Full Abstraction at Package Boundaries of Object-Oriented Languages*

Yannick Welsch and Arnd Poetzsch-Heffter

University of Kaiserslautern, Germany
{welsch,poetzsch}@cs.uni-kl.de

Abstract. We develop a fully abstract trace-based semantics for sets of classes in object-oriented languages, in particular for Java-like sealed packages. Our approach enhances a standard operational semantics such that the change of control between the package and the client context is made explicit in terms of interaction labels. By using traces over these labels, we abstract from the data representation in the heap, support class hiding, and provide fully abstract package denotations. The soundness and completeness of our approach is proven using innovative simulation techniques.

1 Introduction

Systems, components, and libraries have to evolve over time to meet new requirements. In an object-oriented setting, such evolution steps affect the classes used in the implementation. An important aspect for safe evolution is the ability to modularly check for *compatibility*, i.e., whether a new version has the same behavior as the old one in *all* program contexts in which the old version can be used. In particular, every refactoring should guarantee compatibility. Mutual compatibility corresponds to the classical notion of *(contextual) equivalence*: Two sets of classes are equivalent if they exhibit the same operational behavior in every possible context. Proving compatibility or equivalence is challenging because

(1) the number of possible contexts is infinite and contexts are complex
(2) the states and heaps can be significantly different between the versions.

To meet these challenges, we exploit denotational methods. A denotational semantics for classes is called *fully abstract* [11, 13] if classes that have the same denotation are exactly those that are contextually equivalent. In particular, a fully abstract semantics has to abstract from states and heaps to meet challenge (2) above. Proving that two sets of classes are equivalent in the (fully abstract) denotational setting amounts to proving that they have the same denotation.

The central contribution of this paper is the design of such a fully abstract semantics for packages of a Java subset. Furthermore, the paper provides a detailed explanation of the full abstraction proof. To explain our approach in more detail, we consider the following two versions of package xutil:

* This research is funded by the EU project FP7-231620 HATS: Highly Adaptable and Trustworthy Software using Formal Models.

```
package xutil; import java.util.*;

public class Bag implements Collection {
  private ArrayList mList;
  public boolean addAll(Collection c){ BODY }
  ...
}
```

```
package xutil; import java.util.*;
public interface IBag extends Collection {
  public boolean addAll(Collection c);
  ... }
public class Bag implements IBag {
  private MyList mList;
  public boolean addAll(Collection c){ BODY2 }
  ... }
class MyList { IMPL }
```

The goal is to show that the version on the right-hand side has the same behavior as the one on the left-hand side in all possible program contexts. In particular, we address the following language-related challenges:

- change of implementation, e.g., BODY2 instead of BODY, MyList instead of ArrayList
- change of subtype hierarchy, e.g., interface IBag is added
- non-public, encapsulated classes, e.g., the new class MyList is package local
- use of imported types in signatures (e.g., interface Collection in signature of method addAll) and in the implementation (e.g., ArrayList is imported)
- inheritance and casts (not illustrated)

Approach. In our approach, the denotation of a package (or set of classes and interfaces) is expressed by the interactions between code belonging to the package and code belonging to the context. It is defined in *two* steps starting from a standard operational semantics. In the first step, the operational semantics is augmented in a way that the interactions can be made explicit. In the second step, traces of interaction labels are used to semantically characterize the package behavior. A non-trivial aspect is the treatment of inheritance, because with inheritance, some code parts of a class/object might belong to the context and other parts to the package under investigation.

Using traces allows abstracting from the state and heap representation in the old and new version. It solves challenge (2). To obtain a finite representation of all contexts and solve challenge (1), we construct a nondeterministic *most general context* that exactly exhibits the possible behavior of contexts. Using an operational semantics as a starting point has the advantage that we can use simulation relations applied to standard configurations (i.e., heap, stack) for the full abstraction proof. Furthermore, it provides a direct formal relation to Hoare-like program logics and standard techniques for static program analysis.

Related Work. Banerjee and Naumann [4] presented a method to reason about whole-program equivalence in a Java subset. Under a notion of confinement for class tables, they prove equivalence between different implementations of a class by relating their (classical, fixpoint-based) denotations by simulations. Silva, Naumann, and Sampaio [15] extend this work to prove several refactoring laws for whole hierarchies of classes.

Similar to our work, Jeffrey and Rathke [8] give a fully abstract trace semantics for a Java subset with a package-like construct. However, they do not consider inheritance,

down-casting and cross-border instantiation. Using similar techniques, Ábrahám et al. [2] give a fully abstract semantics for a concurrent class-based language (without inheritance and subtyping).

Whereas we use simulation techniques to prove full abstraction, other authors apply (bi-)simulations to relate two program parts. This technique was first used by Hennessy and Milner [7] to reason about concurrent programs. Sumii and Pierce used bisimulations which are sound and complete with respect to contextual equivalence in a language with dynamic sealing [16] and in a language with type abstraction and recursion [17]. Koutavas and Wand, building on their earlier work [9] and the work of Sumii and Pierce, used bisimulations to reason about the equivalence of *single* classes [10] in different Java subsets. The subset they considered includes inheritance and down-casting. Their language, however, neither considers interfaces nor accessibility of types.

Outline. In the following, we illustrate our method to develop fully abstract denotational semantics for Java-like packages. The remainder of this paper is structured as follows. Section 2 introduces the formalized language LPJava and the notion of source compatibility. Section 3 gives the operational and trace semantics of LPJava and presents the full abstraction result. Section 4 shows how full abstraction can be proved by simulation relations and Section 5 gives a general outline on how to prove two components compatible. Section 6 presents directions for future work and concludes. For proofs and further details on the formalization, we refer to the extended report [18].

Notations. We use the *overbar* notation $\overline{x}$ to denote a finite list and the *hat* notation $\widehat{x}$ to denote a set. The empty list and set are denoted by $\bullet$ and the concatenation of list $\overline{x}$ and $\overline{y}$ is denoted by $\overline{x} \cdot \overline{y}$. Concatenation is sometimes implicit by writing terms in juxtaposition. Single elements are implicitly treated as lists/sets when needed. The function last$(\dots)$ returns the last element of a list. The expression $\mathcal{M}[x \mapsto y]$ yields the map $\mathcal{M}$ where the entry with key x is updated with the value y, or, if no such key exists, the entry is added. The empty map is denoted by $\varnothing$ and dom$(\mathcal{M})$ and rng$(\mathcal{M})$ denote the domain and range of the map $\mathcal{M}$.

2 Formalization of LPJava and Source Compatibility

The language considered in the following is a sequential object-oriented language called LPJava (see Fig. 1). It has interfaces, classes and subtyping. To consider the additional challenge of inheritance and type hiding, it also has subclassing and a package system. Classes can extend other classes, and can be declared either package-local or public. Primitive data types (like **int**, etc.) are not considered as they do not provide additional insight. For simplicity, all methods are assumed to be public and all fields to be private. LPJava also allows explicit casting, which leads to more distinguishing power from class contexts. The operator $(p.t)E_1 : E_2$ encodes both an *instanceof* and *cast* operator, i.e., it yields the value of E_2 if the value of E_1 cannot be cast to $p.t$.

