Modular Specification of Encapsulated Object-Oriented Components

Arnd Poetzsch-Heffter and Jan Schäfer

Technische Universität Kaiserslautern, Germany
poetzsch@informatik.uni-kl.de

Abstract. A well-defined boundary of components allows to encapsulate internal state and to distinguish between internal calls that remain inside the component and external calls that have target objects outside the component. From a static point of view, such boundaries define the programmer's interface to the component. In particular, they define the methods that can be called on the component. From a dynamic point of view, the boundaries separate the component state and those parts of the program state outside the component.
In this tutorial paper, we investigate encapsulated components that are realized based on object-oriented concepts. We define a semantics that captures a flexible notion of hierarchical encapsulation with confined references. The semantics generalizes the encapsulation concepts of ownership types. It is used as a foundation for modular behavioral component specifications. In particular, it allows to provide a simple semantics for invariants and an alternative solution for the frame problem. We demonstrate this new specification methodology by typical programming patterns.

1 Introduction

Component-based software is developed by linking components or by building new components from existing ones. A key issue in component-based software development is to specify the component interfaces in an implementation independent way so that components can be used relying only on their specifications. The often cited component definition by Szyperski [36], p. 41, says: "A software component is a unit of composition with contractually specificied interfaces and explicit context dependencies only (...)." In this tutorial paper, we investigate behavioral specification techniques for encapsulated object-oriented components that we call *boxes*. A box instance is a runtime entity that encapsulates a number of objects. Some of these objects are confined to the box, that is, they may not be referenced from outside the box. Other objects of the box may be referenced from the outside. The box model is a generalization of programming models underlying ownership type systems. It simplifies the semantics of specification constructs and handles complex program patterns. In particular, we consider reentrant calls and components with multiple ingoing references. According to the tutorial character of the paper, conveying the general concepts will be favored over technical rigour.

Component specifications are used as documentation and to improve program development and understanding. They can support program testing and runtime checks. Furthermore, they simplify static analysis and are a prerequisite for the verification of program properties. A central goal for specification techniques in all these areas is *modularity*. A modular specification technique allows showing that a component implementation satisfies its specification by only knowing the modules implementing the component and by only using specified properties of referenced components. In particular, knowledge about the application context of a component is not necessary. Modularity is a very important requirement for the scalability of a specification technique, because components can be checked once and for all independent of future application contexts. Unfortunately, due to aliasing and subtyping, modularity is difficult to achieve in an object-oriented setting, in particular if components consist of collaborating objects.

In the remainder of this section, we introduce the programming model underlying boxes (Subsect. 1.1), explain the challenges of modularity in more detail, and shortly describe our approach (Subsect. 1.2). In Sect. 2, we present the box model and its semantic foundations. Section 3 describes the developed specification technique along with its application. A discussion of our approach and its relation to existing work is contained in Sect. 4. The paper is completed by conclusions and topics for future work in Sect. 5.

1.1 Programming Model Based on Boxes

In this subsection, we informally introduce the *box model* consisting of the new concept "Box" and the underlying programming model. We build on the general object-oriented model with classes, objects, (object) references, object-local state represented by instance variables, methods to define behavior, and a type system with subtyping. We use notations from Java and C# (see [14, 22]). Throughout the paper, we make two simplifiying assumptions: We only consider *single-threaded* programs with *terminating* methods[1].

A *box instance* or *box*, for short, is a runtime entity. Like an object, a box is created, has an identity and a local state. However, a box is in general realized by several objects. A prominent example — often discussed in work on ownership and alias control — is a linked list object with iterators (see e.g. [2]). Figure 1 shows such a list box, indicated by a rounded rectangle with a dashed line. A box and its state is characterized by:

- an *owner* object that is created together with the box (the IterList-object in Fig. 1); the owner is one of the boundary objects;
- other *boundary* objects, that is, objects that can be accessed from outside the box (the two Iterator-objects in Fig. 1);
- so-called *confined* objects that may not be accessed from outside the box (the Node-objects in Fig. 1);

[1] These assumptions focus the paper, they are not necessary requirements. On the contrary, one major motivation for the development of the box model is a higher-level model for synchronization in multi-threaded programs.

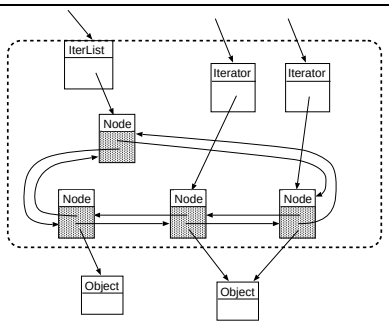

Fig. 1. A box encapsulating the implementation of a linked list.

– so-called *external* objects that are referenced from within the box (the objects outside the dashed line in Fig. 1).

The *concrete state* of a box changes if the state of one of its objects changes or if a new object is created within the box. As part of the concrete state of a box may be hidden, clients of a box usually see only an abstraction of the concrete state, the so-called *abstract state*. Clients use the functionality of a box by calling methods on the owner or the other boundary objects. Thus, to work with a box, a client has to know the constructor and public interface of the owner object and the public interfaces of the other boundary objects. In addition, it is helpful to know the public interfaces of external objects passed to the box, because this is the only way a box can influence the state outside the box, the so-called *box environment*. Figure 2 presents the interfaces of the IterList-box.

The interfaces in Fig. 2 use two notations that go beyond Java or C#. Interfaces marked with the keyword `box` provide a box constructor (IterList in Fig. 2). Thus, a box interface is similar to the public interface of a class without public fields. The annotion of arguments and results by the keyword `external`

```
box interface IterList {
  IterList ();
  external Object get( int index );
  void add( external Object x );
  external Object remove( int index );
  boolean contains(external Object x);
  Iterator  listIterator ();
}
```

```
interface Iterator {
  boolean hasNext();
  external Object next();
  void    remove();
}

interface Object {
  boolean equals(external Object x);
}
```

Fig. 2. Interfaces of lists with iterators.

```
box interface SecretHolder {
  SecretHolder();
  void invite(external Stranger x);
}
```

```
box interface Stranger {
  Stranger();
  void doMeAFavor(external Object x);
}
```

Fig. 3. Exporting objects by calls to external references.

indicates that such arguments and results may refer to objects in the box environment. More precisely, we distinguish three different kinds of arguments and results:

1. *Data objects* are considered global values of the system that can be arbitrarily passed around. They are either of a primitive type like `boolean` or `int` or of a so-called `pure` type. A pure type has only immutable objects and its methods have no side-effects (see [18]). A typical example for a pure type is the type `String` in Java. We assume that types are declared as pure so that it is known whether a type is pure or not.
2. *Boundary objects* are the internal objects of a box that may be *exposed* to the box environment either as a result of a method or by passing them as a parameter to a method called on an external object. Arguments or results that are not pure and have no annotation are considered boundary objects.
3. *External objects* are objects that are passed into the box. Usually, these objects are objects from outside the box. For flexibility reasons, we allow as well to pass boundary objects as external objects into a box. (The semantic details will be treated in Sect. 2.)

Component Categories. Based on the box model, we can categorize components according to their behavior at the boundary. For example, lists with iterators have the following characteristic features:

1. They have a nontrivial encapsulated state that can be manipulated from the outside by "handles", like e.g. iterators (below, we will see that an iterator can change the state of the list and of the other iterators).
2. They import external references only by methods called on boundary objects.
3. They do not call methods with side-effects on external references (below we see, that a call of method `contain` in `IterList` leads to external calls of method `equals`).

The list with iterator example is often used in the literature, because its features are shared by other components, for example by complex data structures with iterators for special traversals or, on a more conceptual level, by file systems with file handlers. Although many specification and verification frameworks already have problems handling components with these feature, we believe for two reasons that we have to cover a larger class of components. One reason is that

more general components are used in practice. We will look at a representative example, namely an instance of the observer pattern, in the next subsection. The other reason is that we would like to use the model to analyse, specify, or exclude more complex component interaction. As an example, consider the boxes in Fig. 3. At first sight, secret holders will keep their secret. There is no method returning anything. However, a secret holder can reference a Stranger-object and it can do the stranger a favor, passing some object to the stranger. Accidentally, it might pass an object of type Secret representing a secret. The stranger could cast the received object down to a Secret-object and extract the secret. Two aspects that have motivated design decision of the box model can be learned from this example:

- External calls should be explicit in specifications.
- Downcasts of external objects should not be allowed.

Before we present the semantic basis for the box model, we look at the main motivation for the model, that is, the challenges of modular specification.

