Grey-Box Specification
and
Runtime Testing
of
Object-Oriented Program Components

Yannick Welsch

May 5, 2008

Software Technology Group
Department of Computer Science
TU Kaiserslautern

Supervisors:
Prof. Dr. Arnd Poetzsch-Heffter
Dipl.-Inf. Jean-Marie Gaillourdet
Ich versichere hiermit, dass ich die vorliegende Masterarbeit mit dem Thema

Grey-Box Specification and Runtime Testing of Object-Oriented Program Components

selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Die Stellen, die anderen Werken dem Wortlaut oder dem Sinn nach entnommen wurden, habe ich durch die Angabe der Quelle, auch der benutzten Sekundärliteratur, als Entlehnung kenntlich gemacht.

______________________________  _________________________
(Ort, Datum)                  Yannick Welsch
Acknowledgements

I would like to thank Prof. Dr. Arnd Poetzsch-Heffter for making it possible to write this thesis. Many thanks go to Dipl.-Inf. Jean-Marie Gaillourd for answering all my questions, for all the insightful conversations during the development of the ideas in this thesis and for his valuable guidance. I would also like to thank all the other members of the Software Technology Group for making me feel at home and giving me the opportunity to share so many interesting discussions. The most special thanks go to my family and friends for their constant support.
Abstract

Formal specification and verification techniques for object-oriented programs have evolved tremendously in the last ten years. However a big short-coming of existing approaches is that most object-oriented languages have no clear notion of component, or if they have, this notion doesn’t fit very well with existing programming practices. Hence, there also is a lack for specification techniques at the program component level. In this thesis, our components are the ones defined by Poetzsch-Heffter, Gaillourdet and Schäfer. They define their program components as dynamic entities which consist of a varying set of objects, and their boundary to the environment. Component behaviours are defined as partial functions from incoming to outgoing messages. Starting from this notion of component behaviour, we build our specification technique.

We define a formal framework for integrating specification languages for components, and give one concrete instance of such a specification language, which is a mixture of a declarative and operational one. We then map the formal concepts to the Java programming language, describe the resulting issues, our concrete implementation and the underlying design decisions. Finally we develop a framework for checking the specifications at runtime.

Zusammenfassung


Wir definieren ein formales Rahmenwerk um Spezifikationssprachen für Komponenten zu integrieren und geben dann eine konkrete Instanz einer solchen Spezifikationssprache an, die teils deklarativ und teils operational ist. In einem nächsten Schritt werden dann die formalen Konzepte auf die Sprache Java übertragen und die damit verbundenen Probleme beschrieben. Abschließend wird eine Umgebung entwickelt, um die Spezifikationen zur Laufzeit zu überprüfen.
Grey-Box Specification
and
Runtime Testing
of
Object-Oriented Program Components

Yannick Welsch
y_welsch@cs.uni-kl.de

May 5, 2008
Contents

1 Introduction 5
  1.1 Components ......................................................... 5
    1.1.1 Scope ......................................................... 6
  1.2 Component Specifications ........................................... 6
    1.2.1 Scope ......................................................... 7
  1.3 Runtime Testing .................................................... 7
  1.4 Related Work ...................................................... 8
  1.5 Goals and Overview ............................................... 8

2 OOBx, an Object-Oriented Language with Program Components 11
  2.1 Introduction ....................................................... 11
  2.2 Syntax and Static Semantics ....................................... 11
  2.3 Informal Semantics ................................................ 12
  2.4 Helper Functions .................................................. 12
  2.5 Standard Semantics ............................................... 15
    2.5.1 Reduction Rules .............................................. 15
    2.5.2 Initial Configuration ....................................... 16
  2.6 Message Semantics ................................................ 16
    2.6.1 Small-Step Message Semantics ............................... 19
    2.6.2 Mix-Step Message Semantics ............................... 19
  2.7 Equivalence of Standard and Message Semantics ................. 22
  2.8 Conclusion ....................................................... 23

3 JBox, Program Components for Java 25
  3.1 Requirements and Goals .......................................... 25
  3.2 The Language ..................................................... 26
  3.3 Runtime ............................................................ 26
  3.4 Conceptual Issues ................................................ 29
    3.4.1 Static Members .............................................. 29
    3.4.2 Multiple Subtyping .......................................... 30
Introduction

Components are widely used as an aid to describe software systems. They are the basic building blocks which structure a software system in its constituents. The concept of components is dependent on the current view on the system, among which we can distinguish the logical view, code structure view, deployment view, runtime view and other views following yet different criteria. Our scope here is restricted to the behavioural specification of program components, meaning components which actually exist as executable code in a program at runtime.

