Towards a Fully Abstract Semantics for Object-Oriented Program Components

Preliminary Version

Arnd Poetzsch-Heffter  Jean-Marie Gaillourdet  Jan Schäfer

University of Kaiserslautern, Department of Computer Science, Gottlieb-Daimler Str., Kaiserslautern, Germany

Abstract

Behavioral semantics for components abstract from implementation details and describe the components’ behavior independent of the components’ implementations. It provides an important foundation for behavioral substitutability and interface specifications. In this paper, we develop and investigate a behavioral semantics for a sequential class-based object-oriented language with aliasing, subclassing, and dynamic dispatch. The implementation of a component includes an owner class $C$ and the classes and interfaces used by $C$. A component instance consists of a dynamically evolving set of objects with a clear boundary to the environment. Behavioral semantics is expressed in terms of the messages crossing the boundary. The types of objects that can cross a boundary are described by the component signature.

We develop a light-weight object-oriented component model and discuss substitutability. We present an object-oriented core language supporting this model and define its operational semantics in a standard way. As a central step towards a behavioral semantics, we develop a message semantics that assigns to each component its part of the heap and stack, localizes object creation to components, and treats inter-component communication by messages. We show that the message semantics is equivalent to the standard semantics. By abstracting from the component states, we obtain a behavioral semantics. Component behaviors are defined as partial functions from incoming messages to outgoing messages. The semantic domain of these behaviors is independent of possible component environments and implementations. We proof that the behavioral semantics is fully abstract with respect to the operational semantics.

Key words: object-oriented language, behavioral semantics, components

Preliminary Version 8 July 2008
1 Introduction

As it is today, object-oriented programming does not provide a well-defined semantical layer between single classes and complete, executable programs with hundreds of classes. Having such a large gap between the constituents of a program and the program itself is a disadvantage for program development and analysis. Here are some aspects illustrating this claim:

• Substitution and refactoring on the level of single classes rarely leads to equivalent programs. Often several related classes have to be exchanged together (cf. Sec. 2).
• For scalability, program analysis should be compositional, i.e., the analysis of a component should use the analysis results of the subcomponents.
• The semantics of a program component should be definable independent of possible program contexts in which it might be used.

The central goal of this paper is to provide a semantical foundation for the specification and analysis of object-oriented program components. By specification, we mean precise descriptions of component interface properties. Analysis includes abstract interpretation, verification, and checks for substitutability. A component realizes program abstractions with well-defined boundaries. At runtime, a program component can be instantiated several times.

In the object-oriented paradigm, the component implementation consists of classes and interfaces. Communication across component boundaries is carried out via method calls on references of objects that are exposed by component instances or passed into them. Program components range from single classes that do not use other classes to provide its functionality, over API classes using other classes and interfaces to application components of different sizes. Components can be built from other components. They provide structural layers between single classes and executable programs.

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

---

Email addresses: poetzsch@informatik.uni-kl.de (Arnd Poetzsch-Heffter), jmg@informatik.uni-kl.de (Jean-Marie Gaillourdet), jschaefer@informatik.uni-kl.de (Jan Schäfer).

1 Supported by the Deutsche Forschungsgemeinschaft (German Research Foundation)
2 Cf. [30] for a realistic software system structured into components.
3 In this paper, we only consider components that react on external stimuli.
(1) Different component implementations can be compared based on the message behavior. This simplifies the notion of behavioral substitutability for components. In particular, an explicit coupling relation between substitutable components is not needed (cp. with the approach in [4]).

(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) Behavioral semantics 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 the following, 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. We do not consider concurrency in this paper.

1.1 Approach, Contributions, Overview

A component in our approach is called a box. Boxes are a light-weight semantics-based component model for OO-programming. A box B is declared in the program together with its interface $I_B$. The box implementation provides a class $C$ implementing $I_B$. $C$ is called the owner class of $B$. The instantiation of $B$ creates a new box instance $b$ and a new $C$-object $x$ in $b$ and returns $x$; $x$ is called the owner of $b$. In this paper, we assume for simplicity that all objects of a box instance $b$ are created locally, that is, either at box creation time or by an object in $b$. If an object in $b$ wants to create an object in an existing box other than $b$, it has to use a method.

Boxes can be tree-structured, that is, a box instance $b$ can have so-called inner boxes. Boxes are similar to ownership contexts. However, we only use the structural aspects of ownership (similar to [5]) and consider owners to be included in their box. Confinement is not enforced. In particular, our model allows arbitrary references going into and out of a box.

The box model is informally introduced in Sec. 2. The main part of the paper develops a behavioral semantics from a small-step operational semantics. Altogether the paper provides the following main contributions:

(1) A core language realizing the box model, a light-weight object-oriented component model. The model is a substantial refinement and extension of the model presented in [24] (Section 3).

---

4 This is similar to a distributed setting with remote method invocation.
An enhanced standard operational semantics for this language capturing
the component structure and providing boundary checks (Section 4).

An equivalent message semantics that assigns to each component its part
of the heap and stack, localizes object creation to components, and treats
inter-component communication by messages (Section 5).

A proof sketch that standard operational and message semantics are
equivalent (Section 6). A full proof is beyond the scope of this paper.
We present the proof sketch to give a detailed description of how both
semantics are related.

A behavioral semantics as abstraction from the message semantics (Sec-
tion 7).

A well-defined notion of substitutability and a proof that the behavioral
semantics is fully abstract w.r.t. the message semantics (Section 8).

Finally, Section 9 concludes and motivates lines of future work.

Although each contribution is interesting on its own, this paper is mainly about
how these constructions fit together. To manage a presentation of this overall
scenario within one paper, we had to focus the proofs to the new aspects and
those that are interesting for the construction.

1.2 Related Work

Ownership and Verification. The basic idea of our component model, namely
to hierarchically structure the heap into dynamically created regions, is bor-
rowed from ownership disciplines. They were originally developed to check con-
finement properties by type systems: see [8] for an introduction and overview;
[3] for a system to check concurrency properties; [25, 10] for generic own-
ership type systems. More expressive structuring techniques are presented in
[2, 18, 29] where the latter was developed to check box properties. In [13],
ownership structuring techniques are used to define and check immutability.
Boogie [5] and other approaches to modular reasoning (see e.g. [20, 17]) use
structured heaps to define the semantics of object invariants, to control the
dependencies of specification statements, and to partition the heap. In Boogie
and our approach, ownership is a semantical notion that is not defined by a
type system, but by other languages constructs.

The importance to modularize reasoning and analysis based on heap structur-
ing is shown as well by [26], which develops a logic for partial heaps, and by
[28] which presents a modular static analysis to identify structural invariants
of heap-manipulating programs.

In [4], Banerjee and Naumann show how confinement properties based on
ownership-structures can be exploited to define and verify the equivalence of
program components. As in our approach, they use a semantics-based notion of ownership. Their technique to establish the equivalence result is different. They use relations for coupling execution states and a simulation-based proof technique whereas we abstract the component implementations separately and compare the abstractions.\footnote{Comparison of abstractions is not treated in this paper.} The work in \cite{4} and our approach both aim at substitutability for components of scalable size. Other work investigates refinement and inheritance relations on the level of classes (see in particular \cite{3}).

\textit{Component Interaction and Abstraction.} Using message sequences to characterize state and behavior of software components is already investigated with other goals or in other settings (see e.g. \cite{9} and later \cite{7}). Nierstrasz defines the notion of request substitutability for objects based on request sequences \cite{21}.

Closest to our work is the work of Jeffrey and Rathke on Java Jr. \cite{16}. The authors develop a trace semantics for components in a sequential class-based object-oriented language. They show full abstractness of their semantics and control types at component boundaries. They also develop a notion of component compatibility. The main difference to our work is the used component model. In Java Jr. a component is a sequence of packages. Components are not instantiable, thus, at runtime, the number of components is fix. In our approach, components can be instantiated at runtime, which allows for an unbounded number of component instances of the same type. In addition to that, component instances are tree-structured at runtime. This allows component instance to transparently reuse other components and have full control over the state of instance of such components. In Java Jr. it is in general not possible for a component to transparently use other components, without having to deal with possible effects due to sharing. Finally, in contrast to our work, Java Jr. does not support downcasts, which has a great impact on the trace semantics and the compatibility of components.

In a previous work, Jeffrey and Rathke \cite{15} developed a fully abstract may testing semantics for a variant of the concurrent object calculus. As in our approach, they use call and return messages crossing component boundaries to handle callback scenarios. They provide an explicit mechanism in their calculus to hide object identifiers whereas our abstraction mechanism is applied only to exposed object identifiers. While our main focus is on heap structure and inheritance, their main focus is on thread-based concurrency where parallel execution is the composition mechanism. The work of Abraham et al. also investigates interface behavior for a concurrent object calculus in \cite{1}. Similar to our work, they use a technique to separate the global stack into compo-
ponent local stacks. Fully abstraction results for sequential and concurrent object calculi are also presented in [31]. Whereas many fundamental aspects are similar to the cited work on object calculi, our approach differs in the following aspects:

- Our approach is developed for a typical object-oriented core language with inheritance, not for a calculus. In particular, the same class can be used by the environment and different components.
- We explicitly consider a hierarchical heap structure. This yields a different and more practical approach to observability and object connectivity.
- We hide types at the component boundaries to achieve improved substitutability.
- Our technique for object identifier abstraction is not based on name binding and alpha conversion, but on identifier translation.

Another approach to the last aspect, i.e., to the problem of how to abstract over object identifiers, is storeless semantics [27].

The paper improves earlier work presented in [24]. It provides a more flexible language support for boxes, handles casts which is practically important, develops for the first time the relation between a standard semantics and the message semantics, improves the behavioral semantics with well-defined interfaces, and provides a full abstractness result.

2 Program Components and their Boundaries

In this section, we illustrate program components in general and boxes in particular by a small example. To improve readability, the example is written in a Java-like syntax [12]. Beyond the features of the core language formally treated in the following sections, we use the primitive types \( \texttt{int} \) and \( \texttt{boolean} \), local variable declarations, and statement sequencing. We use interface and class types as in Java, but do not consider multiple subtyping in this paper. That is why the most general type \( \texttt{Object} \) is an interface type in our setting.

2.1 Declaring, Using, and Implementing Boxes

To describe how boxes are declared, used, and implemented, we consider a simple list component that supports iterators.

\[ \text{\underline{6}} \] Both constructs can be expressed in the core language using the let-expression.

\[ \text{\underline{7}} \] This restriction is only made to simplify the presentation.
Declaring Boxes. The list component is declared as a box with name `BList` and with interface `List`:

```java
box BList : List;

interface List extends Object {
    void add(Object o);
    Iterator iter();
}
```

A new instance of a BList-box is created by the expression `new BList()` that returns a reference to an object of type `List`. This reference can be used to add objects to the list and to obtain iterators for the list. As the iterators provide as well access to the box, we need the interface for `Iterator` and its super interface(s) to understand what an environment can do with the box:

```java
interface Iterator extends Object {
    boolean hasNext();
    Object next();
}

interface Object { }
```

To understand how the box might influence the environment, we need as well the interface types of references that a box can receive from the environment. In our example, this is type `Object`. As `Object` has no methods, the box cannot influence the environment at all if we neglect downcasts. Of course, in many practical examples, received references to objects of the environment are used as targets for method calls.

In summary, a BList-box exposes references of type `List` and `Iterator` and receives references of type `Object`. These three types define the so-called box signature. The user of a box only needs to know the behavior of the types in the box signature. As shown in Sec. 4, the box signature can be derived from the interface `I` given in an box declaration `box B : I`. Knowing the types of references crossing the box boundary, we know by which methods a box can communicate with its environment.

Using Boxes. Boxes are created like objects. The objects that they expose are used in the same way as in other object-oriented language. Here is a simple program fragment that creates a box, adds two elements, asks for an iterator, and uses the iterator:

```java
class User implements Object {
    void doSomething() {
        List myList = new BList();
```
```java
myList.add( new String() );
myList.add( new MyClass() );
Iterator it = myList.iter();
print(it.next());
}
```

**Implementing Boxes.** A box implementation consists of an implementation directive and further classes and interfaces. An implementation directive

```java
box B with C
```

states that class C is used as an implementation of the interface type I of B. When a new B-instance is created, a C-object is returned. The following program fragment sketches a possible implementation of BList:

```java
box BList with SLList;

class SLList implements List {
    Node first;
    void add(Object o){ ... }

    Iterator iter() {
        SLListIter it = new SLListIter();
        it.current = first;
        return it;
    }
}

class Node implements Object { Object elem; Node next; }

class SLListIter implements Iterator {
    Node current;
    boolean hasNext() { return current != null; }

    Object next() {
        Node n = current;
        current = current.next;
        return n;
    }
}
```

In general, a box implementation comprises several classes and interfaces. Some of the classes and interfaces are only developed for the box. Others may be also used by other box implementations. For examples, library classes are typically used in many box implementations. In addition to that, a box implementation can use other boxes.
2.2 Properties to Achieve Substitutability

A box implementation $B1$ can be substituted by a modified or different implementation $B2$ if there is no program context that can observe the difference. Substitution is central to component-oriented programming, as it allows for a separate development of components and program contexts using the components. For example, if we replace the implementation of $\text{BList}$ by an array-based solution, the semantics of class $\text{User}$ should not be affected. Our approach is to develop a behavioral semantics for box implementations such that implementations with the same semantics can be substituted for each other. The behavioral semantics should satisfy the following properties:

1. It should be sufficiently abstract. In particular, it should abstract from the instance level and from the hidden types used for the box implementation.
2. There should be no surprises for a user (see below).
3. The semantics should be expressible using only the box signature.
4. It should be sound, i.e., if two box implementations $B1$ and $B2$ have the same behavioral semantics, there should be no program context observing a difference.

The first three properties show that the development of a behavioral component semantics depends as well on some design aspects. We will briefly discuss these aspects to motivate the design decisions underlying our approach.

Sufficient Abstractness. As we assume that two instances of the same box implementation have the same behavior, the semantics should abstract from the instance level. This property will be achieved by a normal form for object identifiers that is independent of the program context.