1.2 Specification and Modularity

Component specifications express different kinds of properties of components. *Functional* properties relate method results to method arguments. *Structural* properties describe invariants on the reference structure between objects. For example, in the box model, we guarantee that all reference chains from an object outside the box to an object inside the box go through a boundary object. *Frame* properties state the effects of a method call and what is not affected by a call, often called the *non-effects*. The treatment of non-effects is very important in an object-oriented setting, because a method call $X.\mathrm{m}(Y,Z)$ can potentially affect all objects that are reachable from X, Y, Z via reference chains.

In this subsection, we explain what modularity means for box specifications and describe the challenges of modularity.

Box Specifications and Modularity. For the definition of modularity, we have to make some assumptions about the implementation of a box B and need some terminology. In a first step, we assume that an implementation of B consists of a set of classes and interfaces. One of the classes implements B and has a public constructor with the same argument types as the constructor in the box interface of B (the external annotations are not considered). An implementation of a box is called a *box class*. Boxes with such implementations are called *simple*. For an interface I, the *minimally type-closed* set of interfaces containing I, denoted by $MTC(I)$, is the smallest set satisfying the properties:

- $I \in MTC(I)$
- if $J \in MTC(I)$ and T is an argument or result type of J, then T is primitive or $T \in MTC(I)$

An *interface specification* consists of annotations specifying properties of the interface. Our interface specification technique is explained in Sect. 3. An example of an interface specification for `IterList` is shown in Fig. 11, p. 21.

For a box B, we distinguish between three not necessarily disjoint subsets of $MTC(B)$:

- The *exposed* interfaces describe the interfaces of boundary objects including the interface of B.
- The *referenced* interfaces describe the interfaces of external objects.
- The *additional* interfaces are the interfaces in $MTC(B)$ that are neither exposed nor referenced.

A *box specification* consists of the exposed interface specifications and uses the referenced and additional interface specifiations. In many cases, some of the exposed interfaces already exist when a new box class is developed.

Definition (Modularity). A specification technique for simple boxes or box-like components is called *modular* if one can show that the box class satisfies its specification by using only

- the referenced and additional interface specifications, and
- the classes and interfaces of the implementation.

Before we analyse the challenges of modularity, we slightly generalize our implementation concept and, together with it, the notion of modularity. The implementation of a *compound* box may use other boxes. Whereas — from an implementation point of view — this merely structures the complete set of classes and interfaces an implementation consists of, it generalizes the notion of modularity. The implementation of a compound box B consists of a set of classes and interfaces, and a set of box specifications not containing B.[2] The classes can use the specified boxes. As these boxes are encapsulated in B, we call them *inner* boxes of B.

For compound boxes or similar hierarchical components, modularity allows the use of inner box specification, but not of their implementation. Compound boxes have a flavor of composition. However, the box model does not support direct composition without glue code. Box classes are composed by additional program parts that link the box instances at runtime and provide further functionality or adaption.

Challenges of Modularity. Every modular specification and verification framework for object-oriented components with method call semantics is confronted with three main challenges:

1. *Control of specification scope*: The specification of a component C may only express properties that are under full control of C. Otherwise, these properties might be invalidated by other components or program parts that are not known when C's specification is checked.

[2] We claim that even recursive boxes can be allowed, but do not consider it here, because we have not yet investigated it in detail.

2. *Underspecification and reentrance*: A component C can reference another component D through an interface I with a very weak specification. That is, C knows almost nothing about the behavior of D. In particular, methods called from C on D might cause reentrant calls on C that are not specified in I.
3. *Frame problem*: Specifications have to be sufficiently expressive w.r.t. effects and non-effects to the enviroment. For example, if a component C uses components D and E for its implementation, it has to know whether a method call on D causes a side-effect on E or not. Otherwise, it has no information about the state of E after the call.

To illustrate these main challenges of modularity, we consider a simplified instance of the observer pattern. An observable game is a two-player game with alternative moves where a human plays against the computer. One player plays white, the other black. The interfaces of this example are given in Fig. 4. Method `move` allows the human to make a move that is internally answered by the computer, method `swapPlayers` swaps the players, and method `readPos` reads and returns the current position on the board. `MoveDescr` and `Position` are pure types. Method `register` registers observers of the game. The interface of box `GameObserver` describes a simple observer.

To both box interfaces in Fig. 4, we have already added ghost and model variables expressing the box state. The ghost variable `gameObs` in interface `ObservableGame` captures the set of referenced observers of a game, variable `obsGame` holds the observed game of a game observer. The model variables `currentPos`, `player`, and `displayedPos` hold the current position of a game, the

```
box interface ObservableGame {
  references Observer* gameObs;
  model Position currentPos;
  model Color player;
  ObservableGame();
  void move( MoveDescr md );
  void swapPlayers();
  Position readPos();
  void register(external Observer go);
}

interface Observer {GameObserver
  void stateChanged()
}
```

```
box interface GameObserver
              extends Observer {
  references ObservableGame obsGame;
  model Position displayedPos;
  GameObserver(
        external ObservableGame g );
  void stateChanged();

  invariant
  // out of control:
     forall( o in obsGame.gameObs ){
       o instanceof GameObserver
     }
  // problematic:
     this in obsGame.gameObs
}
```

Fig. 4. Interfaces of `ObservableGame` and `GameObserver`

color of the human player, and the displayed position of an observer respectively. (More details are given in Sect. 3.)

Control of Specification Scope. The scope of a specification depends on the component model and encapsulation discipline. In our case, the scope is related to the box. The situation is simple if specifications only depend on the local state of the box, because this state can be controlled. It is less clear if a specification depends on state of referenced boxes. For example, the first invariant of box `GameObserver`, stating that the referenced game has only game observers of type `GameObserver` in Fig. 4, is not locally verifiable, because the interface of `ObservableGame` allows to register observers of other types. The second invariant is more interesting. If the specification of `ObservableGame` guarantees a reasonable behavior of `register` and allows us to prove that the set of game observes is only growing and if we can verify that the constructor `GameObserver` does registration correctly, one might be able to verify the second invariant in a modular way (cf. [4] for a detailed discussion).

Underspecification and Reentrance. In a typical observer scenario, the observed subject has only very little information about the observers. For example, the box `ObservableGame` only knows that observers have a method `stateChanged`. It knows nothing about the behavior of `stateChanged`. Now, let us assume we want to prove that the implementation of method `move` does not swap the players. As method `move` changes the position, it will call method `stateChanged`. Figure 5 shows a scenario in which a somewhat malicious observer calls `swapPlayers` before returning from `stateChanged`. Thus, we can only prove the "non-swap-players" property in a modular way if the specification of `ObservableGame` can restrict reentrant calls. More generally, a modular specification technique has to provide mechanisms to control reentrant calls.

Frame Problem. To illustrate the frame problem, we consider a gaming system that uses `ObservableGame` and `GameObserver` as inner boxes, that is, modular verification has to rely on their specification. A central invariant of the gaming

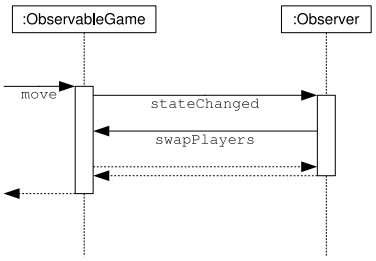

Fig. 5. An observer modifying the state of the game by a reentrant call.

system would be that the current observers `currObs` of the system display the current position of their observed games:

```
forall( o in currObs ){ o.displayPos == o.obsGame.currentPos }
```

This invariant can only be shown if the specification of `move` guarantees that all registered observers are notified. More generally, a method specification must describe its side-effects. Furthermore, we want to know what is left unchanged by a method execution. For example, it should be derivable from the specifications that a move in one game does not affect the state of another game (if this is the case).

Approach. Our specification approach is based on the encapsulation properties of the box model. The box model is semantically founded in the programming language. This is different from most other approaches to modular specification, where the structuring and encapsulation techniques are only part of the specification framework. Having a clear foundation in the language semantics provides a more expressive basis to develop a specification technique. This is important for our goal to separate the behavioral specifications from the implementation.

The remainder of this tutorial paper presents central parts of our approach. In Sect. 2, we demonstrate how the box model can be integrated into the semantics of a simple class-based language. To keep the presentation focused, the language supports only a subset of the constructs that we use in the specification examples. In Sect. 3, we explain the main concepts of our specification technique and show how the specifications profit from the box model.

2 Encapsulated Object-Oriented Components

Heap encapsulation can be achieved by different techniques. We propose here a semantic-based approach, that is, an approach in which the structuring of the heap is part of the programming language semantics.

