A Representation-Independent Behavioral Semantics for Object-Oriented Components

Arnd Poetzsch-Heffter and Jan Schäfer[⋆]

University of Kaiserslautern
{poetzsch,jschaefer}@informatik.uni-kl.de

Abstract. Behavioral semantics abstracts from implementation details and allows to describe the behavior of software components in a representation-independent way. In this paper, we develop a formal behavioral semantics for class-based object-oriented languages with aliasing, subclassing, and dynamic dispatch. The code of an object-oriented component consists of a class and the classes used by it. A component instance is realized by a dynamically evolving set of objects with a clear boundary to the environment. The behavioral semantics is expressed in terms of the messages crossing the boundary. It is defined as an abstraction of an operational semantics based on an ownership-structured heap. We show how the semantics can be used to define substitutability in a program independent way.

1 Introduction

The behavior of object systems is often described as a set of loosely coupled objects with encapsulated state that communicate via messages. However, this conceptual view is only partially reflected by existing object-oriented programming languages. Most of them are trimmed for efficient implementation of local computations. Their semantics is usually given in terms of state-transitions based on global heaps. As they do not support clear boundaries between parts of the heap, modular reasoning and abstraction of representation aspects is much more difficult than in a loosely coupled setting.

If runtime components have well-defined boundaries, their behavior can be completely defined in terms of their reaction to incoming message sequences. Considering only the messages that a client sends to a component makes the semantics independent from the representation of the component states. Such *behavioral semantics* has three advantageous properties:

1. Different component implementations can be compared based on the message behavior. Thus, an explicit coupling relation between the states of the implementations as it is needed in state-based approaches (see in particular the seminal paper [3] on representation independence for OO-programs) is not necessary. This simplifies the notion of behavioral substitutability for components.

[⋆] Supported by the Deutsche Forschungsgemeinschaft (German Research Foundation)

2. Behavioral semantics provides a suitable semantical basis for behavioral component specifications, i.e. for specifications that describe component behavior without referring to the implementation.
3. It simplifies modular analysis, because it is easier to abstract from the execution environment of the component. In particular, we can analyze component implementations without knowing their program contexts.

In this paper, we present an approach to behavioral semantics with the above properties for imperative object-oriented languages like Java and C# that support references, aliasing, subclassing, dynamic dispatch, and recursive types and methods. The main technical challenges were (a) to find a simple, yet powerful notion of runtime components, (b) to support callbacks, and (c) to use well-established semantical techniques for the definition of the behavioral semantics.

Approach and Overview. A runtime component in our approach is called a *box*. A box consists of an *owner object* and a set of other objects. A box is created together with its owner by instantiating the class of its owner. Boxes are tree-structured, that is, a box b can have so-called *inner* boxes. We distinguish two kinds of classes, normal classes and box classes (annotated by the keyword `box`). The instantiation of a normal class creates an object in the *current box*, that is, in the box of the current this-object. The instantiation of a box class creates a new inner box of the current box together with its owner. For simplicity, we do not support the direct creation of objects outside the current box. Such nonlocal creations can only be done by using a method. Note that this is similar to a distributed setting with remote method invocation.

Our approach only uses structural aspects of ownership (similar to [5]). It does not enforce confinement. In particular, our semantical model allows arbitrary references going into and out of a box (In this respect, it is more flexible than that of [3]). For type systems enforcing box confinement, we refer to [21].

The operational semantics for boxes distinguishes between local method calls and calls on objects of other boxes (Sec. 2). From this semantics, we develop a behavioral semantics in two steps (Sec. 3). In the first step, we abstract from box states and consider the concrete message histories at box boundaries. In the second step, we abstract from object identifiers and box environments, getting a semantics that is independent of the program context a component is used in. The remaining sections of the paper show how the semantics can be used to define substitutability (Sec. 4) and contain a discussion of related work and the conclusions.

2 Operational Semantics for Boxes

In this section, we present the operational semantics for our object-oriented core language. Most parts of the semantics follow the reductional style of [13]. The semantics has two new features: (a) It structures the heap into box-local subheaps. (b) It handles non-local method invocations by call and return messages crossing the box boundaries. It can express arbitrary sequences of callbacks.

$P ::=$	$\overline{L}$	programs
$L ::=$	$[\mathsf{box}]\ \mathsf{class}\ C\ \mathsf{extends}\ C'\ \{\overline{D}\ \overline{f};\ \overline{M}\}$	classes
$M ::=$	$C\ m(\overline{C}\ \overline{x})\{e\}$	methods
$e ::=$		expressions
	x	variables
	$\mid\ \mathsf{null}$	null constant
	$\mid\ (C)e$	cast
	$\mid\ \mathsf{new}\ C$	object/box creation
	$\mid\ e.f$	field access
	$\mid\ e.f = e$	field update
	$\mid\ \mathsf{let}\ x = e\ \mathsf{in}\ e$	variable binding
	$\mid\ e.m(\overline{e})$	method call
C, D		class names

Fig. 1. Abstract syntax

2.1 Syntax and Typing

The abstract syntax of our language is shown in Fig. 1. We use similar notations as Featherweight Java (FJ) [14]. A bar indicates a sequence: $\overline{L} = L_1, L_2, \ldots, L_n$, where the length is defined as $|\overline{L}| = n$. Similar, $\overline{C}\ \overline{f}$; is equal to $C_1\ f_1; \ldots; C_n\ f_n$. If there is some sequence $\overline{x}$, we write x_i for any element of $\overline{x}$. We sometimes write $\overline{x} \cdot x$ for adding x to sequence $\overline{x}$, and $\overline{x} \circ \overline{x}'$ for the concatenation of two sequences. The empty sequence is denoted by $\bullet$. *front* returns a sequence without the last element, and *last* returns the last element of a sequence, i.e. $\overline{x} = \mathit{front}(\overline{x}) \cdot \mathit{last}(\overline{x})$. We often apply a function f on a sequence of elements that is only defined on single elements. This means to apply f to each element of the sequence and return the sequence of the results, e.g. $f(x_1, \ldots, x_n) = f(x_1) \cdot f(x_2) \cdots f(x_n)$. We sometimes treat sequences as sets, e.g. if we write $\overline{x}_1 \subseteq \overline{x}_2$, both sequences are implicitly treated as sets.

