics of BOS multi-methods and C++ overloaded methods are different, BOS allows a programmer to wrap overloaded C++ methods. Also, note that there still are many open issues related to the use and implementation of multi-methods in the context of object-oriented languages. Solutions to these issues have been recently proposed in Cecil [Chambers95].

Message resend and delegation. BOS allows the programmer to specify the first object to consider during the message lookup. In the common case, the message lookup starts with the receiver of the message (i.e., **`self`**). In the simplest form of a message

resend, the lookup starts with the parents of the object in which the current method is defined. For example, in Figure 6, assume *aPoint* received the

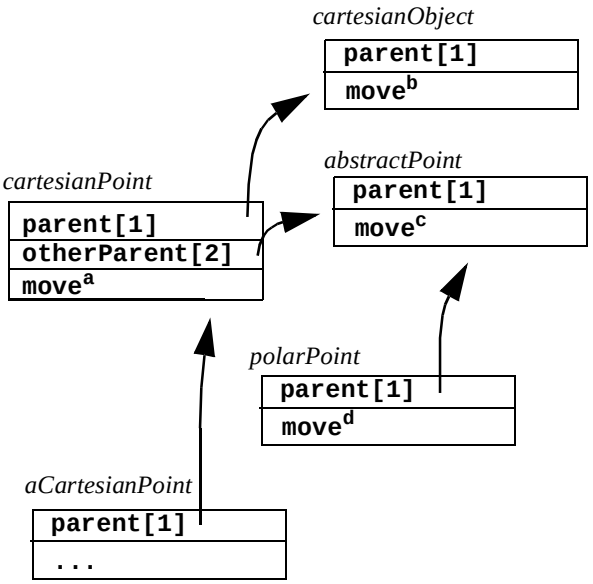

Figure 6. Examples of delegation

move message and **move**^{a} method in *cartesianPoint* is selected. If **move**^{a} simply resends the **move** message, the lookup starts with the highest priority parent (i.e., labeled **parent** in the figure) and the **move**^{b} method in *cartesianObject* is selected. A similar behavior is provided by **super** in Smalltalk.

In the presence of multiple inheritance, this behavior is not always desired. To solve this problem, BOS allows the programmer to specify the parent slot to follow. In the previous example, if the **move**^{a} method in *cartesianPoint* resends the **move** message and specifies the parent slot **otherParent** as a starting point, the **move**^{c} method in *abstractPoint* is selected.

Finally, the programmer may want to resend a message to an object which is not a direct parent of the current method holder, such as an ancestor or a sibling. Using the previous example, the **move**^{a} method may resend the **move** message and specify *polarPoint* as a starting point. In that case, the **move**^{d} method is selected.

Note that message resending in BOS is only a mechanism for explicitly specifying the starting point of the message lookup. Message resending is independent of the current state of the receiver. For example, if the **parent** slot of *aPoint* is modified after the selection of **move**^{a} but prior to any of the three resends describe previously, the same behavior would have been observed.

Note also that the semantics of BOS resend is much simpler than that of SELF 2.0. In case the **move**^{b} method resends the **move** message, the BOS lookup never considers the **move**^{c} method as a candidate, that is, the message lookup never backtracks from the current method holder. However, we have limited experience in developing applications which use multiple inheritance extensively. It is therefore possible that the simultaneous use of slot priorities, multi-methods, and multiple inheritance in BOS leads to undesirable interactions such as those observed in SELF 2.0 [RSmith95].

3.4. Argument binding

Position arguments. BOS supports position arguments, given that their implementation leads to an efficient argument passing algorithm. When actual arguments are not tagged with keywords, actual arguments are bound to formal arguments in the same order they are passed. Since the number of actual arguments and their type have already been checked during the message dispatching step, actual arguments are simply used to initialize the slots of the activation being bound. If the number of actual arguments is less than the total number of slots in the activation (i.e., if some of the optional arguments have not been passed), these last slots maintain their original value (i.e., the default value).

Keyword arguments. Given that their use is more frequent during rapid prototyping and user interface construction, BOS also supports keyword arguments. Here, each actual argument is tagged by a keyword, which is used for matching against the name of the formal arguments. The actual arguments do not have to be in the same order as the formal arguments. The optional arguments do not have to be specified. If formal arguments are specified more than once, or if a required argument is not specified, BOS raises an **argumentMismatch** error, as described previously.

Mixed arguments. BOS allows position and keyword arguments to be mixed in a message send. We observed that this ability is useful during the development of user interfaces, in which methods with a large number of optional arguments are frequent. This enables the programmer to pass the required arguments by position (for efficiency) and the few optionals to be specified by keyword (for brevity). In this case, position arguments are bound first, then keyword arguments are then, as explained in the previous paragraph.