A more difficult problem is the treatment of types used for the implementation of a box. Let’s consider a different implementation for $\text{BList}$ that uses a public type $\text{Vector}$ to implement the interface $\text{List}$. In a naive design of a behavioral semantics, such an implementation can never be substitutable for the implementation given above, because a program context can try to cast the exposed reference to a $\text{Vector}$-object and thereby observe a difference between the $\text{SLList}$- and $\text{Vector}$-implementation. Essentially, there are three possible solutions to this problem: (a) Forbid the use of publicly visible types in implementations. (b) Forbid casts on exposed references. (c) Expose the references only with the type visible in the box signature. As the first two solutions are too restrictive in practice, we choose solution (c).
No Surprises. This property concerns modifications of program contexts using a box. Thus, it is symmetric to component substitutability, but aims at separate development of program contexts. Let us assume that a box $B$ can receive objects of type $S$, but has no subtype of $S$ in its signature. A program context $P$ using $B$ can of course pass subtype objects to $B$, say of a type $T$, $T <: S$. Now, let us assume that the developer of $P$ decides to make a copy of class $T$, renames it to $U$, and uses $U$ instead of $T$; now, $U$ objects are passed to $B$. As neither $T$ nor $U$ are part of $B$’s signature, $B$’s instances should behave the same in both contexts. But in general, this is not true, because $B$ can observe the type difference of the received objects by casting to $T$ or $U$. Thus, a change to use $U$ instead of $T$ in the program context could cause $B$-boxes to behave differently. This would certainly surprise developers of the program context, because $T$ and $U$ are not in the signature of $B$. To support separate development of program contexts and boxes, we use solution (c) as sketched above as well for references received by a box.

Direct Component Semantics. Operational semantics are usually defined for complete executable programs. The semantics of components is only given indirectly in such an approach: One has to consider all possible program contexts and see how the component behaves in these contexts. This indirection complicates modular analysis and verification, in particular as the set of types in possible program contexts is infinite. We aim to provide a direct behavioral box semantics that is expressed in terms of the types of the box signature only. Thus, a box signature forms a symmetric boundary between two boxes – the inner box and the box forming the environment.

3 An Object-Oriented Language with Boxes

This section presents an object-oriented core-language with boxes. It is a classical OO-language that in addition supports box declarations and allows to use box names in creation expressions. The semantics of this languages is investigated in the following sections.

3.1 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: $L = L_1, L_2, \ldots, L_n$, where the length is defined as $|L| = n$. Similar, $T f_1; \ldots; f_n$ is equal to $T_1 f_1; \ldots; T_n f_n$. If there is some sequence $\pi$, we write $x_i$ for any element
of \( \pi \). We write \( \pi \cdot x \) for adding \( x \) to sequence \( \pi \), and \( \pi \circ \pi' \) for the concatenation of two sequences. The empty sequence is denoted by \( \bullet \). Functions \( f \) that are defined on element types are canonically lifted to sequences of elements, i.e. applying \( f \) to a sequence \( \pi \) of elements returns a sequence of the results, \( f(\pi) = f(x_1) \cdot f(x_2) \cdots f(x_n) \). We implicitly treat sequences as sets, e.g. if we write \( \pi_1 \subseteq \pi_2 \), both sequences are treated as sets.

Our language is similar to other core formalizations of Java, namely FJ \[14\] and CLASSICJAVA \[11\]. The main difference of our language is the addition of box declarations and that we extend the \texttt{new} expression to allow the creation of boxes. Our language supports stateful objects, classes, interfaces, aliasing, inheritance, single\(^8\) subtyping (only one supertype), and dynamic dispatch.

The syntax of the language is given in Fig. 1. A set of class, interface, and box declarations \( L \) is called \textit{declaration-complete} iff all names used in \( L \) have a declaration in \( L \). A \textit{program} \( P \) in our language is a declaration complete set \( L \).

A box declaration \texttt{box} \( B : I \) declares a box \( B \) with interface type \( T \). As explained below, the box signature is derived from \( T \). A box implementation directive \texttt{box} \( B \) \texttt{with} \( C \) declares the owner class \( C \) of a box. \( C \) has to be a subtype of \( T \). A program which contains a box declaration without a corresponding implementation directive is called a \textit{program context} for \( B \). A box implementation \( BI \) is a pair \((B, P_B)\) where \( B \) is a box name and \( P_B \) is a minimal declaration complete program which includes the box declaration and implementation directive for \( B \). \( P_B \) is called \( BI \)'s \textit{code base}. The box implementations are the \textit{program components} in our model. A program is called \textit{executable} if it has a box \texttt{Globox} realizing the predefined interface \texttt{Main} implemented by a class \texttt{MainClass} having a method \texttt{main}.

\begin{verbatim}
box Globox : Main with MainClass
    interface Main { Object main(Object input); }
    class MainClass implements Main { ... }
\end{verbatim}

The contextual constraints and the typing of the language are essentially as in Java (see \[12\]) and will not be repeated here. The differences will be explained in the following.

The subtype relation is denoted by \( < \cdot \), i.e. \( T <\cdot T' \) means that \( T \) is a subtype of \( T' \). The subtype relation is the reflexive and transitive closure of the relation induced by the class and interface declarations of the given program (see Fig. 2). We write \( <\cdot^+ \) to explicitly remove reflexivity. We assume a predefined interface \texttt{Object} without any methods. Different from Java is that we enforce single subtyping, i.e. every class is either the subtype of a class or of one

\(^8\) The restriction to single subtyping is only made to simplify the presentation.
interface. An interface can only have a single supertype which can be Object. This restriction is merely to simplify the presentation and not necessary to obtain the results. We do not 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 has a single body expression which yields the return value of the method. Expressions can be variables, the null constant, cast expressions, new-expressions, field accesses, field updates, let-expressions, method calls, and conditionals. If the name in a new-expression denotes a box, a box owner of a new box is created; otherwise, a standard object is created in the current box. let-expressions are used to support local variables and sequential expression evaluation.

3.2 Semantics Overview

The rest of this paper presents several semantics for the language given above. Figure 3 gives an overview. The semantics marked as equivalent are semantics for executable programs. The summarized small-step message semantics is also equivalent to the small-step message semantics, but it is only used as
4 Standard Semantics with Boxes

In this section we present a small-step operational semantics for our language. It uses standard techniques to define object-oriented language semantics, and adds the treatment of boxes. The operational semantics is given in reductional small-step style, that is, we represent a configuration, by the states of the created objects and a partially evaluated expression over dynamic values.
Fig. 4. Definitions for the configurations of the standard semantics

4.1 Configurations

Figure 4 contains the definitions for the configurations. Objects are represented by a globally unique identifier $j$ together with their class name. The state $O$ of an object is represented by a sequence $\tau$ of the values of its fields. We assume that the fields of a class are totally ordered, so we need no mapping from field names to values. The mapping $O$ captures for each object its state.

A box $b$ is either the global box globox which includes all other boxes, or it is created at runtime and represented by a globally unique identifier $k$.

The state of an executing program is represented by a quintuple $\langle C, O, N, P, R \rangle$. $C$ is an execution stack of pending method executions. It also records the box in which the pending expression has been reduced. The mapping $N$ assigns every object the box to which it belongs. The mapping $P$ records the parent relation on boxes, i.e., if $P(k) = k'$ means that $k$ is an inner box of $k'$. This relation is not used by the standard semantics, but helps to show equivalence with the message semantics developed below. $R$ assigns a box signature to every box; we explain box signatures in Subsec. 4.3.

To represent partially evaluated expressions, the syntax of expressions is extended to allow objects and the special variable result – indicating that the expression expects the return value of a pending call.

Figure 5 defines some auxiliary functions to derive information from a program.
box $B$ with $C$  
$\text{impl}(B) = C$

box $B : T$  
$\text{intf}(B) = T$

class $C$ impls $I$  
$\{ T \ f; \ldots \}$

fields$(C) = T \ f$

\[
\begin{align*}
\text{class } C &\text{ extends } C' \{ T \ f; \ldots \} \\
\text{fields}(C) &= T \ f \circ \text{fields}(C')
\end{align*}
\]

\[
\begin{align*}
\text{class } C &\text{ extends } D \{ \ldots ; M \} \\
m \notin M
\end{align*}
\]

\[
\text{mbody}(C, m) = \text{mbody}(D, m)
\]

Fig. 5. Auxiliary functions

4.2 Reduction Rules

The standard reduction semantics is a relation on configurations $S, b, e$ consisting of the state $S$, the currently executing box $b$, and the currently reduced expression $e$. The judgment $S, b, e \rightarrow^r S', b', e'$ expresses that reducing $e$ in state $S$ and box $b$ leads in one step to $S', b', e'$.

An evaluation context $e\Box$ is an expression with a “hole” $\Box$ somewhere inside the expression. We write $e\Box[e]$ to replace the hole in $e\Box$ by expression $e$. A hole in $e\Box$ can only appear at certain positions defined as follows:

\[
e\Box ::= \Box | (T)e\Box | e\Box.f | e\Box.f = e | v.f = e\Box | \text{let } x = e\Box \text{ in } e \\
\text{if } (e\Box = e) \text{ then } e \text{ else } e | \text{if } (v = e\Box) \text{ then } e \text{ else } e \\
| e\Box.m(\tau) | v.m(\tau, e\Box, \tau)
\]

The reduction rules are shown in Figure 6. Object creation creates a new object inside the current box (see rule r-new-obj). The function next yields the next object identifier that is available. Box creation creates a new box $k$ and a new object $j$; it places the new object $j$ into $k$ and $k$ into the current box; $R'$ records the signature for the new box (see rule r-new-box).

Field access is restricted to box local objects by checking that the receiver belongs to the current box (see rules r-field-access and r-field-update). This reflects the idea that a box encapsulates its objects and simplifies the message and behavioral semantics (otherwise messages for field access would be needed).

Method invocations are handled by two rules. The r-call rule replaces the method invocation expression by result and pushes the resulting expression together with the currently executing box onto the execution stack. The currently executing box is set to the box which owns the receiver object of the call and establishes the body expression of the called method as the new reduction expression, after substitution of formal by actual parameters. If an expression
\[
\begin{align*}
\textbf{R-NEW-OBJ} & \quad j = \text{next}(\text{dom } O) \quad o = j:C \quad \text{fields}(C) = T \quad \overline{\text{null}} = |\overline{J}| \\
\langle C, O, N, P, R \rangle, b, e \sqsubseteq [\text{new } C] & \rightarrow_{\text{r}} \langle C, O[j \mapsto \overline{\text{null}}], N[j \mapsto b], P, R \rangle, b, e \sqsubseteq [\text{o}] \\
\textbf{R-NEW-BOX} & \quad k = \text{next}(\text{dom } P) \quad \text{impl}(B) = C \quad o = j:C \quad \text{fields}(C) = T \quad \overline{J} \\
\langle C, O, N, P, R \rangle, b, e \sqsubseteq [\text{new } B] & \rightarrow_{\text{r}} \langle C, O[j \mapsto \overline{\text{null}}], N[j \mapsto k], P', R' \rangle, b, e \sqsubseteq [\text{o}] \\
\textbf{R-FIELD-ACCESS} & \quad o = j:C \quad N(j) = b \quad \text{fields}(C) = T \quad \overline{J} \quad O(j) = \overline{v} \\
\langle C, O, N, P, R \rangle, b, e \sqsubseteq [\text{f} \cdot i] & \rightarrow_{\text{r}} \langle C, O, N, P, R \rangle, b, e \sqsubseteq [\text{v}_i] \\
\textbf{R-FIELD-UPDATE} & \quad o = j:C \quad N(j) = b \quad \text{fields}(C) = T \quad \overline{J} \quad O(j) = \overline{v} \\
\langle C, O, N, P, R \rangle, b, e \sqsubseteq [\text{f} \cdot i = v] & \rightarrow_{\text{r}} \langle C, O, N, P, R \rangle, b, e \sqsubseteq [\text{v}] \\
\textbf{R-CALL} & \quad o = j:C \quad \text{mbody}(C, m) = (\overline{\tau}e) \quad b' = N(j) \quad e' = [o/\text{this}, \overline{\tau}/\overline{\tau}]e \\
\langle C, O, N, P, R \rangle, b, e \sqsubseteq [\text{m} \cdot (\overline{\tau})] & \rightarrow_{\text{r}} \langle C, O, N, P, R \rangle, b', e' \\
\textbf{R-RETURN} & \quad S = \langle [b', e \sqsubseteq \text{result}] \cdot C, O, N, P, R \rangle \\
\langle C, O, N, P, R \rangle, b', e \sqsubseteq [\text{v}] & \rightarrow_{\text{r}} \langle C, O, N, P, R \rangle, b', e \sqsubseteq [\text{v}] \\
\textbf{R-CAST-NULL} & \quad S, b, v \rightarrow_{\text{r}} \langle C, O, N, P, R \rangle, b', e \sqsubseteq [\text{v}] \\
\textbf{R-CAST-OBJ} & \quad \langle C <: T \rangle \quad b' = N(j) \quad T' = \sigma^+(\text{crossedSigs}_{P, R}(b', b), C) \quad T' <: T \\
\langle C, O, N, P, R \rangle, b, e \sqsubseteq [\text{T}] & \rightarrow_{\text{r}} \langle C, O, N, P, R \rangle, b, e \sqsubseteq [\text{j:C}] \\
\textbf{R-LET} & \quad S, b, e \sqsubseteq [\text{let } x = v \text{ in } e] \rightarrow_{\text{r}} S, b, e \sqsubseteq [\text{v/x/e}] \\
\textbf{R-IF-TRUE} & \quad v = v' \\
\langle C, O, N, P, R \rangle, b, e \sqsubseteq [\text{if } (v = v') \text{ then } e \text{ else } e'] & \rightarrow_{\text{r}} S, b, e \sqsubseteq [e] \\
\textbf{R-IF-FALSE} & \quad v \neq v' \\
\langle C, O, N, P, R \rangle, b, e \sqsubseteq [\text{if } (v = v') \text{ then } e \text{ else } e'] & \rightarrow_{\text{r}} S, b, e \sqsubseteq [e'] \\
\end{align*}
\]

Fig. 6. Reduction rules of the standard semantics with boxes
cannot be further reduced, i.e., is a value $v$, and the execution stack is not empty, the top element from the execution stack is used to continue execution: the box and the currently evaluated expression of the caller with $v$ for `result` define the new configuration (see rule `r-return`).

As mentioned before, our semantics restricts casts to enforce the encapsulation of box internal types from the environment and to enforce the encapsulation of environment internal types to a box (rule `r-cast-obj`). Reducing a cast expression is only possible if the type to cast to is element of all box signatures, which are on the path between the owner box of the referencing object and the owner box of the referenced object. The details of the used functions are given in the following section.