It is helpful to compare the general approach with the step from untyped to typed languages. Typed languages provide an additional notion, the types, to formulate certain desirable program properties. Types have a clear foundation in the language semantics (e.g. the type of an object is needed for casts). They avoid runtime errors (e.g. situations where a target object of a call does not have an appropriate method), but most of the checks can be done at compile time so that runtime overhead can be kept small. Our goals are similar. We provide a notion of encapsulation, namely the boxes. Our box model has the following features:

1. They hierarchically structure the heap. In particular, this structure can be used to define the meaning of interface specifications.
2. They distinguish between internal and external calls. This is important for partitioning systems into components.
3. They provide a notion of object confinement. This supports the programmer management of alias control and is important for modular verification.

$$
\begin{array}{rll}
P \in \textbf{Program} & ::= & \overline{D} \\
D \in \textbf{TypeDecl} & ::= & ID \mid BD \mid CD \\
ID \in \textbf{IntfDecl} & ::= & \texttt{interface}\ I\ \texttt{extends}\ \overline{I}\ \{\ \overline{G}\ \} \\
BD \in \textbf{BoxDecl} & ::= & \texttt{box interface}\ B\ \texttt{extends}\ \overline{I}\ \{\ \overline{G}\ \} \\
CD \in \textbf{ClassDecl} & ::= & \texttt{class}\ C\ \texttt{implements}\ \overline{S}\ \{\ \overline{T}\ \overline{f};\ \overline{M}\ \} \\
A \in \textbf{Annotation} & ::= & \texttt{boundary} \mid \texttt{external} \\
m \in \textbf{MethodName} & & \\
G \in \textbf{AbstractMeth} & ::= & A\ S\ m(\ \overline{A}\ \overline{S}\ \overline{x}\ ) \\
M \in \textbf{MethodDecl} & ::= & T\ m(\ \overline{T}\ \overline{x}\ )\ \{\ e\ \} \\
e \in \textbf{Expr} & ::= & x \mid (T)\ e \mid \texttt{new}\ lm\ em\ R(\overline{e}) \mid e.f \mid e.f = e \mid e.m(\overline{e}) \mid \\
 & & \texttt{let}\ x = e\ \texttt{in}\ e \mid \ldots \\
lm \in \textbf{LocModifier} & ::= & \texttt{local} \mid \texttt{global} \\
em \in \textbf{ConfModifier} & ::= & \texttt{confined} \mid \texttt{exposable} \\
x, y \in \textbf{Variable} & & \\
I \in \textbf{InterfaceName} & & R \in \textbf{BoxName} \cup \textbf{ClassName} \\
B \in \textbf{BoxName} & & S \in \textbf{InterfaceName} \cup \textbf{BoxName} \\
C \in \textbf{ClassName} & & T \in \textbf{InterfaceName} \cup \textbf{BoxName} \cup \textbf{ClassName}
\end{array}
$$

Fig. 6. Syntax of OBO.

In this section, we show how boxes can be founded in the language semantics. We believe that the presented approach can be extended to object-oriented programming languages like e.g. Java and C#. Techniques for static checking are shortly discussed in Sect. 4.

A Simple OO-Language with Boxes. We present a simple OO-language supporting the box model, called *OBO*. The language is similar to other OO-kernel languages ([13, 16]). To focus on the central issues, we make the following simplifications:

- OBO supports subtyping, but not inheritance. In particular, the root type `Object` is an interface (see Fig. 2).
- OBO supports only default constructors.
- OBO has no exception handling mechanism. If a runtime error occurs, that is, if there is no applicable semantic rule, we assume that the program aborts.
- In OBO, all pure types, including `String`, are predefined. They have appropriate box interfaces and are treated as external types. That is, in OBO interfaces, we only distinguish between boundary and external types.

Figure 6 shows the syntax of OBO. Within the syntax, lists are denoted by an overline. A program consists of a list of interfaces, box interface, and class declarions, including a startup class `Main` of the form:

```
class Main {  String main( String arg ){ e } }
```

$$
\begin{array}{rll}
b \in \textbf{Box} & ::= & bc \mid \texttt{globox} \\
o \in \textbf{Object} & ::= & (b, C, j; em) \\
r \in \textbf{Reference} & ::= & \langle o, T, rk \rangle \\
rk \in \textbf{RefKind} & ::= & \texttt{intn} \mid \texttt{extn} \\
v \in \textbf{Value} & ::= & null \mid r \\
OS \in \textbf{ObjState} & ::= & \langle \overline{v} \rangle \\
BS \in \textbf{BoxState} & ::= & \{o : OS\} \\
H \in \textbf{Heap} & ::= & \{b : BS\} \\
F \in \textbf{StackFrame} & ::= & \{x : v\} \\
j \in \textbf{ObjId} & & \\
bc \in \textbf{CreatedBox} & &
\end{array}
$$

Fig. 7. Dynamic entities of OBO.

Interfaces and box interfaces are as explained in Subsect. 1.1. For each box interface B in a program, there has to be exactly one class implementing B. This class is denoted by $class(B)$ in the semantics below. Context conditions are essentially like in Java, in particular the typing rules apply. The subtype relation is denoted by $\preceq$.

The creation expression starting with keyword `new` deserves consideration. It allows to create instances of a class or box. The instance can be created locally or globally. Within a program, global creation is only allowed for the predefined pure types of OBO. That is, all other creations take place in the box of the this-object. A more general creation scheme is desirable for programmers, but would complicate our kernel language and is beyond the scope of this presentation. Instances can be created as confined or exposable. References to confined instances may not leave the box in which the instance is created whereas exposable instances may be passed out. More precise explanations of the expressions are given together with the semantics.

Semantics. A box could be represented by the owner object that is created together with the box. This is certainly an appropriate solution for implementations and is essentially the idea of Boogie where ownership is represented by ghost variables. Here, we use an approach with explicit box instances. As formalized in Fig. 7, a box is either the predefined global box `globox` or a *created* box. A created box has a *parent* box from which it was created. The transitive reflexive closure of the child-parent relation is called the inclusion relation, denoted by $\subseteq$. In particular, we have for any created box b: $b \subset parent(b) \subseteq \texttt{globox}$. The creation of boxes is formalized similar to object creation. We assume that there is a sufficiently large set of boxes from which we choose and allocate a box. Details are described together with the operational semantics below.

Each box b is confined in one of its ancestors. This so-called *confining* box is denoted by $confIn(b)$. The meaning of the confinement is that boundary objects

$box : \mathbf{Object} \to \mathbf{Box}$
$box\ (b, \_, \_; \_) = b$

$thisBox : \mathbf{StackFrame} \rightharpoonup \mathbf{Box}$
$thisBox(F) = box(F(this))$

$receivingBox : \mathbf{lm} \times \mathbf{StackFrame} \to \mathbf{Box}$
$receivingBox(global, F) = \texttt{globox}$
$receivingBox(local, F) \ = thisBox(F)$

$passable : \mathbf{Value} \times \mathbf{Box} \times \mathbf{A} \to \mathbf{Bool}$
$passable(null, b_r, a) = true$
$passable(\langle (b, C, \_; conf), \_, \_\rangle, b_r, external) = (br \leq b)$
$passable(\langle (b, C, \_; expo), \_, \_\rangle, b_r, external) = (br \leq confIn(b))$
$passable(\langle (b, C, \_; conf), \_\rangle, b_r, boundary) = (b = br)$
$passable(\langle (b, C, \_; expo), \_, \_\rangle, b_r, boundary) = (br \leq confIn(b) \wedge b \leq br)$

$adapt : \mathbf{Value} \times \mathbf{S} \times \mathbf{Box} \times \mathbf{A} \to \mathbf{Value}$
$adapt(null, S, br, a) = null$
$adapt(\langle o, \_, \_\rangle, S, br, external) = \langle o, S, extn\rangle$
$adapt(\langle (b, C, j; em), \_, \_\rangle, S, br, boundary) = \langle (b, C, j; em), C, intn\rangle$ if $br = b$
$= \langle (b, C, j; em), S, extn\rangle$ otherwise

Fig. 8. OBO's auxiliary functions.

of b may not be accessed from outside $confIn(b)$. The confining box will be determined at box creation. We assume that we can create new boxes for any given parent with an appropriate confining box.

An object is represented as a triple consisting of its box, its class, and an identifier to distinguish objects of the same class in a box. Furthermore, an object carries the information whether it is confined to its box b or whether it can be exposed (see Fig. 7). In the latter case, it is confined to $confIn(b)$. To distinguish between internal and external calls, we access objects over explicitly modeled references. A reference is a triple consisting of the referenced object, a type, and a reference kind. The type is a supertype of the object type and is used to prevent illegal downcasts. The reference kind distinguishes between internal objects, and objects that are considered external. (Notice, that this is similar to external references in JavaCard applets [34]; see as well Sect. 4).