Variadic methods. An alternate way to handle methods which can take a large number of optional arguments is the use of variadic methods. Given a method `M`, if the last required argument is of type `traits rest`, all remaining arguments are collected into a `rest` object which is bound to that argument. A `rest` object is a collection which contains a vector part and a table part. Unbound position arguments are added to the vector part in the order they were passed, while any unbound keyword arguments are added to the table part using the argument keyword as a table key.

Argument constraints. As previously described, activations are represented with block objects and arguments are represented as slots in blocks. From the programmer's point of view, the actual arguments are checked for type conformance with formal arguments when the block is activated. However, checking argument constraints is not necessary at this point, given that the message dispatch algorithm guarantees that the formal and actual arguments are consistent by the time a method block is activated. Repeating the constraints check is unnecessary.

4. Implementation issues

In this section, we describe some of the major issues we encountered while implementing BOS and satisfying the requirements described in Section 2. Since the number of such issues is too large to allow discussion here, we focus on three: the BOS architecture, external code wrapping, and garbage collection.

4.1. Architecture

Our requirement of an evolutionary development environment is that it accommodates legacy code, either interpreted or compiled, written in different languages. Also, the mechanisms developed for wrapping existing code can be used during hardening to reimplement performance critical components.

Tcl accommodates compiled code by providing wrapping mechanisms. SELF accommodates Smalltalk by providing a translator. Our approach was to package BOS as a C library, rendering it independent of any syntax (unlike Tcl). Consequently, BOS can be used even in purely compiled languages without the overhead of an unneeded parser. Instead, primitives provided by BOS are available as a small set of C structures and functions. The code segment of blocks is either a C function or encoded as a stream of virtual machine instructions, similar to Smalltalk byte code. The instruction set of the BOS virtual machine is a simple stack machine with less than 15 instructions for pushing objects on and popping them from the run-time stack, sending messages, and raising exceptions. Object creation and modification, block activation, control structures, and arithmetic operations are all currently implemented as message sends.

Drawing on lessons learned from using Tcl, we also provided a large number of convenience functions and macros to support common cases encountered when writing C blocks. A parser for stitch (see Section 5.1), an object-oriented language inspired by the SELF syntax, is also provided in the form of a separate library, though its use is not required by BOS. Moreover, we envision the development of parsers on top of BOS for implementing other languages, such as SELF and Smalltalk, which would address the issue of accommodating interpreted legacy code.

To facilitate wrapping legacy code, the memory layout of BOS objects respects the same rules as C structures. Except for space overhead at the beginning of the object, there is a one to one mapping between BOS objects and C structures containing primitive types. Figure 7 contains an example of such a mapping. The top part of the figure shows a BOS

object referring to a character, a double, and a string (all of which are considered objects). The bottom part of the figure shows the equivalent C structure. Note that BOS strings are not equivalent to C strings.

This mapping reduces the need for converting across representations and copying values when passing arguments. It also allows an object to be cast to a structure in methods implemented in C, thus enabling the programmer to bypass the message sending mechanism when accessing state. Although this practice introduces an explicit distinction between state and behavior (which is normally hidden in BOS), we found it critical near the end of the code-hardening cycle when run-time speed becomes a much more important issue than modularity. This also facilitates the wrapping of existing components by reducing the amount of glue code required and by avoiding data duplication. The following section addresses in more detail the issues related to wrapping legacy code.

```
// a sample BOS object
   (|
      c (traits char).
      d (traits double).
      s (traits string).
   |)

// its corresponding C structure
struct {

   /* per-object overhead */
   _BOS_HEADER_;

   /* slots */
   char c;
   double d;

   /* s is a handle (sizeof(pointer)) */
   Bos_Object s;
};
```

Figure 7. A BOS object (using the stitch notation) and equivalent C structure

4.2. External code wrapping

The mechanisms provided by BOS to wrap compiled legacy code are similar in concept to those provided by Tcl. We addressed two drawbacks of Tcl: data representation and the amount of glue code needed (see Section 8.1).

Representing objects as C structures partially addresses the drawback of Tcl's representation of values as character strings. However, our approach assumes that the legacy code does not access or modify memory space in front of the structure (which is managed by BOS). When wrapping around C++ classes (which exhibit this problem), the programmer addresses this issue by modifying the inheritance hierarchy such that wrapped classes all inherit from the C++ `Wrapper` class provided by BOS. Finally, if the modification of the inheritance hierarchy is not possible (e.g., the classes to be wrapped are part of a commercial product), wrapper objects may be defined as a structure containing a single pointer to the C++ object being wrapped. BOS provides the C programmer with primitives to hide and manage these pointers and requires that any access to the BOS object are done through methods. Although this last approach requires a slightly larger amount of glue code (i.e., one method per exported C++ method), we observed that the wrapping overhead is still smaller than the amount of glue code required by Tcl. Tcl requires the programmer to convert wrapped data from a string representation to primitive types and to explicitly check the argument types, which is not the case in BOS (see Section 8.1).