### 4.3 Box Trees and Box Signatures

The parent relation $\mathcal{P}$ assigns to each box $k$ its parent box $b$. The relation forms a tree with `globbox` at the root. For $\mathcal{P}(k) = b$, we also write $k \prec \mathcal{P} b$. The transitive closure of $\prec$ is denoted by $\prec^+$ and the transitive, reflexive closure by $\prec^*$.

As explained in Sec. 2, a component has to declare which types are known at its boundary. In our model, this is captured by the notion of a `box signature`. Box signatures define a type boundary for boxes. Their purpose is to encapsulate implementation types of a box and to decouple a box from the types of its environment. This boundary allows to change the implementation of types used in a box implementation without changing the visible box behavior. This restriction to box signatures has to be reflected in casts, as they allow to observe the concrete dynamic type of an object.

The signature of a box $B$ is a finite set of types. These types are belong to both the code base of the box and of the environment. A box declaration `box B : I` implicitly defines the signature $B_{\text{sig}}$ of the box $B$. It is the smallest set satisfying:

1. $I \in B_{\text{sig}}$.
2. If $T \in B_{\text{sig}}$, all types appearing as parameter or return types in a method signature of $T$ belong to $B_{\text{sig}}$.
3. If $T \in B_{\text{sig}}$, the supertypes of $T$ belong to $B_{\text{sig}}$.

Thus, `Object` is part of every box signature.

When casting a reference to a certain type $T$ we have to check whether type $T$ is in the intersection of all box signatures which have to be crossed when following the box tree from the current box to the box owning the cast object.
Definition 1 (crossedBoxes) The function crossedBoxes yields the sequence of boxes whose signatures has to be crossed when following the box tree to get from one box to another box.

\[
\text{crossedBoxes}_P(b, b') \overset{\text{def}}{=} \begin{cases} 
\bullet & \text{if } b = b' \\
\cdot \text{crossedBoxes}_P(P(b), b') & \text{if } b \prec_P b' \\
\text{crossedBoxes}_P(b, P(b')) \cdot b' & \text{if } b' \prec_P b \\
\cdot \text{crossedBoxes}_P(P(b), b') & \text{else}
\end{cases}
\]

It is easy to see that the parent function \( P \) is never applied to the root of the box tree and that the distance of \( b \) and \( b' \) in the tree is always reduced. Thus, \( \text{crossedBoxes}_P \) is well-defined if \( P \) defines a tree. Note that \( \text{crossedBoxes}_P(b, B') \) never contains \text{globox}. Thus, we can safely apply lifted signature mappings \( \mathcal{R} \) to the elements of \( \text{crossedBoxes}_P(b, B') \):

\[
\text{crossedSigs}_{P, \mathcal{R}}(b, b') \overset{\text{def}}{=} \mathcal{R}(\text{crossedBoxes}_P(b, b'))
\]

The type restriction of references that cross box boundaries is done by the function \( \sigma \) that takes a box signatures \( B_{\text{sig}} \) and a type \( T \) and yields the most specific supertype of \( T \) in \( B_{\text{sig}} \):

\[
\sigma(B_{\text{sig}}, T) \overset{\text{def}}{=} T' \quad \text{where } T' \in B_{\text{sig}} \land T' <: T \land \not\exists T'' \in B_{\text{sig}} : T <: T'' <:+ T'
\]

Here, the single subtype restriction made for our language simplifies the presentation. The most specific supertype which is allowed for a given type in a sequence of box signatures, is expressed by \( \sigma^+ \):

\[
\sigma^+(\bullet, T) \overset{\text{def}}{=} T \\
\sigma^+(B_{\text{sig}} \cdot B_{\text{sig}}, T) \overset{\text{def}}{=} \sigma^+(B_{\text{sig}}, \sigma(B_{\text{sig}}, T))
\]

4.4 Initial Configurations

Given an executable program \( P \), let \( e_{\text{main}} \) be the body of the main method; \( C \) some class in \( P \); \( O_0 \) some reference closed object store capturing the input objects; \( N_0 \) such that it maps all \( j \) in \( \text{dom} \; O_0 \) to \text{globox}; \( j_0: \text{Main}, j_1:C \in O_0 \). Then, the initial configuration is defined as:

\[
\langle \bullet, O_0, N_0, \emptyset, \emptyset, \text{globox}, [j_0: \text{Main}/\text{this}, j_1:C/\text{input}] \rangle e_{\text{main}}
\]

\footnote{Recall that we implicitly lift functions defined on single elements to lists of elements, cf. Sec. \[3\].}

\footnote{The single subtype restriction can be eliminated by considering sets of types for external references.}
5 Message Semantics

The standard semantics demonstrates how heap-structures in general and boxes in particular can be expressed in a semantics with global state. In this section we present a small-step message semantics for our object-oriented language with boxes. This message semantics distributes the global state into box local states. Method calls between different boxes are represented by messages. The representation of references to the same object may vary between different boxes, i.e., references address objects relative to a current box and not absolute as usual. Furthermore we distinguish between reference representations inside a box, so-called intra-box references and those that are passed over a boundary, so-called inter-box references. Consequently, the representation of references has to change during transmission of messages from one box to another. This technique to deal with references has two important advantages:

(1) Objects can be created locally without knowing the identifiers of all other objects in the system.
(2) The representation of objects can be normalized at boundaries. This is important to make the representation of objects passing a boundary of a box b independent of b’s environment and of b’s implementation.

Method calls are handled by two messages, a call message and a return message. Messages may be executed locally or passed to other boxes. Analogous to the distinction between inter- and intra-box references, we distinguish between inter- and intra-box messages. Sequences of inter-box messages are later used to describe the denotation of a box implementation.

5.1 Configurations

Figure 7 shows the needed definitions for the configurations of the message semantics. They will be explained in the following subsections.

5.1.1 Reference and Message Representations

Object references are divided into intra-box references $o_b$ and inter-box references $o_d$. Intra-box references are used throughout the reduction rules that handle box internal computations. They are stored in locations of the box local object store. Inter-box references are only used in inter-box messages, i.e., during the transmission of messages across box boundaries. They are generated when a reference to an object crosses a boundary for the first time. Intra-box references are either created during object instantiation, box instantiation or are generated when a reference crossed a box boundary for the first time.
o_b ::= b.j:T  
 intra-box references
o_d ::= d.j:T  
 inter-box references
v_b ::= o_b | null  
 intra-box values
v_d ::= o_d | null  
 inter-box values
O ::= v_b  
 object states
B ::= ⟨C, O, I, R⟩  
 box states
C ::= δb  
 execution stacks
eb ::= e | b  
 execution stack elements
e ::= . . . | result | o_b  
 extended expressions
r ::= e | n_b | n_d  
 reduction terms
n_b ::=  
  | →o_b.m(ν_b)  
  | ←ν_b  
  inter-box messages  
  call message  
  return messages
n_d ::=  
  | o_d.m(ν_d)  
  | ν_d  
  inter-box messages  
  call messages  
  ingoing return messages  
  outgoing return messages
O ::= j → O  
 object stores
I ::= k → B  
 inner boxes
R ::= b → Q  
 box mappings
Q ::= (B_{sig}, M)  
 signature-boundary tuple
M ::= o_b ↔ o_d  
 intra-inter mappings
B_□ ::= □ | ⟨C, O, I_□, R⟩  
 box states with a hole
I_□ ::= I[k → B_□]  
 inner boxes with a hole
b ::= own | env | k  
 box prefixes
d ::= in | out  
 direction prefixes
j ∈ N  
 object identifiers
k ∈ N  
 box identifiers

Fig. 7. Definitions for the configurations of the message semantics

The different kinds of references capture the idea of “seeing” an object from different boxes. This context dependent representation allows us to use relative references. That is, the same object is denoted by different relative references in different boxes. Figure 8 shows some examples. The light gray areas represent boundaries of boxes, the white areas inside the boundaries represented the boxes. Thus, we see a box with two inner boxes. The small rectangles with dark gray background show how the reference for an object is represented in the box to which the object belongs; this is indicated by the prefix own. The small rectangles with a white background show the reference representation of objects within other boxes and at the boundary. Rectangles that are connected by an arrow represent the same physical object, but from different perspectives. The rounded boxes represent intra-inter mappings R, the arrows crossing a rounded box are the elements of that mapping. In the message semantics, R also handles the renaming of references. Now, we explain the details of the reference representation (cf. Fig. 7).
**Intra-Box References.** An intra-box reference $o_b = b.j : C$ consists of a box prefix $b$ which identifies the relative position of the owner box of $o_b$, a box-local identifier $j$, and the dynamic type $C$ of the object. If $b = \text{own}$, we say that $o_b$ is a local reference, otherwise it is a non-local reference, $b = \text{env}$ means the object is owned somewhere in the environment of the current box. For local references, we know that $j$ is in the domain of the local object store of the box, $j \in \text{dom} \mathcal{O}$; for non-local references, we know that $b.j$ is in the domain of the object mapping $\mathcal{R}(b)$, $b.j \in \text{dom} \mathcal{R}(b)$ where $\mathcal{O}$ and $\mathcal{R}$ belong to the state of the currently executing box as will be explained below.

**Inter-Box References.** An inter-box reference $o_d = d.j : C$ is specific for a boundary between two boxes. Because the boxes are structure into a tree, boundaries are asymmetric. They have an inner and an outer side. If $d = \text{out}$ then $o_d$ refers to an object belonging to the outside, otherwise it refers to an object in the box or some inner box.
Boundary Type Adaption. As explained in Sec. 2 and with the standard semantics, the type of an object crossing a boundary of a box \( B \) should not be more specific as the type in \( B_{\text{sig}} \). Whereas the standard semantics performs the corresponding type adaption in one step within cast expressions, the messages semantics realizes the type adaption stepwise whenever a reference crosses a boundary.

Consider the scenario in Fig. 8 with \( C_2 \prec I_3 \prec I_2 \prec I_1 \). We assume that the outer box signature consists of \( I_1 \), \( C_0 \), and \( C_1 \), that the signature of the left inner box consists of \( C_1 \), \( C_3 \), and \( I_2 \), and that the signature of the right inner box consists of \( C_1 \), \( C_4 \), and \( I_3 \). In addition, they all include \texttt{Object}. Every type that occurs in an inter-box reference has to be part of the box signature corresponding to that boundary. For example, reference \texttt{env.2:} \( I_2 \) originally had type \( C_2 \) which does not belong to the signature of the left inner box, namely \( C_1 \), \( C_3 \), and \( I_2 \). That is why the type was adapted to \( I_2 \). The references that were adapted to a supertype are surrounded with a double rectangle in Fig. 8. The type adaption of a reference makes sure that the original type of a non-local object can only be observed up to the types of the box signature.

Messages. Like references and for the same reasons, messages are represented differently within a box, \textit{intra-box messages}, and at the boundary of boxes \textit{inter-box messages} (cf. Fig. 7). Intra-box messages are based on intra-box references and have no direction. Inter-box messages are based on inter-box references and have a direction. For a call, this direction is given by the direction prefix of the receiver. For a return, it is part of the message representation.

5.1.2 States

In the message semantics, the box tree is represented by a box environment and the currently executing box. The box environment captures the state of all surrounding boxes and their children except for the current box. The state of a box consists of a local call stack \( C \), an object store \( O \), an inner box store \( I \) and a mapping \( R \):

- Similar to the standard semantics, \( C \) comprises the contexts of active method calls. In addition, it captures routing information, e.g. a routing information \texttt{env} on top of the stack means that the next return message to be handled has to be forwarded to the environment.
- \( O \) stores the values of the objects belonging to the current box.
- \( I \) captures the states of the inner boxes.
- \( R \) maintains for every inner box as well as for the environment a bijective \textit{object mapping} \( M \) from intra-box to inter-box references and vice versa.
To concisely describe the box environment of the current box, we introduce box states with a “hole” $B□$. A box state with a hole is a normal box state which has at one position of the box tree a “hole” $□$. That position is either the root or the state of an inner box. In the latter case, we use a special inner box mapping with a hole $I□$. Such a mapping is a standard inner box mapping which yields for one inner box a box state with a hole. We write $B□[A]$ to replace the hole in $B□$ with $A$, where $A$ is either a box state with a hole or a box state without a hole. Note that $B□[□] = B□$ and that the hole can appear at the top position in which case $B□ = □$.

5.2 Auxiliary Functions and Object Mappings

Before presenting the reduction rules of the message semantics, we need to define some auxiliary functions and object mappings. By dom and cod we denote the domain and co-domain of a mapping, respectively.

**Definition 2 (id)** The function id returns the identifier of a reference.

\[
id(b.j:T) \overset{\text{def}}{=} j
\]

\[
id(d.j:T) \overset{\text{def}}{=} j
\]

**Definition 3 (next)** The function next returns the next free primitive identifier for a given list of references or primitive identifiers.

\[
\text{next}(\bullet) \overset{\text{def}}{=} 0
\]

\[
\text{next}(\overline{k}) \overset{\text{def}}{=} 1 + \max(\overline{k})
\]

\[
\text{next}(\overline{j}) \overset{\text{def}}{=} 1 + \max(\overline{j})
\]

The functions forward and receive (see Fig. 9) translate intra-box messages to inter-box messages and vice versa using the bijective mapping in $Q$ of intra- and inter-box references. They use the functions export and import to translate the references (Fig. 10). All four functions take an additional parameter $b$ or $d$ which designates the boundary for which the inter-box reference is to be translated.