Our language supports stateful objects, aliasing, inheritance, and dynamic dispatch. It is similar to other core formalizations of Java, namely FJ [14] and ClassicJava [13]. The main difference is the distinction between box and normal classes that provides the structuring of the heap into boxes. A set of classes $\overline{L}$ is called *declaration complete* iff all names used in $\overline{L}$ have a declaration in $\overline{L}$. A program in our language is a declaration complete set of classes $\overline{L}$. The smallest declaration complete program for a class C is called the *code base* of C. In this paper, code bases for the box classes are used as a simple notion of *program components*. In practice, the code base of a box class would be structured by module systems separating the box-local part and the code bases of the inner boxes. We consider this (interesting) aspect beyond the scope of this paper.

Contextual constraints and typing rules are essentially as in Java. The subtype relation will be denoted by $<:$, i.e. $C <: D$ means that C is a subtype of D. We assume that the most general class `Object` is a normal class without fields and methods. A subclass of a box class has to be as well a box class. We do not

$b ::= o \mid \mathsf{globox}$	boxes
$o ::= \langle j, b, C \rangle$	objects
$v ::= o \mid \mathsf{null}$	values
$O ::= \overline{v}$	object states
$B ::= \langle ES, OS, IB \rangle$	box states
$ES ::= \overline{r}$	execution stacks
$OS ::= \overline{j} \mapsto \overline{O}$	object stores
$IB ::= \overline{j} \mapsto \overline{B}$	inner boxes
$n ::=$	messages
$o \rightarrow o'.m(\overline{v})$	call message
$\mid \; o \leftarrow o'.m{:}v$	return message
$e ::= ... \mid \mathsf{result} \mid o$	reduction expressions
$t ::= n \mid r$	terms
$r ::= o \rightarrow o'.m\{e\}$	call
j	object identifier

Fig. 2. Dynamic entities and extended expression syntax

support overloading of methods and require that an overriding method has the same signature as the overridden method. We do not consider field hiding, so all fields declared in a class must have names different from the inherited fields. A method only has a single body expression which is also the return value of the method. Expressions can be variables, the null constant, cast expressions, new-expressions, field accesses, field updates, let-expressions, and method calls. If the class in a new-expression is a box class, a a new box together with its owner object is created; otherwise, a normal object is created in the current box. Let-expressions support local variables and sequential composition.

$$class(\langle \_, \_, C \rangle) = C \qquad owner(\langle \_, b, \_ \rangle) = b \qquad \frac{\mathsf{box\ class}\ C\ \mathsf{extends}\ C'\ \{\ldots\}}{boxClass(C)}$$

$$\frac{boxClass(class(o))}{box(o) = o} \qquad \frac{\neg boxClass(class(o))}{box(o) = owner(o)} \qquad \frac{\_\ \mathsf{class}\ C\ \mathsf{extends}\ C'\ \{\ \overline{D}\ \overline{f};\ \ \ldots\}}{fields(C) = \overline{D}\ \overline{f} \circ fields(C')}$$

$$\frac{\_\ \mathsf{class}\ C\ \mathsf{extends}\ \_\{\ldots\ \ D\ m(\overline{D}\ \overline{x})\{e\}\ \ \ldots\}}{method(C, m) = D\ m(\overline{D}\ \overline{x})\{e\}} \qquad \frac{\_\ \mathsf{class}\ C\ \mathsf{extends}\ C'\{\ldots\ \ \overline{M}\}\quad m \notin \overline{M}}{method(C, m) = method(C', m)}$$

$$\frac{owner(b) = b'}{b \prec b'} \qquad \frac{b \prec b' \qquad b' \prec b''}{b \prec b''} \qquad \begin{array}{rcl} address(o \rightarrow o'.m(\overline{v})) & = & box(o') \\ address(o \leftarrow o'.m{:}v) & = & box(o) \end{array}$$

Fig. 3. Auxiliary functions

2.2 Operational Semantics

Our operational semantics supports the structuring of the heap into boxes. Its central feature is *box locality*: the rules only refer to the heap parts of the current box and its inner boxes. Box locality is a prerequisite for the abstraction technique in Sec. 3. The semantics is mainly given in reductional small-step style, that is, we represent an evaluation state by a partially evaluated expression over dynamic values and by the states of the created objects. Fig. 2 contains the needed definitions. A box is either represented by its owner object or by the constant `globox` denoting the global box that contains all other boxes. An object is uniquely defined by an identifier j and by its box b. To avoid an extra mapping from objects to their classes, we add the class name as a third component to the object representation. Two objects $\langle j_1, b_1, C_1\rangle$ and $\langle j_2, b_2, C_2\rangle$ are different iff $j_1 \neq j_2$ or $b_1 \neq b_2$. Working with identifiers that only need to be unique within the box allows to create new objects in a box without knowing the identifiers of outside objects or objects in inner boxes.

The state of an object is represented by the values for its fields. The state of a box b consists of its execution state, the state of the objects with owner b and the state of the inner boxes. The execution state is a stack of pending method executions. It is used to handle callbacks. For example, if a method executing in box b leads to a call on an object outside of b, this call can call back on b's objects and so forth. Calls to and returns from methods on non-local objects are handled by messages. To represent partially-evaluated expression, the expression syntax of Fig. 2 is extended. An expression can be an object or the keyword `result` indicating that the expression expects the result of a pending call.

Figure 3 shows auxiliary functions needed by our semantics. The *box* function returns the box of an object o. Objects o of box classes represent their own box. Otherwise the box is represented by the object's owner. The relation $b \prec b'$ expresses that b is a direct or indirect inner box of b'. The reflexive closure of $\prec$ is denoted by $\preceq$. An object o is called *box-local* to box b iff $box(o) = b$. In particular, the owner of a box b is box-local to b (see fourth rule in Fig. 3). It *is in* box b iff $box(o) \preceq b$. Otherwise, o *is outside of* b.