Finally, the last difficult issue we encountered when wrapping legacy code was the co-existence of code requiring the explicit management of memory (as in C and C++) with garbage collection provided by BOS. This is the subject of the next section.

4.3. Garbage collection

Explicit memory management (i.e., allocation, reallocation, and freeing of memory chunks) provides the programmer with the ability to optimize the application for speed and space. However, explicit memory management also introduces the potential for fatal errors. Garbage collection addresses memory management by freeing any structures that are not referenced. However, in addition to significant speed and space overhead, garbage collection introduces a variance in response time, thus frustrating the user with unpredictable performance. In an evolutionary development environment, garbage collection should be provided such that its overhead is

spread across run-time in a predictable manner, and such that it can be explicitly controlled by the programmer (i.e., the programmer should be able to tune garbage collection to trade-off space with speed). However, the programmer should not have to be burdened with detailed knowledge of the garbage collector. Moreover, garbage collection should co-exist with any other essential features provided by the environment (e.g., legacy code wrapping).

Memory management in BOS (which is handled by the **`clone`** method and the garbage collector) is implemented using standard C memory management primitives only. Object structures are allocated and freed individually on a per need basis. We avoided using operating system specific memory management primitives in order to enhance portability and ensure compatibility with legacy components. The BOS garbage collector does not compact memory. Instead, additional memory fragmentation is controlled by caching common memory structures (i.e., by pooling objects).

The garbage collector is implemented as an incremental mark and sweep algorithm [Wilson94] which is packaged in a separate run-time thread. The amount of work done by the garbage collector is managed by a feedback control algorithm which readjusts the garbage collector's parameters based on the process size and the rate of allocation. Moreover, the programmer has direct control over the algorithm's parameters, which enables fine tuning the garbage collector to trade-off between speed and space. However, the feedback algorithm performs well in most situations and thus, makes such an intervention a rare occurrence.

The interaction with legacy code is handled by way of callbacks. When the programmer wraps legacy code, he may specify three callbacks for allocating, scanning, and freeing memory chunks unknown to BOS. These callbacks are invoked when the wrapped object is cloned, marked, or swept, respectively.

5. Tools

BOS comes with several tools for the construction of object-oriented programs. Two of them are described in this section: the stitch language, which supports the prototyping phase of software development, and the graphical user interface toolkit, which allows users to easily create interfaces by cloning interface objects.

5.1. stitch

The stitch programming language began as an implementation of the SELF language on top of BOS. However, as the implementation of stitch proceeded it became apparent that there were concepts of BOS that were not supported by the SELF syntax. As a result, the stitch syntax was modified such that it could use all of the features of BOS including:

- slot types
- position and keyword argument passing
- optional arguments
- variadic methods
- multiple results

stitch provides syntactic sugar for creating blocks and objects in the form of an object and a block constructor, respectively. Figure 8 shows two examples of stitch syntax. The first example is an object constructor showing the creation of an object from **`prototypes object`** with a slot called **`a`** of type **`traits number`** with a priority **`0`** and a default value of **`5`**. The second example is a block constructor which shows the creation of a block containing the required arguments **`a`** (typed as **`traits number`**) and **`b`** (typed as **`traits object`** by default); a variadic argument collector, **`args`**; and a local variable, **`c`**.

```
// Example 1: A stitch object
(prototypes object|
   a(traits number)[0] = 5.
|).

// Example 2: A stitch block
[
   a(traits number).
   b
|
   args(traits rest).
   c
||
      ...
].
```

Figure 8. stitch syntax examples

5.2. BOSTk

Tk, as provided in the standard Tcl/Tk release, consisted of two major components:

- A replacement for the X toolkit intrinsics which sought to work around a number of fundamental problems with X windows so as to hide them from the Tcl/Tk programmer;
- A set of user interface “widgets” (components) which more or less mimicked the Motif graphical style, but which could be used in a much more flexible and immediate way via Tcl.

