Toward Specification and Composition of BoxScript Components

H. Conrad Cunningham
Computer & Information Science
University of Mississippi
(662) 915-5358
cunningham@cs.olemiss.edu

Yi Liu
Computer & Information Science
University of Mississippi
(662) 915-7602
liuyi@cs.olemiss.edu

Pallavi Tadepalli
Computer & Information Science
University of Mississippi
(662) 915-7602
pallavi@cs.olemiss.edu

ABSTRACT
BoxScript is a Java-based, component-oriented programming language whose design seeks to address the needs of teachers and students for a clean, simple language. This paper briefly describes BoxScript and presents the authors' preliminary ideas on specification of components and their compositions.

Categories and Subject Descriptors
D.2.4 [**Software Engineering**]: Software/Program Verification – *class invariants*, *formal methods, programming by contract.*

General Terms
Design, Languages, Verification.

Keywords
Component, composition, specification, BoxScript.

1. INTRODUCTION
The goal of component-oriented programming is to enable a software system to be built quickly and reliably by assembling separately developed software components to form the system. The system should be flexible enough to be readily adapted to changing requirements by replacing, adding, or removing components. The concepts and languages that support this approach should be taught to students in computing science and software engineering programs.

In 2002 the first author taught an advanced software engineering class focused on Component Software in which the second and third authors were students [4]. The class used an approach to design similar to the "UML Components" approach of Cheesman and Daniels [2]. For the programming projects, the class used the Enterprise JavaBeans (EJB) component model and technology. EJB is a component model for building server-side, enterprise-class applications [11]. The complexity of the EJB technology meant it was not ideal for use in an academic course. The technology got in the way of teaching the students how to "think in components" cleanly. The students had to map their designs into the EJB technology [2, 7] and struggle to master enough of the technology to complete their term projects.

As a result, the second author undertook the design of a simple, component-oriented language with features that support its use in teaching. This language, called BoxScript, is described in section 2. Section 3 discusses the authors' preliminary ideas on how to specify BoxScript components and their composition formally and section 4 summarizes and identifies areas for further work.

2. BOXSCRIPT
BoxScript is a Java-based, component-oriented programming language whose design seeks to address the needs of teachers and students for a clean, simple language. The component concept is shown in Figure 1 [5].

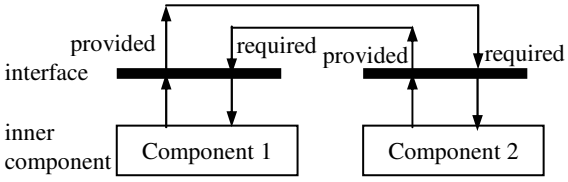

Figure 1. Components and Their Interconnections

A component is called a *box*. A box is a strongly encapsulated module that hides its internal details while only exposing its interfaces. There are two types of interfaces. A *provided interface* describes the operations that a box implements and that other boxes may use. A *required interface* describes the operations that the box requires and that must be implemented by another box. A BoxScript interface is represented syntactically by a Java interface, that is, by a set of related operation signatures. Each occurrence of an interface in a box has an *interface handle*, which identifies that occurrence uniquely within the box, and a *type*, which is the Java interface type. Each box has a corresponding box description (`.box`) file that gives the needed declarations.

An *abstract box* is a box that describes the provided and required interfaces but does not implement the provided interfaces. An abstract box should be implemented by concrete boxes, i.e., atomic or compound boxes. Figure 2a shows an abstract box description `DateAbs`. Its provided interface `DayCal` calculates the day of the week for a date. Figure 2b shows another abstract box description `CalendarAbs`, which has one provided and one required interface. Its provided interface `Display` takes the time range and displays the calendar accordingly.

```
abstract box DateAbs
{  provided interface DayCal Dc;
     //Dc is handle of interface DayCal
}
```

Figure 2a. DateAbs.box

```
abstract box CalendarAbs
{  provided interface Display Dis;
   required interface DayCal  DayC;
}
```