Messages. A message $o \rightarrow o'.m(\overline{v})$ contains the sender, o, the receiver, o', the method name m, and the method parameters $\overline{v}$. We distinguish between *call messages* ($\rightarrow$) and *return messages* ($\leftarrow$). Note that for return messages the sender of the message is the object which originally called the message, and the receiver is the receiver of that call, because the original receiver sends the answer back to the original sender. The explicit representation of the sender allows to avoid a stack mechanism. Stacks in combination with callbacks would breach box locality and cause a problem for the abstraction in Sec. 3.

The *address* a of a message n is the box of the object to which the arrow points. A message n that has either the sender or the receiver in a box b is called an *ingoing* message for b if a is in b. and an *outgoing* message otherwise. The receiver object and the method parameters of a message n are called the *parameters*, $params(n)$, of n. Non-null parameters that are in b are called *inner* parameters of n, the others are called *outer* parameters, denoted by $\mathit{inner}(b, n)$ and $\mathit{outer}(b, n)$ respectively. We say that a return message *matches* a call message if the method names, the sender objects and the receiver objects are the same.

Judgements and Rules. To achieve box locality, we seperate the state of a box from the state of the enclosing boxes and guarantee that execution in a box b only modifies objects in b. The semantical rules specify two different judgements. The outside view to a box is represented by the judgement

$$b \vdash (B, n) \Downarrow (B', n')$$

expressing the fact that sending message n to box b in box state B leads to a terminating execution in b with a reply message n' that has an address outside b. B' is the state of b when n' is sent. For example, the message n could be a call and n' the corresponding return or an intermediate call to an outside object. Analogous to a judgement of big-step operational, the judgement allows to abstract from the execution steps within boxes.

Execution within a box is formalized by a reduction semantics. A triple $b{:}B, t$ is called a *configuration* consisting of a box b, its state B, and a term t representing an execution to be performed in b. A single execution step has the form:

$$b{:}B, t \quad \rightsquigarrow \quad b{:}B', t'$$

We write $\rightsquigarrow^*$ for the transitive, reflexive closure of $\rightsquigarrow$. Note that a reduction step only modifies the state of box b. States of other boxes remain unchanged. In the semantics, so-called *evaluation contexts* represent partially evaluated expressions. An evaluation context $\mathcal{E}$ is an expression with a "hole" $[\,]$ somewhere inside the expression. We write $\mathcal{E}[e]$ to mean that the hole in $\mathcal{E}$ is replaced by expression e. A hole in $\mathcal{E}$ can only appear in certain positions defined as follows:

$$\mathcal{E} ::= [\,] \mid (T)\mathcal{E} \mid \mathcal{E}.f \mid \mathcal{E}.f = e \mid v.f = \mathcal{E} \mid \mathsf{let}\ x = \mathcal{E}\ \mathsf{in}\ e \mid \mathcal{E}.m(\overline{e}) \mid v.m(\overline{v}, \mathcal{E}, \overline{e})$$

Similar to the evaluation context $\mathcal{E}$, we define a context $\mathcal{R}$ as a call with a hole, and we write $\mathcal{R}[\![e]\!]$ to replace that hole by an expression e.

$$\mathcal{R} ::= o \rightarrow o'.m\{\mathcal{E}\}$$

The rules for the reduction relation are given in Fig. 4. The upper part describes the evaluation of expressions, the lower part the handling of calls, returns, and messages. Casts are only allowed for box-local objects (R-CAST-OBJ). Thus, casts cannot be used to distinguish outside objects. This property will be used in Sec. 3 for the abstraction of outside objects (cf. the proof of the Abstraction Lemma). A more flexible cast rule would complicate the abstraction. Instantiating a normal class (R-NEW-OBJ) adds a new object to the object state of the box. Instantiating a box class adds a new box and object with an initial box state to the inner boxes. The new object identifier has to be unique with respect to all objects having the same owner. Note that in both cases a new

R-CAST-NULL

$$\frac{}{b{:}B, \mathcal{R}[\![(C)\mathsf{null}]\!] \quad \rightsquigarrow \quad b{:}B, \mathcal{R}[\![\mathsf{null}]\!]}$$

R-CAST-OBJ

$$\frac{box(o) = b \qquad class(o) <: C}{b{:}B, \mathcal{R}[\![(C)o]\!] \quad \rightsquigarrow \quad b{:}B, \mathcal{R}[\![o]\!]}$$

R-NEW-OBJ

$$\frac{\begin{array}{c}\neg boxClass(C) \qquad j \notin (dom(OS) \cup dom(IB)) \\ fields(C) = \overline{D}\ \overline{f} \qquad |\overline{\mathsf{null}}| = |\overline{f}| \qquad o = \langle j, b, C\rangle\end{array}}{b{:}\langle ES, OS, IB\rangle, \mathcal{R}[\![\mathsf{new}\ C]\!] \quad \rightsquigarrow \quad b{:}\langle ES, OS[j \mapsto \overline{\mathsf{null}}], IB\rangle, \mathcal{R}[\![o]\!]}$$

R-NEW-BOX

$$\frac{\begin{array}{c}boxClass(C) \qquad j \notin (dom(OS) \cup dom(IB)) \\ fields(C) = \overline{D}\ \overline{f} \qquad |\overline{\mathsf{null}}| = |\overline{f}| \qquad o = \langle j, b, C\rangle \qquad B = \langle \bullet, \{j \mapsto \overline{\mathsf{null}}\}, \varnothing\rangle\end{array}}{b{:}\langle ES, OS, IB\rangle, \mathcal{R}[\![\mathsf{new}\ C]\!] \quad \rightsquigarrow \quad b{:}\langle ES, OS, IB[j \mapsto B]\rangle, \mathcal{R}[\![o]\!]}$$

R-FIELD-READ

$$\frac{o = \langle j, \_, C\rangle \qquad box(o) = b \qquad fields(C) = \overline{D}\ \overline{f} \qquad OS(j) = \overline{v}}{b{:}\langle ES, OS, IB\rangle, \mathcal{R}[\![o.f_i]\!] \quad \rightsquigarrow \quad b{:}\langle ES, OS, IB\rangle, \mathcal{R}[\![v_i]\!]}$$