We assume that every class has a default constructor. Similar as for Java [5], the default constructor has the same access modifier as its class. A *package* (denoted by Q or R) has a name and consists of a sequence of type declarations. We assume that

$$
\begin{array}{rcl}
K,X,Y & ::= & \overline{Q} \\
Q,R & ::= & \textbf{package}\ p\ ;\ \overline{D} \\
D & ::= & [\textbf{public}]\ \textbf{class}\ c\ \textbf{extds}\ p.c\ \textbf{impls}\ \overline{p.i}\ \{\ \overline{F}\ \overline{M}\ \} \\
 & | & [\textbf{public}]\ \textbf{interface}\ i\ \textbf{extds}\ \overline{p.i}\ \{\ \overline{M}\ \} \\
F & ::= & \textbf{private}\ p.t\ f\ ; \\
M & ::= & \textbf{public}\ p.t\ m(\overline{p.t\ v})\ \left(\ ;\ |\ \{\ E\ \}\ \right) \\
E & ::= & x \mid \mathsf{null} \mid \mathsf{new}\ p.c() \mid (p.t)E : E \mid E.f \mid E.f = E \\
 & | & \mathsf{let}\ p.t\ x = E\ \mathsf{in}\ E \mid E.m(\overline{E}) \mid E == E\ ?\ E : E
\end{array}
$$

$$
\begin{array}{rcl}
t & ::= & c \mid i \\
c & \in & \text{class names} \\
i & \in & \text{interface names} \\
p,q & \in & \text{package names} \\
f & \in & \text{field names} \\
m & \in & \text{method names} \\
x & \in & \text{variable names}
\end{array}
$$

Fig. 1. Abstract syntax of LPJava

packages are sealed (cf. [6], Sect. 2), meaning that once a package is defined no new class and interface definitions can be added to the package. Types are fully qualified by their package name.

A *codebase*, which consists of a list of packages, is denoted by K, X or Y. If it satisfies all the well-formedness conditions of the language, i.e., well-formed type hierarchy, well-typedness of all expressions, etc., we write $\vdash K$ (or $\vdash X$, $\vdash Y$) and call such a codebase a *component*. Note that components are definition-complete. To join two codebases into a larger codebase, we write them in juxtaposition (i.e., KX). If we join a codebase K and a component X, we often call K a (class) *context* of X.

A prerequisite for two components to have the same behavior is that whenever the first component can be joined with a context into a larger component, then the second one can be joined as well using the same context. This property,[1] focusing solely on typing and not behavioral aspects, is called *source compatibility* [8]:

Definition 1 (Source compatibility). *A component Y is* source compatible *with a component X if for any codebase K: $\vdash KX$ implies $\vdash KY$.*

It is important to notice that this does not allow automatic checking that a component Y is source compatible with X, because the definition quantifies over an infinite set of contexts. However, a set of checkable conditions that are necessary and sufficient for Y to be source compatible with X can be given. We describe them in the following.[2] The package names occurring in X must exactly be those occurring in Y. Every public type defined in X must appear in Y. For public types of X, every method which is part of the type (declared or inherited) in X must also have a method with the same signature (i.e. same parameter and return type) in Y and vice versa. The subtype hierarchy between public types of X must be maintained in Y.

Theorem 1. *A component Y is source compatible with a component X if and only if the checkable conditions between X and Y described above hold.*

[1] Note that the definition is not symmetric. This allows Y to be a more *refined* version of X.

[2] As this paper focuses on behavioral aspects, we do not explain here why these conditions are necessary and sufficient. However, explanations and proofs can be found in [18].

$E ::= \ldots \mid r$		extd. expressions
$r ::= j \mid \mathsf{null}$		reference
$\mathcal{O} ::= j \mapsto H$		heap
$H ::= (V, L, p.c, \overline{r})$		heap entry
$\mathcal{F} ::= (E \mid \mathcal{E})_L{:}p.t$		typed stack frame
$V ::= \mathsf{internal} \mid \mathsf{exposed}$		exposure flag
$L ::= \mathsf{ctxt} \mid \mathsf{comp}$		origin location
$j \in$	object identifiers	

Fig. 2. Semantic entities for LPJava

$t ::= \overline{l}$		trace
$l ::= \mu! \mid \mu? \mid \tau$		label
$\mid \mathsf{error} \mid \mathsf{halt}$		
$\mu ::= \mathsf{call}\ o.m(\overline{v})$		call message
$\mid \mathsf{rtrn}\ v$		return message
$o ::= j{:}T^{\alpha}$		abstracted object
$v ::= o \mid \mathsf{null}$		label value
$T^{\alpha} ::= \langle \widehat{p.c}, \widehat{p.i}, \widehat{m} \rangle$		abstracted type

Fig. 3. Syntax of traces

3 Trace Characterization of Component Behavior

In this section, we characterize the behavior of a component X in terms of its possible interaction traces with program contexts. We first define the interaction traces of X with a specific program context. Then, we introduce nondeterministic expressions that allow for the definition of most general contexts which simulate all possible contexts. Finally, we state the full abstraction result.

3.1 Enhanced Operational Semantics

We enrich a standard semantics in such a way that the interactions between a component X and its program context K become explicit and call it the enhanced semantics. In particular, we define the traces of interactions between X and K. The rules in Fig. 4 describe the (enhanced) small-step operational semantics of LPJava. Auxiliary functions are defined in Fig. 5.

The operational rules for a component X in a context K are based on a labelled small-step reduction judgement of the form $\zeta \overset{l}{\rightsquigarrow} \zeta'$ with configurations $\zeta = KX, \mathcal{O}, \overline{\mathcal{F}}$. The heap $\mathcal{O}$ is a map from object identifiers to heap entries and $\overline{\mathcal{F}}$ is a list of typed stack frames (see Fig. 2). The configurations are augmented with additional information. However, this does not change the standard operational behavior. The flag L ranges over $\{\mathsf{ctxt}, \mathsf{comp}\}$ and indicates whether entities belong to the context K or to the component X. It is used in stack frames to mark if the code (E or $\mathcal{E}$) that is part of this stack frame originates from X or K. It is also used in heap entries to denote whether the object has been created by code of X or K. The flag V is used in heap entries to denote whether an object created in the context has been exposed to the component or vice versa. Objects are always created internally (see **RI-NewObj**) but can over time be exposed when they are passed from (to) the component to (from) the context.

Stack frames are associated with either the component X or the context K. The topmost stack frame contains an expression E and all other stack frames contain an evaluation context $\mathcal{E}$. An evaluation context $\mathcal{E}$ (see [19]) is an expression with a *hole* $[]$ somewhere inside the expression. We write $\mathcal{E}[E]$ to mean that the hole in $\mathcal{E}$ is replaced

by expression E. A hole in $\mathcal{E}$ can only appear at certain positions defined as follows:

$$\begin{aligned}\mathcal{E} ::= {} & [\,] \mid \mathcal{E}.f \mid \mathcal{E}.f = E \mid r.f = \mathcal{E} \mid \mathsf{let}\ p.t\ x = \mathcal{E}\ \mathsf{in}\ E \mid (p.t)\mathcal{E} : E \mid \mathcal{E}.m(\overline{E}) \\ & \mid r.m(\overline{r}, \mathcal{E}, \overline{E}) \mid \mathcal{E} == E\ ?\ E : E \mid r == \mathcal{E}\ ?\ E : E\end{aligned}$$

We say that X *controls execution* if code of X is executed; otherwise K controls execution. The function currloc(ζ) from Fig. 5 is used to determine who controls execution. An interaction is a change of control (see **RI-Call-Boundary** and **RI-Return-Boundary**). Note that only interactions allocate or deallocate stack frames, i.e., calls within the context or the component are not handled using the stack (see **RI-Call-Intern**). Labels record changes of control. An interaction trace is a finite sequence of labels (Fig. 3). Interaction is considered from the viewpoint of the component. Input labels (marked by ?) express a change of control from the context to the component; output labels (marked by !) express a change from the component to the context. There are input and output labels for method invocation and return, as well as labels for well-formed and abrupt program termination. The labels for method invocation and return include the parameter and result values together with their *abstracted types* (explained later).