The important aspect of these functions is that they normalize the references: When a reference passes a boundary for the first time, a new representation is chosen. Its identifier only depends on the references that have passed the boundary before. This is realized by the function next. Thus, whatever a box implementation or the environment looks like, the $n$th reference coming in or going out has always the same representation.
forward\((k, \leftarrow v_b, Q) \overset{\text{def}}{=} ([v_d, Q'])\) with \((v_d, Q') = \text{export}(\text{in}, v_b, Q)\)

forward\((\text{env}, \leftarrow v_b, Q) \overset{\text{def}}{=} ([v_d, Q'])\) with \((v_d, Q') = \text{export}(\text{out}, v_b, Q)\)

receive\((k, \leftarrow v_b, Q) \overset{\text{def}}{=} ([v_d, Q'])\) with \((v_d, Q') = \text{import}(k, v_b, Q)\)

receive\((\text{env}, \leftarrow v_b, Q) \overset{\text{def}}{=} ([v_d, Q'])\) with \((v_d, Q') = \text{import}(\text{env}, v_b, Q)\)

receive\((b, \leftarrow o_d.m(v_d), Q) \overset{\text{def}}{=} ([v_d, Q'])\) with \((v_d, Q') = \text{import}(b, o_d, Q)\)

receive\((b, \leftarrow o_d.m(v_d), Q) \overset{\text{def}}{=} ([v_d, Q'])\) with \((v_d, Q') = \text{import}(b, v_o, Q)\)

forward\((b, \rightarrow (b'.j : T).m(v_b), Q) \overset{\text{def}}{=} ([o_d.m(v_o), Q']_{n+1})\)

with \((o_d, Q_0) = \text{export}(b', k, o_b, Q)\)

\((v_{d_0}, Q_1) = \text{export}(b', k, v_{b_0}, Q_0)\)

\vdots

\((v_{d_n}, Q_{n+1}) = \text{export}(b', k, v_{b_n}, Q_n)\)

receive\((k, \rightarrow v_b, Q) \overset{\text{def}}{=} ([v_d, Q'])\) with \((v_d, Q') = \text{import}(k, v_b, Q)\)

receive\((\text{env}, \rightarrow v_b, Q) \overset{\text{def}}{=} ([v_d, Q'])\) with \((v_d, Q') = \text{import}(\text{env}, v_b, Q)\)

receive\((b, \rightarrow o_b.m(v_b), Q) \overset{\text{def}}{=} ([v_b, Q']_{n+1})\)

with \((o_b, Q_0) = \text{import}(b, o_b, Q)\)

\((v_{b_0}, Q_1) = \text{import}(b, v_{d_0}, Q_0)\)

\vdots

\((v_{b_n}, Q_{n+1}) = \text{import}(b, v_{d_n}, Q_n)\)

Fig. 9. Definitions of the forward and receive functions

5.3 Reduction Rules

The message semantics is expressed by the judgment

\[ B_\square, B, r \xrightarrow{rM} B_\square', B', r' \]

which means that reducing an expression \(r\) or handling a message \(r\) in box state \(B\) and under box environment \(B_\square\) leads in one step to \(B_\square', B', r'\). The reduction rules are shown in three figures: rules for the reduction of expressions are shown Fig. 11, rules translating expressions to intra-box messages and vice-versa are shown in Figure 12, and rules translating intra-box messages to inter-box messages and vice-versa are shown in Figure 13.

The expression reduction rules are essentially like the ones in the standard semantics. They only use the state of the currently executing box and do not refer to the box environment. One main difference lies in the \(\text{RM-CAST-OBJ}\) rule compared to the \(\text{R-CAST-OBJ}\) rule of the standard semantics. The new
\[
\begin{align*}
\text{export}(b, \text{null}, Q) & \overset{\text{def}}{=} (\text{null}, Q) \\
\text{import}(b, \text{null}, Q) & \overset{\text{def}}{=} (\text{null}, Q)
\end{align*}
\]

\[
\begin{align*}
\text{import}(b, d.j:T, (B_{\text{SIG}}, M)) & \overset{\text{def}}{=} \\
\begin{cases}
(M^{-1}(o_d), (B_{\text{SIG}}, M)) & o_d \in \text{cod}(M) \\
(o_b, (B_{\text{SIG}}, \{o_b \leftrightarrow o_d\} \cup M)) & (b = k \land d = \text{in}) \lor (b = \text{env} \land d = \text{out}) \\
\bot & \text{else}
\end{cases}
\end{align*}
\]

\[
\begin{align*}
\text{export}(b, b'.j:T', (B_{\text{SIG}}, M)) & \overset{\text{def}}{=} \\
\begin{cases}
(M(o_b), (B_{\text{SIG}}, M)) & o_b \in \text{dom} M \\
(o_d, (B_{\text{SIG}}, M \cup \{o_b \leftrightarrow o_d\})) & \text{else}
\end{cases}
\end{align*}
\]

Fig. 10. Definition of the import and export functions

The rule does not need to perform the type adaption, as this is done when reference pass boundaries (see below).

The rule for creating new boxes, \((\text{RM-NEW-BOX})\), is interesting. It shows how the inter-intra-box mappings are initialized for the object created in the new box.

The intra-box message rules (Fig. 12) handle method calls and returns. Method calls are translated by rule \((\text{RM-HANDLE-CALL})\) into an intra-box call message. The fact that the call is started locally is recorded on the stack. If the current
Fig. 11. Expression reduction rules of the message semantics

reduction term is a single value, which means that a method execution has been terminated, this value is translated to an intra-box return message (RM-HANDLE-RETURN). Message are carried to the receiver box by rule explained below. On arrival, they are executed. An intra-box call message is executed.
by rule (RM-EXEC-CALL) if the receiver object \( o_b \) belongs to the current box. Intra-box return messages are executed if top stack element is an expression with a result hole (RM-EXEC-RTRN). They are handled by taking this expression and replacing result by the return value.

The inter-box message rules (Fig. 13) handle the communication between different boxes. They use the functions forward and receive for the translation between intra-box and inter-box messages. This translation not only converts the different identifiers but also adapts the type of the reference to the most specific type which is allowed by the box signature of the crossed box boundary. There are eight different cases. The first four rules translate intra-box messages into inter-box messages. Depending on the receiver object, an intra-box call message can either be forwarded to the environment (RM-FWD-CALL-ENV) or to some inner box (RM-FWD-CALL-INNER). When forwarding a return message, the execution stack must be considered (RM-FWD-RTRN-ENV, RM-FWD-RTRN-INNER). If a message is sent to an inner box, the hole of the box environment is replaced by the currently executing box, with the new hole being the state of the receiving inner box. The last four rules handle inter-box message and translate them to intra-box messages. The inter-box message can either come from the environment (RM-RCV-CALL-ENV, RM-RCV-RTRN-ENV) or the currently executing box is sending a message to the environment (RM-RCV-CALL-INNER, RM-RCV-RTRN-INNER). In the latter case the executing box is changed to the parent box and the box environment is changed to the environment of the parent box.
5.4 Initial Configurations

Initial configurations of the message semantics are similar to those of the standard semantics. Given an executable program $P$, let $e_{\text{main}}$ be the body of the main method; $C$ some class in $P$; $O_0$ some reference closed object store. Then, the initial configuration is defined as:

$$\square, \{\bullet, O_0, \varnothing, \varnothing\}, [\text{own}_{j_0}: \text{Main/this}, \text{own}_{j_1}: C/\text{input}] e_{\text{main}}$$
5.5 Discussion

The presented message semantics has three central properties:

1. The semantics is equivalent to the standard semantics. Each step in the standard semantics corresponds to one or several steps in the message semantics. If it corresponds to several steps, the number of steps depends on the distance of the sender and receiver in the box tree.

2. Only the last two rules \textsc{rm-rcv-call-inner} and \textsc{rm-rcv-rtrn-inner} make use of the state of the box environment.

3. The references are normalized for each boundary in the sense that the \textit{n}th reference coming in or going out always has the same representation, independent of the box implementation and program context.

The first property guarantees that the message semantics, though complex, reflects the expected runtime behavior. The two other properties breach the way to a behavioral semantics.

6 Equivalence of Standard and Message Semantics

In this section we present a proof sketch that the message semantics of Sec. 5 is equivalent to the standard semantics of Sec. 4. In particular, programs executed with either semantics are not distinguishable by the programmer or user.

To justify this proposition we show that the standard semantics is bisimilar to a so-called \textit{summarized} message semantics. The summarized message semantics is derived from the message semantics by hiding all states, which have no bisimilar state in the standard semantics. We argue that there are no infinite successive chains of those hidden states and they only occur when the control flow is jumping to or back from a message in another box. Thus, all states which are the result of a state change in the standard semantics have a bisimilar state in the summarized message semantics.

6.1 Bisimulation and Summarized Message Semantics

We use a definition of strong bisimulation, following the one in [19, p. 88].

\textbf{Definition 4 (Strong Bisimulation)} Let \((S_a, \Lambda, \rightarrow_a)\) and \((S_b, \Lambda, \rightarrow_b)\) be labeled state transition systems with sets of states \(S_i\), sets of labels \(\Lambda\), and transition relations \(\rightarrow_i\). \(R \subseteq (S_a \times S_b)\) is called a strong bisimulation relation...
if

\[
\forall \lambda \in \Lambda; s, s' \in S_a; t \in S_b: \left(R(s, t) \land \left(s \xrightarrow{\lambda} s'\right)\right) \\
\Rightarrow \left(\exists t' \in S_b: t \xrightarrow{\lambda} t' \land R(s', t')\right)
\]  

(4.1)

\[
\forall \lambda \in \Lambda; s \in S_a; t, t' \in S_b: \left(R(s, t) \land \left(t \xrightarrow{\lambda} b t'\right)\right) \\
\Rightarrow \left(\exists s' \in S_a: s \xrightarrow{\lambda} s' \land R(s', t)\right)
\]  

(4.2)

In the following, we denote the set of all configurations \(S, b, e\) of the standard semantics by \(CONF_r\), and the set of all configurations \(B, B, e\) of the message semantics by \(CONF_{rm}\). We use the metavariable \(c\) to range over configurations, augmented by the indices \(r\) and \(rm\) if the semantics is not clear from the context.

For all rules of all semantics presented in this article, labels are given at their definition site with a prefix in order to relate a rule to a certain semantics. When talking about labels of rules in a mathematical sense, we omit that prefix, e.g. \(r, rm\). The set of labels without the prefix \(r-\) of the reduction rules of the standard semantics (Fig. 6) is denoted by \(\Lambda^*\).

There is no strong bisimulation between the standard semantics and the small-step message semantics, because in the latter semantics the calls and returns are handled by several reduction steps. The intermediate configurations have no corresponding configuration in the domain of standard semantics. Therefore, we define a variant of the message semantics which omits those configurations.

**Definition 5 (Summarized Message Semantics)** The summarized message semantics relation \(\xrightarrow{\text{rm}}\) is identical to the message semantics relation \(\xrightarrow{\text{rm}}\), with the following exception: Every occurring chain of steps of the form

\[
C_0 \xrightarrow{\text{HANDLE-CALL}}_{\text{rm}} C_1 \xrightarrow{\text{FWD-CALL-\_}}_{\text{rm}} C_{1+i} \xrightarrow{\text{RCV-CALL-\_}}_{\text{rm}} C_{2+n} \xrightarrow{\text{EXEC-CALL}}_{\text{rm}} C_{3+n}
\]

is replaced by a single step \(C_0 \xrightarrow{\text{CALL}}_{\text{rm}} C_{3+n}\) and every chain of steps of the form

\[
C_0 \xrightarrow{\text{HANDLE-RTRN}}_{\text{rm}} C_1 \xrightarrow{\text{FWD-RTRN-\_}}_{\text{rm}} C_{1+i} \xrightarrow{\text{RCV-RTRN-\_}}_{\text{rm}} C_{2+n} \xrightarrow{\text{EXEC-RTRN}}_{\text{rm}} C_{3+n}
\]

is replaced by a single step \(C_0 \xrightarrow{\text{RETURN}}_{\text{rm}} C_{3+n}\)

Let \(m\) be the number of box boundaries between the calling and the called method, then the length of the replaced chains is exactly \(2 + 2 \ast m\). As the
number of box boundaries is always finite, the length of the replaced chains is finite, too.

The most challenging part of the equivalence between the two semantics is the definition of the bisimulation relation. This is the topic of the next subsection.

6.2 Bisimulation Relation on Configurations

We define a bisimulation relation between the states of the standard and message semantics. As we use definitions from both semantics, we use \( R \) and \( RM \) as additional indices to resolve ambiguities.

**Definition 6 (bp)** The function \( \text{bp} \) returns for a given box environment the path of box identifiers which leads from the root to the box hole.

\[
\begin{align*}
\text{bp}(\square) & \triangleq \bullet \\
\text{bp}(\langle C, O, I[k_{RM} \mapsto B\square], R \rangle) & \triangleq k_{RM} \cdot \text{bp}(B\square)
\end{align*}
\]

An identifier translator function \( \pi \) maps box-local object or box identifiers from the semantics domain of the message semantics to the semantics domain of the standard semantics. \( \Box \) is used as the box identifier of the current box.

**Definition 7 (Identifier Translator Functions)** Let \( \pi : k_{RM} \times (j_{RM} \cup \{\Box\}) \rightarrow (j_R \cup b_R) \) be a function. Then, we call \( \pi \) an identifier translator function. The set of all identifier translator functions is denoted by \( \Pi \). We write \( \pi_K \) for a partially applied \( \pi \).

The following definition extends a identifier translator \( \pi \) to the value level. It requires the whole context \( B\square, B \) because it needs the mappings at each box boundary in order to handle references to non-local objects.

**Definition 8 (Value Translator Function)** Let \( \pi \) be an identifier translator function. We write \( \hat{\pi} \) for \( \pi \) lifted to values. \( \hat{\pi} : B\square \times B \times (v_b \cup \{\Box\}) \rightarrow v_R \cup b_R \)

\[
\begin{align*}
\hat{\pi}(B\square, B, null) & \triangleq null \\
\hat{\pi}(B\square, B, \Box) & \triangleq \pi(\text{bp}(B\square), \Box) \\
\hat{\pi}(B\square, B, \text{own}.j_{RM}.T) & \triangleq \pi(\text{bp}(B\square), j_{RM}).T
\end{align*}
\]
We use $[\hat{\pi}\Box B, B] e_{\text{rm}}$ as notation for a substitution of all values $v_b$ in $e_{\text{rm}}$ with their corresponding $v_b = \hat{\pi}(\Box B, B, v_b)$. This results in a $e_{\text{r}}$.

Using $\pi$ and $\hat{\pi}$, we stepwise define a mapping $\text{glob}_\pi$ from (box environment state,box state)-pairs of the message semantics to (state,box)-pairs in the standard semantics. That is, $\text{glob}_\pi$ merges the tree of heaps into a global heap and the local stacks into a global stack. $\text{glob}_\pi$ is parameterized by a given identifier translator function $\pi$. According to the state of the standard semantics, the definition comes in five steps. This first step defines a function that merges the distributed execution stacks into one global execution stack.