R-FIELD-WRITE

$$\frac{o = \langle j, \_, C\rangle \qquad box(o) = b \qquad fields(C) = \overline{D}\ \overline{f} \qquad OS(j) = \overline{v}}{b{:}\langle ES, OS, IB\rangle, \mathcal{R}[\![o.f_i = v]\!] \quad \rightsquigarrow \quad b{:}\langle ES, OS[j \mapsto [v/v_i]\overline{v}], IB\rangle, \mathcal{R}[\![v]\!]}$$

R-LET

$$\frac{}{b{:}B, \mathcal{R}[\![\mathsf{let}\ x = v\ \mathsf{in}\ e]\!] \quad \rightsquigarrow \quad b{:}B, \mathcal{R}[\![[v/x]e]\!]}$$

R-SEND-CALL-MSG

$$\frac{ES' = o'' \rightarrow o.m\{\mathcal{E}[\mathsf{result}]\} \cdot ES}{b{:}\langle ES, OS, IB\rangle, o'' \rightarrow o.m\{\mathcal{E}[o'.m'(\overline{v})]\} \quad \rightsquigarrow \quad b{:}\langle ES', OS, IB\rangle, o \rightarrow o'.m'(\overline{v})}$$

R-SEND-RTRN-MSG

$$\frac{}{b{:}B, o \rightarrow o'.m\{v\} \quad \rightsquigarrow \quad b{:}B, o \leftarrow o'.m{:}v}$$

R-EXEC-CALL-MSG

$$\frac{box(o) = b \qquad method(class(o), m) = \_\ m(\_\ \overline{x})\{e\}}{b{:}B, o' \rightarrow o.m(\overline{v}) \quad \rightsquigarrow \quad b{:}B, o' \rightarrow o.m\{[o/this, \overline{v}/\overline{x}]e\}}$$

R-EXEC-RTRN-MSG

$$\frac{box(o) = b \qquad ES = (o'' \rightarrow o.m\{e\}) \cdot ES'}{b{:}\langle ES, OS, IB\rangle, o \leftarrow o'.m'{:}v \quad \rightsquigarrow \quad b{:}\langle ES', OS, IB\rangle, o'' \rightarrow o.m\{[v/\mathsf{result}]e\}}$$

R-FORWARD-INNER

$$\frac{address(n) \preceq b' \qquad b' = \langle j, b, \_\rangle \qquad b' \vdash (IB(b'), n) \Downarrow (B', n')}{b{:}\langle ES, OS, IB\rangle, n \quad \rightsquigarrow \quad b{:}\langle ES, OS, IB[j \mapsto B']\rangle, n'}$$

R-BOX-BIG-STEP

$$\frac{b{:}B, n \quad \rightsquigarrow^* \quad b{:}B', n' \qquad address(n') \not\preceq b}{b \vdash (B, n) \Downarrow (B', n')}$$

Fig. 4. Rules of the operational semantics

object identifier can be determined based on box-local information. Rules R-FIELD-READ and R-FIELD-WRITE only allow field access on box-local objects. Thus, object creations and field updates only need box-local information and only affect box-local state. All other effects have to achieved via method calls.

A method call is treated by sending a message with the current receiver as sender (R-SEND-CALL-MSG). The evaluation state $o'' \rightarrow o.m\{\mathcal{E}[\mathsf{result}]\}$ of the current method exection is recorded on the box-local execution stack. The placeholder result marks the position for the result. If the body of a method is fully evaluated, a return message is sent (R-SEND-RTRN-MSG). If the receiver of a call message is in box b (R-EXEC-CALL-MSG), the method is executed with the actual parameters. A return message with address in b pops the pending call from the execution stack, substitutes the result, and continues evaluation (R-EXEC-RTRN-MSG). According to rule R-FORWARD-INNER, a message n with an address in an inner box b' of b is forwarded to b'. If it terminates with a reply message n', the state of b' is updated and n' is handled in b. The last three rules cannot be applied if the address of a message n' is outside b. This is the case in which box-local execution terminates with reply method n' (R-BOX-BIG-STEP).

A program is called *executable* iff it contains a class of the form: `class Main extends Object { D main(C p){e}}`. It is executed with start configuration $globox{:}\langle \bullet, \{j_p \mapsto input, \ldots, j_0 \mapsto \bullet\}, \varnothing\rangle, o_0 \rightarrow o_0.main(o_p)$ where $o_0 = \langle j_0, globox, Main\rangle$, $o_p = \langle j_p, globox, C\rangle$, and j_0 and j_p are distinct object identifiers. "*input*" denotes the field values of the parameter object o_p, and the dots indicate the possibility to have additional objects in the start configuration that are referenced by o_p. This allows to encode interesting input in the absence of primitive data types. Program execution can have three *outcomes*:

1. It can terminate normally in a configuration $globox{:}B', o_0 \leftarrow o_0.main{:}v$, i.e. with terminated method main and return value v.
2. It can end up in some configuration different from the above such that no rule is applicable (e.g. a field access on a nonlocal object). We consider this as *abortion*. For space limitation, we do not handle such exceptional cases explicitly.
3. It can diverge.

It is easy to verify that in any configuration at most one rule is applicable. Thus, the semantics is deterministic. Although determinism is not needed in principle for our approach, having a deterministic language simplifies the presentation in the following sections.

3 Behavioral Semantics for Boxes

In the following, we assume that an executable program P is given containing a box class C with code base K. We define a behavioral semantics for C and K, which is independent of the representation of the box states and independent of the environment in which C is used. The latter is not yet achieved because in general the box state encoding still uses identifiers of objects from classes not belonging to C. We reach this goal in three steps. *First*, we define a so-called *interface semantics* which takes a box, a box state, and a message for the box and results in the next state of the box and its answer message. This semantics is directly based on our big-step judgement from above. In a *second* step we abstract from the state of the box by using so-called concrete message histories. A history represents the state of a box without referring to the objects of the box and their field values. The *history semantics* defines how a message n is executed in a box b with a history H. Histories still refer to box and object identifiers. *Third*, we abstract from boxes, their execution environments, and object identifiers by defining *abstract histories*. An abstract history is essentially an equivalence class of concrete histories of boxes with the same box class. Abstract histories are used to define a precise behavioral semantics: Given an abstract history and an abstracted message, it yields the abstract answer of a box class.