BOS provides a user interface toolkit based on Tk; however, instead of Tcl, this version of Tk (BOSTk) is based on BOS. All widgets are represented as BOS objects, and the normal notions of inheritance, refinement, and so on are available to the programmer. In addition, BosTk objects are easily created and manipulated from stitch thus facilitating the rapid prototyping of interfaces. For example, Figure 9 shows how a dialog box can be prototyped in just a few lines of stitch. The stitch code shown was simply entered at the stitch command line to create the dialog box shown.

```
world _DefineSlots: (|
    dialog. message. okButton
|).

dialog: (tk prototypes toplevel clone:
    (tk top)).

message: (tk prototypes label clone:
    dialog text:”Error!” relief:#ridge).

message packAppend: side:#top
    fill:#both expand:true.

okButton: (tk prototypes button clone:
    dialog text:”OK” relief:#groove
    block: [|d=dialog|| d destroy ]).

okButton packAppend: side:#top padY:20.

dialog title: “Dialog”.

dialog map.
```

Figure 9. BOSTk Dialog Box Example

6. Point Example

The following example incrementally demonstrates several features of BOS. First, it shows how to create a **`point`** object in stitch. Then, a user interface is constructed to display the **`point`**. The example demonstrates that the **`point`** object can be reimplemented in C to gain speed. Then the **`point`** object is implemented in C++ to demonstrate how existing code can be wrapped, or used, by BOS. Finally, the example presents how the brokering mechanism provided with BOS can be used to cleanly separate the implementation of the user interface displaying the **`point`** from the actual **`point`** itself.

Point Object in stitch. Figure 10 represents the creation of a **`point`** object in stitch. The **`point`** object has two slots, **`x`** and **`y`**, both typed as **`traits integer`**. In addition, a **`move`** method is defined on **`traits point`** which resets the coordinates of the **`point`**. To use the point object, the programmer would clone the prototypical point and interact with it via messages.

```
defineProtoAndTrait: #object #point.

prototypes point _DefineSlots: (|
 x (traits integer).
 y (traits integer).
|).

traits point _DefineSlots: (|
  // Move the point to new coordinates
  move =
    [
      dx (traits integer).
      dy (traits integer).
    |
    ||
      x: (x + dx).
      y: (y + dy).
      self. // Return the point
    ].
|).
```

Figure 10. stitch point object

User Interface to Point Object. After creating the **`point`** object we can create an interface in stitch by

cloning BOSTk objects. The resulting window interface, called the **pointDisplay** object, is shown in Figure 11. The interface contains two entry lines for the x and y coordinates and a canvas which contains a displayed point. The displayed point is a representation of the **point** because it is merely how the user interface displays the **point** object; it is not the **point** object itself.

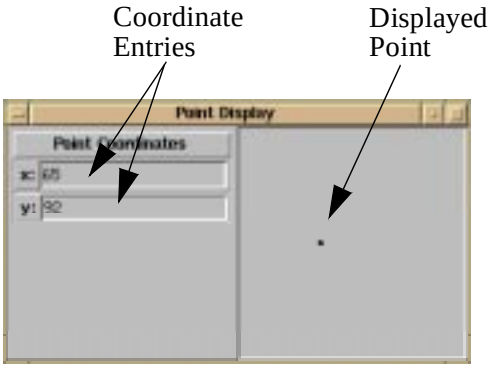

Figure 11. Point Display

Now we need to interact with the **point** object from the interface. Figure 12 shows the **canvasMove** method which implements a mechanism for interaction. The **canvasMove** method is invoked when the user drags the displayed point on the canvas: it retrieves the coordinates from the X event of where the user moved the mouse to, sends the **move** message to the **point** object, and sends itself the **redisplay** message. In addition, the interface has an **entryMove** method which moves the **point** in a similar manner when the user enters new coordinates in the entry lines.

The user can drag the displayed point around or enter new coordinates and the result is that the **point** object is sent the **move** message with its new coordinates and then the coordinates are updated and the displayed point is redrawn.

Point Object in C. When an object definition becomes stable, it can be partially or completely reimplemented in C for efficiency. Figure 13 demonstrates how the **point** object definition and **move** method would be implemented in C. It is important to note that this does not preclude the addition of new stitch methods to the **point** object (now in C).

Point Object in C++. To extend this example further, assume that the **point** object is implemented in C++. Figure 14 shows the C++ code necessary to implement a **Point** class.