Figure 7 contains as well the definitions of heaps and stack frames. Heaps map boxes to box-local state reflecting the structuring of the store. Figure 9 presents the big-step operational semantics for OBO. The underlying judgment $H, F \vdash e \Rightarrow v, H'$ expresses that the evaluation of an expression e in a state with heap H and stack frame F has v as result and H' as resulting heap. The

$$\frac{F(x) = v}{H, F \vdash x \Rightarrow v, H} \text{(E-VAR)}$$

$$\frac{H, F \vdash e \Rightarrow null, H_0}{H, F \vdash (T)\ e \Rightarrow null, H_0} \text{(E-CAST NULL)}$$

$$\frac{H, F \vdash e \Rightarrow r, H_0 \qquad r = \langle \_, T_1, \_ \rangle \qquad T_1 \preceq T}{H, F \vdash (T)\ e \Rightarrow r, H_0} \text{(E-CAST OBJ)}$$

$$\frac{\begin{array}{c} b \notin dom(H) \qquad b_0 = receivingBox(lm, F) \qquad parent(b) = b_0 \\ confIn(b) = \text{if } em = conf \text{ then } b_0 \text{ else } confIn(b_0) \qquad class(B) = C \\ o = (b, C, j; expo) \qquad fields(C) = \overline{T}\ \overline{f} \qquad BS = \{o \mapsto \langle \overline{null} \rangle\} \text{ with } |\overline{null}| = |\overline{f}| \end{array}}{H, F \vdash \texttt{new}\ lm\ em\ B() \Rightarrow \langle o, B, \texttt{extn} \rangle, H[b \mapsto BS]} \text{(E-NEW BOX)}$$

$$\frac{\begin{array}{c} b = thisBox(F) \qquad o = (b, C, j, em) \\ o \notin dom(H(b)) \qquad fields(C) = \overline{T}\ \overline{f} \qquad BS = H(b)[o \mapsto \langle \overline{null} \rangle] \text{ with } |\overline{null}| = |\overline{f}| \end{array}}{H, F \vdash \texttt{new local}\ em\ C() \Rightarrow \langle o, C, \texttt{intn} \rangle, H[b \mapsto BS]} \text{(E-NEW OBJ)}$$

$$\frac{H, F \vdash e \Rightarrow \langle o, C, \texttt{intn} \rangle, H_0 \qquad fields(C) = \overline{T}\ \overline{f} \qquad H_0(box(o))(o) = \overline{v}}{H, F \vdash e.f_i \Rightarrow v_i, H_0} \text{(E-FIELD)}$$

$$\frac{\begin{array}{c} H, F \vdash e_0 \Rightarrow \langle o, C, \texttt{intn} \rangle, H_0 \qquad fields(C) = \overline{T}\ \overline{f} \\ b = box(o) \qquad H_0(b)(o) = \overline{v} \qquad H_0, F \vdash e_1 \Rightarrow v, H_1 \qquad BS = H_1(b)[o \mapsto [v/v_i]\overline{v}] \end{array}}{H, F \vdash e_0.f_i = e_1 \Rightarrow v, H_1[b \mapsto BS]} \text{(E-FIELDUP)}$$

$$\frac{\begin{array}{c} H, F \vdash e \Rightarrow \langle o, C, \texttt{intn} \rangle, H_0 \qquad H_0, F \vdash e_1 \Rightarrow v_1, H_1 \cdots H_{n-1}, F \vdash e_n \Rightarrow v_n, H_n \\ mbody(m, C) = \overline{x}.e_b \qquad H_n, \{this \mapsto \langle o, C, \texttt{intn} \rangle, \overline{x} \mapsto \overline{v}\} \vdash e_b \Rightarrow v_m, H_m \end{array}}{H, F \vdash e.m(\overline{e}) \Rightarrow v_m, H_m} \text{(E-INVK INTN)}$$

$$\frac{\begin{array}{c} H, F \vdash e \Rightarrow \langle o_0, S, \texttt{extn} \rangle, H_0 \\ o_0 = (b_r, C, \_; em) \qquad a_m\ S_m\ m(a_1\ S_1, \ldots, a_n\ S_n)\ signatureOf\ m\ in\ S \\ H_0, F \vdash e_1 \Rightarrow v_1, H_1 \quad \cdots \quad H_{n-1}, F \vdash e_n \Rightarrow v_n, H_n \\ passable(v_1, b_r, a_1) \quad \cdots \quad passable(v_n, b_r, a_n) \qquad mbody(m, C) = \overline{x}.e_b \\ H_n, \{this \mapsto \langle o_0, C, \texttt{intn} \rangle, \overline{x} \mapsto \overline{adapt(v_i, T_i, b_r, a_i)}\} \vdash e_b \Rightarrow v_m, H_m \\ passable(v_m, thisBox(F), \texttt{external}) \end{array}}{H, F \vdash e.m(\overline{e}) \Rightarrow adapt(v_m, T_m, thisBox(F), a_m), H_m} \text{(E-INVK EXTN)}$$

$$\frac{H, F \vdash e_0 \Rightarrow v_0, H_0 \qquad H_0, F[x \mapsto v_0] \vdash e \Rightarrow v, H_1}{H, F \vdash \texttt{let}\ x = e_0\ \texttt{in}\ e \Rightarrow v, H_1} \text{(E-LET)}$$

Fig. 9. Rules of OBO's operational semantics.

semantics uses auxiliary functions from Fig. 8. In the following, we discuss the interesting rules.

Rule (E-CAST OBJ) shows how the distinction between references and objects is used to restrict downcasts. A downcast is only allowed to the type of the reference, not to the type of the object.

Rule (E-NEW BOX) captures the creation of a box instance b together with its owner object o. A possible approach would have been to make the functions *parent* and *confIn* part of the state and enlarge their domain whenever a box is allocated. As our core language does not support modification of the box structure after box creation, we use a simpler formalization that avoids this. We assume a rich box structure with an infinite carrier set $\mathbf{CreatedBox} \cup \{\texttt{globox}\}$ and functions *parent* and *confIn* having the following properties:

1. *parent* defines a tree structure on **CreatedBox** with root `globox`.
2. For all nodes b and all ancestor nodes b_a of b, the set of b's children with $\mathit{confIn}(b) = b_a$ is infinite (where the ancestor relation is the transitiv reflexive closure of the parent relation).

Box creation means to choose a box b from the box structure that is not yet part of the heap. Let b_0 be the receiving box which is either the global box (recall that this is only allowed for pure types) or the box of the current this-object. The new box b is chosen such that its parent is b_0 and its owner is confined either in b_0 or in $\mathit{confIn}(b_0)$ depending on the encapsulation modifier *em*. The second property above guarantees that such a new box always exists. Rule (E-NEW OBJ) shows the creation of objects. Objects may only be created locally.

Rules (E-FIELD) and(E-FIELDUP) show that read and write accesses are only allowed through internal references. Below we show that expressions only evaluates to internal references, if the referenced object is in the same box as the current this-object.

Rules (E-INVK INTN) and (E-INVK EXTN) describe internal and external calls. An internal call is an ordinary dynamically bound method call. The function *mbody* yields the parameters and body of method m in class C. The receiver object of an external call might belong to a different box than the current this-object. Thus, values can be passed from one box to another. The value *null* can always be passed. To check whether a reference r is passable, we distinguish two cases:

- r is considered external in the receiving box b_r: Then the confined box of r's object has to include b_r (otherwise, confinement is violated).
- r should be boundary in the receiving box b_r: Then, it has to be checked that r's object is included in b_r (otherwise it cannot be a boundary object of b_r) and that the confined box of r's object includes b_r (otherwise, confinement is violated).

If a reference is passable, it has to be adapted to the new box. The adapted reference is an internal reference if the parameter annotation is `boundary` and the receiving box is equal to the box of the referenced object. Otherwise, it is an external reference.

Execution of an OBO program starts with a configuration containing an object X of class `Main` in the global box and a stack frame F for executing method `main` with $F(this) = X$ and $F(arg)$ referencing the input string. The result of executing the expression in the body of `main` is the result of the program execution.

Properties. The box semantics has two central properties: Internal references never leave their box, and objects are never referenced from outside their confining box. To formulate these properties precisely, we interpret each semantic rule as a recursive procedure that takes the heap, stack frame, and expression as parameters and returns the expression value and heap as results. The antecedents of the rule constitute the body of the procedure, a sequence of checks and recursive calls. Execution of an OBO program corresponds to a call tree of these recursive procedures. A *pre-configuration* consists of the input heap and stack frame and the value null, a *post-configuration* consists of the result heap, the input stack frame, and the result value. A configuration is either a pre- or post-configuration. For all configurations (H, F, v) of a OBO program execution the following holds:

- Internal references never leave their box, that is:
  if $H(b)(o_0) = \overline{v}$ with $v_i = \langle o, T, intn \rangle$, then $box(o) = b$;
  if $thisBox(F) = b$ and $F(x) = \langle o, T, intn \rangle$, then $box(o) = b$;
  if $thisBox(F) = b$ and $v = \langle o, T, intn \rangle$, then $box(o) = b$.
- An object o is never referenced from outside its confining box $b_{cf(o)}$:
  if $H(b)(o_0) = \overline{v}$ with $v_i = \langle o, \_, \_ \rangle$, then $b \subseteq b_{cf(o)}$;
  if $thisBox(F) = b$ and $F(x) = \langle o, \_, \_ \rangle$, then $b \subseteq b_{cf(o)}$;
  if $thisBox(F) = b$ and $v = \langle o, \_, \_ \rangle$, then $b \subseteq b_{cf(o)}$.