**Definition 9** ($\text{glob}_\pi^C$)

$$
\text{glob}_\pi^C(\Box, \langle \cdot, O, I, R \rangle) 
\triangleq \cdot
$$

$$
\text{glob}_\pi^C(B_\Box, e_{\text{rm}} \cdot C, O, I, R) 
\triangleq (\pi(B_\Box, \Box), [\hat{\pi}_{B_\Box, B}] e_{\text{rm}} \cdot \text{glob}_\pi^C(B_\Box, C, O, I, R))
$$

$$
\text{glob}_\pi^C(B_\Box, e_{\text{rm}} \cdot C, O, I, R) 
\triangleq \text{glob}_\pi^C(\Box, \langle k \mapsto \Box, C, O, I, R \rangle)
$$

$$
\text{glob}_\pi^C(B_\Box, \langle k \mapsto C, O, I, R \rangle) 
\triangleq \text{glob}_\pi^C(\Box, \langle k \mapsto \Box, C, O, I, R \rangle)
$$

In the next step, we define the merge of the box-local object stores into one object store.
Definition 10 ($\text{glob}_\pi^O$)

\[
\text{glob}_\pi^O(B \Box, \langle \mathcal{C}, \mathcal{O}, \mathcal{I}, \mathcal{R} \rangle) \overset{def}{=} \{ \pi(B \Box, B, j) \mapsto \hat{\pi}(\mathcal{O}(j)) \mid j \in \text{dom} \mathcal{O} \} \\
\cup \bigcup_{j \in \text{dom} \mathcal{I}} \text{glob}_\pi^O(B \Box[\langle \mathcal{C}, \mathcal{O}, \mathcal{I} \rangle \mapsto \big], \mathcal{R} \rangle, \mathcal{I}(j))
\]

glob^N_\pi$ derives the object–box relation from the box-local stores.

Definition 11 ($\text{glob}_\pi^N$)

\[
\text{glob}_\pi^N(B \Box, \langle \mathcal{C}, \mathcal{O}, \mathcal{I}, \mathcal{R} \rangle) \overset{def}{=} \{ \pi(bp(B \Box, j)) \mapsto \pi(bp(B \Box, \Box)) \mid j \in \text{dom} \mathcal{O} \} \\
\cup \bigcup_{k \in \text{dom} \mathcal{I}} \text{glob}_\pi^N(B \Box[\langle \mathcal{C}, \mathcal{O}, \mathcal{I} \rangle \mapsto \big], \mathcal{R} \rangle, \mathcal{I}(k))
\]

The following definition derives the explicit box tree relation.

Definition 12 ($\text{glob}_\pi^P$)

\[
\text{glob}_\pi^P(\Box, \langle \mathcal{C}, \mathcal{O}, \mathcal{I}, \mathcal{R} \rangle) \overset{def}{=} \bigcup_{k \in \text{dom} \mathcal{I}} \{ \pi(k, \Box) \mapsto \text{globox} \} \cup \text{glob}_\pi^P(\langle \mathcal{C}, \mathcal{O}, \mathcal{I} \mapsto \big], \mathcal{R} \rangle, \mathcal{I}(k))
\]

\[
\text{glob}^P_\pi(B \Box[\langle \mathcal{C}, \mathcal{O}, \mathcal{I} \rangle \mapsto \big], \mathcal{R} \rangle, \mathcal{I}(k)) \overset{def}{=} \{ \pi(bp(B \Box[\langle \mathcal{C}, \mathcal{O}, \mathcal{I} \rangle \mapsto \big], \Box)) \mapsto \pi(bp(B \Box, \Box)) \} \\
\cup \bigcup_{k' \in \text{dom} \mathcal{I}} \text{glob}^P_\pi(B \Box[\langle \mathcal{C}, \mathcal{O}, \mathcal{I} \rangle \mapsto \big], \mathcal{R} \rangle, \mathcal{I}(k'))
\]

The next definition maps boxes to their box signatures.

Definition 13 ($\text{glob}_\pi^R$)

\[
\text{glob}_\pi^R(\Box, \langle \mathcal{C}, \mathcal{O}, \mathcal{I}, \mathcal{R} \rangle) \overset{def}{=} \bigcup_{k \in \text{dom} \mathcal{I}} \{ \pi(k, \Box) \mapsto \mathcal{R}(\text{env}) \} \cup \text{glob}_\pi^R(\langle \mathcal{C}, \mathcal{O}, \mathcal{I} \mapsto \big], \mathcal{R} \rangle, \mathcal{I}(k))
\]
\[ \text{glob}_\pi^R(B\square[(C, O, I[k \mapsto \square], R)], \langle C', O', I', R' \rangle) \triangleq \\
\{ \pi(\text{bp}(B\square[(C, O, I[k \mapsto \square])), \Xi) \mapsto R(\text{env}) \}_{\pi} \} \\
\cup \bigcup_{k' \in \text{dom} I'} \text{glob}^R_{\pi}(B\square[(C, O, I[k \mapsto (C', O', I'[k' \mapsto \square], R')], R)], I(k')) \]

The last definition in this sequence integrates the above functions:

**Definition 14 (glob)** The function \( \text{glob}_\pi \) translates a global state from the message semantics domain to the standard semantics domain. Let \( \pi \in \Pi \).

\[ \text{glob}_\pi(B\square, B) \triangleq ((C_\pi, O_\pi, N_\pi, P_\pi, R_\pi), b_\pi) \]

with

\[
\begin{align*}
C_\pi &= \text{glob}^C_\pi(B\square, B) \\
O_\pi &= \text{glob}^O_\pi(B\square[B]) \\
P_\pi &= \text{glob}^P_\pi(B\square[B]) \\
R_\pi &= \text{glob}^R_\pi(B\square, B) \\
b_\pi &= \pi(\text{bp}(B\square), \Xi)
\end{align*}
\]

The function \( \text{glob}_\pi \) is the key to formulate the bisimulation. The next steps provide a helper function to retrieve used intra-box references and the development of the properties that identifier translator functions have to satisfy.

**Definition 15 (brr)** The function \( \text{brr} \) returns the set of all intra-box references relative to a given box, i.e. all objects owned by this box and all objects that have passed the boundary of this box so far.

\[ \text{brr}((C, O, I, R)) \triangleq (\text{dom} O) \cup \bigcup_{Q \in \text{cod} R} (\text{dom} Q) \]

**Definition 16 (\( \pi \)-inv)** Let \( B\square, B \) be a global state in the message semantics. The predicate \( \pi\text{-inv}(B\square, B, \pi) \) holds iff all of the following conditions are satisfied where \( ((C_\pi, O_\pi, N_\pi, P_\pi, R_\pi), b_\pi) = \text{glob}_\pi(B\square, B) \):

\[ \pi(\bullet, \Xi) = \text{globbox} \]

\[ \forall (\text{own}, j_{\text{hm}}, T), (\text{own}, j'_{\text{hm}}, T') \in \text{brr}(B) . \]

\[ \pi(\text{bp}(B\square), j_{\text{hm}}) = \pi(\text{bp}(B\square), j'_{\text{hm}}) \Rightarrow j_{\text{hm}} = j'_{\text{hm}} \cdot \] (16.1)

\[ \forall o_b, o'_b \in \text{brr}(B) . \hat{\pi}(B\square, B, o_b) = \hat{\pi}(B\square, B, o'_b) \Rightarrow o_b = o'_b \]

\[ \forall (b_{\text{hm}}, j_{\text{hm}}, T') \in \text{brr}(B) . \exists j'_r, T, b'_r \cdot j'_r = \pi(\text{bp}(B\square), j_{\text{hm}}) \wedge b'_r = N_r(j'_r) \]

\[ \Rightarrow T' = \sigma^+(\text{crossedSigs}_{P_\pi, R_\pi}(b'_r, b_\pi), T) \] (16.3)

\[ \forall (\text{own}, j_{\text{hm}}, T) \in \text{brr}(B) . \hat{\pi}(B\square, B, O_{\text{hm}}(j_{\text{hm}})) = O_\pi(\pi(\text{bp}(B\square), j_{\text{hm}})) \] (16.4)

\[ \forall j_r \in \text{dom} O_\pi . \exists j_{\text{hm}} \in \text{dom} O_{\text{hm}} . \pi(\text{bp}(B\square), j_{\text{hm}}) = j_r \] (16.5)

\[ \forall j_r \in \text{dom} O_\pi . \exists j_{\text{hm}} \in \text{dom} O_{\text{hm}} . \pi(\text{bp}(B\square), j_{\text{hm}}) = j_r \] (16.6)
Equation (16.1) expresses the base case for identifier translation; in particular, $\pi(\text{bp}(\Box), \Box) = \text{globox}$. Equation (16.2) states that there must not be two different local objects which are mapped to the same object identifier. Equation (16.3) is a generalization of (16.2) to all intra-box references known to this box. Equation (16.4) states that $\pi$ may only translate references which obey the type restrictions given by box signatures. Equation (16.5) essentially expresses that $\hat{\pi}_{B, B}$ is a homeomorphism from the standard semantics objects to message semantics objects. The last equation, (16.6), is surjectivity of $\pi_{\text{bp}(B)}$.

Definition 17 ($\pi$-inv-all) We require this invariant for a $\pi$ to be compatible with the global state of a message semantics configuration.

$$\pi\text{-inv-all}(B, B, \pi) \iff \forall B' \exists B''. (B' = B''[B] \Rightarrow \pi\text{-inv}(B', B'', \pi))$$

Definition 18 ($=\pi$) The equality on expressions modulo identifier translation by $\pi$ is denoted by $=\pi$. Let $\pi \in \Pi$.

$$e = \pi_{B, B} e_{\text{rm}} \iff (e = [\hat{\pi}_{B, B}]e_{\text{rm}})$$

Finally, we define a relation $\approx$ on configurations of the standard and message semantics (and also of the summarized message semantics).

Definition 19 ($\approx$)

\begin{align*}
(S, b, e_r) \approx (B, B, e_{\text{rm}}) & \iff \exists \pi \in \Pi. \pi\text{-inv-all}(B, B, \pi) \\
& \Rightarrow (e_r = \pi_{\text{bp}(B), B} e_{\text{rm}} \land (S, b = \text{glob}_{\pi_{\text{bp}(B)}}(B, B)))
\end{align*}

6.3 The Strong Bisimulation Theorem

Now, we have all definitions necessary to formulate and sketch that $\approx$ is a strong bisimulation relation between the standard semantics and the summarized message semantics.

Theorem 1 Consider the labeled state transition systems $(\text{CONF}_r, \Lambda^*, \rightarrow_r)$ and $(\text{CONF}_r, \Lambda^*, \rightarrow_{\text{rm}})$. There exists a strong bisimulation relation between them.

Proof Sketch

Initial configurations. The definition of strong bisimulation (Def. 4) does not ensure there are configurations related by $\approx$. Therefore, we show first
that initial configurations in the standard semantics have a related initial configuration in the message semantics.

Let \( \pi_0(\bullet, \text{own}.j:T) \overset{\text{def}}{=} j:T \) and \( \pi_0(\bullet, \boxtimes) \overset{\text{def}}{=} \text{glob} \). \( \pi_0 \) relates all initial configuration of both semantics domains. As every initial configuration includes only objects in the \text{glob} \), and \( \text{bp}(\square) = \bullet, \pi_0 \) is sufficient to map every object from the message semantics domain to the standard semantics. The construction of \( O_0^R \) and \( N_0^R \) from \( O_{0RM}^R \) is straightforward.

\[ \approx \text{ is a strong bisimulation relation. } \]

To show that \( \approx \) is strong bisimulation relation for the standard and summarized message semantics, we have to provide an identifier translator function \( \pi_{\text{post}} \) for the postconfiguration. The construction uses the of the identifier translator \( \pi_{\text{pre}} \) of the preconfiguration.

We can reuse \( \pi_{\text{pre}} \) directly as long as there is neither an object nor a box created. Only when a \( \text{new-obj} \) or a \( \text{new-box} \) rule was applied we have to extend \( \pi_{\text{pre}} \). This is done by the function \( \text{suc}c \):

**Definition 20 (Successor of identifier translator functions)** Let \( \lambda \in \Lambda^* \), \( s, s' \in C_R, t, t' \in C_{RM} \) with \( (s \approx_{\pi} t), (s \overset{\lambda}{\overset{\text{R}}{\rightarrow}} s'), (t \overset{\lambda}{\overset{\text{RM}}{\rightarrow}} t') \), and let \( t = B_{\square}, (C_{RM}, O_{RM}, I_{RM}, R_{RM}), e_{RM} \) and \( s = (C_R, O_R, N_R, P_R, R_R), b, e_{R} \) then:

\[
\text{suc}c(\pi, s, t) \overset{\text{def}}{=} \begin{cases}
\pi((k, \text{next}(\text{dom} O_{RM}))) \mapsto \text{next}(\text{dom} O_{R}) & \text{if } e_{RM} = e_{R}[\text{new } C] \\
\pi((k', \boxtimes) \mapsto \text{next}(\text{dom} P_{R})) & \text{if } e_{RM} = e_{R}[\text{new } B] \\
\pi & \text{else}
\end{cases}
\]

with \( k = \text{bp}(B_{\square}) \) and \( k' = \text{bp}(B_{\square}[\cap C_{RM}, O_{RM}, I_{RM}[\text{next}(\text{dom} I_{RM}) \mapsto \square], R_{RM}]) \)

To prove Theorem 1, we have to show (4.1) and (4.2) for all labels, i.e. all rules. Let \( c_r, c_r' \in CONF_R \) and \( c_{RM}, c_{RM}' \in CONF_{RM} \) with \( c_r \approx c_{RM} \) and let \( \pi_{\text{pre}} \) be the existing identifier translator. We show that \( c_r' \approx c_{RM}' \) where the needed identifier translator is obtained by \( \text{suc}c \) from \( \pi_{\text{pre}} \).

In the following, we will consider the most interesting rules.

- **NEW-OBJ** For object creation, the following two equations express the property that has to derived from the definitions:

\[
\text{glob}_n^O(B_{\square}, (C_{RM}, O_{RM}, I_{RM}, R_{RM}))[j_{RM} \mapsto \text{null}, I_{RM}, R_{RM})] = \text{glob}_n^O(B_{\square}, (C_{RM}, O_{RM}, I_{RM}, R_{RM}))[\pi((\text{bp}(B_{\square}), j_{RM}) \mapsto \text{null})]
\]
\[
glob^N_\pi(B □, \langle C_{\text{RM}}, O_{\text{RM}}, I_{\text{RM}}, R_{\text{RM}} \rangle) = \glob^N_\pi(B □, \langle C_{\text{RM}}, O_{\text{RM}}, I_{\text{RM}}, R_{\text{RM}} \rangle)[\pi(bp(B □), j_{\text{RM}}) \mapsto \pi(bp(B □), \Box)]
\]

Note that \(\pi\text{-inv-all}\) is reduced to \(\pi\text{-inv}\) for the current box, as all other boxes are unchanged.