A *program context* is a context that has a public class $p.c$ with a main method lang.Object main(), where the class $p.c$ is called a *startup class*. It is executed by calling main. In the following we assume that the startup class is always main.Main and is defined in the context K. The initial configuration initζ_{KX} is then defined as $KX, \mathcal{O}, \mathcal{F} \cdot \bullet$ where $\mathcal{O} \stackrel{\text{def}}{=} \emptyset[j \mapsto (\mathsf{internal}, \mathsf{ctxt}, \mathsf{main.Main}, \overline{\mathsf{null}})]$ and $\mathcal{F} \stackrel{\text{def}}{=} E[j/\mathsf{this}]_{\mathsf{ctxt}}{:}\mathit{lang.Object}$, if E is the body of the main method.

Traces. In the following, we consider the traces (i.e. finite lists of labels) which are generated by steps of the operational semantics. To compare traces of different components and with different contexts, we abstract from package local types and types declared in the context. Package local types should not appear in the labels, because different components might use different local types. Types in labels are *abstracted* (see typeabs$_{KX}(p.c)$ in Fig. 5) to a representation which only preserves the information (1) which public supertypes of $p.c$ belong to the component (2) which of their methods are not overridden by the context. The reason for (1) is that these are the types of X that can be used in cast expressions in the context. Based on the label, it becomes thus clear which cast expressions will succeed and which not. The reason for (2) is that, based on the label, we know the methods that, if invoked with the object as receiver, lead to changes of control. As X defines a finite set of types (denoted by $\mathcal{T}_X$), there are only a finite set of abstracted types that can occur in traces with X. This set is denoted by $\mathcal{T}_X^\alpha$ and can be constructed from X.

To abstract from internal τ steps of a computation, we provide a large step version of the enhanced semantics (denoted $\stackrel{t}{\Longrightarrow}$) that is defined by:

$$\frac{}{\zeta \stackrel{\bullet}{\Longrightarrow} \zeta} \qquad \frac{\zeta \stackrel{t}{\Longrightarrow} \zeta' \qquad \zeta' \stackrel{\tau}{\rightsquigarrow}_* \zeta'' \qquad \zeta'' \stackrel{l}{\rightsquigarrow} \zeta''' \qquad l \neq \tau}{\zeta \stackrel{t \cdot l}{\Longrightarrow} \zeta'''}$$

Every large step represents a finite number of τ steps (denoted by $\stackrel{\tau}{\rightsquigarrow}_*$, the reflexive, transitive closure of $\stackrel{\tau}{\rightsquigarrow}$) followed by a non-$\tau$ step. Note that τ does not appear in labels

RI-Internal-Step

$$\frac{\mathcal{O}, E \dashrightarrow^{L}_{KX} \mathcal{O}', E'}{KX, \mathcal{O}, \mathcal{E}[E]_L{:}p.t \cdot \overline{\mathcal{F}} \overset{\tau}{\rightsquigarrow} KX, \mathcal{O}', \mathcal{E}[E']_L{:}p.t \cdot \overline{\mathcal{F}}}$$

RI-NewObj

$$\frac{j \notin dom(\mathcal{O}) \qquad H = (\mathsf{internal}, L, p.c, \overline{\mathsf{null}})}{\mathcal{O}, \mathsf{new}\ p.c() \dashrightarrow^{L}_{KX} \mathcal{O}[j \mapsto H], j}$$

RI-Call-Intern

$$\frac{\mathrm{type}_{\mathcal{O}}(j) = p.c \qquad \langle m, \_, E\rangle \in_{KX} p.c \qquad \mathrm{mdecl}_{KX}(p.c, m) = L}{\mathcal{O}, j.m(\overline{r}) \dashrightarrow^{L}_{KX} \mathcal{O}, E[j/\mathsf{this}, \overline{r}/\overline{v_p}]}$$

RI-Call-Boundary

$$\frac{\begin{array}{c}\mathrm{type}_{\mathcal{O}}(j) = p.c \qquad \langle m, \_, E\rangle \in_{KX} p.c \\ \mathrm{mdecl}_{KX}(p.c, m) = \neg L \qquad l = \mathrm{mcall}_{KX}(j, m, \overline{r}, \mathcal{O}, L) \qquad \mathcal{O}' = \mathrm{expose}(j\overline{r}, \mathcal{O})\end{array}}{KX, \mathcal{O}, \mathcal{E}[j.m(\overline{r})]_L{:}p.t \cdot \overline{\mathcal{F}} \overset{l}{\rightsquigarrow} KX, \mathcal{O}', E[j/\mathsf{this}, \overline{r}/\overline{v_p}]_{\neg L}{:}p'.t' \cdot \mathcal{E}_L{:}p.t \cdot \overline{\mathcal{F}}}$$

RI-Return-Boundary

$$\frac{l = \mathrm{mrtrn}_{KX}(r, \mathcal{O}, L) \qquad \mathcal{O}' = \mathrm{expose}(r, \mathcal{O})}{KX, \mathcal{O}, r_L{:}p'.t' \cdot \mathcal{E}_{\neg L}{:}p.t \cdot \overline{\mathcal{F}} \overset{l}{\rightsquigarrow} KX, \mathcal{O}', \mathcal{E}[r]_{\neg L}{:}p.t \cdot \overline{\mathcal{F}}}$$

RI-Fail

$$\frac{E = \mathsf{null}.f_i \vee E = \mathsf{null}.f_i = r \vee E = \mathsf{null}.m(\overline{v})}{KX, \mathcal{O}, \mathcal{E}[E]_L{:}p.t \cdot \overline{\mathcal{F}} \overset{\mathsf{error}}{\rightsquigarrow} KX, \mathcal{O}, \bullet}$$

RI-Halt

$$\frac{}{KX, \mathcal{O}, r_{\mathrm{ctxt}}{:}p.t \overset{\mathsf{halt}}{\rightsquigarrow} KX, \mathcal{O}, \bullet}$$

Fig. 4. Transition rules for the enhanced small-step semantics, using the helper judgement $\dashrightarrow^{L}_{KX}$ for internal steps that are local to an evaluation context. The rules **RI-Cast**, **RI-Let**, **RI-If**, **RI-FieldSel** and **RI-FieldUp** are not shown but can be found in [18].

of large steps. Large steps always jump to the state right after the next non-τ label has been generated.

Termination ($\zeta\downarrow$) and divergence ($\zeta\uparrow$) are defined in the usual way. We write $\zeta\downarrow$ iff there exists a terminal configuration $\zeta_f = KX, \mathcal{O}, \bullet$ such that $\zeta \overset{t}{\Longrightarrow} \zeta_f$. Note that termination occurs if and only if the last label is either error or halt. We write $\zeta\uparrow$ iff the execution diverges. As can be seen from the transition rules, evaluation is deterministic (up to object naming).