A proof sketch of these properties is contained on the appendix. A further discussion of the box model is given in Sect. 4.

3 Modular Specification of Box Interfaces

We have seen that a box is a generalization of an object. Consequently, we reuse language concepts for class specifications, in particular from JML [18], and for refinement of object-oriented programs, in particular from [7]. Our contributions in this area are the extension of such techniques to the box model and full implementation independence which is important in a component setting. We present the specification technique along with the examples introduced in Sect. 1 and explain central aspects of box specifications.

The specification technique is concerned with the internal state of the box, with access to a box from the enviroment, and with accesses of the box to the enviroment. A box specification addresses four aspects:

- Declarative object-oriented models that are used to express box state and method behavior in an implementation independent way.

- Specification of encapsulation aspects consisting of method argument and result annotations and a mechanism to keep track of the references that are exposed to the environment and that refer to external objects.
- The internal state of a box as described by model variables of its boundary objects. Model variables are specification only variables that are visible to the client of the box. The initial state of a box is described together with the constructor. The state space can be restricted by invariants.
- The behavior of methods contained in exposed interfaces. In addition to declarative pre-post specifications, the technique supports abstract statements and a simple approach to achieve modularity in the context of reentrant calls.

In the following, we first introduce declarative models. Then, we explain specifications for boxes that can only be accessed through their owner, focusing on the specification of external calls. Finally, we show how multiple-access boxes can be handled.

3.1 Declarative Models

To be applicable to components, specifcations should not refer to implementation parts. We achieve implementation independency by using declarative models. A declarative model provides the types, values, and mathematical functions to explain the state space of a box. A specification framework usually provides standard models for datatypes like sets and lists. Other models may be component specific and must be developed by the component developer. For the following, we assume the parametric datatypes `Set<A>` and `List<A>` as standard models. We use Java-like notation for the interfaces and indicate by the keywork `pure` that method calls of these types have no side-effects.

```
pure interface Set<T> {
  static Set<T> empty();
  boolean contains(T elem);
  Set<T> insert(T elem);
  Set<T> delete(T elem);
}
```

```
pure interface List<T> {
  static List<T> empty();
  boolean  contains(T elem);
  T        nth(int index);
  int      length();
  List<T>  appendLast(T elem);
  List<T>  delete(T elem);
  Set<T>   toSet();
}
```

As an example for a component specific model, we consider the declarative model the `ObservableGame` interface of Fig. 4. Moves are described by a string and are checked and converted to values of type `MoveDescr` by a function `mkMoveDescr`. The type `Position` models the positions of the game and provides functions for the initial position, for checking whether a move is legal in a position, and for yielding the new position after a move.

```
pure interface Color {
  static Color WHITE = new Color();
  static Color BLACK = new Color();
  Color other();
  // WHITE.other() == BLACK
  // BLACK.other() == WHITE
}

pure interface MoveDescr {
  static MoveDescr mkMvDescr(String s);
}
```

```
pure interface Position {
  static  Position initial ();
  boolean legal( MoveDescr md );
  Position doMove(MoveDescr md);
  // if this.legal(md),
  //    return position after move,
  // else return this
}
```

Different techniques can be used to specify declarative models. As declarative models are simpler than state-based models with side-effects, an informal description is often sufficient. JML uses functional programming [17]. The Larch approach uses abstract datatypes ([15]). We applied logic-based specification techniques in previous work ([31]).

3.2 Specifying Single-Access Boxes

A box that can only be accessed through its owner is called a *single-access box*. Such a box exposes only one reference to clients, that is, it essentially behaves like one object (of course, it can be implemented by many objects). We explain the specification technique along with the specification of `ObservableGame` in Fig. 10.

Encapsulation. As explained in Sect. 2, encapsulation is based on the box model of the underlying programming language. From the method signatures in Fig. 10, one can see that only the contructor exposes a reference, namely the owner, and that only method `register` imports external non-pure references to the box. To keep track of imported references, the specification technique supports the declaration of specification-only ghost variables using the keyword `references`. The variables are either of a type T or of a type $T*$ where T is a reference type. In the latter case, the variable stores sets of references of type T (we do not use type `Set<T>` to distinguish between references to sets of T and sets of references to T-objects).

Box State. The state of a box consists of the internal state of the box and the set of external and exposed references. For single access boxes, the internal box state can be associated with the owner. The state space is expressed by so-called *model variables*. Model variables are similar to instance variables, but are specification-only variables. As shown in Fig. 10, the state of `ObservableGame`-boxes is captured by two model variables. Variable `currentPos` holds the current position of the game and variable `player` records the color of the human player.

The initial state of a box after termination of the constructor is described together with the constructor (cf. Fig. 10). The techniques for specifying constructors are the same as those for methods and are explained below.

```
box interface ObservableGame
{
  references Observer* gameObs;
  model Color player;
  model Position currentPos;

  ObservableGame()
    ensures  player == Color.WHITE && currentPos == Position.initial()
         &&  gameObs == Set.empty() ;

  pure Position readPos()
    ensures result == currentPos ;

  void register( external Observer go )
    requires go != null
    ensures  gameObs == pre(gameObs).insert(go)
         &&  unchanged([currentPos,player]) ;

  void move( MoveDescr md )
    behavior if( currentPos.legal(md) ) {
               currentPos = currentPos.doMove(md);
               foreach( o in gameObs ){ o.stateChanged(); }
               forsome( cmd : currentPos.legal(cmd) ){
                 currentPos = currentPos.doMove(cmd);
                 foreach( o in gameObs ){ o.stateChanged(); }
               }
             }
    ensures  unchanged([player,gameObs]) ;
    invokable this.readPos() ;

  void swapPlayers()
    behavior player = other(player);
             forsome( cmd : currentPos.legal(cmd) ){
               currentPos = currentPos.doMove(cmd));
               foreach( o in gameObs ){ o.stateChanged(); }
             }
    ensures  unchanged([gameObs]) ;
    invokable this.readPos() ;
}

interface Observer {
  void stateChanged()
    behavior arbitrary ;
}
```

Fig. 10. Behavioral specification of box ObservableGame.

Method Behavior. According to the semantics of Sect. 2, the general execution behavior of a method m on box B can be described by a list of execution segments. A *segment* is either internal or external. Internal segments consist of box internal execution actions not containing an external call. External segments correspond to external calls. They are indexed by the receiver object and the sent message. External segments are called *simple* if they do not cause a reentrant call to B. Otherwise each call back to B is again described by a list of execution segments. The specification of a method should describe:

- the precondition, if any[3],
- the *local effects*, that is, the modifications of the box state,
- the *frame effects*, that is, the external calls to the box environment, and
- restrictions on reentrant calls.

The new aspect of our technique is the treatment of effects and frame properties. We distinguish between local and external effects. Local effects may be underspecified. That is, a method may modify more than the specification reveals. For external effects, it is the other way round. A method may only perform external calls that are mentioned in the specification. Thus, a specification provides a guarantee that no effects or only certain effects can happen.

We use language constructs from precondition-postcondition-style specifications and from refinement-style specifications. The precondition, indicated by the keyword `requires`, is given by a boolean expression (e.g. the precondition of method `register` in Fig. 10 requires that the argument is non-null). Methods that do not change the box state and have no effects on the box environment can be declared as pure (e.g. the method `readPos` in Fig. 10 is a pure method). The local and frame effects of a method are specified by an ensures clause or a behavior clause.

As usual, an *ensures clause* is expressed by a boolean expression. It may use the prestate values of *box-local variables*, denoted by an application of the operator `pre` to the variable name, and their poststate values as well as the return value of the method, denoted by the identifier `result`. As abbreviation, we use the operator `unchanged` taking a list of variables and stating that the value of the variables is the same in pre- and poststate (the specification of `register` illustrates its use).

A *behavior clause* consists of an abstract statement describing the internal segments of the method and when external calls happen. Abstract statements might be nondeterministic. For example, the specification of `move` in Fig. 10 calls method `stateChanged` for each registered observer. The order of the calls is not fixed by the foreach-statement (compare [7]). We support as well a nondeterministic forsome-statement: If there exists an element satisfying the given predicate, one such element is chosen and the body is executed. Otherwise, the body is not executed. The completely unknown behavior is denoted by arbitrary (the specification of `stateChanged` provides a typical example).

[3] A missing precondition is equivalent to precondition `true`.