- **NEW-BOX** In order to prove \(\pi\text{-inv-all}\) after a box creation, we have to show \(\pi\text{-inv}\) for the surrounding and for the newly created box. Furthermore, the equivalence of the global states has to be shown, i.e.:

\[
glob_\pi(B □, \langle C_{\text{RM}}, O_{\text{RM}}, I_{\text{RM}}, R_{\text{RM}} \rangle) = \glob_\pi(B □, \langle C_{\text{RM}}, O_{\text{RM}}, I_{\text{RM}}, R_{\text{RM}} \rangle)[\pi(bp(B □), j_{\text{RM}}) \mapsto \pi(bp(B □), \Box)]
\]

\[
S \mapsto S_\pi^{\pi'}(B □, \langle C_{\text{RM}}, O_{\text{RM}}, I_{\text{RM}}, R_{\text{RM}} \rangle) \mapsto S \mapsto S_\pi^{\pi'}(B □, \langle C_{\text{RM}}, O_{\text{RM}}, I_{\text{RM}}, R_{\text{RM}} \rangle)[\pi(bp(B □), j_{\text{RM}}) \mapsto \pi(bp(B □), \Box)]
\]

\[
I = \text{intf}(B)
\]

\[
(o_d, Q) = \text{export}(\text{env}, \text{own.0}:C, (\text{bsig}(I), \emptyset))
\]

\[
(k_{\text{RM}}, j_{\text{RM}}:T, Q') = \text{import}(k, o_d, (\text{bsig}(I), \emptyset))
\]

then

\[
glob_\pi(B □, \langle C_{\text{RM}}, O_{\text{RM}}, I_{\text{RM}}', R_{\text{RM}}' \rangle) = \langle C_R, O_R', N_R', P_R', R_R', b_R \rangle
\]

\[
O_R' = O_R[j_R \mapsto \text{null}]
\]

\[
N_R' = N_R[j_R \mapsto k_R]
\]

\[
P_R' = P_R[k_R \mapsto b_R]
\]

\[
R_R' = R_R[k_R \mapsto \text{bsig}(I)]
\]

\[
j_R = \pi'(bp(B □) \cdot k_{\text{RM}}, j_{\text{RM}})
\]

\[
k_R = \pi(bp(B □) \cdot k_{\text{RM}}, \Box)
\]

- **CALL** With the call rule we use natural induction over the number of box boundaries between the caller and the callee. The induction argument "un-rolls" the definition of CALL in the summarized message semantics (Definition 5). In both cases (4.1) and (4.2) the proof goal is directly applicable as induction hypothesis.
The globalized states of \( c_{\text{RM}} \), \( c_{\text{RM},1} \), and \( c_{\text{RM},2i} \) are all equal. Only the last step (EXEC) mutates the globalized state. By globalized state we mean the result of \( \text{glob applied to } B, B \) of a configuration \( c_{\text{RM}} = B, B, e_{\text{RM}} \). The configurations are mutated, because the reference mappings may become updated and the \( e_{\text{RM}} \) of \( c_{\text{RM}} \) is replaced by a message after the HANDLE step. Furthermore, there are multiple forwarding and receiving rules. They all have to be handled by case distinction over the direction in which to send the message, i.e. an inner box or the environment.

- **RETURN** This case is analog to the call rule.
- **CAST-OBJ** The cast rule of the standard semantics for non-null references is the only one, which is not defined as usual. It constrains casts in a way that is equivalent to the type restriction of references through box signatures in the message semantics. This is expressed by invariant \((16.4)\).

The proofs for the other rules follow a similar pattern and are not discussed here. □

The proof sketch of the equivalence between the standard semantics and the message semantics allows in particular to systematically transfer techniques that are developed based on the standard semantics to the message semantics. Furthermore, it relates the full abstractness result that we developed in the next sections for the message semantics back to the classical semantics.

7 Behavioral Semantics for Boxes

Starting from the small-step message semantics described in Sec. 2, this section defines a behavioral semantics that is based on sequences of incoming messages. The behavioral semantics allows to compare boxes with respect to their behavior at the boundary. It abstracts:

1. from the concrete states of boxes: In particular, it abstracts from the
types, number, and states of objects in a box and only considers replies to incoming messages. Normalization of references in the message semantics guarantees that the semantics does not depend on which objects are allocated in the box.

(2) from the concrete states of box environments: Only the incoming messages sent by the environment to the box are considered. Normalization of references in the message semantics guarantees that the semantics does not depend on which objects are allocated in the environment.

(3) from box instances: As only the normalized references crossing the boundary are relevant for the behavior, different box instances have the same semantics.

The development is done in two steps. First, we go from the small-step message semantics to a message semantics that considers all reductions between a message sent to a box and the corresponding reply as one big-step. Then, we do the abstractions mentioned above.

7.1 Mix-step Message Semantics

The small-step message semantics has two rules that move the execution focus of the current box to one of its inner boxes (RM-FWD-CALL-INNER, RM-FWD-RTRN-INNER) and two rules that move the focus to the directly surrounding box (RM-RCV-CALL-INNER, RM-RCV-RTRN-INNER; cf. Figure 13). In all other rules, the box environment remains the same. The central idea underlying the development of the behavioral semantics is to translate the small-step message semantics of Sec. 5 into a mix-step message semantics. The mix-step semantics uses a small-step judgment to express box local executions and a big-step judgment to relate messages sent to inner boxes to the replies. Thus, a rule expressing the return from the box to its environment is no longer necessary. More importantly, we can eliminate the box environment from the box-local judgment:

\[ \mathcal{B}, r \rightarrow_{\text{rmm}} \mathcal{B}', r' \]

It reads “in box state \( \mathcal{B} \), \( r \) can be reduced in one step to \( r' \) with new box state \( \mathcal{B}' \)”. As the big-step judgment is used to express all small steps between the forwarding of a message to an inner box \( b \) and the receipt of the next reply from \( b \), we can simply use the transitive closure of \( \rightarrow_{\text{rmm}} \) (denoted by \( \rightarrow_{\text{rmm}}^{+} \)) to express such steps in the semantics.

The mix-step message semantics is obtained from the semantics in Fig. 11, 12 and 13 as follows:

(1) Remove the rules (RM-FWD-CALL-INNER), (RM-FWD-RTRN-INNER), (RM-RCV-CALL-INNER), and (RM-RCV-RTRN-INNER) that move the focus be-
Each rule in Fig. 14 is constructed from two of the removed rules. For exam-
ple, the box environment is left out in the configurations.

RMM-FWD-CALL-INNER-CALL
n_b = (k, j : T_m(\pi_b))
(n_d, Q') = forward(k, n_b, R(k))
I(k), n_d \rightarrow_{\text{RMM}}^+ B, n'_d
n'_d = 1(out, j : T_m(\pi_d))
(n'_b, Q'') = receive(k, n'_d, Q')

\langle C, O, I, R \rangle, n_b \rightarrow_{\text{RMM}} \langle k \cdot C, O, I[k \mapsto B], R[k \mapsto Q''] \rangle, n'_b

RMM-FWD-CALL-INNER-RTRN
n_b = (k, j : T_m(\pi_b))
I(k), n_d \rightarrow_{\text{RMM}}^+ B, n'_d
n'_d = 1(out, j : T_m(\pi_d))
(n'_b, Q'') = receive(k, n'_d, Q')

\langle C, O, I, R \rangle, n_b \rightarrow_{\text{RMM}} \langle k \cdot C, O, I[k \mapsto B], R[k \mapsto Q''] \rangle, n'_b

RMM-FWD-RTRN-INNER-CALL
n_b = \leftarrow v_b
(n_d, Q') = forward(k, n_b, R(k))
I(k), n_d \rightarrow_{\text{RMM}}^+ B, n'_d
n'_d = 1(out, j : T_m(\pi_d))
(n'_b, Q'') = receive(k, n'_d, Q')

\langle C, O, I, R \rangle, n_b \rightarrow_{\text{RMM}} \langle k \cdot C, O, I[k \mapsto B], R[k \mapsto Q''] \rangle, n'_b

RMM-FWD-RTRN-INNER-RTRN
n_b = \leftarrow v_b
(n_d, Q') = forward(k, n_b, R(k))
I(k), n_d \rightarrow_{\text{RMM}}^+ B, n'_d
n'_d = 1(out, j : T_m(\pi_d))
(n'_b, Q'') = receive(k, n'_d, Q')

\langle C, O, I, R \rangle, n_b \rightarrow_{\text{RMM}} \langle k \cdot C, O, I[k \mapsto B], R[k \mapsto Q''] \rangle, n'_b

Fig. 14. Rules of mix-step semantics that handle messages sent to inner boxes
between boxes.

(2) Replace the old judgment in the remaining rules by the box local judg-
ment, i.e., eliminate the box environment B_\Box from all rules and re-
name the rules by changing the prefix “RMM” to “RMM” (rule of mix-step
message semantics). We call these the adopted rules.

(3) Add the rules of Fig. 14.

Each rule in Fig. 14 is constructed from two of the removed rules. For ex-
ample, rule (RMM-FWD-CALL-INNER-CALL) is constructed from (RM-FWD-CALL-
INNER) and (RM-RCV-CALL-INNER) as follows: Take the premises of both
rules, add the big-step judgment which corresponds to the postconfiguration of
(RM-FWD-CALL-INNER) and the preconfiguration of (RM-RCV-CALL-INNER),
and construct the conclusion of the new rule from the preconfiguration of (RM-
FWD-CALL-INNER) and the postconfiguration of (RM-RCV-CALL-INNER). Of
course, the box environment is left out in the configurations.

Initial configurations are the same as for the small-step message semantics,
dropping the empty environment. A program execution according to the mix-
step semantics is a tree of traces:

**Definition 21** Let P be a program and c' be an initial configuration for P.

The derivation deriv_{RM}(P, c') of P with c' is a trace according to the small-
step message semantics. Each step consists of a rule name, a preconfiguration and a postconfiguration such that the preconfiguration of the first step is $c^i$ and of every other step is the postconfiguration of the previous step.

The derivation $\text{deriv}_{RMM}(P, c^i)$ of $P$ with $c^i$ in the mix-step semantics is a tree-structured trace of the following form: A step is either a small step according the modified rules of the small-step semantics or a box derivation. A box derivation can be terminating or nonterminating. It is terminating, if one of the rules in Fig. 14 is applicable. A terminating box derivation consists of a subderivation for $I(k), n_d \rightarrow^{+}_{\text{RMM}} B, n'_d$ followed by the step according the conclusion of the rule. It is nonterminating, if the preconfiguration satisfies the conditions for one of the rules in Fig. 14, but the trace starting in $I(k), n_d$ does not lead to a configuration $B, n'_d$ with an outgoing message. In that case, the trace starting in $I(k), n_d$ is considered as the subderivation.

A derivation is called complete if no further rule is applicable. Otherwise, it is called partial.

A derivation of the mix-step semantics is called pruned if some of the subderivations are eliminated. The pruned derivations in which all subderivations are eliminated is called the global trace of $P$ with $c^i$.

In the mix-step semantics, the box environment is captured by the tree-structure of the derivation and its subderivations. Every subderivation corresponds to a box. The depth of a subderivation in the derivation tree equals the depth of the corresponding box in the box tree. More precisely, whenever a box derivation is performed, the associated subderivation describes the execution trace for an inner box with local identifier $k$.

The use of the big-step judgment does not lead to nondeterminism, because within the rules of Fig. 14, the reduction term $n'_d$ denotes an outgoing message and the mix-step semantics no longer has a rule to handle such messages. Thus, a trace starting in $I(k), n_d$ either stops at $B, n'_d$ such that there is no configuration with outgoing message before $B, n'_d$ in the trace, or it does not terminate.

**Theorem 2** The small-step and mix-step message semantics are equivalent in the sense that there is a translation from the derivations according to one semantics to the derivations according to the other with identical local box states, and vice versa.

**Proof.** As described above, each rule in Fig. 14 is constructed from two of the removed rules. The translation exploits this correspondence between the rule systems. We present the translation from small-step to mix-step derivations in some detail. As the translation of the other direction is very similar,
we leave out the technical details.

Translation from small-step to mix-step derivation: We stepwise construct the derivation for the mix-step semantics from the small-step trace. On a stack $cs$, we record the configurations at box entry. Let $c_{RM}$ denote the current configuration of the small-step derivation and $c_{RM}M$ denote the current configuration of the mix-step derivation. The function $rbe$ removes the box environment from a small-step configuration. Initially, $c_{RM} = c_{RM}^i, c_{RM}M = rbe(c_{RM}^i), cs = \bullet$.

For the translation of a step $c_{RM} \rightarrow_{RM} c_{RM}'$ with rule label $l$, we consider the following cases:

1. $l$ denotes a rule that is adopted by the mix-step semantics. Then, use the adopted rule: $c_{RM} \rightarrow_{RM} rbe(c_{RM}')$. The stack is not modified.
2. $l = (\text{RM-FWD-CALL-INNER})$ with an instance of $B_{\square}, \langle C, O, I, R \rangle, n_b \rightarrow_{RM} B_{\square}[\langle C, O, I[k \leftarrow \square], R[k \leftarrow Q] \rangle], I(k), n_d$:
   Push $\langle C, O, I, R \rangle, n_b$ on the stack. Start a subderivation with current configuration $I(k), n_d$. Note that the pushed configuration satisfies the conditions for the preconfiguration in rule $(\text{RMM-FWD-CALL-INNER-CALL})$ and $(\text{RMM-FWD-CALL-INNER-RTRN})$.
3. $l = (\text{RM-FWD-RTRN-INNER})$ with an instance of $B_{\square}, \langle k \cdot C, O, I, R \rangle, n_b \rightarrow_{RM} B_{\square}[\langle C, O, I[k \leftarrow \square], R[k \leftarrow Q] \rangle], I(k), n_d$:
   Push $\langle k \cdot C, O, I, R \rangle, n_b$ on the stack. Start a subderivation with current configuration $I(k), n_d$. Note that the pushed configuration satisfies the conditions for the preconfiguration in rule $(\text{RMM-FWD-RTRN-INNER-CALL})$ and $(\text{RMM-FWD-RTRN-INNER-RTRN})$.
4. $l = (\text{RM-RCV-CALL-INNER})$ with an instance of $B_{\square}[\langle C, O, I[k \leftarrow \square], R \rangle], B, n'_d \rightarrow_{RM} B_{\square}, \langle k \cdot C, O, I[k \leftarrow B], R[k \leftarrow Q] \rangle, n'_d$:
   Finish the subderivation. Let $c_{RM}M = \text{pop}(cs)$.
   If the message in $c_{RM}M$ is a call configuration $B_{\square}, \langle C, O, I, R \rangle, n_b$, it satisfies the conditions in the premise; let $(n_d, Q) = \text{forward}(k, n_b, R(k))$ and $(n'_d, Q') = \text{receive}(k, n'_d, Q)$. Then:
   $c_{RM} \rightarrow_{RMM} (k \cdot C, O, I[k \leftarrow B], R[k \leftarrow Q'])n'_b$ is a transition with $(\text{RMM-FWD-CALL-INNER-CALL})$. We use here the fact that a trace between an application of $(\text{RM-FWD-CALL-INNER})$ and $(\text{RM-RCV-CALL-INNER})$ never modifies $R$ of the box environment.
   If the message in $c_{RM}M$ is a return configuration $B_{\square}, \langle C, O, I, R \rangle, n_b$, it satisfies the conditions in the premise; let $(n_d, Q) = \text{forward}(k, n_b, R(k))$ and $(n'_d, Q') = \text{receive}(k, n'_d, Q)$. Then:
   $c_{RM} \rightarrow_{RMM} (k \cdot C, O, I[k \leftarrow B], R[k \leftarrow Q'])n'_b$ is a transition with $(\text{RMM-FWD-RTRN-INNER-CALL})$. Again, we use here the fact that a trace between an application of $(\text{RM-FWD-RTRN-INNER})$ and $(\text{RM-RCV-CALL-INNER})$ never modifies $R$ of the box environment.
5. $l = (\text{RM-RCV-RTRN-INNER})$: This case is analogous to Case 4.
Translation from mix-step to small-step derivation: For the translation of mix-step derivations to small-step derivations, we record the current box environment in $cbe$. This allows us to restore the correct box environment after exit to a box. For brevity, we omit the details of this proof part. □

Theorem 2 together with Theorem 1 affirms the fact that our language even if based on the mix-step message semantics is equivalent to a standard object-oriented language extended by some additional box-related checks.