In order to deal with the non-deterministic choice of fresh object identifiers, we introduce (object) renamings. A *renaming* is a bijective relation on object identifiers. We write ρ for such a relation. We can then consider traces equivalent (or related) if they are equal modulo a renaming.

Definition 2 (Related traces). $t_1 \equiv^{\rho} t_2$ *iff the object identifiers appearing at the same positions in the traces are related under* ρ *and the types appearing at the same position are equal. If we are not interested in a particular* ρ*, we omit it for brevity.*

In the following, we use the straightforward generalization of this definition of *equality modulo a renaming* ($\equiv^{\rho}$) to arbitrary syntactic structures. The observable behavior of a program run can then be reduced to the traces it exposes.

$$
\begin{array}{ll}
\mathrm{expose}(r,\mathcal{O}) & \stackrel{\text{def}}{=} \begin{cases} \mathcal{O} & \text{if } r = \mathsf{null} \\ \mathcal{O}[j \mapsto (\mathsf{exposed}, L, p.c, \overline{r})] & \text{if } r = j \wedge \mathcal{O}(j) = (\_, L, p.c, \overline{r}) \end{cases} \\
\mathrm{mcall}_{KX}(j,m,\overline{r},\mathcal{O},L) & \stackrel{\text{def}}{=} \mathsf{call}\ \mathrm{objectabs}_{KX}(j,\mathcal{O}).m(\mathrm{objectabs}_{KX}(\overline{r},\mathcal{O}))\ \mathrm{fromdir}(L) \\
\mathrm{mrtrn}_{KX}(r,\mathcal{O},L) & \stackrel{\text{def}}{=} \mathsf{rtrn}\ \mathrm{objectabs}_{KX}(r,\mathcal{O})\ \mathrm{fromdir}(L) \\
\mathrm{fromdir}(L) & \stackrel{\text{def}}{=} \begin{cases} ? & \text{if } L = \mathsf{ctxt} \\ ! & \text{if } L = \mathsf{comp} \end{cases} \\
\mathrm{objectabs}_{KX}(r,\mathcal{O}) & \stackrel{\text{def}}{=} \begin{cases} \mathsf{null} & \text{if } r = \mathsf{null} \\ j{:}\mathrm{typeabs}_{KX}(\mathrm{type}_{\mathcal{O}}(r)) & \text{if } r = j \end{cases} \\
\mathrm{typeabs}_{KX}(p.c) & \stackrel{\text{def}}{=} \langle \widehat{p.c}, \widehat{p.i}, \widehat{m} \rangle \text{ where } \widehat{p.c} \cup \widehat{p.i} \text{ are the supertypes of } p.c \text{ that are} \\
 & \quad \text{in pubtypes}(X) \text{ and} \\
 & \quad \widehat{m} = \{m \mid m \in \mathrm{methods}_X(\widehat{p.c} \cup \widehat{p.i}) \wedge \mathrm{mdecl}_{KX}(p.c,m) = \mathsf{comp}\} \\
\mathrm{pubtypes}(X) & \stackrel{\text{def}}{=} \{p.t \mid p.t \in \mathcal{T}_X \wedge \mathrm{public}_X(p.t)\} \quad \text{(public types defined in } X\text{)} \\
\mathrm{currloc}(KX,\mathcal{O},\overline{\mathcal{F}}) & \stackrel{\text{def}}{=} L \text{ if } \overline{\mathcal{F}} = E_L{:}p.t \cdot \overline{\mathcal{F}'} \quad \text{(location of top of stack)} \\
\mathrm{type}_{\mathcal{O}}(r) & \stackrel{\text{def}}{=} \begin{cases} \bot & \text{if } r = \mathsf{null} \\ p.c & \text{if } \mathcal{O}(r) = (\_,\_,p.c,\_) \end{cases} \\
\neg L & \stackrel{\text{def}}{=} L' \text{ where } L \neq L' \text{ (similar for } \neg V\text{)} \\
\mathrm{filter}(\mathcal{O},V) & \stackrel{\text{def}}{=} \{j \in \mathcal{O} \mid \mathcal{O}(j) = (V,\_,\_,\_)\} \\
\mathrm{filter}(\mathcal{O},L) & \stackrel{\text{def}}{=} \{j \in \mathcal{O} \mid \mathcal{O}(j) = (\_,L,\_,\_)\} \\
\mathrm{filter}(\mathcal{O},V,L) & \stackrel{\text{def}}{=} \mathrm{filter}(\mathcal{O},V) \cap \mathrm{filter}(\mathcal{O},L) \\
\mathrm{visible}(\mathcal{O},L) & \stackrel{\text{def}}{=} \mathrm{filter}(\mathcal{O},\mathsf{exposed}) \cup \mathrm{filter}(\mathcal{O},\mathsf{internal},L) \\
\mathrm{abs}_X(\langle \widehat{p.c}, \widehat{p.i}, \widehat{m} \rangle) & \stackrel{\text{def}}{=} \langle \widehat{p.c}', \widehat{p.i}', \widehat{m}' \rangle \text{ where } (\widehat{p.c}' \cup \widehat{p.i}') = (\widehat{p.c} \cup \widehat{p.i}) \cap \mathrm{pubtypes}(X) \\
 & \quad \text{and } \widehat{m}' = \{m \mid m \in (\widehat{m} \cap \mathrm{methods}_X(\widehat{p.c} \cup \widehat{p.i}))\}
\end{array}
$$

Fig. 5. Helper definitions, where $\mathrm{mdecl}_{KX}(p.c,m)$ yields the location L where the method body has been declared (searching from the class $p.c$ upwards) and $\mathrm{methods}_X(p.t)$ yields the method names of declared and inherited methods in $p.t$.

Definition 3 (Traces). *The* traces *of a component X with a program context K are:* $\mathrm{Traces}(KX) \stackrel{def}{=} \{t \mid \exists \zeta : \mathrm{init}\zeta_{KX} \stackrel{t}{\Longrightarrow} \zeta\}$

Note that $\mathrm{Traces}(KX)$ is closed w.r.t. renaming, i.e., if $t \in \mathrm{Traces}(KX)$ and $t' \equiv t$, then also $t' \in \mathrm{Traces}(KX)$. Furthermore, $\mathrm{Traces}(KX)$ is prefix-closed and only refers to public types in X.