A method execution satisfies a behavior clause if it implements one of the possible behaviors. The order of internal actions is not relevant as long as the specified state before an external call is correct. An ensures clause may be added to a behavior clause, to explicitly state properties that can be derived from the behavior clause. It is not allowed to specify additional properties. Methods `move` and `swapPlayers` show a combination of behavior and ensures clauses.

In connection with the box model, specifications based on abstract statements provide a new solution to the frame problem. The effects of a method are seperated into two parts: the box-local effects and the external effects. Local effects may be underspecified, that is, the behavior clauses need not completely determine the state changes. In particular, subboxes (not treated in this paper) can refine the local effects, for example w.r.t. extended state. We could have used modifies clauses to express what remains unchanged within a box. However, as the abstract state space of the box is known, one can as well simply list what remains unchanged (see for example the ensures clause of method `move` in Fig. 10). The real difference between our approach and the modifies-clause approach to the frame problem concerns the modifications to the environment. In our approach, a method specification has to define all possible external calls. In particular, if no external call is specified, it is a guarantee that there are no effects to the environment. In the case of external modifications, the modifies-clause approach needs to describe the unknown effected state by some abstract variables and later specify the dependencies between the abstract variables and the concrete environment. This is often a difficult task. On the other hand, the verification techniques for modifies clauses are more advanced than for our approach based on external calls.

Finally, a method specification can limit the acceptable reentrant calls. Without an invokable clause, all reentrant calls are allowed. If an invokable clause is given, like, for example, in the specification of the methods `move` and `swapPlayers` in Fig. 10, only the listed reentrant calls are acceptable. If the client of the box does not prevent inacceptable reentrant calls, the box need no longer satisfy its contract. As shown in Subsect. 1.2, the restriction on reentrant calls is needed for modularity.

3.3 Specifying Multiple-Access Boxes

Boxes with multiple boundary objects and boundary objects of different types provide additional specification challenges. We will focus here on three aspects:

1. Control of the exposed references to boundary objects.
2. Box state with multiple objects and invariants.
3. Internal method calls in specifications.

We explain these aspects along with the `IterList` example (see Fig. 11 and 12). The specification essentially formulates the behavior of the Java class `LinkedList` (see [35]).

```
box interface IterList  {
  exposes    Iterator* iters;
  references Object* elems;
  model List<Object> value;

  IterList ()
    ensures value==List.empty() && iters==Set.empty() && elems==Set.empty();

  invariant
    forall( it1, it2 in iters ){ (it1.pos<=it2.pos && !it1.valid) ==> !it2.valid }

  pure external Object get( int index )
    requires 0 <= index && index < value.length();
    ensures  result == value.nth(index);

  void add( external Object x )
    behavior foreach( it in iters ){
               if( it.pos == value.length() ) it.valid = false;
             }
             value = value.appendLast(x);
             elems = toSet(value);

  external Object remove( int index )
    requires 0 <= index && index < value.length()
    behavior foreach( it in iters ){
               if( it.pos >= index ) it.valid = false;
             }
             result = value.nth(index);
             value  = value.delete(index);
             elems  = value.toSet();

  pure boolean contains ( external Object x )
    behavior if( x == null ) {
               result = value.contains(null);
             } else {
             result = false;
             foreach( y in value ) {
               if( x.equals(y) ) {
                 result = true; break;
             } } }

  Iterator listIterator ()
    ensures !pre(iters).contains(result)  && iters.contains(result)
         && result.myList == this         && result.pos == 0
         && result.valid == true          && result.hasCurrent == false ;
}
```

Fig. 11. Behavioral specification of box `IterList`.

Controling Exposed References. In addition to its owner, a box can expose other boundary objects. Similar to the outgoing references (e.g., variable `elems` keeps track of outgoing references in the specification of Fig. 11[4]), a specification has to control exposed references in special ghost variables declared with the keyword `exposes`. For example, the `IterList` specification captures the set of exposed iterators in variable `iters` (see Fig. 11). Updates of the exposes variables are described in the method specifications; we say that exposed references are *registered*. An implementation satisfies an exposes specification if only registered references are exposed.

```
interface Iterator
{
  model IterList myList;
  model int pos;
  model boolean valid;
  model boolean hasCurrent;

  invariant
    ( 0 <= pos ) && ( valid ==> pos <= myList.value.length() )

  pure boolean hasNext()
    requires valid
    ensures  result == (pos < myList.value.length())

  external Object next()
    requires valid  && (pos < myList.value.length())
    behavior result = myList.get(pos);
             pos++;
             hasCurrent = true;

  void remove()
    requires valid && hasCurrent
    behavior myList.remove(pos);
}
```

Fig. 12. Behavioral specification of interface `Iterator`.

[4] We enforce that all outgoing, non-pure references are registered. We investigate whether it is sufficient to register only references that are used in external calls. That could reduce the specification overhead; in particular, the ghost variable elems would become dispensable.

Box State Revisited. Whereas in single-access boxes the box state can be associated with the owner object, this is not appropriate for multiple-access boxes. For a client of the box, it is more natural to distribute the box state over the boundary objects, in particular, because newly created boundary objects often make it necessary to extend the state space. For the `IterList` example, distributing the state means that part of the box state is associated with the owner (namely the represented list, see model variable `value` in Fig. 11) and other parts are associated with the iterators (namely the current iterator position in the list and the information whether an iterator is valid and has a current element; see model variables `pos`, `valid`, and `hasCurrent` in Fig. 12).

In addition to the state of the boundary objects, the relationship between boundary objects has to be modeled. As demonstrated by the model variable `myList` in Fig. 12, references between boundary objects can be used. It is allowed to access the state of referenced objects in specifications, because the references are encapsulated within the box. For example, the invariant of the `Iterator` interface accesses the value of the associated list and the invariant of the `IterList` interface accesses the exposed iterators.

The box model provides a fairly straightforward semantics for invariants. The invariants of a box B have to hold whenever execution is outside B, that is, in any configuration with $box(F(this)) \not\subseteq B$. In particular, they have to hold before outgoing calls. Note that there may as well be reentrant calls from inner boxes (although, one can argue that this is bad programming style). However, these calls are under the control of the box designer. To achieve modularity, invariants may only refer to box local state. This is for example true for the invariants given in Fig. 11 and 12. The requirement is not satisfied by the invariants given in Fig. 4. The references clause in `GameObserver` states that the observed game `obsGame` is outside the box. Thus, its fields may not be used in the invariants. To handle such invariants and the one given on page 9 in our approach, one has define a box `GamingSystem` that encapsulate `ObservableGame` and `GameObserver`. This can be a very lightweight box providing only methods for creating observable games and game observers. It would be a multiple-access box having observable games and game observers as boundary objects. The invariants in question would become part of the box specification of `GamingSystem`.

Internal Calls. Whereas the support of external calls is necessary to handle effects and non-effects to the environment in an abstract way, internal calls are important to structure interface specifications and to improve reuse. A good example is the `remove` method of the iterator in Fig. 12. The behavior is simply described by an internal call to the `remove` method of the list. If this was not allowed, one would have had to describe the complex behavior of the latter method twice. Notice that the specification of the remove method in IterList demonstrates as well how the exposes variables can be used to describe effects to all boundary objects.

Distributing the box state and supporting internal calls over the boundary objects has the additional advantage that interface specifications can be used for different boxes. For example the Iterator-specification of Fig. 12 could be

used for different collection boxes if the model variable `myList` is generalized to a more abstract type.

4 Discussion and Related Work

In this section, we relate the presented techniques to existing work and shortly discuss our design decisions. The section is structured according to the main aspects of the specification technique.

Encapsulation. Encapsulation is a central technique to achieve modularity ([26] discusses the relation between encapsulation- and visibility-based modularity). The current techniques for object-based encapsulation build on ownership concepts (see [9, 8]). The basic idea is to guarantee the owner-as-dominator property: All reference chains from an object in the global context to an object X in a non-global context go through X's owner. The box model incorporates two extensions to the original ownership model:

- In [6, 8], objects of inner classes are allowed as boundary objects. We generalized this to arbitrary boundary objects. At the moment, we do not have a checking technique for our full model. In particular, we cannot support programs in which boundary objects are passed back into a box as explicit parameters. Our currently developed checking approach (see [33]) builds on a variant of ownership domains [2].
- The universe type system [23, 24] realizes a slightly different discipline (see [11]) and controls references leaving a context. Such references have to be read-only in the universe model.

There are different approaches to specify and check ownership. Most of the work cited above uses type systems. Many aspects of our approach are inspired by the assertion-based ownership technique of Boogie [5]. Within Boogie, each object X has a ghost variable referencing X's owner. These ghost variables can be used for specification and verification purposes and support ownership transfer [19].