7.2 Behavioral Semantics

As the judgment $B, n_d \rightarrow_{\text{nm}} B', n'_d$ does not refer to the current box environment, the behavior of a box is independent of the environment. Furthermore the big-step judgment gives us a relation expressing the boundary behavior. In this section, we define a more abstract boundary relation for a box that does not refer to the concrete box states $B$ and $B'$. More precisely, let $BI = (B, P_B)$ be a box implementation with signature $B_{\text{sig}}$. The denotation $\llbracket BI \rrbracket$ of $BI$ is a partial function taking an incoming message sequence $\pi_d$ and an incoming message $n_d$ as arguments and yielding a message as reply. The message sequence $\pi_d$ abstractly represents the prestate $B$ of the box when $n_d$ is received.

The technical challenge to achieve full abstractness is to restrict the domain of $\llbracket BI \rrbracket$ to message sequences that occur in program contexts. The beautiful and practically important property of our approach is that the definition of $\llbracket BI \rrbracket$ only uses program declarations defined in $BI$. In particular, we need not quantify over the set of possible program contexts. The partial function $\llbracket BI \rrbracket$ is defined in two steps. First, we provide a constructive specification of possible sequences of incoming and outgoing messages. Then, we use the mix-step message semantics to define the replies.

Definition 22 A call message $\langle d.j:T \rangle.m(\overline{v})$ is type correct w.r.t. a box implementation $BI$, iff

- $T$ and the values in $\overline{v}$ are null or have a type in $B_{\text{sig}}$,
- type $T$ has a method $m$ (either declared or inherited), and
- $\overline{v}$ is a type correct parameter list for $m$.

A return message is type correct if its value is null or has a type in $B_{\text{sig}}$.

The set of type correct incoming and outgoing messages of $BI$ are denoted by $INM_{\text{mi}}$ and $OUTM_{\text{mi}}$, respectively; their union, the set of boundary messages, is $BDM_{\text{mi}} : = INM_{\text{mi}} \cup OUTM_{\text{mi}}$. The finite sequences over these sets are denoted by an overline, e.g. $\overline{BDM_{\text{mi}}}$. The function $\text{kind} : BDM_{\text{mi}} \rightarrow \{ic, ric, oc, roc\}$
retrieves the information whether a boundary message is an incoming call, a return from an incoming call, an outgoing call, or a return from an outgoing call. Furthermore, we need the following notions:

**Definition 23** A boundary message sequence $n_d$ is called stack-correct iff the sequence $\text{kind}(n_d)$ is a prefix of a word generated by the following context free grammar $\Gamma$:

$$\Gamma = (\{\text{EXECIN}, \text{EXECOUT}\}, \{\text{ic, ric, oc, roc}\}, PROD, \text{EXECOUT})$$

with $PROD :$

- $\text{EXECOUT} ::= \text{ic EXECIN ric EXECOUT} \mid \epsilon$
- $\text{EXECIN} ::= \text{oc EXECOUT roc EXECIN} \mid \epsilon$

The grammar associates each return message in a stack-correct message sequence with a unique matching call message. It makes sure that execution starts with an incoming call, that there is no return without a matching call, and that the directions of the messages are compatible.

A boundary message sequence $n_d$ is called call-correct iff it is stack-correct and the values in all return messages are type-correct with respect to the matching call message.

The objects used in a boundary message sequence have to certify constraints with respect to the exposed and received objects:

**Definition 24** The set of exposed objects $\text{exp}(n_d)$ of a boundary message sequence $n_d$ for box $BI$ exactly contains the initial object $in_0:T$ of $BI$ and the objects occurring in $n_d$ with direction prefix $in$.

The set of received objects $\text{rec}(n_d)$ of a boundary message sequence $n_d$ for box $BI$ exactly contains the objects occurring in $n_d$ with direction prefix $out$.

An object is called a passed object of $n_d$ iff it is either exposed or received. A boundary message sequence $n_d$ is called pass-correct if it is empty or if its longest proper prefix $n'_d$ is pass-correct and the following holds where $n_d = n'_d \cdot n_d$:

- for incoming messages $n_d$, $n_d$ only has objects as parameters that have been passed by $n'_d$ or are “newly” received objects that do not appear in $n'_d$; in the latter case, they have to be next objects according to the normalization technique explained in Subsec. [5.1]

- for outgoing messages $n_d$, $n_d$ only has objects as parameters that have been passed by $n'_d$ or that are “newly” exposed objects that do not appear in $n'_d$; in the latter case, they have to be next objects according to the normalization technique.
The boundary message sequences in $BDM_{bi}$ that are call- and pass-correct are called boundary behaviors. The set of boundary behaviors of $BI$ is denoted by $BDB_{bi}$. Note that the check whether a boundary message sequence is in $BDB_{bi}$ is purely syntactical and easy to perform. The notion of boundary behaviors is used to define which incoming messages are considered to be admissible in a box state.

We use the big-step judgment of the mix-step message semantics $B, r \rightarrow^{+}_{\text{MM}} B', r'$ to define the behavioral semantics, i.e., the denotation of boxes. A denotation is a partial function in $INM_{bi} \times INM_{bi} \rightarrow OUTM_{bi}$ where the first argument represents the abstract state of the box and the second argument is the incoming message to be handled. For the definition, we use two recursively defined partial functions. Function $\text{expand}_{bi}$ maps an incoming message sequence to the corresponding boundary behavior, i.e., to a message sequence that also contains the replies. Function $\text{concr}_{bi}$ maps an incoming message sequence to the state of the box after execution of the messages. Conceptually, $\text{concr}_{bi}$ is a concretization function that maps the abstract representations of box states for boxes with signature $B_{bi}$ to the concrete states according to box implementation $BI$. Throughout the paper, we assume that the cons operation is strict and that the judgments yield false for undefined arguments:

$$
\text{expand}_{bi}(\bullet) \overset{\text{def}}{=} \bullet
$$

$$
\text{expand}_{bi}(\pi_d \cdot n_d) \overset{\text{def}}{=} \text{expand}_{bi}(\pi_d) \cdot n_d \cdot n_d' \text{ if } \text{expand}_{bi}(\pi_d) \cdot n_d \in BDB_{bi} \text{ and } \exists B' : \text{concr}_{bi}(\pi_d), n_d \rightarrow^{+}_{\text{MM}} B', n_d'
$$

$$
\text{expand}_{bi}(\pi_d \cdot n_d) \overset{\text{def}}{=} \perp \text{ otherwise}
$$

$$
\text{concr}_{bi}(\bullet) \overset{\text{def}}{=} B \text{ where } B = (\bullet, \{0 \mapsto \text{null}\}, \varnothing, \{\text{env} \mapsto Q\})
$$

$$
\text{and } (\cdot, Q) = \text{export(}\text{env, own.}0:C, (\text{bsig}(I), \varnothing)\text{)}
$$

$$
\text{and } C \text{ is the owner class of } BI,
$$

$$
\text{and } I \text{ is the type of } BI
$$

$$
\text{concr}_{bi}(\pi_d \cdot n_d) \overset{\text{def}}{=} B' \text{ if } \text{expand}_{bi}(\pi_d) \cdot n_d \in BDB_{bi} \text{ and } \exists n_d' : \text{concr}_{bi}(\pi_d), n_d \rightarrow^{+}_{\text{MM}} B', n_d'
$$

$$
\text{concr}_{bi}(\pi_d \cdot n_d) \overset{\text{def}}{=} \perp \text{ otherwise}
$$

$\text{expand}_{bi}$ and $\text{concr}_{bi}$ are well-defined, because the big-step judgment is deterministic (see above). $\text{expand}_{bi}(\pi_d)$ is defined if $\text{concr}_{bi}(\pi_d)$ is defined. Furthermore, if $\text{expand}_{bi}(\pi_d)$ is defined, then it is defined for all prefixes of $\pi_d$. Based on $\text{expand}_{bi}$, we define the denotation of a box implementation as a curried
partial function:

\[ [BI] : \text{INM}_{\text{bi}} \rightarrow (\text{INM}_{\text{bi}} \rightarrow \text{OUTM}_{\text{bi}}) \]

\[ [BI] \overline{n_d} n_d \overset{\text{def}}{=} n'_d \quad \text{if} \quad \text{expand}_{\text{bi}}(\overline{n_d} \cdot n_d) = \text{expand}_{\text{bi}}(\overline{n_d}) \cdot n_d \cdot n'_d \neq \bot \]

\[ [BI] \overline{n_d} n_d \overset{\text{def}}{=} \bot \quad \text{otherwise} \]

Abstracting over \( BI \), we obtain a semantics for box implementations. This semantics is independent of the program context in which a box is used. To abstract in the mix-step semantics from box implementations and to compare traces of programs using different box implementations, we replace the big-step judgment by the corresponding box denotation. The inner box mapping \( I \) captures for an inner box \( k \) the box implementation and the incoming message history. The denotation-based messages semantics is obtained from the mix-step semantics by the following translation steps:

1. Change the definition of the inner box mapping \( I \) given in Fig. 7 to 
   \[ I \ ::= \ k \mapsto (BI, INM_{bi}) \] where \( BI \) denotes box implementations. By \( I^1(k) \) and \( I^2(k) \), we refer to the first and second component of \( I(k) \) respectively.
2. Replace in rule (RMM-NEW-BOX) the equation for \( I' \) by 
   \[ I' = I[k \mapsto (\text{boxImpl}(B), \bullet)] \] where \( \text{boxImpl}(B) \) yields the box implementation for \( B \) in the current program.
3. Modify the rules in Fig. 14 as follows:
   - Replace \( I(k), n_d \overset{\text{rmm}}{\rightarrow} B, n'_d \) in the premises by \( I^1(k), I^2(k) n_d = n'_d \)
   - Replace \( I[k \mapsto B] \) in the conclusions by \( I[k \mapsto (I^1(k), I^2(k) \cdot n_d)] \)

Derivations of the behavioral semantics are defined as for the mix-step semantics (Def. 21) where the subderivation for the big-step rules are started in the configuration \( \text{concr}_{\text{bi}}(I^2(k)), n_d \) where \( BI = I^1(k) \).

**Theorem 3** Mix-step and denotation-based message semantics are equivalent in the sense that there is a translation from the derivations according to one semantics to the derivations according to the other with identical local box states, and vice versa. In particular, the box denotations are sound w.r.t. the operational semantics.

**PROOF.** From the definitions above, we obtain the following equivalence:

\[ [BI] \overline{n_d} n_d = n'_d \]

\[ \iff \quad \text{expand}_{\text{bi}}(\overline{n_d}) \cdot n_d \in BDB_{\text{bi}} \quad \text{and} \quad \exists B' : \text{concr}_{\text{bi}}(\overline{n_d}), n_d \overset{\text{rmm}}{\rightarrow} B', n'_d \]
Mix-step and denotation-based message semantics only differ in the representation of the state of inner boxes. We stepwise construct pairs of partial derivations \( (D_{rmm}, D_{dm}) \) for the mix-step and denotation-based message semantics starting from an initial configuration of a program. We only perform a derivation step if a step is possible in both semantics. As the two partial derivations always have the same structure and length, there is a one-to-one correspondence between the configurations, and in particular between the occurring inner box mappings \( I_{rmm} \) and \( I_{dm} \). By induction on the derivation length, we show that the following invariant holds for all such corresponding mappings

\[
\text{dom} I_{rmm} = \text{dom} I_{dm}
\]

\[
\forall k \in \text{dom} I_{rmm} \cdot I_{rmm}(k) = \text{concr}_{br} I_{dm}(k), \quad \text{where } BI = I_{rmm}^1(k)
\]

and that if one derivation can do a next step then the other can also do a next step. In particular, we can use the above invariant to translate the representation of inner box states according to one semantics to that of the other semantics.

The innerbox mappings are only modified by rule \text{rmm-new-box} and the rules in Fig. 14 as well as their new counterparts in the denotation-based message semantics. Obviously, the invariant is established when a box is created. Now, consider the situations in which rules of Fig. 14 are applied. We consider the following three cases:

**Case 1:** Neither a rule of the mix-step semantics is applicable to the postconfiguration of \( D_{rmm} \) nor a rule of the abstract semantics is applicable to \( D_{dm} \). Then, both derivations get stuck and by induction hypothesis the invariant holds for all intermediate configurations.