Ideally, software systems should be assembled from libraries of existing or sometimes newly written components, just as in other engineering disciplines e.g. hardware. In reality, most software systems are written from scratch where components are not explicitly found in the actual code, making software production more a craft than an engineering discipline. This is partly due to the lack of concepts in programming languages to define and integrate components.

Another important aspect which comes with component software and reuse is the specification of these program components. Here again, most components are not specified at all or only the most important entry points are specified informally at best. What we want are simple, yet powerful formal specifications, because, in the end, we want to be able to check conformance between a component specification and implementation and also do client reasoning. This allows the packaging of a component in binary form, after checking for conformance to the specification, and distributing it to a user which can then use the component specification to check if he uses the component in the right way, the specification basically being the contract between the component vendor and user.

In the next sections, we will explain the notions of component, component specification and runtime testing, narrow down the scope and give a short overview of this master thesis.

1.1 Components

We will explain the concept of a component: They are basically program parts which can be reused by other (larger) program parts. Thus composition and reusability are key
1 Introduction

concepts of component software. Components can vary largely in size, from few lines of code to tens of thousands of lines, making it even more difficult to find a good mapping from concept to language abstraction. To make safe reuse of a component possible, components need to have interfaces for provided and required services through which interactions with other components can occur.

We will look at components in an object-oriented setting. In an initial step, we have to map the abstract concept of components to a concrete concept in the object-oriented programming language. In functional programming languages like SML, one established language abstraction for components are functors. The notion of component is mapped onto a functor or structure, interfaces correspond to signatures, required components are functor parameters and composition corresponds to functor application. We will not discuss the shortcomings of this approach, as we only want the reader to get a feel of how a component may be mapped to an actual language abstraction.

Current object-oriented programming languages are better at expressing small components than expressing larger ones. In these settings, the notion of component is often associated with a class. Other approaches tie the notion of component to modules or packages, libraries, even whole frameworks, processes or web services, if they exist under a certain form in the language.

Although a lot of effort and progress has already been done in the field of program verification and checking for object-oriented systems, none of the existing specification techniques have a well defined component notion which fit in our opinion a natural notion of component in an object-oriented system, or these components are not actual program components. In other cases, these kind of component types can be found to be far too restricting for practical use.

1.1.1 Scope

The components we consider in this thesis are runtime components, which represent a part of the runtime state of the program, and are described more thoroughly by Arnd Poetzsch-Heffter, Jean-Marie Gaillourdet and Jan Schäfer in [20]. They define their program components as dynamic entities which, in the context of object-oriented programming, consist of a varying set of objects, and their boundary to the environment. They also describe the behaviour of components as the possible sequence of messages which pass the component boundaries. We build our concept of component specification on this message semantics, and reuse the theoretic results in [20] to derive our own results.

1.2 Component Specifications

Component specifications usually provide a good abstraction of the internal behaviour of a component. They can serve as contract between component vendor and user, or exist for the simple purpose of documentation. For checking specification conformance, we need formal specifications. Our scope is restricted to the specification of the observable component behaviour limited to functional aspects. There are basically two different styles of formal specifications. In the algebraic or property-oriented style, operations are described solely in terms of each other by relating their input and output values. Model-based specifications are closer to implementations, as they operate on a state.
1.3 Runtime Testing

As they are closer to high-level implementations, they are often easier to understand for most people. While algebraic specifications allow for a concise specification of small examples (e.g. stack specification with axioms like \( \text{top}(\text{push}(s, x)) = x \)), the model-based approach scales better for large systems.

When specifying services, it is often easier to provide an operational description instead of a purely descriptive one. When talking about behaviour of a component, we can distinguish between black- and white-box view on a component. For a good description of why black-box specifications are insufficient and white-box specifications too much, we direct the reader to [5]. So if neither black- nor white-box approaches provide us with the right abstraction level, we have to find a middle ground. Grey-box specifications are a trade-off between black- and white-box specifications, where the former do not provide enough information, e.g. the external operation by the component (required interfaces), and the latter burden the component deployer with too much information.