3.2 Most General Context

Although the traces abstract from the context, we still have to consider the traces of all possible program contexts in order to describe the full behavior of a component. This issue is addressed in this section by constructing a most general context κ_X that enables all possible interactions that X can engage in. The context κ_X represents exactly all contexts that X can have. Compared to a concrete context, κ_X abstracts over types, objects, and operational steps. To represent κ_X, we extend LPJava by nondeterministic

$$\textbf{MGC-Skip}\quad \frac{}{\mathcal{O}, \mathsf{nde} \dashrightarrow_{KX}^{\mathsf{ctxt}} \mathcal{O}, \mathsf{nde}}$$

$$\textbf{MGC-NewObj}\quad \frac{j \notin dom(\mathcal{O}) \qquad p.c \in \mathcal{T}_{KX} \qquad \mathrm{public}_{KX}(p.c)}{\mathcal{O}, \mathsf{nde} \dashrightarrow_{KX}^{\mathsf{ctxt}} \mathcal{O}[j \mapsto (\mathsf{internal}, \mathsf{ctxt}, p.c, \overline{\mathsf{null}})], \mathsf{nde}}$$

$$\textbf{MGC-PrepareCall}\quad \frac{\begin{array}{c} j, r_i \in \mathrm{visible}(\mathcal{O}, \mathsf{ctxt}) \cup \{\mathsf{null}\} \qquad \mathrm{type}_{\mathcal{O}}(j) = p.c \\ \langle m, \overline{q.t} \twoheadrightarrow \_, \_ \rangle \in_{KX} p.c \qquad \mathrm{mdecl}_{KX}(p.c, m) = \mathsf{comp} \qquad \overline{\mathrm{type}_{\mathcal{O}}(r) \leq_{KX} q.t} \qquad x \text{ fresh} \end{array}}{\mathcal{O}, \mathsf{nde} \dashrightarrow_{KX}^{\mathsf{ctxt}} \mathcal{O}, \mathsf{let}\ lang.Object\ x = j.m(\overline{r})\ \mathsf{in}\ \mathsf{nde}}$$

$$\textbf{MGC-PrepareRtrn}\quad \frac{r \in \mathrm{visible}(\mathcal{O}, \mathsf{ctxt}) \cup \{\mathsf{null}\} \qquad \mathrm{type}_{\mathcal{O}}(r) \leq_{KX} p.t}{KX, \mathcal{O}, \mathsf{nde}_{\mathsf{ctxt}} : p.t \cdot \overline{\mathcal{F}} \overset{\tau}{\rightsquigarrow} KX, \mathcal{O}, r_{\mathsf{ctxt}} : p.t \cdot \overline{\mathcal{F}}}$$

$$\textbf{MGC-Fail}\quad \frac{}{KX, \mathcal{O}, \mathsf{nde}_{\mathsf{ctxt}} : p.t \cdot \overline{\mathcal{F}} \overset{\mathsf{error}}{\rightsquigarrow} KX, \mathcal{O}, \bullet}$$

Fig. 6. Transition rules for the most general context

expressions ($E ::= \ldots \mid \mathsf{nde}$) and corresponding reduction rules (Fig. 6). Nondeterministic expressions are only allowed in these most general contexts for LPJava components. Reducing a non-deterministic expression can lead to the creation of new objects, a well-formed cross-border method call / return using **RI-Call-Boundary / RI-Return-Boundary**) or abrupt program termination.

In order to distinguish contexts of the previous section from most general contexts, we call the previous ones *deterministic* contexts. Contexts then simply subsume both deterministic and most general contexts.

Construction of κ_X. The most general context of X is denoted by κ_X. Let mgc be a package name not occurring in X. For each abstracted type $T^\alpha = \langle \widehat{p.c}, \widehat{q.i}, \widehat{m} \rangle \in \mathcal{T}_X^\alpha$, we construct a class of the form:

---

package mgc; **public class** c **extends** $q.d$ **implements** $\overline{q.i}$ { $\widehat{M}$ }

---

where (1) c is a class name that is unique for each abstracted type T^α, (2) $q.d$ is the smallest class in $\widehat{p.c}$, (3) $\widehat{M}$ are the methods with signature from $\mathrm{methods}_X(\widehat{p.c} \cup \widehat{q.i})$ which do not have names in $\widehat{m}$ and with nde as body. The idea behind this construction is that abstracting the type of the constructed class yields the abstracted type from which the class was constructed (i.e. $\mathrm{typeabs}_{KX}(mgc.c) = T^\alpha$). The context κ_X also has an additional class Main, which is the startup class of $\kappa_X X$:

---

package main; **public class** Main { lang.Object main() { nde } }

Using this definition of most general context, we can give the denotation of a component X as the set of traces generated by the most general context of X. Note that this definition solely depends on X.

Definition 4 (Denotation of a component). *The denotation of a component X is defined as* $\mathrm{Traces}(\kappa_X X)$.

3.3 Full Abstractness

The standard notion of testing or contextual compatibility [12] states that every program context which terminates with the first component must also terminate with the second component.

Definition 5 (Testing compatibility). *A component Y is* testing compatible *with X if Y is source compatible with X and for any deterministic program context K of X:* $\text{init}\zeta_{KX}\downarrow$ *implies* $\text{init}\zeta_{KY}\downarrow$.

The definition of testing compatibility quantifies over all possible program contexts and, as outlined for the challenges in the introduction, cannot be used in general for proving that two components are compatible. We therefore give an alternative definition that is based on the aforementioned denotations of components.

Definition 6 (Behavioral compatibility). *A component Y is* behaviorally compatible *with X if Y is source compatible with X and* $\text{Traces}(\kappa_X X) \subseteq \text{abs}_X(\text{Traces}(\kappa_Y Y))$.

We can not simply state trace inclusion, as Y may have more public types than X (see Def. 1). We must abstract from these additional types in the traces with the function $\text{abs}_X(T^\alpha)$ defined in Fig. 5. Finally we can state our main theorem, namely that the compatibility notions of Def. 5 and Def. 6 coincide.

Theorem 2 (Full abstraction). *Consider two components X and Y. Then Y is behaviorally compatible with X iff Y is testing compatible with X.*

The following lemmas are the main ones needed to prove full abstraction. Each of these are proven using simulation relations (see Section 4). The first two lemmas show that components and contexts compute the next label only based on the trace history, i.e., that the trace contains all relevant information.

Lemma 1 (Component independency). *Consider two contexts K_1 and K_2 for X such that $t \in \text{Traces}(K_1 X)$ and $t \in \text{Traces}(K_2 X)$ and $\text{last}(t) = \mu?$. Then $t \cdot l \in \text{Traces}(K_1 X)$ implies $t \cdot l \in \text{Traces}(K_2 X)$.*

Lemma 2 (Context independency). *Let Y be source compatible with X, K be a context for X and Y, and $t \in \text{Traces}(KX)$ and $t \in \text{abs}_X(\text{Traces}(KY))$ and $t = \bullet$ or $\text{last}(t) = \mu!$. Then, $t \cdot l \in \text{Traces}(KX)$ implies $t \cdot l \in \text{abs}_X(\text{Traces}(KY))$.*

The following two lemmas state that the most general context for a component X simulates exactly all possible contexts for X.

Lemma 3 (Trace abstraction). *Let K be a deterministic program context of component X. Then,* $\text{Traces}(KX) \subseteq \text{Traces}(\kappa_X X)$.

Lemma 4 (Trace concretization). *Let X be a component and $t \in \text{Traces}(\kappa_X X)$. Then, there is a deterministic program context K of X with $t \in \text{Traces}(KX)$.*

4 Simulations

In this section, we show how the main lemmas from the previous section can be proven using simulation relations on the runtime configurations. Before we can relate two configurations, we define in Section 4.1 a few well-formedness properties that the configurations must satisfy. We then define preorder relations over well-formed runtime configurations in Section 4.2. We show in Section 4.3 how these preorder relations are preserved by small operational steps. We also show that they are simulation relations on the large-step semantics in Section 4.4. Finally we describe how the main (trace-based) lemmas from the previous section can be proven using these simulation relations.