**Case 2:** A rule of the abstract semantics is applicable to \( D_{dm} \). We only have to consider the rules corresponding to those in Fig. 14. The applicability implies \( [I^1(k)] I^2(k) \cdot n_d = n'_d \), thus the equivalence above yields:

\[
\exists B' : \text{concr}_{br} I_{dm}(I^2(k)), n_d \rightarrow^{+}_{rmm} B', n'_d
\]

From the induction hypothesis \( I_{rmm}(k) = \text{concr}_{br} I_{dm}(I^2(k)) \) with \( BI = I_{rmm}^1(k) \), we conclude

\[
\exists B' : I_{rmm}(k), n_d \rightarrow^{+}_{rmm} B', n'_d
\]

Thus, the rule in the mix-step semantics is also applicable. As the semantics is deterministic, \( B' \) is uniquely determined and \( B' = \text{concr}_{br} I_{dm}(I^2(k) \cdot n_d) \). Consequently, the invariant holds after the common derivation step as well.

**Case 3:** A rule of the mix-step semantics is applicable to \( D_{rmm} \). The reasoning
is analogous. However, we have to show that in all such situations

$$\text{expand}_m(T^m_{\text{msg}}) \cdot n_d \in BDB_m \text{ where } BI = T^i(k)$$

that is, we have to show that messages sequences occurring at the boundary of a box in the mix-step message semantics satisfy the properties of the Definitions 22, 23 and 24. Type correctness and stack correctness can be proved as usual for the standard semantics and are maintained in the two equivalent transformations of the semantics. The pass correctness is a consequence of the definitions of forward and receive in Fig. 9. A detailed technical proof of these properties is lengthy, does not provide new inside and is beyond the scope of this paper. □

8 Substitutability and Fully Abstractness

In this section, we provide formal notions for program contexts and for the use of a box in a program context. We show that the behavioral semantics is fully abstract with respect to the mix-step message semantics.

A program with box declaration box $B : I$ that has no implementation directive for $B$ is called a program context for $B : I$. The set of program contexts for $B : I$ is denoted by $P_{B : I}$. It is straightforward to extend notions for programs to program contexts. In particular, a program context $PC$ has a set of initial configurations (cf. Subsec. 5.4) which we denote by $IC_{PC}$. Note that $IC_{PC}$ is non-empty for any $PC$. A program context $PC$ for $B : I$ and a box implementation $BI$ for $B : I$ are called compatible, if the union of the declarations of $PC$ and $BI$ is a program. This union will be denoted by $PC[BI]$. Compatibility in particular implies that $PC$ and $BI$ do not contain different classes with the same name. Two box implementations $B_1$ and $B_2$ for $B : I$ are called compatible, if the union of the declarations of $B_1$ and $B_2$ is a program.

In the following, we assume that program contexts contain the declarations for Main, Globox, and MainClass according to Sec. 3 i.e. the combination of a program context with a box implementation yields an executable program. Without loss of generality, we also assume that the other box implementations do not use Main, Globox, and MainClass.

To make the notion of full abstractness precise, we need a notion of “observable program equivalence”. We could consider executable programs as boxes and use their behavioral semantics as appropriate notion for observable program equivalence. In fact, this would simplify the presentation, and seems to us to be a good choice to define programming language semantics in the future. However, for this paper that sets up the notion of boxes, we prefer a
notion that is directly based on the operational semantics. As our programs do not communicate with the environment, we assume that the environment can observe the object store of the global box. The intuition is that if the global box communicated with the environment, the communication would depend on this store.

**Definition 25** Two programs $P_1$ and $P_2$ are considered to be observational equivalent w.r.t. a set of initial configurations $I\mathcal{C}$, $P_1 \equiv_{I\mathcal{C}} P_2$, iff the global traces $T_1$ of $P_1$ and $T_2$ of $P_2$ according to the mix-step message semantics with start configuration $c^i \in I\mathcal{C}$ satisfy the following property: Both traces are either of the same finite length or they are infinite, and the object store $O^{i}_1$ in the $i$th configuration of $T_1$ has to be equal to the object store $O^{i}_2$ in the $i$th configuration of $T_2$ for all configurations with index $i$.

Based on this notion, we can state our central result that the behavioral semantics is fully abstract for compatible boxes:

**Theorem 4** Let $B_1$ and $B_2$ be two compatible implementations of box $B : I$.

$$ [B_1] = [B_2] \iff \left( \forall PC \in \mathcal{P}^{B_1} : \text{compatible}(PC, B_1) \land \text{compatible}(PC, B_2) \Rightarrow PC[B_1] \equiv_{I\mathcal{C}_{PC}} PC[B_2] \right) $$

**PROOF.** *Implication from left to right:* Consider derivations of $PC[B_1]$ and $PC[B_2]$ of the mix-step message semantics with start configuration $c^i$. According to Theorem 3, they are equivalent to derivations of denotation-based semantics. Because of $[B_1] = [B_2]$, subderivations for $B_1$ and $B_2$ either both yield no reply or the same reply. Thus, the derivations of their outer boxes are the same. In particular, the object stores in the global traces are the same.

*Implication from right to left:* Proof by contradiction: We assume that $[B_1] \neq [B_2]$ and show that there exists a program context $PC$ for which the following holds:

$$ \text{compatible}(PC, B_1) \land \text{compatible}(PC, B_2) \land \neg (PC[B_1] \equiv_{I\mathcal{C}_{PC}} PC[B_2]) $$

According to the assumption, there exists incoming message sequences $\pi_d$ and incoming messages $n_d$ such that $[B_1](\pi_d, n_d) \neq [B_2](\pi_d, n_d)$. For sequences with minimal length, we get $\text{expand}_{B_1}(\pi_d) = \text{expand}_{B_2}(\pi_d)$. We construct a program context $PC$ that generates the incoming message sequence $\pi_d \cdot n_d$:

- $PC$ contains declarations for $\text{Main}$, $\text{Globox}$, $\text{MainClass}$, and $\text{Object}$ as well as the declarations for the types in $B_{\text{sig}}$, as they are given in $B_1$. As $B_1$ and $B_2$ are compatible, they have the same declarations for the types in $B_{\text{sig}}$. 

49
• Let \( \text{REC} = \text{rec}(\text{expand}_{n1}(\pi_d) \cdot n_d) \) denote the received objects in \( \text{expand}_{n1}(\pi_d) \cdot n_d \), i.e., the objects that belong to the environment. For each \( o = (\text{out}, j : T) \in \text{REC} \) that appears as the receiver of a method call in \( \text{expand}_{n1}(\pi_d) \), we construct a class \( C_o \) with \( C_o \prec : T \). \( C_o \) has a field \( \text{refmain} \) to access the initial object of type \text{MainClass}.

• In \text{MainClass}, we declare a field for each passed object in \( \text{expand}_{n1}(\pi_d) \cdot n_d \). Thus, all passed objects can be read and written by \text{MainClass} and via the \( \text{refmain} \)-field as well by the classes \( C_o \).

• Method \text{main} first initializes the fields of \text{MainClass} by creating an object for each \( o \in \text{REC} \) of class \( C_o \), storing \( o \) in its corresponding field and storing itself to \( o.\text{refmain} \). Then, it creates a \( B \)-box. Then, it performs the outer execution according to \( \text{expand}_{n1}(\pi_d) \cdot n_d \) (see below).

• \( C_o \) contains an implementation for each method \( m \) called on \( o \) in \( \text{expand}_{n1}(\pi_d) \).

• If the supertype of \( C_o \) is an interface it contains some dummy implementation for the other methods. If \( m \) is called \( n \) times on \( o \) in \( \text{expand}_{n1}(\pi_d) \), \( n + 2 \) auxiliary methods are declared. The first \( n \) methods perform the outer execution according to the \( n \)-th call in \( \text{expand}_{n1}(\pi_d) \cdot n_d \) (see below).

• To dispatch to these auxiliary methods, we add for each auxiliary method \( m_{a_i}, i \in \{1 \ldots (n+2)\} \) of \( m \) a new field \( f_{m_i} \) to \( C_o \), which holds a key object. An additional field \( f_{mc} \) is used to store the key of the next method to be dispatched. When \( m \) is called the first time all fields \( f_{m_i} \) are assigned different newly created objects, which act as keys. The first auxiliary method is called and field \( f_{mc} \) is set to the value of \( f_{m_1} \). For each following invocation of \( m \) the dispatching code checks, by using the if-expression, the value of \( f_{mc} \) against each key field. It then calls the corresponding auxiliary method and sets \( f_{mc} \) to the next key. If the last key matches, which is the case for all calls following \( \pi_d \cdot n_d \), the method simply returns some type correct object. According to this construction, every outgoing call in \( \text{expand}_{n1}(\pi_d) \) and \( \text{expand}_{n1}(\pi_d) \cdot n_d \), if this is defined, is handled by a different auxiliary method.

• Now, we explain how to construct the method bodies to “perform the outer execution according to \( \pi_d \cdot n_d \)”: As each method (except \text{main}) corresponds to a specific outgoing call in \( \text{expand}_{n1}(\pi_d) \), we can determine which parameters are objects that are exposed for the first time. In a first step, these objects are stored into the corresponding fields of \text{MainClass}. According to the grammar in Def. 23 each call execution consists of a prefix of a sequence of pairs \( ic, ric \) and a final return \( roc \). The possible callbacks between the \( ic \)'s and \( ric \)'s are handled by other methods. The parameters for the incoming calls \( ic \) can be taken from the respective fields in \text{MainClass}. After the return from a call, new exposed return values are stored.

• \text{MainClass} has two methods \text{mkDistinction1} and \text{mkDistinction2} with no parameter which set specific fields \( fx1 \) and \( fx1 \) of the initial object.

There are four different reasons (and their symmetric counterparts) of why \( [B1](\pi_d, n_d) \neq [B2](\pi_d, n_d) \). For each of these cases, we extend the above
modify the object store, we get observably different traces for
the program context (if necessary) to produce a pair of different object stores in
the global trace. The idea is that at the point where the first reply appears that
is distinct in the traces for PC[B1] and PC[B2], mkDistinction1 is called in
the trace for PC[B1] and mkDistinction2 in the traces for PC[B2]. As these
methods are not called before the distinction appeared and as they differently
modify the object store, we get observably different traces for PC[B1] and
PC[B2].

(1) [B1](\pi_d, n_d) \neq \bot and [B2](\pi_d, n_d) = \bot: In this case, the global trace of
PC[B1] is longer than that of PC[B2].

(2) [B1](\pi_d, n_d) = [o_d, m(\pi_d)] \neq [o'_d, m'(\pi'_d)] = [B2](\pi_d, n_d). There are three
cases how the call messages may differ from each other.
(a) \o_d \neq \o'_d. Let \o_d = d.j:T and \o'_d = d'.j':T'. We know that d and d'
are both out. j must be different from j', as otherwise the above
construction yields T = T'. Let n, n' be the number of invocations
so far of o_d.m and o'_d.m', respectively. We now modify the classes
C_{o_d} and C_{o'_d} described above. We change the n + 1 auxiliary method
in C_{o_d} for m and the n' + 1 auxiliary method in C_{o'_d} for m'. To
the latter implementation we add the call \textit{refmain.mkDistinction1()},
to the former the call \textit{refmain.mkDistinction2()}. As there are no
previous calls to these methods, they cause the objects stores in the
global traces to become different.
(b) m \neq m' \wedge o_d = o'_d. This case is handled like above.
(c) \nu_d \neq \nu'_d \wedge o_d = o'_d \wedge m = m' for some i. Let \pi be the formal parameter
variables of method m. To handle that case, we add to the n+1 auxili-
ary method the expression \texttt{if(x_i == e) \textit{refmain.mkDistinction1()}
else \textit{refmain.mkDistinction2()}} where e depends on \nu_d. If \nu_d =
null then e \equiv null. Otherwise \nu_d = \nu'_d for some object \o'_d. Let f be the
field in \texttt{MainClass} which stores the reference to object \o'_d. Then e \equiv
\textit{refmain.f}. We assume here that if \o'_d is a new object which has not
been seen by the environment yet that it already has been assigned
in m to \texttt{refmain.f} before we execute the if-expression.

(3) [B1](\pi_d, n_d) = [\nu_d \neq \nu'_d = [B2](\pi_d, n_d). This is done similar to the
last subcase of the case above. We only have to modify the method that
invokes the method to which the return belongs and after return compare
the result value.

(4) [B1](\pi_d, n_d) = [o_d.m(\pi_d) \neq \nu'_d = [B2](\pi_d, n_d). For this case, we modify
the corresponding auxiliary method for m by adding a call to the distinc-
tion method \textit{refmain.mkDistinction1()}. Furthermore, we add a
call \textit{refmain.mkDistinction2()} to the method that invokes the method
that belongs to the return [\nu'_d, immediately after the method invocation.

\Box
9 Conclusions and Future Work

Boxes are a light-weight structuring technique with interesting semantical properties. The box declaration is essentially an annotation that allows object-oriented programmers to indicate a boundary between components and their users. As in most ownership disciplines the boundary is on the instance level.

In this paper, we focused on the semantical properties of boxed OO-programs. Based on the box structure, we developed a message semantics that is equivalent to an enhanced standard operational semantics. By abstracting from box states, we developed a behavioral semantics for boxes. To avoid the exposition and reception of hidden types and to allow separate development of components and client programs, we proposed an adaptation technique for types at the box boundary. To check the soundness and conciseness of the behavioral box semantics, we proved that the behavioral semantics a fully abstract w.r.t. the underlying message semantics. This result is substantiated by the fact that the message semantics is equivalent to the enhanced standard semantics.

We believe that our approach leads to a number of threads for future work that are of practical and theoretical interest:

- There are several open questions concerning the development of the box model to full object-oriented programming languages.
- Even if programmers have to use a language without boxes, they should use the gained insights for the development of substitutable components. A number of programming rules can be derived based on the foundations. E.g., it is dangerous to use a public class for the implementation of an interface that is visible at the component boundary. At least, objects of such classes should not be exposed.
- Considering every program as a box in future programming languages, it would be semantically simpler to treat realistic input and output streams. In addition, the behavioral semantics provides a natural equivalence relation for program components with input and output.
- A driving force underlying this work was the goal to find a component semantics that is independent of program contexts. The next step is to use the developed semantics as a basis for component specification (see [23]) and verification, in particular for proving that one component can substitute another one in all possible contexts.
- The semantical framework is as well suited for modular static analysis of program components (cf. [28] for such an approach based on a different component notion).
- Another interesting aspect is that a given implementation can be used as implementation for boxes with different interfaces. This gives rise to a or-
dering relation on component behavior.

Last, but not least, the extension of the box model to concurrency is a promising line of future research as it allows to combine local data invariants for parts of the heap with synchronization at box boundaries.

References

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.