The interface semantics *isem* for boxes is defined as a partial function from boxes, box states, and messages to *outcomes*, *oc*, where the outcome is either a pair consisting of a box state and a reply message or one of the constants *ABORT* or *DIVERGE*. More precisely:

Definition 1 (Interface Semantics). *Let b be a box, B a state of b and n an ingoing message for b. We define the interface semantics,* isem*, as*

$$isem(b,B,n) = \begin{cases} B', n' & \text{if } b \vdash (B,n) \Downarrow (B',n') \\ ABORT & \text{if } b{:}B,n \ \rightsquigarrow^* \ b{:}B',n' \text{ and there are} \\ & \text{no } B'',n'' \text{ with } b{:}B',n' \ \rightsquigarrow \ b{:}B'',n'' \\ DIVERGE & \text{otherwise} \end{cases}$$

3.1 History-Based Semantics

To become representation-independent we define a semantics that does not directly refer to the state of a box. The idea is to reconstruct the state from the incoming messages of a box by starting with an empty state and sequentially applying the *isem* semantics:

$$\begin{array}{lll} state(b,\bullet) & = \langle\varnothing,\varnothing,\varnothing\rangle & \\ state(b,\overline{n}\cdot n) & = B' & \text{if } state(b,\overline{n}) = B \text{ and } isem(b,B,n) = B',n' \\ state(b,\overline{n}\cdot n) & = undefined & \text{otherwise} \end{array}$$

It is clear that not every arbitrary sequence of incoming messages for a box, leads to a valid state. In particular, only objects that have been earlier exposed by a method call or return are permitted as parameters and a return message has to match the last callback from the box. Valid sequences of incoming messages are called *concrete histories.*

Definition 2 (Concrete History, Admissible Message). *A concrete history H is a quadruple consisting of a box b, denoted by $box(H)$, a sequence of incoming messages, $ims(H)$, a sequence of pending calls, $pcs(H)$, and a sequence of exposed objects, $exp(H)$. Every concrete history H with $box(H) = b$ satisfies the following conditions:*

– *If $ims(H) = \bullet$, then $pcs(H) = \bullet$ and $exp(H) = \{b\}$.*
– *If $ims(H) = n_1, \ldots, n_z$, then*
  1. *there is a concrete history H' with $box(H') = b$ and $ims(H') = n_1, ..., n_{z-1}$.*
  2. *n_z is* admissible *for H', which means that*
     (a) n_z is an ingoing message for b
     (b) $inner(b, n_z) \subseteq exp(H')$
     (c) if n_z is a return message then n_z matches $last(pcs(H'))$
  3. *$isem(b, state(b, ims(H')), n_z) = (B, n)$*
  4. *$exp(H) = exp(H') \circ inner(b, n)$*
  5. $pcs(H) = \begin{cases} pcs(H') & \text{if } n_z \text{ is a call, and } n \text{ is a return} \\ pcs(H') \cdot n & \text{if } n_z \text{ is a call, and } n \text{ is a call} \\ front(pcs(H')) & \text{if } n_z \text{ is a return and } n \text{ is a return} \\ front(pcs(H')) \cdot n & \text{if } n_z \text{ is a return, and } n \text{ is a call} \end{cases}$

Based on the notions of concrete histories, we define a representation-independent semantics for boxes. The big advantage of a representation independency is that it allows to compare two boxes with different representations without the need to relate their representations (see [3]).

Definition 3 (History-Based Semantics). *Let H be a concrete history with $box(H) = b$. Let n be an admissible message for H. We define the history-based semantics,* hsem*, as*

$$hsem(H, n) = \begin{cases} n' & \text{if } oc = (B, n') \\ oc & \text{otherwise} \end{cases} \quad \text{where } oc = isem(b, state(b, ims(H)), n)$$

3.2 Behavioral Semantics

The incoming message sequence of a concrete history still contains objects and types that depend on the execution environment. In order to abstract from concrete objects we introduce abstract objects $\tilde{o}$. An abstract object is represented by a natural number i, and a subscript indicating whether it is an inner or outer object with respect to a certain box.

$$\tilde{o} ::= i_{in} \mid i_{out}$$

The precise types of the objects are not recorded, as we want to compare histories for implementations with different types. The needed type information can be derived from the method signature in an abstract message, which is defined as follows. Let C be a box class with code base K, and let D be a class in K; *abstract messages* $\tilde{n}$ of C have the form

$$\tilde{n} ::= \tilde{o}.D{::}m(\overline{\tilde{v}}) \mid D{::}m{:}\tilde{v}$$

The left one is a call message and the right one is a return message. A message $\tilde{n}$ is an *ingoing call* if $\tilde{o}$ is an inner object, otherwise it is an *outgoing call.* The method parameters $\tilde{v}$ are abstract objects or null (*abstract values*). The function *inner* on abstract messages returns the set of abstract inner objects occurring as parameters of the message. *params* return the message parameters, i.e. $\tilde{o} \cdot \overline{\tilde{v}}$. D::m denotes method m in class D. This *class qualification* of method names is needed because the receiver type is no longer represented and methods in incomparable classes may have the same name.

Let η be a bijection from concrete objects to abstract objects. We say that a concrete message n *corresponds to* an abstract message $\tilde{n}$, denoted by $n \simeq_\eta \tilde{n}$, if and only if the method names and message kinds are the same, the class qualification of $\tilde{n}$ is the largest supertype of the receiver in which the method is defined, and the parameters are the same under η, i.e. $\eta(params(n)) = params(\tilde{n})$.