Figure 2b. CalendarAbs.box

An *atomic box* is the basic element in BoxScript. It does not contain any other boxes. It must supply an implementation for

each provided interface, that is, a Java class that `implements` the interface. The description of an atomic box gives the box name and, if appropriate, the name of the abstract box it implements. It also gives its provided and required interfaces by listing their interface types and handles. Figure 3a and 3b show atomic box descriptions `Date` and `Calendar`. `Date` implements `DateAbs` and `Calendar` implements `CalendarAbs`.

```
box Date implements DateAbs
{  provided interface DayCal Dc; }
```

Figure 3a. Date.box

```
box Calendar implements CalendarAbs
{ provided interface Display Dis;
  required interface DayCal  DayC;
}
```

Figure 3b. Calendar.box

A *compound box* is a box composed from other boxes. It does not implement its provided interfaces, but uses the implementations provided by its constituent boxes. Each constituent box is given an identifier, called its *box handle*, to enable it to be uniquely identified as a participant within the composition. The box description for a compound box not only supplies the information given in the atomic box, but also specifies (1) the boxes from which this compound box is composed, (2) the sources of its provided and required interfaces, and (3) the connection information that describes how the constituent box interfaces are "wired" together. To provide flexibility, a compound box can be declared to be `composed from` either concrete or abstract boxes. `BuildCalendar` (in Figures 4a and 4b) is composed from abstract boxes `DateAbs` and `CalendarAbs`. When we configure `BuildCalendar`, we substitute concrete boxes such as `Date` and `Calendar` for the corresponding abstract boxes.

```
abstract box BuildCalendarAbs
{  provided interface Display D;}
```

Figure 4a. BuildCalendarAbs.box

```
box BuildCalendar implements
                    BuildCalendarAbs
{  composed from DateAbs boxD,
          CalendarAbs boxC;
 // boxD is box handle for DateAbs
 // boxC is box handle for CalendarAbs
  provided interface
      Display D from boxC.Dis;
  connect boxC.DayC to boxD.Dc;
}
```

Figure 4b. BuildCalendar.box

BoxScript uses the box handles to expose and connect the interfaces of the constituent boxes. The `composed from` declaration in `BuildCalendar` assigns `boxD` and `boxC` as the box handles for `DateAbs` and `CalendarAbs`, respectively. The `provided interface` and `required interface` declarations give the types of the interfaces, their interface handles, and their sources. The source is a box handle and interface handle associated with a constituent box. In `BuildCalendar`, interface handle `D` identifies an interface of type `Display` that is mapped to interface handle `Dis` of the box with box handle `boxC` (i.e., `CalendarAbs`). The `connect` statement connects a required interface of one box to a provided interface of another. In `BuildCalendar`, the required interface with handle `DayC` of the box with box handle `boxC` (i.e., `boxC.DayC`) is connected to the provided interface with handle `Dc` of the box with box handle `boxD` (i.e. `boxD.Dc`).

The composition of boxes into a compound box hides all provided interfaces that are not explicitly exposed and must expose every required interface that is not wired to a provided interface of a box within the composition. In the example, provided interface `Display` is exposed. Figures 5a and 5b illustrate the composition process.