In order to access the C++ **Point** class, a BOS **point** object still needs to be implemented, but this time, instead of containing **x** and **y** slots the BOS

```
/* Method to move point from binding */
canvasMove =
  [
    e (tk traits event)
  |
    mx. my. px. py
  ||
    // Position of mouse
    mx: (displayCanvas canvasX: (e x)).
    my: (displayCanvas canvasY: (e y)).
    // Position of display point
    px: (uiPoint xc).
    py: (uiPoint yc).
    // Move point object by difference
    point move: (mx - px) (my - py).
    // Redisplay coordinates and point
    redisplay
  ].
```

Figure 12. CanvasMove method in stitch

```
/* BOS Point Object Definition */
typedef struct {
  /* per-object overhead */
  _BOS_HEADER_;
  /* slots */
  int x;
  int y;
} BosPoint;


/* Bos point move method */
BosDefineCMethod(BosPointMoveCM)
{
  Bos_ReturnCode code = BOS_OK;
  BosPoint **point;
  int dx, dy;

  point = self->v_object;
  dx = msgv[0].ma_value.v_int;
  dy = msgv[1].ma_value.v_int;
  (*point)->x = (*point)->x + dx;
  (*point)->y = (*point)->y + dy;
  BosReturnSelf();

  return BOS_OK;
}
```

Figure 13. C point object

```
class Point {
  int x;
  int y;
 public:
  void move(int, int);
}

void Point::move(int dx, int dy)
{
  x = x + dx;
  y = y + dy;
}
```

Figure 14. C++ point object

point has an internal slot containing a pointer to the actual C++ **point**. In addition, the BOS **point** must implement a C method for each C++ method it needs to access. For example, Figure 15 shows the wrapper code for the C++ **move** method. The wrapper code defines a BOS C method called **BosPointMoveCM** which translates the arguments and calls the C++ **move** method. Now, the programmer can move the **point** by sending the **move** message to the BOS **point** object which in turn invokes the **move** method on the C++ **point** object.

```
/* Bos wrapper for point move method */
BosDefineCMethod(BosPointMoveCM)
{
  Bos_ReturnCode code = BOS_OK;
  BosPoint **point;
  int dx, dy;

  point = self->v_object;
  dx = msgv[0].ma_value.v_int;
  dy = msgv[1].ma_value.v_int;
  (*point)->p->move(dx, dy);
  BosReturnSelf();

  return BOS_OK;
}
```

Figure 15. BOS/C++ wrapper

In this way, existing C++ objects can be accessed from BOS (or stitch) through message sends. For this example, this means that the **point** object defined by the stitch code in Figure 10 can be replaced by an equivalent C or C++ object. Since the object is being accessed by message sends in both cases, none of the calling code (Figure 12) needs to be modified. Figure 16 depicts how the code is executed.

The Broker. The broker, an object provided with

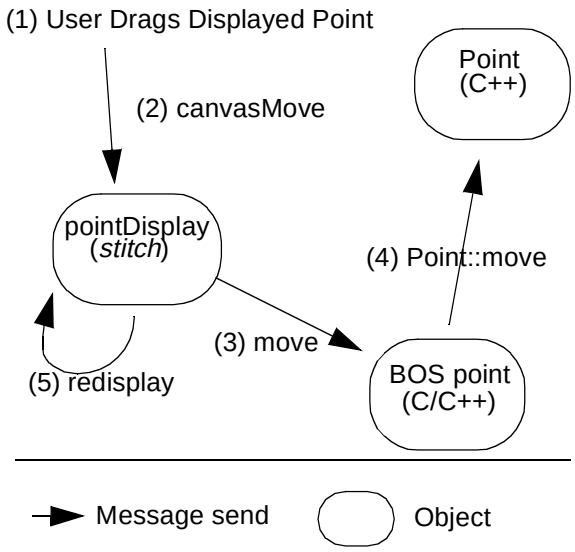

Figure 16. User moving point

BOS, is a tool for coordinating changes to objects. In this example, we are interested in the user interface being updated when the **point** object is sent the **move** message. To accomplish this, two things must happen. First, the **pointDisplay** object registers a callback with the broker to be invoked when the **moved** event is generated by the **point** object. Second, the BOS **point** object generates the **moved** event each time after it invokes the C++ **move** method. This way, any time the **point** is moved, the user interface and any other object registering an interest are notified. Figure 17 depicts how the code is executed in this scenario. The major benefit of structuring the code in this manner is that the user interface will be redisplayed properly no matter how the **point** is moved (e.g., the **move** message could be sent to the **point** from the stitch command line).