Definition 4 (History Abstraction). *Let H be a concrete history. An* abstraction *G of H consists of a sequence of incoming abstract messages $ims_a(G)$, a sequence of pending abstract calls $pcs_a(G)$, and a sequence of exposed abstract objects $exp_a(G)$ such that the following conditions are satisfied:*

- *There exists a bijection η from concrete objects occurring in H as parameters of messages or as exposed objects, and abstract objects occurring in G, such that*
$$\eta(o) = \begin{cases} i_{in} & \text{if } o \preceq b \\ i_{out} & \text{otherwise} \end{cases}$$
- $\eta(exp(H)) = exp_a(G)$
- $ims(H) \simeq_\eta ims_a(G)$
- $pcs(H) \simeq_\eta pcs_a(G)$

We call G an *abstract history* for a box class C iff G is an abstraction of some concrete history H with $class(box(H)) = C$.

In general, there are many different abstractions of a concrete history, as the object numbers of abstract messages can be arbitrarily chosen. To give every concrete history a unique abstraction, we define a normalization of abstract histories. To normalize an abstract history G we rename all message numbers appearing in G in such a way that the first occurring number is 1 and all following numbers are always advanced by 1. By $absHis(H)$ we denote the *normalized abstraction* of concrete history H and call the resulting history a *normalized abstract history.*

Given a concrete history H and a concrete message n, where all parameters of n already occur in H, we define the function $absMsg(H, n)$ to result in abstract message $\tilde{n}$ as follows: let $G = absHis(H)$ and let η be the bijection of Def. 4; then $\tilde{n}$ is an abstract message that corresponds to n, and the parameters in n are equal to the abstract parameters in $\tilde{n}$ under bijection η. If there are new objects in n which are not in the domain of η, η is extended in a normalized way.

Given a concrete history H and an admissible message n for H. Let $oc = hsem(H, n)$. We denote the abstraction of outcome oc by $absOutcome(oc, H, n)$. If $oc \in \{ABORT, DIVERGE\}$, then $absOutcome(oc, H, n) = oc$. Otherwise, we abstract the message oc w.r.t. H and n in the same way as we defined $absMsg$.

The central property of our abstraction is formulated by the following lemma. It states that abstract histories can express the behavior of a box class and its code base independent of the box instances and of the program in which the box is used. More precisely:

Lemma 1 (Abstraction). *Let C be a box class with code base K, and P_1 and P_2 be two programs containing K. Let b_1 and b_2 be boxes of C in executions of P_1 and P_2 resp.; furthermore let H_1 and H_2 be concrete histories for b_1 and b_2 such that $absHis(H_1) = absHis(H_2)$. If n_1 and n_2 are admissible messages for H_1 and H_2 resp. with $absMsg(H_1, n_1) = absMsg(H_2, n_2)$, then*

$$absOutcome(hsem(H_1, n_1), H_1, n_1) = absOutcome(hsem(H_2, n_2), H_2, n_2) \ .$$

Proof. A detailed formal proof is beyond the scope of this paper. Here, we give an outline of the central ideas. The proof runs by induction on the length of H_1 (note $|ims(H_1)| = |ims(H_2)|$). Let H_i^k denote the prefix of H_i containing the first k messages.

Induction invariant: For all $k \in \{0, \ldots, |ims(H_1)|\}$ there exists a bijection β^k from the objects and boxes occurring in $state(b_1, ims(H_1^k))$ to the objects and boxes occurring in $state(b_2, ims(H_2^k))$ such that

$$\begin{array}{ll} class(o) = class(\beta_k(o)) & \text{for the objects in } state(b_1, ims(H_1^k)) \\ state(b_1, ims(H_1^k)) = state(b_2, ims(H_2^k)) \downarrow \beta^k & \end{array}$$

where $state(b_2, ims(H_2^k)) \downarrow \beta^k$ denotes the box state that is obtained from $state(b_2, ims(H_2^k))$ by replacing all objects o and boxes b by $\beta^k(o)$ and $\beta^k(b)$.

The induction basis follows from rule R-NEW-BOX. Induction step: Because of $absMsg(H_1^k, n_1^k) = absMsg(H_2^k, n_2^k)$, both messages are of the same kind. If they are call messages the receiver has to be an object o_i in b_i. Because the messages n_i^k are admissible, o_i is in $exp(H_i^k)$, thus, o_1 is in the domain of β^k so that $class(o_1) = class(o_2)$. Thus, $absHis(H_1^k) = absHis(H_2^k)$ yields that the methods are the same, in particular, they have the same signature. Thus, the parameter lists have the same length. For parameter objects p_1 of n_1 that already occur in H_1^k, $absHis(H_1^k) = absHis(H_2^k)$ yields that $\beta^k(p_1) = p_2$. Parameter objects not occurring in H_1^k or H_2^k are outside objects (otherwise they are present in $exp(H_i^k)$). Note that they may have different dynamic types. Because of $absHis(H_1^k) = absHis(H_2^k)$, there is a bijection from the parameter objects not occurring in H_1^k to those not occurring in H_2^k that is consistent with the position in the parameter list of n_i^k. By β_+^k we denote the extension of β^k to the object not occurring in H_1^k. A similar construction has to be done, if n_i^k are return messages. In that case, the pending call sequence is used to identify the addressees of the messages.

Now, we have corresponding start states $state(b_1, ims(H_1^k))$ and $state(b_2, ims(H_2^k))$ with corresponding incoming messages. The rules of Fig. 4 keep the correspondence, because none of the rules depend on the concrete object or box identifiers or on the concrete type of outside objects (that is the reason why we do not allow to cast outside objects). Thus, either both executions abort, diverge, or produce corresponding replies, that is replies with the same abstraction. □

Based on the Abstraction Lemma we formulate a behavioral semantics for box classes. Let G be an abstract history. An abstract message $\tilde{n}$ is called *admissible for G* if and only if

- $\tilde{n}$ is an ingoing abstract message, and
- $inner(\tilde{n}) \subseteq exp_a(G)$, and
- if $\tilde{n}$ is a return, i.e. $\tilde{n} = D{::}m{:}\tilde{v}$, then it matches the last pending call, i.e. $last(pcs_a(G)) = i_{out}.D{::}m(\overline{\tilde{v}})$, for some abstract values $\overline{\tilde{v}}$.

Definition 5 (Behavioral Semantics). *Let G be a normalized abstract history and let $\tilde{n}$ be an admissible message for G. We define the behavioral semantics,* bsem*, as*

$$bsem(G, \tilde{n}) = absOutcome(hsem(H, n), H, n)$$

where H is any concrete history for a box b with $G = absHis(H)$, and n is any admissible concrete message for H with $\tilde{n} = absMsg(H, n)$.