1.2.1 Scope

The specification technique developed in this master thesis is a combination of a declarative and operational one for grey-box specifications, which can encode call-backs without complicated encodings and where some constructs for nondeterminism are introduced for higher abstraction. In the end, we want to be able to check conformance between specifications and implementations and also do client reasoning. We develop a framework for specification languages based on the aforementioned message semantics for components. These specification languages cover the following properties [6]:

- state transformation
- parameter values and return value
- external calls

We also give a concrete instance of such a specification language. An important factor for the specification language is not to lay too much restriction on the implementation space. Thus non-deterministic constructs and abstract states are introduced. With our technique we can describe the internal (abstract) state transitions of the component and also callbacks out of the component (effects on the environment of the component).

1.3 Runtime Testing

As explained earlier, formal specifications provide a nice foundation for conformance checking. Runtime testing and checking allows to validate that a system works accordingly to its specifications in certain scenarios. Whereas it does not make such strong guarantees as can be achieved by verification, it can reveal presence of defects.

Using our component specifications, we want to do conformance checking for the component implementations they are associated with. Our focus in this thesis will however be on the specification part, whereas we always keep checkability in mind, which will be reflected in some design decisions.
1.4 Related Work

Among the related work is JML [15], the Java Modeling Language, which can be used to specify the behaviour of Java modules. The JML specification technique is a mix between the LARCH [13] approach and Design by Contract [17] as found in Eiffel. It basically allows the behavioural specification of methods and/or abstract data types. The introduction of the JML reference manual [15] first talks about specification of program modules, but soon it becomes clear to the reader that the term module means class or interface for that particular document. Throughout the whole document, the word component is never used. If the notion of program component is fixed as to a class instance, JML allows to specify program components. A lot of useful extensions to JML have been proposed during the last decade, but the whole concept somehow lacks formal foundations, as most of these extensions are only described informally or, if formally, for some subpart of the Java language. The technique described in this paper has a strong formal basis, which serves as a useful basis for extensions. Whereas earlier versions of JML only allowed for a descriptive nature of program behaviour, recent support [21] for model programs was added to JML, but maintaining a too strong mapping to the code and hence giving not enough flexibility to the programmer in our opinion. Although a way of ownership, through the Universe type system, has found its way into JML and which structures code, e.g. form components, no facility is provided to specify the behaviour of these components. Both concepts seem to be somewhat unrelated. The Spec♯ language [2], with similar goals to the JML approach, has the concept of components, but under a restricted form. Our component notion allows for far more complex object graphs. The reason why JML and Spec♯ do not have more complex component notions is partly due to them being very strongly oriented towards formal verification purposes. Some of the work done for JML and Spec♯ would also fit very well into our specification technique, and an integration of some of our ideas into their work might lead to an interesting result.

The concept of Trace-Assertions in Jass [9, 19] is also similar to our trace-based approach. The main difference is that we specify the partial traces for components which can be large groups of objects, whereas their approach describes traces on the class instance level. We also tried to provide a more natural syntax for our specification language.

Another way to specify programs is through higher level specification techniques, e.g. Abstract State Machines. These specifications can be executed (see ASML and Spec Explorer [7], a model-based testing tool). The goal we want to achieve is slightly different as we don’t want to completely specify a program, but put restrictions on valid executions. Our specifications are also syntactically closer to the actual code. To check if a program satisfies an ASM specification, they must be somehow put in relation to each other. This is what we partly get for free.

1.5 Goals and Overview

Starting from the presented notion of component, we want to provide a formal framework for describing and checking grey-box specifications for object-oriented program components. We will then present one particular instance of such a specification lan-

8
guage. The theoretical concepts will be mapped onto the programming language Java. A checking framework is also to be developed to check conformance of implementations to specifications at runtime.

In the rest of this section we give an outline to the main part of the thesis, all of which is graphically represented in Fig. 1.1. We start in Chapter 2 by giving an introduction to OOBox (Object-Oriented Language with Boxes), the object-oriented language with components which has been presented in [20] with its various semantics. From there on, we present our own work. In Chapter 3 we describe the implementation of the given language in Java, which we will call JBox. In Chapter 4 we formally describe a generic extension to the OOBox language to embed concrete specification languages, called OOBoxSpec, and give the corresponding (informal) embedding in Java in Chapter 5. In Chapter 6 we give a formal description of a concrete embedding called OOCS (Object-Oriented Component Specification Language) and the corresponding embedding for Java in Chapter 7, called JCS (Java Component Specification Language). Each of the afore mentioned chapters also contains a short evaluation, a discussion of further concepts, design decisions and issues.