4.1 Well-formed Runtime Configurations

Before giving the definition of well-formed runtime configuration, we first define a few helper functions. The function $\text{stackabs}_L(\overline{\mathcal{F}})$ yields all the L-tagged stack frames of $\overline{\mathcal{F}}$ and $\text{fieldrestrict}^L_{KX}(p.c, \overline{r})$ yields all field values of $\overline{r}$ that are defined in classes of L that are superclasses of $p.c$ or $p.c$ itself. The function objectrefs(...) yields all object identifiers contained in a syntactic element. We can then define well-formed runtime configurations.

Definition 7 (Well-formed runtime configuration). *A runtime configuration $\zeta = KX, \mathcal{O}, \overline{\mathcal{F}}$ is well-formed (denoted by $\mathsf{wf}(\zeta)$) if*

- *ζ is well-typed (standard definition, not detailed further here)*
- *Top of stack $\overline{\mathcal{F}}$ is an expression of the form $\mathcal{E}[E]$, the rest are contexts of the form $\mathcal{E}$*
- *Stack frames in $\overline{\mathcal{F}}$ are alternatively from* comp *and from* ctxt *and the lowest stack frame is from* ctxt
- *Store consistency:* $\text{objectrefs}(\text{rng}(\mathcal{O})) \subseteq \text{dom}(\mathcal{O})$
- *Stack consistency and separation:* $\forall L : \text{objectrefs}(\text{stackabs}_L(\overline{\mathcal{F}})) \subseteq \text{visible}(\mathcal{O}, L)$
- *Only L-visible objects can be accessed from L-visible objects:* $\forall j \in \text{visible}(\mathcal{O}, L)$ *with* $\mathcal{O}(j) = (\_, \_, p.c, \overline{r})$ *we have* $\text{objectrefs}(\text{fieldrestrict}^L_{KX}(p.c, \overline{r})) \subseteq \text{visible}(\mathcal{O}, L)$
- *Internal objects of X are of a type of X:* $\forall(\_, \mathsf{comp}, p.c, \_) \in \text{rng}(\mathcal{O}) : p.c \in \mathcal{T}_X$
- *Internal objects of K have their X fields untouched:* $\forall(\mathsf{internal}, \mathsf{ctxt}, p.c, \overline{r}) \in \text{rng}(\mathcal{O}) : \text{fieldrestrict}^{\mathsf{comp}}_{KX}(p.c, \overline{r}) = \overline{\mathsf{null}}$

Initial program states are well-formed and well-formedness is preserved by small-step operational steps.

4.2 Preorder Relations $\preccurlyeq^{\rho}_{L}$

The preorder relations $\preccurlyeq^{\rho}_{\mathsf{comp}}$ and $\preccurlyeq^{\rho}_{\mathsf{ctxt}}$ relate two well-formed runtime configurations if their comp or ctxt part is similar. This allows us for example to relate runtime configurations when configurations only differ in the context or component (see e.g. Lemmas 1 and 2). We give their definition in the following. Note that helper functions are given in Fig. 5.

Definition 8 (Preorder relation $\preccurlyeq_L^\rho$). *Consider two well-formed configurations* $\zeta_1 = K_1X_1, \mathcal{O}_1, \overline{\mathcal{F}_1}$ *and* $\zeta_2 = K_2X_2, \mathcal{O}_2, \overline{\mathcal{F}_2}$ *such that* X_2 *is source compatible with* X_1*. We write* $\zeta_1 \preccurlyeq_L^{\rho_e} \zeta_2$ *if* ρ_e *is a renaming from* $\mathsf{filter}(\mathcal{O}_1, \mathsf{exposed})$ *to* $\mathsf{filter}(\mathcal{O}_2, \mathsf{exposed})$ *and there is a renaming* ρ_i *from* $\mathsf{filter}(\mathcal{O}_1, \mathsf{internal}, L)$ *to* $\mathsf{filter}(\mathcal{O}_2, \mathsf{internal}, L)$ *and* $\rho = \rho_e \cup \rho_i$ *such that*

- *if* $L = \mathsf{ctxt}$ *then* $K_1 = K_2$ *else* $X_1 = X_2$
- $\mathrm{currloc}(\zeta_1) = \mathrm{currloc}(\zeta_2)$
- $\mathrm{stackabs}_L(\overline{\mathcal{F}_1}) \equiv^\rho \mathrm{stackabs}_L(\overline{\mathcal{F}_2})$
- *If* $j_1 \equiv^\rho j_2$ *with* $\mathcal{O}_1(j_1) = (V_1, L_1, p_1.c_1, \overline{r_1})$ *and* $\mathcal{O}_2(j_2) = (V_2, L_2, p_2.c_2, \overline{r_2})$*, then*
  - $V_1 = V_2$
  - $L_1 = L_2$
  - $\mathrm{fieldrestrict}^L_{K_1X_1}(p_1.c_1, \overline{r_1}) \equiv^\rho \mathrm{fieldrestrict}^L_{K_2X_2}(p_2.c_2, \overline{r_2})$
  - *if* $L_1 = L$ *then* $p_1.c_1 = p_2.c_2$ *else* $\mathrm{typeabs}_{K_1X_1}(p_1.c_1) = \mathrm{abs}_{X_1}(\mathrm{typeabs}_{K_2X_2}(p_2.c_2))$

In the following, we explain the definition of $\preccurlyeq_L^\rho$. We first require that there is a renaming from the exposed objects of the first to the exposed objects of the second configuration. We also require that there is a renaming between the (internal) objects that are created by L. We then require that for both configurations the execution is at the same place (either in code of the component or the context). Furthermore, we require the parts of the stack that consist of code from L to be equal under the object renaming. For related objects, the heap entries must also match in the following way. The exposure and location flags must be the same. The values of fields that are defined in L must be equal under the object renaming. At last, the dynamic type of related objects must be equal if they are created by L. Otherwise, they must have the same abstracted types.

The relations $\preccurlyeq_L^\rho$ can be considered as simulation relations on the large-step semantics. We illustrate this in the following. Initial states are in the relation.

Lemma 5 (Initial states are related under $\preccurlyeq_{\mathsf{comp}}$). *Consider two program contexts* K_1 *and* K_2 *such that* K_1X *and* K_2X *are well-formed. Then* $\mathrm{init}\zeta_{K_1X} \preccurlyeq_{\mathsf{comp}} \mathrm{init}\zeta_{K_2X}$.

Lemma 6 (Initial states are related under $\preccurlyeq_{\mathsf{ctxt}}$). *Consider two components* X_1 *and* X_2 *such that* X_2 *is source compatible with* X_1 *and* KX_1 *and* KX_2 *are well-formed. Then* $\mathrm{init}\zeta_{KX_1} \preccurlyeq_{\mathsf{ctxt}} \mathrm{init}\zeta_{KX_2}$.

We first present how the relations $\preccurlyeq_L^\rho$ are preserved by steps of the small-step operational semantics and later extend it to the large-step one.

4.3 Small-step Semantics

We consider four different cases. We distinguish whether the steps are labelled by τ or another label. We also distinguish whether the steps are initiated in the context or the component.

Lemma 7 (τ-steps in $\neg L$ preserve $\preccurlyeq_L$). *Assume that* $\zeta_1 \preccurlyeq_L^\rho \zeta_2$ *and* $\mathrm{currloc}(\zeta_1) = \neg L$*. If* $\zeta_1 \overset{\tau}{\rightsquigarrow} \zeta_1'$ *then* $\zeta_1' \preccurlyeq_L^\rho \zeta_2$*. Similarly, if* $\zeta_2 \overset{\tau}{\rightsquigarrow} \zeta_2'$ *then* $\zeta_1 \preccurlyeq_L^\rho \zeta_2'$.