Example Discussion. This example demonstrates several features of BOS and how they interact. From the example, we can see how pieces of code from different languages can operate together through the BOS message sending mechanism—any of the components shown could have been implemented in stitch, C, or C++. In addition, the path from prototyping to code hardening becomes clear. Objects can be prototyped in stitch and then moved to C as they become stable. Furthermore, this does not have to be done all at once—that is, an object can have both

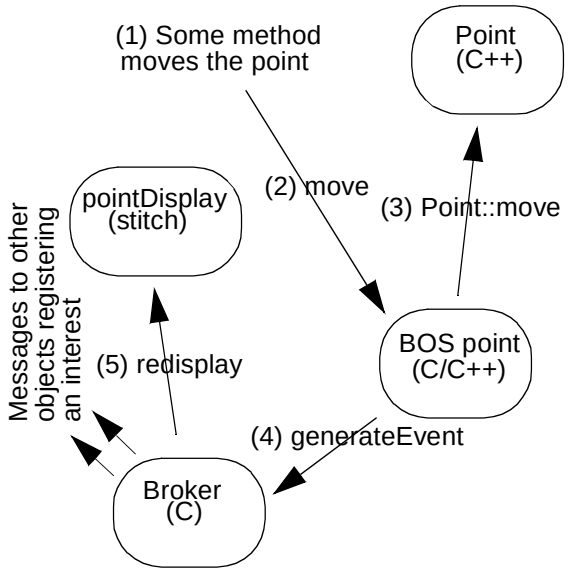

Figure 17. User moving point with broker

stitch and C methods implemented on it. Finally, the brokering mechanism demonstrates how the user interface of an application and the domain of the application can be cleanly separated. This further supports evolution towards hardened code because changes to the user interface can be made independently of the domain. In other words, the domain can become stable even while the user interface is experiencing a great degree of flux.

7. Evaluation

In this section we give anecdotal accounts of the development and usage of BOS.

BOS development. BOS has been under development since early 1991. The first prototype of BOS was implemented in Tcl. After assessing the usefulness of the first prototype, a second prototype was reimplemented in C in 1992. Since then, BOS has undergone several major revisions (e.g., the message-sending mechanism has been refined several times, multi-threaded support has been added, and the garbage collection mechanism has been improved).

Currently, the BOS library is about 60,000 lines of C. It is packaged as a C library which implements the object model, the message sending mechanism, multi-threaded support, and garbage collection. A parser (stitch), a user interface toolkit (BOSTk), an event-broker, and several interfaces to database management systems are included as components in the toolkit.

Applications. One of the first applications to be built with BOS was AutoGraph, an object-oriented application framework for configuring train control software. The development of AutoGraph, characterized by two teams collaborating on different parts of the system, required application domain objects to be implemented in C++ and user interface objects to be implemented in stitch. To accomplish this, BOS objects were implemented to wrap around C++ domain classes which allowed all of the functionality of the C++ code to be accessible from BOS. This meant that the user interface of AutoGraph could be prototyped entirely in stitch while the domain objects were developed in C++.

ACES was a large application built with stitch for recording engineering design history. ACES, like AutoGraph, used the BOS user interface toolkit extensively and made use of the encapsulation techniques of BOS to integrate an off-the-shelf RTF parser into ACES. The RTF parser was used to create BOS objects from documents.

While BOS was used to implement several applications, the primary reason for its development was the implementation of *n*-dim, an information modeling environment for collaborative engineering design. The development of the current version of *n*-dim began concurrently with the C implementation of BOS. BOS has undergone several major revisions since then, most of which were driven by the requirements of *n*-dim. Initially, *n*-dim was essentially a modeling engine without a user interface or storage system. As time went on, both a user interface and a storage system were added. The user interface was implemented using BOSTk. The storage system was implemented using the BOS C API to create an object interface to the Postgres database management system. The simplicity and openness of BOS facilitated this type of prototyping and evolution. Experiences similar to ours have been reported with NewtonScript [WSmith95]. Several features of BOS were used more as *n*-dim matured. For example, type signatures were used more extensively as code

became stable, multi-methods were used to clean up and make selected methods more efficient, methods which were stable or which were considered performance bottlenecks were moved into C, and multiple inheritance was used to simplify the *n*-dim object hierarchy. Currently, *n*-dim consists of about 35,000 lines of stitch code and 10,000 lines of C code.

To date, nine programmers (including the two designers of the current BOS prototype) have used the BOS toolkit to develop and deliver applications.

Shortcomings. Despite these successes with BOS, we have encountered several limitations, the foremost being performance. Although BOS performs well enough to create usable applications, its performance does not yet compete with many other object-oriented programming languages such as SELF or C++. On the other hand, given that SELF was able to implement such an efficient run-time compiler, it stands to reason that such optimizations could be implemented for BOS.

The simple mechanism for argument constraint checking in BOS (i.e., strict inheritance) can cause confusion since the same object can serve as both an instance and a type simultaneously. We do not see this directly as a problem, but it can be confusing for programmers new to BOS.