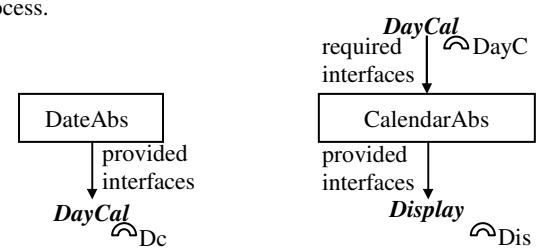

Figure 5a. DateAbs and CalendarAbs

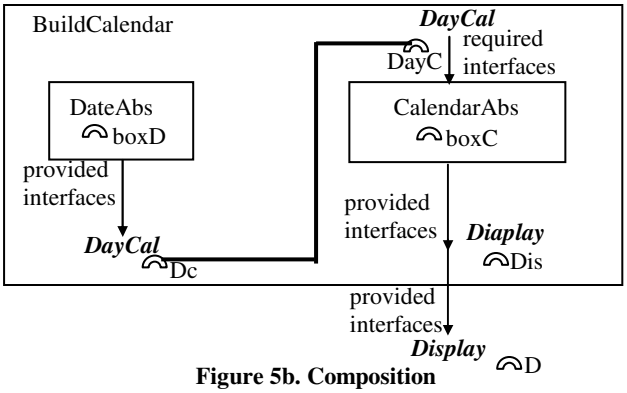

Figure 5b. Composition

Atomic and compound boxes may either be standalone or implementations of abstract boxes. All the implementations of an abstract box are *variants* of the abstract box. The intention is that one variant can be safely substituted for another. When one box substitutes for another, the substitute must *satisfy* the specification of the original box. A variant's provided interfaces should supply at least the operations of the abstract box and the variant's required operations should be at most those of the abstract box.

3. SPECIFICATION

In BoxScript, as in the Cheesman-Daniels approach [2], one basic unit for specifying functionality is the interface. An interface is a set of operation signatures (name, parameter types and order, and return value types) that are related. BoxScript uses Java interfaces for its interface types.

In the Cheesman-Daniels approach, the semantics of an interface is specified in terms of an *interface information model* [2], which is expressed graphically as a UML type (class) diagram augmented by Object Constraint Language (OCL) [12] invariants. For BoxScript, we simplify the presentation and consider the information model to consist of a pair `(V,I)`, where `V` is a set of abstract variables representing the *abstract state* of the component instance associated with the interface and `I` is an *invariant* representing the valid values of the abstract state.

An *invariant* is an assertion that must be kept true in all states of a box that are visible to its clients [6]. We attach invariants to an interface to specify the unchanging properties of the objects that implement the interface. In the model, symbol `I` denotes the conjunction of all the invariants attached to an interface.

We specify the semantics of an individual operation using *precondition* and *postcondition* assertions. A precondition expresses the requirements that any call of the operation must satisfy. That is, it gives valid values of the operation arguments and the interface's abstract state from which the operation can be safely called. A postcondition expresses properties that are ensured in return by the execution of the call. It gives the results of the operation in terms of the arguments and abstract state. We require any operation that is called with the precondition true to terminate eventually with the postcondition true.

To provide precise specification about the relationships of operations calls to each other, we can include *history sequences* [3], which record the sequence of operation calls. This allows assertions about the sequences to appear in the invariants, preconditions, and postconditions.

A box interface `x` *extends* box interface `y` (syntactically) if and only if type(`x`) = type(`y`) or type(`y`) extends type(`x`) in the Java type system. That is, all the operation signatures in `y` also appear in `x`, but `x` may have additional operations. Type extension does not allow either covariant or contravariant changes to operations.

Box interface `x` satisfies interface `y` when `x` provides at least the operations required by `y` and the operations of `x` have an equivalent meaning to the matching operations in `y`. More formally, box interface `x` *satisfies* box interface `y` if and only if:

- `x` *extends* `y`
- `I(x) & C(x,y) ⇒ I(y)`
- `(∀m : m ∈ y :`
  `(pre(y,m) & C(x,y)& I(y) ⇒ pre(x,m))`
  `& (post(x,m)& C(x,y)& I(x) ⇒ post(y,m)))`

Above, `I(x)` refers to the invariant for `x` and `pre(x,m)` and `post(x,m)` refer to the precondition and postcondition, respectively, for operation `m` on interface `x`. Assertion `C(x,y)` is a coupling invariant that relates the equivalent aspects of the interface information models for `x` and `y`.

The above definition of *satisfaction* is motivated by Meyer's treatment of inheritance in the design by contract approach (and the Eiffel language) [8] and the concept of a coupling invariant in program and data refinement [9].

The second basic unit of specification is the box. A box is a program module that encapsulates some functionality behind its provided interfaces. A client of the box may call an operation on a provided interface. To carry out this operation, a box may invoke operations on its required interfaces, each of which is connected to a provided interface of some box. The specification for a provided interface must be satisfied by the implementation of the box; the specification for a required interface must be satisfied by a provided interface of some box.