Our work was also inspired by techniques for applet isolation in Java Card. To control accesses from other applets, Java Card allows such external accesses only through so-called shareable interfaces. Dynamic checks enforce this condition at runtime [34]. The semantics that was used in a paper on static checking of applet isolation has already separated references and objects and supported context information in the references [12]. The distinction between internal and external references is also motivated by role models for objects, in particular those described in [28].

Invariants and Reentrant Calls. Invariants are a central specification technique, for both hidden implementation aspects and visible properties of the abstract state of a component. Unfortunately, it is fairly difficult to provide a modular semantics for them in object-oriented programming. In [26], Müller et

al. investigate the most common semantic approach based on so-called visible states. The visible state semantics requires that invariants have to hold in pre- and post-states of all calls to public methods. They show that a naive visible state semantics does not work for layered objects structures and provide a refined version. This so-called relevant invariant semantics inspired our approach in which box invariants have to hold whenever execution is outside the box. Our semantics is simpler as it does not enforce invariants to hold in pre- and poststates of internal method calls.

The Boogie methodology supports a more dynamic approach. The execution points at which invariants have to hold are expressed by special specification constructs ([5, 27]) that are placed in the program. This makes the approach difficult to use for an implementation independent setting. A very strong feature of the Boogie approach is its flexible support for inheritance.

To extend our box-based invariant semantics and our basic mechanism to exclude unwanted reentrant calls, we can build on work on typestates [10] and on friend concepts [4]. Typestates can be used to represent sets of invokable methods that allow reentrance under certain specified conditions. The friends concept would enable us to support invariants that access state outside the box (like the invariants in Fig. 4).

Frame Problem. The specification of frame properties is a notoriously difficult problem. The main source of the problem for object-oriented programming is the weak knowledge about the effects or non-effects in the context of extendible state and virtual methods (recall the `ObservableGame` example from Fig. 4). Most existing solutions are based on abstraction techniques and lists of variables that might be modified ([20, 23, 21, 25]). Specifications based on abstract statements do not explicitly list the modifiable variables, but associate modifications with methods ([7]). As described in Subsect. 3.2, we use the same approach for external effects.

5 Conclusions and Future Work

We presented an object-oriented kernel language OBO that supports a notion of dynamically created encapsulation regions called boxes. The implementation of a box can create and encapsulate other boxes. The box model distinguishes between internal and external references and supports object confinement.

In this paper, we used boxes as a semantic foundation for object-oriented software components. We described a behavioral specification technique for component instances that supports dynamically created interface objects and external modifying calls that might lead to reentrance.

In summary, we consider the presented work a further step to close the gap between specification techiques for programs and implementation independent specifications for components. Directions of future work include:

- the extension to all features of object-orientation; in particular, a behavioral subtype relation between box interfaces is needed;

- the development of powerful checking techniques;
- the adaption of existing verification techniques to the box model;
- the exploitation of boxes for higher-level concurrency models.

In addition, we are interested in more theoretical aspects like representation indepence (cf. [3]), calculi for OO-programming [1] and components [32], and questions related to specification completeness.

Acknowledgement. We thank the reviewers and Ina Schaefer for their helpful comments on previous versions of this paper.

References

1. Erika Ábrahám, Marcello M. Bonsangue, Frank S. de Boer, Andreas Grüner, and Martin Steffen. Observability, connectivity, and replay in a sequential calculus of classes. In Frank S. de Boer et. al., editors, *Formal Methods for Components and Objects, FMCO 2004*, volume 3657 of *LNCS*, pages 296–316. Springer-Verlag, 2005.
2. Jonathan Aldrich and Criag Chambers. Ownership domains: Separating aliasing policy from mechanism. In Odersky [29], pages 1–25.
3. Anindya Banerjee and David A. Naumann. Representation independence, confinement and access control. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'02)*, pages 166–177. ACM Press, January 2002.
4. Michael Barnett and David A. Naumann. Friends need a bit more: Maintaining invariants over shared state. In Dexter Kozen and Carron Shankland, editors, *Mathematics of Program Construction, 7th International Conference, MPC 2004*, volume 3125 of *Lecture Notes in Computer Science*, pages 54–84. Springer-Verlag, 2004.
5. Mike Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6), 2004.
6. Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In OOPSLA'02 [30], pages 211–230.
7. Martin Büchi. *Safe Language Mechanisms for Modularization and Concurrency.* PhD thesis, Turku Centre for Computer Science, May 2000.
8. Dave Clarke. *Object Ownership and Containment.* PhD thesis, University of New South Wales, July 2001.
9. Dave Clarke, John Potter, and James Noble. Ownership types for flexible alias protection. In *Proceedings of the 13th ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA'98)*, pages 48–64. ACM Press, October 1998.
10. Robert DeLine and Manuel Fähndrich. Typestates for objects. In Odersky [29], pages 465–490.
11. Werner Dietl and Peter Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology*, 4(8):5–32, 2005.
12. Werner Dietl, Peter Müller, and Arnd Poetzsch-Heffter. A type system for checking applet isolation in Java Card. In *Construction and Analysis of Safe, Secure and Interoperable Smart devices (CASSIS 2004)*, volume 3362 of *Lecture Notes in Computer Science*. Springer-Verlag, March 2004.

13. Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. A programmer's reduction semantics for classes and mixins. *Formal Syntax and Semantics of Java*, 1523:241–269, 1999.
14. James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java^{TM} Language Specification – Second Edition*. Addison-Wesley, June 2000.
15. J. V. Guttag and J. J. Horning. *Larch: Languages and Tools for Formal Specification*. Texts and Monographs in Computer Science. Springer-Verlag, 1993.
16. Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(3):396–450, May 2001.
17. Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A notation for detailed design. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, chapter 12, pages 175–188. Kluwer, 1999.
18. Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML. Technical Report No. 98-06z, Iowa State University, 2004.
19. K. R. M. Leino and P. Müller. Object invariants in dynamic contexts. In Odersky [29], pages 491–516.
20. K. Rustan M. Leino and Greg Nelson. Data abstraction and information hiding. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 24(5):491–553, 2002.
21. K. Rustan M. Leino, Arnd Poetzsch-Heffter, and Yunhong Zhou. Using data groups to specify and check side effects. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation (PLDI'02)*, pages 246–257. ACM Press, June 2002.
22. Microsoft. *C# Language Specification*. 2001.
23. Peter Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.
24. Peter Müller and Arnd Poetzsch-Heffter. Universes: A type system for alias and dependency control. Technical Report 279–1, Fernuniversität Hagen, 2001.
25. Peter Müller, Arnd Poetzsch-Heffter, and Gary T. Leavens. Modular specification of frame properties in JML. *Concurrency and Computation: Practice and Experience*, 15(2):117–154, 2003.
26. Peter Müller, Arnd Poetzsch-Heffter, and Gary T. Leavens. Modular invariants for layered object structures. Technical Report 424, ETH Zürich, Chair of Software Engineering, 2005.
27. David A. Naumann. Assertion-based encapsulation, object invariants and simulations. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *Formal Methods for Components and Objects, Third International Symposium, FMCO 2004*, volume 3657 of *Lecture Notes in Computer Science*, pages 251–273. Springer-Verlag, 2005.
28. James Noble, Jan Vitek, and John Potter. Flexible alias protection. In Eric Jul, editor, *Proceedings of the 12th European Conference on Object-Oriented Programming (ECOOP'98)*, volume 1445 of *Lecture Notes in Computer Science*, pages 158–185. Springer-Verlag, July 1998.
29. Martin Odersky, editor. *Proceedings of the 18th European Conference on Object-Oriented Programming (ECOOP'04)*, volume 3086 of *Lecture Notes in Computer Science*. Springer-Verlag, June 2004.
30. *Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA'02)*. ACM Press, November 2002.

31. Arnd Poetzsch-Heffter. Specification and verification of object-oriented programs. Habilitationsschrift, Technische Universität München, 1997.
32. Riccardo Pucella. Towards a formalization for COM part I: the primitive calculus. In OOPSLA'02 [30], pages 331–342.
33. Jan Schäfer and Arnd Poetzsch-Heffter. Simple fuzzy ownership domains. Unpublished. Preliminary version. Available at `http://softech.informatik.uni-kl.de/~janschaefer`.
34. Sun Microsystems, Inc., Palo Alto, CA. *Java Card™ 2.1.1 Virtual Machine Specification*, May 2000.
35. Sun Microsystems, Inc. *Java™ 2 Platform, Standard Edition, v 1.4.2 API Specification*, 2003. `http://java.sun.com/j2se/1.4.2/docs/api/`.
36. Clemens Szyperski, Dominik Gruntz, and Stephan Murer. *Component Software — Beyond Object-Oriented Programming*. Addison-Wesley, second edition, 2002.

Appendix

The appendix present proof sketches for the properties given in Sect. 2.