In Chapter 8 we briefly describe what we have achieved so far, conclude about the advantages and drawbacks of our methodology, recapitulate what makes the approach so unique and interesting, and motivate lines of future work.
Figure 1.1: Overview of the document structure and relationship of the different chapters, OOBox being the formalism we base all the rest on.
In this chapter we characterise the object-oriented program components which we will later use as a basis for our specification language. While we encourage the reader to first have a look at [20], which contains all of the presented material in detail, we try to give an as complete overview as possible of the material. We only treat the aspects which are important for elaboration in the next chapters.

2.1 Introduction

We present the syntax and semantics of a standard sequential object-oriented language enhanced with boxes, the light-weight program components. The syntax is very common and similar to that found in other formal descriptions (see Classic Java, Featherweight Java etc.). A few constructs for managing the boxes are introduced. Then semantics for the language are given.

In a first step a standard semantics with boxes is presented for the language we will call OOBox (Object-Oriented Language with Boxes). Then a message semantics is developed which can be seen as describing the message behaviour at the boundary of a component. As the different semantics are given, they are proven to be equivalent according to [20]. The concept of box denotations are then defined in terms of the message semantics. To show the equivalence of the denotation-based message semantics and the standard semantics, ”intermediate” semantics are introduced to ease the proofs, namely the small-step, the summarised small-step and the mix-step semantics.

We will shortly introduce the different semantics, and repeat the proof results, but will not repeat them in detail. In the next chapters we will reuse these semantics, in order to construct our results for specifications.

2.2 Syntax and Static Semantics

What follows are the syntax and static semantics of a standard object-oriented language with classes, interfaces, aliasing, inheritance, single sub-typing (only one super-type) and dynamic dispatch. The language is enhanced with component (box) declarations...
and an expression to create new instances of the components (boxes). A box $B$ with interface type $I$ is declared as "box $B : I". To create an (runtime) instance of a box, we first have to associate a given box implementation to a box declaration. This is done using the "box $B$ with $C$" syntax, where $C$ denotes the implementation class compatible with $B$. A box has exactly one interface and one implementation associated to it. The complete abstract syntax of OOBox is given in Fig. 2.1. The notation used is similar to that of Featherweight Java [14], which we refrain from repeating. As a short recapitulation, what distinguishes the OOBox syntax from standard object-oriented languages are the box declaration, the implementation directive and the box creation expression.

A box declaration consists of an interface type which is the so-called entry point to access the box. The box signature, which has an impact on the boundary of a box, can be derived from this interface type as described in section 2.4. A box implementation maps an actual implementation class (the so-called owner class) to the interface, which it must be a subtype of. The box implementations are the actual program components. The contextual and typing constraints are not repeated here, as they can be found in [20]. The noticeable differences from a standard object-oriented language are the following ones:

- Single sub-typing: Each class is only subtype of a single interface or class.
- $Object$, root of the class hierarchy, is an interface with no methods.
- No overloading of methods
- Overriding methods must have the same signature as overridden methods (e.g. no covariant return types)

The formulated restrictions are only to simplify the given presentation. In our implementation of the concepts for Java, we handle the more common cases of course.

2.3 Informal Semantics

We shortly present an informal description of the semantics for OOBox. A box is basically a runtime entity which aggregates a varying set of objects. However, each object can only belong exactly to one box over its lifetime. Boxes are structured into a tree-like hierarchy, where the topmost box is called $globox$ (the global box). Box signatures define the type boundary for boxes, describing the types which are known at the boundary. The idea here is to encapsulate implementation types of a box and decouple internal and external types, allowing a box implementation to be changed without changing the visible behaviour.

2.4 Helper Functions

In this section we describe the different helper functions we need to describe the reduction rules concisely. Part of these helper functions are defined in-place where they are used, whereas we present the most common and important ones here. Sometimes we
\[ P ::= T \quad \text{program} \]
\[ L ::= \text{class } C \text{ (extends } C' \mid \text{impls } I) \{ T \mid \overline{M} \} \quad \text{class} \]
\[ M ::= \text{interface } I \text{ extends } I' \{ M_{SIG} \} \quad \text{interface} \]
\[ T ::= \text{box } B : I \quad \text{box declaration} \]
\[ M ::= \text{box } B \text{ with } C \quad \text{box implementation} \]
\[ M_{SIG} ::= T \ m(T \overline{x}) \quad \text{method signature} \]
\[ e ::= x \quad \text{variable} \]
\[ e ::= \text{null} \quad \text{null constant} \]
\[ e ::= (T) \quad \text{cast} \]
\[ e ::= \text{new } C \quad \text{object creation} \]
\[ e ::= \text{new } B \quad \text{box creation} \]
\[ e ::= e.f \quad \text{field access} \]
\[ e ::= e.f = e' \quad \text{field update} \]
\[ e ::= \text{let } x = e \text{ in } e' \quad \text{variable binding} \]
\[ e ::= e.m(\overline{r}) \quad \text{method call} \]
\[ e ::= \text{if}(e == e') e'' \text{ else } e''' \quad \text{conditional expression} \]