Lemma 2. *bsem is well-defined.*

Proof. Let G be a normalized abstract history and $\tilde{n}$ be an abstract admissible message for G. By definition there exists a concrete history H with $G = absHis(H)$. If we can show that it is possible to choose an admissible message n for H with $\tilde{n} = absMsg(H, n)$, the Abstraction Lemma provides well-definedness, because it guarantees that the abstract outcome does not depend on the choice of H and n.

Let η_G be the bijection of G. If $\tilde{n}$ is an ingoing call, then choose an arbitrary outside sender, and choose arbitrary outside objects for outer parameters not handled by η_G. Use all other objects according to η_G^{-1} given by $ims_a(G)$. Otherwise if $\tilde{n}$ is an ingoing return $D{::}m{:}\tilde{v}$, then let $o \rightarrow o'.m(\overline{v}) = last(pcs(H))$ and let $last(ims_a(G)) = i_{out}.D{::}m(\overline{\tilde{v}})$. The compatibility of $pcs(G)$ yields that $o \leftarrow o'.m{:}v$ is an admissible message where $v = \eta_G^{-1}(\tilde{v})$ if $v \in dom(\eta_G)$ or v is some correctly typed object not in $dom(\eta_G)$ otherwise. □

4 Substitutability

In this section, we discuss how our behavioral semantics can be exploited to handle substitutability in object-oriented programming. Central for the exploitation is the representation independency of the semantics based on a well-defined boundary of the runtime components.

A program component K_1 can be substituted by another component K_2 in a program context P if both components have the same behavior in all executions of P. The application of this notion of substitutability to existing OO-languages faces two problems: 1. Beyond classes, there is no suitable standard concept of a program component; and considering only single classes does not scale. 2. Defining "same behavior" without a sufficiently abstract notion of behavior is doable but complex (see [3] and the discussion in Sec. 5).

Our approach gives answers to both problems. A code base of a box class C is a well-defined notion for flexible program components. Having an explicit behavioral semantics makes it straightforward to define substitutability and equivalence for box classes:

Definition 6 (Substitutability, Equivalence). *Let C_1 and C_2 be two box classes with code bases K_1 and K_2 such that $C_1 = C_2$ or C_2 is a subclass of C_1. (C_2, K_2) is called a* behavioral substitute *of (C_1, K_1) iff*

- *every abstract history G of C_1 is an abstract history of C_2 and*
- *$bsem(C_1, G, \tilde{n}) = bsem(C_2, G, \tilde{n})$ for every abstract history G of C_1 and every admissible abstract message $\tilde{n}$ of G.*

Two code bases K_1 and K_2 for a class C are called equivalent *iff (C, K_1) is a behavioral substitute of (C, K_2) and vice versa.*

Of course, a behavioral substitute can have more behavior. For example, class C_2 or objects exposed by C_2 can have more methods. Thus, they have more admissible messages. However, these messages cannot be used in program contexts in which C_1 is eligible.

Intuitively, a software component SC_2 is substitutable for a component SC_1 if replacing a usage of SC_1 in a program by a usage of SC_2 yields an equivalent program. As we only have a notion of equivalence for box classes, this intuitive meaning gets the following formulation in our setting:

Lemma 3 (Substitution). *Let D, C_1, and C_2 be box classes with code bases K, K_1, and K_2 such that C_1 is used in K and C_2 is a subclass of C_1. Let K' be the code base for D obtained by replacing a creation expression new $C_1()$ by new $C_2()$ in K (as K' is declaration complete it includes K_2). If (C_2, K_2) is a behavioral substitute of (C_1, K_1), then (D, K) and (D, K') are equivalent.*

A proof of the lemma is beyond the scope of this paper. It basically shows that in any context of an executable program an instance of D with code base K can simulate an instance of D with code base K' and vice versa. As in the proof of the Abstraction Lemma, it is crucial that we permit downcasts of an object o only in o's box and that we do not provide an instance-of operator. Consider for example a program context in which (D, K) exposes an owned C_1-object o_1 and (D, K') exposes an owned C_2-object o_2 instead. Casting o_1 and o_2 in this context to C_2 would yield different outcomes and the simulation would fail.

Discussion. The main point of our notion of behavior and substitutability is that the abstraction needed to compare different implementations is given by the behavioral semantics and is independent of the components to be compared and the contexts in which they should execute. Thus, one can do the comparison without component specific coupling relations or specifications.

In one respect, our notion is less flexible than notions of substitutability that are based on classical component specifications. Whereas in our setting admissibility of messages is defined only in terms of the operational semantics, component specifications can and usually do restrict the set of admissible messages by preconditions. Looking at it the other way round, by refining the notion of admissible message, our approach could be used as a semantics for behavioral specifications of program components where a specification defines:

1. The set of possible abstract component states.
2. The admissible messages in a state (using preconditions).
3. The reply to messages and the abstract state in which the reply is sent.

Abstract components states represent and possibly further abstract histories. As a specification can exclude some messages by preconditions, a specification allows less histories than the semantics. The central difference between the classical pre-postcondition approach for behavioral subtyping (see [18]) and a specification technique based on our approach is the treatment of callbacks and effects to the environment. The extensions to the classical approach treat callbacks by supporting controlled dependencies across abstraction boundaries (see e.g. [4]). Our approach suggests to focus on the messages crossing the component boundary. This will simplify the verification of the component and shift part of the burden to the program that connects the components under consideration.

5 Related Work

In [3], Banerjee and Naumann show how confinement properties based on ownership-structures can be exploited to define and verify the equivalence of program components. Like in our approach, they use a semantics-based notion of ownership. Different is the technique to establish the equivalence result. They use relations for coupling execution states and a simulation-based proof technique whereas we abstract the implementations separately and compare the abstractions. The work in [3] and our approach both aim at substitutability for components of scalable size. Other work investigate refinement and inheritance relations on the level of classes (see in particular [2]).