Definition (Configuration). A *configuration* is a triple $\langle H, F, v\rangle$, consisting of a heap H, a stack frame F and a value v.

Definition (Intern-Closed Configuration). A configuration (H, F, v) is called *intern-closed*, denoted by $(H, F, v) \vdash \triangledown$, if the following conditions hold:

1. If $H(b)(o_0) = \overline{v}$ with $v_i = \langle o, \_, \texttt{intn}\rangle$, then $box(o) = b$
2. If $thisBox(F) = b$ and $F(x) = \langle o, \_, \texttt{intn}\rangle$, then $box(o) = b$
3. If $thisBox(F) = b$ and $v = \langle o, \_, \texttt{intn}\rangle$, then $box(o) = b$

Theorem 1. *If* $(H, F, null) \vdash \triangledown$ *and* $H, F \vdash e \Rightarrow v, H_0$ *then* $(H_0, F, v) \vdash \triangledown$

Note that initial configurations appearing in premises of evaluation rules are all *intern-closed*, which is (and needs to be) shown in the proof below.

Proof. Suppose $(H, F, null) \vdash \triangledown$ and $H, F \vdash e \Rightarrow v, H_0$. We show that $(H_0, F, v) \vdash \triangledown$. As F does not change, we only have to show conditions (1) and (3) of the intern-closed definition. We do an induction on the evaluation rules.

(E-VAR) Immediate.
(E-CAST NULL) By the induction hypothesis.
(E-CAST OBJ) By the induction hypothesis.
(E-NEW BOX) (1) follows by assumption $(H, F, null) \vdash \triangledown$ and the fact that $H(b)(o) = \overline{null}$. (3) follows because $v = \langle o, B, \texttt{extn}\rangle$.
(E-NEW OBJ) (1) follows by assumption $(H, F, null) \vdash \triangledown$ and the fact that $H(b)(o) = \overline{null}$. (3) follows from the premises $b = thisBox(F)$ and $o = (b, C, j, em)$.
(E-FIELD) By the induction hypothesis.
(E-FIELDUP) By the induction hypothesis we get $(H_0, F, \langle o, C, \texttt{intn}\rangle) \vdash \triangledown$ and $(H_1, F, v) \vdash \triangledown$. With the premise $b = box(o)$ it follows $thisBox(F) = b$. Let $v = \langle o_1, \_, \texttt{intn}\rangle$. Hence $thisBox(F) = box(o_1)$ and so $b = box(o_1)$. Thus $(H_1[b \mapsto BS], F, v) \vdash \triangledown$.

(E-INVK INTN) By the induction hypothesis we get $(H_0, F, \langle o, C, \texttt{intn}\rangle) \vdash \triangledown$. Hence $thisBox(F) = box(o)$. By repeated application of the induction hypothesis we get $(H_i, F, v_i) \vdash \triangledown$, for $0 < i \leq n$. Let $F_m = \{this \mapsto o, \overline{x} \mapsto \overline{v}\}$. Hence $thisBox(F_m) = thisBox(F)$ and so $(H_n, F_m, null) \vdash \triangledown$. Applying the induction hypothesis leads to $(H_m, F_m, v_m) \vdash \triangledown$. And finally, $(H_m, F, v_m) \vdash \triangledown$.

(E-INVK EXTN) By repeated application of the induction hypothesis we get $(H_0, F, \langle o_0, S, \texttt{extn}\rangle) \vdash \triangledown$ and $(H_i, F, v_i) \vdash \triangledown$, for $0 < i \leq n$. Let $F_m = \{this \mapsto \langle o_0, C, \texttt{intn}\rangle, \overline{x} \mapsto \overline{adapt(v_i, T_i, b, a_i)}\}$. Note that $thisBox(F_m) = b$. We have to show that $\forall x$ with $F_m(x) = \langle o_x, T_x, \texttt{intn}\rangle$. $box(o_x) = b$. The *adapt* function turns all references into `extn` references, except for the case, where $a_i = \texttt{boundary}$. In that case, however, $b = box(o_x)$. Hence $(H_n, F_m, null) \vdash \triangledown$. By the induction hypothesis, $(H_m, F_m, v_m) \vdash \triangledown$. Let $v_r = adapt(v_m, T_m, thisBox(F), a_m)$. The adapt function only returns an `intn` reference if $thisBox(F) = box(v_m)$, thus $(H_m, F, v_r) \vdash \triangledown$.

(E-LET) By the induction hypothesis.

□

Definition. $cf(o)$ is defined as follows.

cf : **Object** $\rightarrow$ **Box**
$cf(b, C, k; \texttt{confined}) = b$
$cf(b, C, k; \texttt{exposable}) = confIn(b)$

Definition (Confined Configuration). A configuration (H, F, v) is called *confined*, denoted by $(H, F, v) \vdash \Delta$, if the following conditions hold:

1. If $H(b)(o_0) = \overline{v}$ with $v_i = \langle o, \_, \_\rangle$, then $b \subseteq cf(o)$
2. If $thisBox(F) = b$ and $F(x) = \langle o, \_, \_\rangle$, then $b \subseteq cf(o)$
3. If $thisBox(F) = b$ and $v = \langle o, \_, \_\rangle$, then $b \subseteq cf(o)$

Theorem 2. *If* $(H, F, null) \vdash \Delta$ *and* $H, F \vdash e \Rightarrow v, H_0$ *then* $(H_0, F, v) \vdash \Delta$

Like above, the proof also shows that all initial configurations of rule premises are confined.

Proof. Suppose $(H, F, null) \vdash \Delta$ and $H, F \vdash e \Rightarrow v, H_0$. We show that $(H_0, F, v) \vdash \Delta$. As F does not change, we only have to show conditions (1) and (3) of the confinedness definition. We do an induction on the evaluation rules.

(E-VAR) Immediate.
(E-CAST NULL) By the induction hypothesis.
(E-CAST OBJ) By the induction hypothesis.
(E-NEW BOX) (1) follows by the fact that $H(b)(o) = \overline{null}$. (3): Depending on e_m, $confIn(b)$ is either b_0 or $confIn(b_0)$. Hence $cf(o)$ is either b, b_0, or $confIn(b_0)$. As $parent(b) = b_0$, it follows $b \subset b_0$. In addition, $b \subseteq confIn(b_0)$. Thus $b \subseteq cf(o)$.
(E-NEW OBJ) (1) follows by the fact that $H(b)(o) = \overline{null}$. (3): From the premise $box(o) = b$ we get $cf(o) \in \{b, confIn(b)\}$. Hence $b \subseteq cf(o)$.

(E-FIELD) (1) by the induction hypothesis. (3): Let $thisBox(F) = b$, for some b. By Theorem 1, $b = box(o)$. Hence $b \subseteq cf(o)$.

(E-FIELDUP) Let $thisBox(F) = b_1$, for some b_1. By Theorem 1, $b_1 = box(o)$. Hence by the premise $b = box(o)$, it follows $b_1 = b$. Let $v = \langle o_v, \_, \_ \rangle$. Hence,by Theorem 1, $box(o_v) = b$. Hence $b \subseteq cf(o_v)$. Thus $(H_1[b \mapsto BS], F, v) \vdash \Delta$.

(E-INVK INTN) By repeated application of the induction hypothesis we get $(H_i, F, v_i) \vdash \Delta$, for $0 < i \leq n$. Let $F_m = \{this \mapsto \langle o, C, \texttt{intn} \rangle, \overline{x} \mapsto \overline{v}\}$. Hence, $thisBox(F) = thisBox(F_m)$. Hence, $(H_n, F_m, null) \vdash \Delta$. We can apply the induction hypothesis and get $(H_m, F_m, v_m) \vdash \Delta$. Thus, $(H_m, F, v_m) \vdash \Delta$.

(E-INVK EXTN) By repeated application of the induction hypothesis we get $\underline{(H_i, F, v_i) \vdash \Delta}$, for $0 < i \leq n$. Let $F_m = \{this \mapsto \langle o_0, C, \texttt{intn} \rangle, \overline{x} \mapsto \underline{adapt(v_i, T_i, b, a_i)}\}$. The $passable(v_i, b, a_i)$ premises ensure that $b \subseteq box(v_i)$. Hence it follows $(H_n, F_m, null) \vdash \Delta$. We can apply the induction hypothesis and get $(H_m, F_m, v_m) \vdash \Delta$. Let $adapt(v_m, T_m, thisBox(F), \texttt{external}) = \langle o_r, \_, \_ \rangle$. We must show that $thisBox(F) \subseteq cf(o_r)$ But this is ensured by the premise $passable(v_m, T_m, thisBox(F), \texttt{external})$. Thus $(H_m, F, \langle o_r, \_, \_ \rangle) \vdash \Delta$.

(E-LET) By the induction hypothesis.

□