Lemma 8 ($\preccurlyeq_L$ **simulates** τ**-steps in** L)**.** *If* $\zeta_1 \preccurlyeq_L^\rho \zeta_2$ *and* $\zeta_1 \overset{\tau}{\rightsquigarrow} \zeta_1'$ *and* $\mathrm{currloc}(\zeta_1) = L$*, then* $\zeta_2 \overset{\tau}{\rightsquigarrow} \zeta_2'$ *and* $\zeta_1' \preccurlyeq_L^\rho \zeta_2'$.

In the following lemmas, we use the notion of *consistency* between renamings. Two renamings are consistent if the union of both relations yields a renaming again (i.e., they agree on the common value pairs).

Lemma 9 (Similar messages from $\neg L$ **preserve** $\preccurlyeq_L$)**.** *If* $\zeta_1 \preccurlyeq_L^\rho \zeta_2$ *and* $\zeta_1 \overset{l_1}{\rightsquigarrow} \zeta_1'$ *and* $\mathrm{currloc}(\zeta_1) = \neg L$ *and* $\zeta_2 \overset{l_2}{\rightsquigarrow} \zeta_2'$ *and* $l_1 \equiv^{\rho_l} \mathrm{abs}_{X_1}(l_2) \neq \tau$ *and* ρ_l *minimal and consistent with* ρ*, then* $\zeta_1' \preccurlyeq_L^{\rho \cup \rho_l} \zeta_2'$.

Lemma 10 ($\preccurlyeq_L$ **simulates messages from** L)**.** *If* $\zeta_1 \preccurlyeq_L^\rho \zeta_2$ *and* $\zeta_1 \overset{l_1}{\rightsquigarrow} \zeta_1'$ *and* $\mathrm{currloc}(\zeta_1) = L$ *and* $l_1 \neq \tau$*, then* $\zeta_2 \overset{l_2}{\rightsquigarrow} \zeta_2'$ *and* $l_1 \equiv^{\rho_l} \mathrm{abs}_{X_1}(l_2)$ *and* ρ_l *minimal and consistent with* ρ *and* $\zeta_1' \preccurlyeq_L^{\rho \cup \rho_l} \zeta_2'$.

4.4 Large-step Semantics

The four lemmas of the previous subsection can be extended to large steps and then to many large steps (i.e. program runs). For single large steps, we only state the lemma where a step is simulated.

Lemma 11 ($\preccurlyeq_L$ **simulates large step from** L)**.** *If* $\zeta_1 \overset{l_1}{\Longrightarrow} \zeta_1'$ *and* $\mathrm{currloc}(\zeta_1) = L$ *and* $\zeta_1 \preccurlyeq_L^\rho \zeta_2$*, then* $\zeta_2 \overset{l_2}{\Longrightarrow} \zeta_2'$ *and* $l_1 \equiv^{\rho_l} \mathrm{abs}_{X_1}(l_2)$ *and* ρ_l *minimal and consistent with* ρ *and* $\zeta_1' \preccurlyeq_L^{\rho \cup \rho_l} \zeta_2'$.

We then relate many large steps. For deterministic contexts, we can state that if we have a run starting from a state and another run starting from a related state which emits a similar trace as the first run, then the end states are related. We generalize this in the following lemma, where we also consider non-deterministic (i.e. most general) contexts.

Lemma 12 (Multiple large steps preserve $\preccurlyeq_L$)**.** *If* $\zeta_1 \overset{t_1}{\Longrightarrow} \zeta_1'$ *and* $\zeta_2 \overset{t_2}{\Longrightarrow} \zeta_2'$ *and* $\zeta_1 \preccurlyeq_L^\rho \zeta_2$ *and* $t_1 \equiv^{\rho_t^{12}} \mathrm{abs}_{X_1}(t_2)$ *and* ρ_t^{12} *minimal and consistent with* ρ*, then* $\exists \zeta_3'$ *such that* $\zeta_2 \overset{t_3}{\Longrightarrow} \zeta_3'$ *and* $t_1 \equiv^{\rho_t^{13}} \mathrm{abs}_{X_1}(t_3)$ *and* ρ_t^{13} *minimal and consistent with* ρ *and* $\zeta_1' \preccurlyeq_L^{\rho \cup \rho_t^{13}} \zeta_3'$.

The states ζ_1' and ζ_2' might not be related, as during the runs $\zeta_1 \overset{t_1}{\Longrightarrow} \zeta_1'$ and $\zeta_2 \overset{t_2}{\Longrightarrow} \zeta_2'$, the same most general context might have chosen different executions as it is non-deterministic. For example, it may create more objects that are internal to it in one execution than in another, but still generate a similar trace. For deterministic contexts, however, $\zeta_1' \preccurlyeq_L \zeta_2'$.

We can finally prove Lemmas 1 and 2. We know that initial states are related and that after the traces are executed, the states thereafter are still related (by Lemma 12). By

Lemma 11, we then know that the second configuration can respond in a similar way to the first one.

To prove Lemmas 3 and 4, however, we need stronger relations that not only relate the comp part of the configurations, but also relate the most general context to the deterministic context (which we denote by $\lll^{\rho}$ for trace abstraction and $\ggg^{\rho}$ for trace concretization). The proof then works in a similar way, where similar lemmas as before have to be proven for the relations $\preccurlyeq^{\rho}_{\mathsf{comp}} \cap \lll^{\rho}$ and $\preccurlyeq^{\rho}_{\mathsf{comp}} \cap \ggg^{\rho}$.

5 Proving Compatibility

In this section, we give proof obligations that are needed in order to prove two components compatible. We also describe why the proof method is complete and sketch how the direct connection of the trace semantics to the operational semantics can be exploited to prove compatibility.

In order to prove that a component Y is behaviorally compatible with X, the following steps are necessary. First, Y must be proven source compatible with X. This can be directly done by the checks detailed in Section 2. The more difficult part is to prove, as per Def. 6, that $\mathrm{Traces}(\kappa_X X) \subseteq \mathrm{abs}_X(\mathrm{Traces}(\kappa_Y Y))$. It is sufficient to prove that $\mathrm{Traces}(\kappa_X X) \subseteq \mathrm{abs}_X(\mathrm{Traces}(\kappa_X Y))$, which follows from the property that $\mathrm{Traces}(\kappa_X Y) \subseteq \mathrm{Traces}(\kappa_Y Y)$.

The proof is done by induction on the length of the traces. The empty trace is trivially in both sets. As induction step, assume $t \cdot l$ such that (1) $t \in \mathrm{Traces}(\kappa_X X)$, (2) $t \in \mathrm{abs}_X(\mathrm{Traces}(\kappa_X Y))$, and (3) $t \cdot l \in \mathrm{Traces}(\kappa_X X)$. The proof goal is then to show that $t \cdot l \in \mathrm{abs}_X(\mathrm{Traces}(\kappa_X Y))$. We distinguish two cases, based on the form of the last label in the trace t:

Case $t = \bullet$ or $\mathrm{last}(t) = \mu!$: The claim follows directly by Lemma 2.