A *box's information model* is formed by joining the information models of its provided interfaces. It may have new abstract state variables and a *box invariant* that defines the validity of the box's state. For a box `B`, let `I(B)` be its box invariant, `C(B)` be the coupling invariant that ties it to the interface information models, and `prov(B)` be the provided interfaces. For any box `B`, it must be the case that:

`(∀p: p ∈ prov(B): I(p)) & C(B) ⇒ I(B)`

An *atomic box* must supply implementations for its provided interfaces as a cluster of Java classes. The implementations of the interfaces within an atomic box may interact directly with each other and share internal state. A provided interface thus must preserve the invariants of all the box's provided interfaces [10]. A convenient way to achieve this is for all of the provided interfaces of an atomic box to have the same information model `(V,I)`.

A *compound box* composes one or more other boxes to form the "larger" box. As is the case with any box, a compound box has a specification as described above. It has a box information model (i.e., abstract state and box invariant) and interface specifications for the provided and required interfaces. The box invariant ties together the information models of the provided interfaces to form the information model for the compound box.

As with the atomic box, a compound box must provide implementations for its provided interfaces and it may use its required interfaces in doing so. However, unlike the atomic box, the compound box defers the implementation of a provided interface to one of its constituent boxes. The interface handle in the compound box is either the same as in the constituent box or it may be an *alias* that is linked to an interface of the constituent box. Similarly, a required interface of the compound box may be a required interface of one or more constituent boxes. A constituent box may have provided interfaces that are not exposed by the compound box. However, a required interface of a constituent box must either be exposed outside the compound box or be satisfied by some provided interface within the compound box. Thus the box invariant for a compound box must relate the properties expected for its interfaces to the related properties of the corresponding interfaces of constituent boxes.

More formally, for any compound box `B`, the following must hold:

- `(∀p: p ∈ prov(B): I(p)) & C(B) ⇒ I(B)`
- `(∀p: p ∈ prov(B):`
  `(∃D,q: D ∈ const(B) & q ∈ prov(D)`
  `& q = alias(B,p): q` *satisfies* `p))`
- `(∀D,r: D ∈ const(B) & r ∈ req(D):`
  `(∃s: s ∈ req(B) & r = alias(B,s):`
  `s` *satisfies* `r)` **OR**
  `(∃E,q: E ∈ const(B) & q ∈ prov(E)`
  `& connected(B,r,q): q` *satisfies* `r))`

In the above, `const(B)` denotes the set of boxes that are composed to form compound box `B`, `alias(B,q)` is the function that maps an interface `q` of compound box `B` to an interface in a constituent box, `req(B)` is the set of required interfaces of box `B` and `connected(B,r,p)` is an assertion that required interface `r` is connected to provided interface `p`. This information is available from the box description. The box invariant may be used in showing that one interface within the box satisfies another.

Consider a valid relationship between a concrete box `B` and an abstract box `A` that it *implements*. Clearly, if abstract box `A` specifies the presence of a provided interface `p`, then concrete box `B` *must* have a provided interface that satisfies `p`. If concrete box `B` has a required interface `r`, then abstract box `A` *must* specify a required interface that satisfies `r`. In terms of operations, the provided interfaces of `B` should supply at least the operations of `A`, and the required operations of `B` should be at most those of `A`. A similar situation occurs if we consider an abstract box that *extends* another abstract box.

More formally, box B *satisfies* box A if and only if:

- `I(B) & C(A,B) ⇒ I(A)`
- `(∀p: p ∈ prov(A): (∃q: q ∈ prov(B):`
  `handle(q) = handle(p) & q` *satisfies* `p))`
- `(∀r: r ∈ req(B):(∃s: s ∈ req(A):`
  `handle(r) = handle(s) & s` *satisfies* `r))`

Above, `C(A,B)` denotes a coupling invariant for the refinement of the information model when replacing `A` by `B`. In particular, `C(A,B)` serves as the coupling invariant for showing that the interfaces of `B` have the needed satisfaction relationship with the corresponding interfaces of `A`. The notation `handle(p)` refers to the interface handle of interface `p`.