Another point of confusion stems from being able to use mixed arguments (i.e., both position and keyword arguments) in a message send. Since position arguments are bound first and keyword arguments are bound second, it is not intuitive which combination of the two will result in a successful binding. The following piece of code demonstrates this:

```
[a. b ||| (a + b) print] value: a:5 6.
```

In this example, BOS binds the position argument (**6**) first, resulting in it being bound to formal argument **a** of the block. Next, BOS attempts to bind the keyword argument (**a:5**), but fails and generates an error since it has already bound **a**. On the other hand, any other combination of arguments (i.e., **a:5 b:6, 5 b:6**, or **5 6**) will work as expected. Essentially, this shortcoming is the result of a trade-off between performance and accepting the occasional odd case.

Finally, interactions between slot priorities, multi-methods, and multiple inheritance have not been thoroughly investigated.

8. Related work: Tcl and SELF

When investigating programming environments for supporting evolutionary development, we particularly focused on Tcl and SELF. In this section, we describe briefly the strengths and weaknesses of these two environments with respect to evolution.

8.1. Tcl

Tcl is a procedural language which is primarily based on other scripting languages (such as csh). It was initially provided as an embeddable library which could be included in any application which required a scripting language. Later, with the emergence of Tk, Tcl became an environment enabling rapid prototyping of user interfaces.

Strengths. Tcl is based on a very small set of concepts. It is easy to learn and leads to compact programs. Also, Tcl allows the modification of code without requiring restarting the application, thus shortening the modify-test-debug cycle. Development times of up to an order of magnitude shorter than developments with C have been reported [Ousterhout94]. Given that Tcl was designed as an embeddable scripting language, it provides mechanisms for wrapping compiled code, thus enabling its use as an integration language.

Weaknesses. Tcl represents all data as strings. Although this simplifies the implementation and increases the portability of Tcl interpreters, it places the responsibility of argument and type checking on each individual command. This means that a substantial portion of each C commands is spent checking types and converting arguments to primitive types and the results to strings. This adds considerable run-time overhead, especially when the application is data intensive (i.e., when the arguments passed among commands are numerous and large).

Another weakness of Tcl is the lack of support for development scale. Tcl, being a scripting language, lacks the modularity and type-checking necessary for the construction of large programs by

multiple developers.

8.2. SELF

SELF is a prototype-based language which was designed for exploratory prototyping. In addition to designing a simple, yet powerful object model, the SELF project has also focused on performance issues and contributed several novel compiler optimizations. For example, the SELF 2.0 interpreter has been reported as being as much as twice as fast as Smalltalk [Ungar92].

Strengths. Like Tcl, SELF is based on a small set of concepts which leads to a short learning curve and to compact programs. It also supports a dynamic model, which enables the modification of code without restarting the application. Unlike Tcl, SELF is a type-safe, object-oriented language. Consequently, SELF should exhibit better development scalability as the size of the application and the number of developers grow. Given that SELF was also designed with prototyping user interfaces in mind, it is conceivable that it would display similar or greater benefits if used on a comparable scale to Tcl.

Weaknesses. SELF is not widely used for commercial development for operational reasons mostly. Currently, SELF runs only on one hardware platform (SPARC) and has not focused on compiled legacy code wrapping.

9. Conclusion

BOS attempts to support evolutionary development by providing a pragmatic implementation of a prototype-based model. The BOS object model is sufficiently close to the SELF object model to yield the same advantages. Similarly, BOS allows for the wrapping of compiled code as in Tcl, while addressing some of its shortcomings. With the addition of several often used components to the BOS toolkit, such as Tk, operating systems threads, and a parser for a simple syntax, programmers are able to prototype user interfaces and domain software rapidly, evolve and stabilize their design, and address performance constraints by moving run-time consuming methods into a compiled language. Addressing performance issues by wrapping compiled code allowed the implementation of the BOS virtual machine to remain simple, thus portable and maintainable. We believe that this will also enable us to continue our exploration of prototype-based features and evaluate further refinements without significantly impacting existing code written with BOS. Finally, we envision a broader accessibility and use of BOS in the near future.

10.Acknowledgments

This work has been supported by the Engineering Design Research Center, an NSF Engineering Research Center. Support has also been provided in part by Asea Brown Boveri Ltd. and Union Switch and Signal, Inc.

We would like to thank: Eric Gardner, Suresh Konda, Russ Milliken, Jayachandra Reddy, and Mark Thomas for their patience and feedback while using BOS; and Eswaran Subrahmanian and Arthur Westerberg for their endless support in many forms without which this project could not have taken place. We would also like to extend special thanks to David Ungar for helping us to understand, clarify, and write about many of the concepts in this paper.

11.References

[ANSI83] ANSI, Inc. *The Programming Language Ada® Reference Manual.* ANSI/MIL-STD-1815A-1983, Springer-Verlag, New York, 1983.