Case $\mathrm{last}(t) = \mu?$: The initial configurations of $\kappa_X X$ and $\kappa_X Y$ are related by $\preccurlyeq_{\mathsf{ctxt}}$ due to Lemma 6. By Lemma 12, the configurations right after the trace t are related by $\preccurlyeq_{\mathsf{ctxt}}$ as well. It then suffices to prove that the second configuration can run and generate a next label whenever the first configuration runs and generates this label. There are different approaches for proving this. One possibility is to capture how the comp part of both configurations are related. This relation, usually called *coupling invariant* [3], only needs to relate the comp parts of the configuration. Note that these parts remain untouched by the context during its steps. Another possibility is to relate the runtime configurations to the traces, i.e., for each component X establish a relation between the traces of $\mathrm{Traces}(\kappa_X X)$ and runtime configurations of $\kappa_X X$ (which we call a *canonical representation invariant*). We consider the description of more detailed approaches as future work.

Completeness of the Proof Method. Although we do not consider in this paper how a coupling invariant can be specified, we can show that such an invariant always exists if two components are behaviorally compatible. This completeness result comes basically for free from our full abstraction proof approach. If two components are behaviorally compatible, then there always exists a coupling invariant (simulation relation) $\mathcal{R}$ which can roughly be constructed as follows:

– $(\text{init}\zeta_{\kappa_X X}, \text{init}\zeta_{\kappa_X Y}) \in \mathcal{R}$.
– If $\text{init}\zeta_{\kappa_X X} \overset{t_1}{\Longrightarrow} \zeta_1$ and $\text{init}\zeta_{\kappa_X Y} \overset{t_2}{\Longrightarrow} \zeta_2$ and $t_1 \equiv t_2$, then $(\zeta_1, \zeta_2) \in \mathcal{R}$.

We know already that for each $(\zeta_1, \zeta_2) \in \mathcal{R} : \zeta_1 \preccurlyeq^{\rho}_{\mathsf{ctxt}} \zeta_2$. We also know that the ctxt part is left untouched when the component executes (and the execution is independent of the context). Thus, a user-specified invariant only needs to talk about the comp part of the configuration. If the invariant only talks about the comp part, then we also have the guarantee that it remains untouched by the context, which allows us to disregard steps in the (most general) context.

Weaker Compatibilites. Sometimes we are not interested in preserving the full behavior of components in an evolution step. For example, we might be interested in checking that the new version of a component has the same behavior as the old one with respect to a subset of its interface methods. In this case, we can compare only those traces including these methods. This is a typical evolution scenario. Similarly, the approach can be adapted to prove that components behave the same in a restricted set of contexts. Weaker compatibilities can be considered for example by (1) providing a more restrictive definition of the most general context (e.g. disallow the context to call a certain method) (2) giving an abstraction function on the traces or (3) specifying method contracts that must be satisfied by contexts.

6 Conclusion and Future Work

We have presented a fully abstract trace-based semantics for packages of an object-oriented class-based language. We have also the shown the relation of the trace-based to the operational semantics and provided a proof outline of the full abstraction result where simulation relations are used. We see this as foundational work for relating trace-based specifications to components (i.e. sets of classes), proving sets of classes compatible or equivalent and proving refactoring transformations behavior preserving.

A particular reason for why the trace-based approach fits so well is that object-oriented programming is based on message passing. We are not sure whether deriving fully abstract semantics using the mixed trace-based/operational semantics approach fits well for other non-OO settings, but plan on exploring this in the future. Furthermore, we would like to work out a concrete proof technique which consists of providing a specification language for describing coupling relations, extending existing program logics for OO programs [1, 14] to our setting and automatically generating proof obligations to be used by interactive or automatic theorem provers.

Acknowledgements. We thank Christoph Feller and Jean-Marie Gaillourdet for commentating on earlier drafts of this paper. We also thank the anonymous reviewers of SBMF 2011 for their valuable feedback.

References

1. Abadi, M., Leino, K. R. M.: A Logic of Object-Oriented Programs. In: Dershowitz, N. (ed.) Verification: Theory and Practice. LNCS, vol. 2772, pp. 11–41. Springer, Heidelberg (2003)
2. Ábrahám, E., Bonsangue, M. M., Boer, F. S. de, Steffen, M.: Object Connectivity and Full Abstraction for a Concurrent Calculus of Classes. In: Liu, Z., Araki, K. (eds.) ICTAC 2004. LNCS, vol. 3407, pp. 37–51. Springer, Heidelberg (2004)
3. Back, R.-J. J., Akademi, A., Wright, J. V.: Refinement Calculus: A Systematic Introduction. Springer, Heidelberg (1998)
4. Banerjee, A., Naumann, D. A.: Ownership confinement ensures representation independence for object-oriented programs. In: Journal of the ACM 52.6, pp. 894–960. (2005)
5. Gosling, J., Joy, B., Steele, G., Bracha, G.: The Java Language Specification, Third Edition. The Java Series. Addison-Wesley, Boston, Mass. (2005)
6. Grothoff, C., Palsberg, J., Vitek, J.: Encapsulating Objects with Confined Types. In: OOPSLA, pp. 241–253 (2001)
7. Hennessy, M., Milner, R.: On Observing Nondeterminism and Concurrency. In: ICALP, pp. 299–309 (1980)
8. Jeffrey, A., Rathke, J.: Java Jr.: Fully Abstract Trace Semantics for a Core Java Language. In: ESOP, pp. 423–438 (2005)
9. Koutavas, V., Wand, M.: Bisimulations for Untyped Imperative Objects. In: Sestoft, P. (ed.) ESOP 2006. LNCS, vol. 3924, pp. 146–161. Springer, Heidelberg (2006)
10. Koutavas, V., Wand, M.: Reasoning About Class Behavior. In: Informal Workshop Record of FOOL 2007 (Jan. 2007)
11. Milner, R.: Fully Abstract Models of Typed Lambda-Calculi. In: Theor. Comput. Sci. 4.1, pp. 1–22. (1977)
12. Morris, J. H.: Lambda-Calculus Models of Programming Languages. Tech. rep. 57. MIT Laboratory for Computer Science (1968)
13. Plotkin, G. D.: LCF Considered as a Programming Language. In: Theor. Comput. Sci. 5.3, pp. 223–255. (1977)
14. Poetzsch-Heffter, A., Müller, P.: A Programming Logic for Sequential Java. In: Swierstra, S. D. (ed.) ESOP 1999. LNCS, vol. 1576, pp. 162–176. Springer, Heidelberg (1999)
15. Silva, L., Naumann, D. A., Sampaio, A.: Refactoring and Representation Independence for Class Hierarchies: Extended Abstract. In: Proceedings of the 12th Workshop on Formal Techniques for Java-Like Programs. FTFJP 2010, 8:1–8:7. ACM, New York, NY, USA (2010)
16. Sumii, E., Pierce, B. C.: A Bisimulation for Dynamic Sealing. In: Theoretical Computer Science 375. (2007)
17. Sumii, E., Pierce, B. C.: A Bisimulation for Type Abstraction and Recursion. In: Journal of the ACM 54. (2007)
18. Welsch, Y., Poetzsch-Heffter, A.: Full Abstraction at Package Boundaries of Object-Oriented Languages. Tech. rep. 384/11. Available at `http://softech.cs.uni-kl.de/Homepage/PublikationsDetail?id=157`. University of Kaiserslautern (May 2011)
19. Wright, A. K., Felleisen, M.: A Syntactic Approach to Type Soundness. In: Inf. Comput. 115.1, pp. 38–94. (1994)