Using message sequences to characterize state and behavior of software components is not a new idea (see e.g. [12] and later [10]). Nierstrasz defines the notion of request substitutability based on request sequences [20]. Broy uses call and return messages to characterize the behavior of methods in a component specification framework ([8]). Abraham et al. investigate interface behavior for a concurrent object calculus in [1]. Like we do, they use call and return messages crossing component boundaries and stacks to handle callback scenarios, but object identifiers are only abstracted with respect to alpha-conversion.

A large core of literature explores behavioral subtyping for object-oriented programming based on class and method specifications. That is, the behavioral subtype relation is not defined in terms of the semantics of the given classes, but in terms of programmer defined specifications that abstract class behavior (see [18]). These techniques build on specification languages for object-oriented programs (e.g. JML for Java [16], Spec# for C# [6]). Leavens and Naumann describe the relation between specification, semantics, and behavioral subtyping in a very concise way [15]. A specification technique with refinement that explicitly handles outgoing messages is developed in [9].

Ownership concepts were originally developed to check confinement properties by type systems: see [11] for an introduction and overview; [7] for a system to check concurrency properties; and [24, 21] for a type system and an type inference technique to check boxes. Boogie [5] and other approaches to modular reasoning (see e.g. [19, 17] use ownership structures to define the semantics of object invariants, to control the dependencies of specification statements, and to partition the heap. The importance to modularize reasoning and analysis based on heap structuring is shown as well by [22], which develops a logic for partial heaps, and by [23] which presents a modular static analysis to identify structural invariants of heap-manipulation programs.

6 Conclusions

We presented a behavioral semantics for flexible object-oriented components with multiple ingoing and outgoing read-write references. The semantics is obtained by a two step abstraction from an extended operational semantics. The semantics formalizes the behavioral aspects that are relevant to a user of the component. We discussed the relation to substitutability and specification-based behavioral subtyping.

A semantics based notion of component behavior has the advantage that it can be used by all language-processing tools and techniques. The abstraction from the execution environment is important for modular static analysis techniques. It guarantees that a "most general client" that generates all admissible message sequences for the component can be used for static analysis. Future work include the refinement of the component model, in particular the transfer of inner boxes from one box to another, the enhancement of our specification and checking techniques, and an extension of the approach to concurrent object-oriented programming.

Acknowledgments. We thank Peter Müller and the anonymous reviewers for their helpful comments.

References

[1] E. Ábrahám, A. Grüner, and M. Steffen. Abstract interface behavior of object-oriented languages with monitors. In *FMOODS*, pages 218–232, 2006.

[2] R.-J. Back, A. Mikhajlova, and J. von Wright. Class refinement as semantics of correct object substitutability. *Formal aspects of computing*, 12(1):18–40, 2000.

[3] A. Banerjee and D. A. Naumann. Ownership confinement ensures representation independence for object-oriented programs. *Journal of the ACM*, 52(6):894–960, 2005.

[4] M. Barnett and D. A. Naumann. Friends need a bit more: Maintaining invariants over shared state. In D. Kozen and C. Shankland, editors, *MPC 2004*, volume 3125 of *LNCS*, pages 54–84. Springer, 2004.

[5] M. Barnett, R. DeLine, M. Fähndrich, K. R. M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6), 2004.

[6] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *CASSIS 2004*. Springer, 2004. LNCS 3362.

[7] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Proc. OOPSLA 2002*, pages 211–230. ACM Press, Nov. 2002.

[8] M. Broy. A core theory of interfaces and architecture and its impact on object orientation. In *Architecting Systems with Trustworthy Components*, pages 26–47. Springer, 2006. LNCS 3938.

[9] M. Büchi. *Safe Language Mechanisms for Modularization and Concurrency*. PhD thesis, Turku Centre for Computer Science, May 2000.

[10] A. Chakrabarti, L. de Alfaro, T. A. Henzinger, M. Jurdzinski, and F. Y. C. Mang. Interface compatibility checking for software modules. In *CAV '02*, pages 428–441. Springer, 2002.

[11] D. Clarke. *Object Ownership and Containment*. PhD thesis, University of New South Wales, July 2001.

[12] L. de Alfaro and T. A. Henzinger. Interface automata. In *ESEC/FSE-9*, pages 109–120. ACM Press, 2001.

[13] M. Flatt, S. Krishnamurthi, and M. Felleisen. A programmer's reduction semantics for classes and mixins. *Formal Syntax and Semantics of Java*, 1523:241–269, 1999.

[14] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *TOPLAS*, 23(3):396–450, May 2001.

[15] G. T. Leavens and D. A. Naumann. Behavioral subtyping, specification inheritance, and modular reasoning. Technical Report TR06-20a, Computer Science, Iowa State University, 2006.

[16] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: a behavioral interface specification language for Java. *SIGSOFT Softw. Eng. Notes*, 31(3):1–38, 2006.

[17] K. R. M. Leino and P. Müller. Object invariants in dynamic contexts. In M. Odersky, editor, *Proc. ECOOP 2004*, pages 491–516. Springer, June 2004. LNCS 3086.

[18] B. Liskov and J. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, 1994.

[19] P. Müller. *Modular Specification and Verification of Object-Oriented Programs*. Springer, 2002. LNCS 2262.

[20] O. Nierstrasz. Regular types for active objects. In *Proc. OOPSLA '93*, pages 1–15. ACM Press, Oct. 1993.

[21] A. Poetzsch-Heffter, K. Geilmann, and J. Schäfer. Inferring ownership types for encapsulated object-oriented program components. In *Program Analysis and Compilation, Theory and Practice: Essays Dedicated to Reinhard Wilhelm*. Springer, 2007. LNCS 4444.

[22] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of LICS'02*, pages 55–74, 2002.

[23] N. Rinetzky, A. Poetzsch-Heffter, G. Ramalingam, M. Sagiv, and E. Yahav. Modular shape analysis for dynamically encapsulated programs. In *European Symposium on Programming (ESOP'07)*. Springer, March 2007.

[24] J. Schäfer and A. Poetzsch-Heffter. A parameterized type system for simple loose ownership domains. *Journal of Object Technology*, June 2007. to appear.