[Blaschek84] G. Blaschek. *Object-Oriented Programming with Prototypes*. Springer Verlag, New York, 1984.

[Boehm88] B.W. Boehm. “A Spiral Model of Software Development and Enhancement.” *IEEE Transactions on Software Engineering*. pp. 61-72, May, 1988.

[Budde92] R. Budde, K. Kautz, K. Kuhlenkamp, & H. Züllighoven. *Prototyping: An Approach to Evolutionary System Development.* Springer Verlag, New York, 1992.

[Chambers95] C. Chambers. “Typechecking and Modules for Multi-Methods.” *ACM Transactions on Programming Languages (TOPLAS)*, Vol. 17, No. 9, November, 1995.

[Cox91] B. Cox. *Object-Oriented Programming: an Evolutionary Approach*. Reading, Mass. 1991.

[Finger93] S. Finger, E. Subrahmanian, & E. Gardner.

"Design Support System for Concurrent Engineering: A Case Study in Large Power Transformer Design." In *ICED 93 Conference Proceedings*, Vol. 3, pp. 14–36, 1993.

[Gabriel91] R.P. Gabriel, J.L. White, & D.G. Bobrow. "CLOS: Integrating Object-Oriented and Functional Programming." *Communications of the ACM*, Vol. 34 No. 9, pp. 28–38, September, 1991.

[Goldberg83] A. Goldberg & D. Robson. *SMALLTALK-80: The Language and its Implementation*. Addison-Wesley, Reading, Mass., 1983.

[Jacobson92] I. Jacobson, M. Christerson, P. Jonsson and G. Overgaard, *Object-Oriented Software Engineering: A Use Case Driven Approach*, Addison-Wesley Publishing Company, New York, 1992.

[Kernighan78] B.W. Kernighan & D.M. Ritchie. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, N.J., 1978.

[Krogh96] B. Krogh, S. Levy, A. Dutoit, & E. Subrahmanian. "Strictly Class-Based Modeling Considered Harmful." *Proceedings of the 29th Hawaii International Conference on System Sciences (HICSS)*. Maui, Hawaii, January 1996.

[Levy96a] S. Levy, A.H. Dutoit, E. Gardener, R. Patrick, D. Cunningham, & J. Uzmack. *BOS Reference Manual*. Engineering Design Research Center, Carnegie Mellon University, Pittsburgh, PA, 1996.

[Levy96b] S. Levy, A.H. Dutoit, E. Gardener, R. Patrick, D. Cunningham, & J. Uzmack. *Stitch Programmer's Manual*. Engineering Design Research Center, Carnegie Mellon University, Pittsburgh, PA, 1996.

[Ousterhout94] J.K. Ousterhout. *Tcl and the Tk toolkit*. Addison-Wesley, Reading, Mass., 1994.

[Rumbaugh93] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, & W. Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[RSmith95] R.B. Smith & D. Ungar. "Programming as an Experience: The Inspiration for Self." *ECOOP'95 Conference Proceedings*, Aarhus, Denmark, August, 1995.

[WSmith95] W.R. Smith. "Using a Prototype-Based Language for User Interface: The Newton Project's Experience." *OOPSLA 95 Conference Proceedings*, pp. 61–72. ACM, 1995.

[Stonebraker86] M. Stonebraker & L. Rowe ed. The Postgres Papers, Technical Report M86/85, Univ. of California, Berkeley, CA, 1986.

[Stroustrup91] B. Stroustrup. *The C++ programming language*. 2nd ed. Addison-Wesley, Reading, Mass. 1991.

[Subrahmanian93] E. Subrahmanian, S. Konda, S. Levy, I. Monarch, Y. Reich, & A. Westerberg, "Computational Support for Shared Memory in Design," *Automation Based Creative Design: Current Issues in Computers and Architecture*. Eds. A. Tzonis & I. White, Elsevier Publishers, Amsterdam, 1993.

[Ungar91a] D. Ungar and R.B. Smith. SELF: The Power of Simplicity. *LISP and Symbolic Computation*, Vol. 4, No. 3, pp. 187–205, 1991.

[Ungar92] D. Ungar, R.B. Smith, C. Chambers, & U. Hölzle. "Object, Message, and Performance: How They Coexist in Self." *Computer*, Vol. 25, No. 10, pp. 53-64 October, 1992.

[Wilkins89] D. J. Wilkins et al. "CINERG: A Design Discovery Experiment." *Proceedings of the NSF Engineering Design Research Conference*, pp. 161–182. National Science Foundation, June 1989.

[Wilson94] P. R. Wilson. "Uniprocessor Garbage Collection Techniques." (Long Version). Submitted to ACM Computing Surveys, 1994.