A compound box may be composed of abstract boxes. At runtime, an instance of a variant of the abstract box is configured into the instance of the compound box. As noted above, the variant must satisfy the specification for the abstract box it implements. That is, the variant is the same as the abstract box from the perspective of its specification. Thus the box invariant of the compound box can transparently address the different variants.

4. CONCLUSION

BoxScript is a Java-based, component-oriented programming language that is under development by the authors. Its design seeks to address the needs of teachers and students by providing a simple and clean language, yet one that can be used to solve practical problems. It introduces a notation for components and their composition but uses the Java language (which is familiar to most students) to express the internal details of components.

This paper briefly describes the concepts of BoxScript and presents the authors' preliminary ideas on formal specification of BoxScript components and their compositions. Although formal specification and verification were not design goals for BoxScript, its relatively simple design, which is based on strongly encapsulated modular units, seems to be amenable to the application of formal techniques. The ideas outlined in this paper do seem promising, but considerable work is needed to elaborate the formalism and experiment with the pragmatics of the approach. In particular, several examples need to be worked out to demonstrate the concepts and techniques. It will also be helpful to adapt the BoxScript approach to enable use of techniques and tools such as those associated with the Java Modeling Language (JML) [6].

The approach sketched in this paper is likely insufficient to capture the full semantics of calls to the required interfaces, in particular, calls that may lead to reentrance into the calling box (e.g., call-backs). The greybox approach [1] or a similar technique may be needed to enable verification of compound boxes.

This paper approaches specification of program semantics in a manner that is language-oriented, that is, somewhat bottom-up and compositional. The ideas should also be addressed from a software engineering perspective, seeking techniques that can be applied effectively in a more top-down, decompositional manner.

5. ACKNOWLEDGMENTS

This work was supported, in part, by a grant from Acxiom Corporation titled "The Acxiom Laboratory for Software Architecture and Component Engineering (ALSACE)." The design and implementation of BoxScript is part of a Liu's dissertation project under the direction of Cunningham. We thank the reviewers for their many helpful comments on this paper.

6. REFERENCES

[1] M. Büchi and W. Weck. *The Greybox Approach: When Blackbox Specifications Hide Too Much*, Technical Report No. 297a, Turku Centre for Computer Science, Finland, August 1999.

[2] J. Cheesman and J. Daniels. *UML Components: A Simple Process for Specifying Component-Based Software***,** Addison Wesley, 2001.

[3] H. C. Cunningham and Y. Cai. "Specification and Refinement of a Message Router," In *Proceedings of the Seventh International Workshop on Software Specification and Design*, IEEE, December 1993.

[4] H. C. Cunningham, Y. Liu, P. Tadepalli, and M. Fu. "Component Software: A New Software Engineering Course," *Journal of Computing Sciences in Colleges*, Vol. 18, No. 6, pp. 10-21, June 2003.

[5] W. Fleisch. "Applying Use Cases for the Requirements Validation of Component-Based Real Time Software," In *Proceedings of the Second IEEE International Symposium on Object-Oriented Real-Time Distributed Computing,* p. 75, IEEE, 1999.

[6] G. T. Leavens and Y. Cheon. "Design by Contract with JML," draft paper, Iowa State University, August 2004.

[7] Y. Liu and H. C. Cunningham. "Mapping Component Specifications to Enterprise JavaBeans Implementations," In *Proceedings of the ACM Southeast Conference*, pp. 177-181, April 2004.

[8] B. Meyer. *Object-Oriented Software Construction*, Second Edition, Prentice Hall, 1997.

[9] C. Morgan. *Programming from Specifications*, Prentice Hall International, 1994.

[10] P. Müller. *Modular Specification and Verification of Object-Oriented Programs,* Lecture Notes in Computer Science 2262, Springer-Verlag, 2002.

[11] I. Singh, B. Stearns, M. Johnson, and the Enterprise Team. *Designing Enterprise Applications with the J2EE™ Platform*, Second Edition. Addison Wesley, 2002.

[12] J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modeling with UML*, Addison-Wesley, 1999.