\( C, D \in \text{class names} \)
\( I \in \text{interface names} \)
\( B \in \text{box names} \)

Figure 2.1: Abstract syntax of OOBx
apply functions defined for a certain type to a list of elements of this type, by which
we mean that we apply the function to each element of the list and concatenate the
resulting values.

As seen earlier, box signatures are an important concept in the given presentation to
decouple internal and external types. They are defined as follows:

**Definition 2.1.** The box signature \( B_{\text{sig}} \) of a box \( B \) with interface \( I \) is the smallest set
of types satisfying the following criteria:

- \( I \in B_{\text{SIG}} \)
- If \( T \in B_{\text{SIG}} \), then all types appearing in method signatures of \( T \) are in \( B_{\text{SIG}} \)
- If \( T \in B_{\text{SIG}} \), all the super-types of \( T \) are in \( B_{\text{SIG}} \)

Another useful function is to determine the sequence of boxes whose signature has to
be crossed when following the box tree from one box to another one. One thing to note
is that this path has in general the form of an arc, first going up then down. The box
tree is determined by the parent relation \( P \). For \( P(k) = b \), we also write \( k \prec_P b \). The transitive closure of \( \prec \) is written \( \prec^+ \) and the transitive, reflexive closure \( \prec^* \). We can
then define the function \( \text{crossedBoxes} \) which yields this particular sequence of boxes.

**Definition 2.2.**

\[
\text{crossedBoxes}_P(b, b') = \text{crossedBoxesOut}_P(b, b') \cdot \text{crossedBoxesIn}_P(b, b')
\]

where

\[
\text{crossedBoxesOut}_P(b, b') = \begin{cases} 
\bullet & \text{if } b' \prec^*_P b \\
\mathbf{b} \cdot \text{crossedBoxesOut}_P(P(b), b') & \text{else}
\end{cases}
\]

\[
\text{crossedBoxesIn}_P(b, b') = \begin{cases} 
\bullet & \text{if } b = b' \\
\text{crossedBoxesIn}_P(P(b), P(b')) \cdot b & \text{if } b' \prec^*_P b \\
\text{crossedBoxesIn}_P(P(b), b') & \text{else}
\end{cases}
\]

\( \text{crossedBoxesOut} \) represents the part of the sequence where we go upwards in the
tree, whereas \( \text{crossedBoxesIn} \) represents the part where we go downwards. This distinc-
tion is of some importance later and was not made in the original paper. Note that
we use the \( \cdot \) symbol as list concatenation as well as list element prepending symbol.

Another important function is the type restriction function \( \sigma \) which restricts a type
\( T \) to a given box signature \( B_{\text{sig}} \), yielding the most specific super-type of \( T \) in \( B_{\text{SIG}} \). The function \( \sigma^+ \) does the same for a sequence of box signatures, restricting the type step
by step:

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

To describe the reduction rules of the standard and message semantics, evaluation
contexts \( e \) are used which describe expressions with a hole \( e \) somewhere inside that
expression. We write \( e[f|e'] \) to replace the hole in \( e \) by \( e' \). A hole can only appear at
certain positions in an expression:

14
2.5 Standard Semantics

In this section, we present the standard semantics for OOBox. The semantics is quite similar to that found in other formal descriptions (see Classic Java, Featherweight Java etc.), whereas only the relevant concepts to the component model are kept. A reductional small-step style was chosen to represent the semantics, where a configuration is represented by the states of the created objects, the currently executing box and a partially evaluated expression (expression with a hole) over dynamic values.

The definitions for the configurations are shortly described in Fig. 2.2. Objects and boxes are represented by globally unique identifiers. The state is represented by the tuple \( \langle C, O, N, P, R \rangle \), where \( C \) is the execution stack of pending methods consisting of the expression with a hole to be reduced and the box it will be reduced in, \( O \) is the object store mapping object identifiers to states (the field values), \( N \) maps objects to the boxes they belong to, \( P \) is the parent relation on boxes and \( R \) yields the box signature for a given box.

2.5.1 Reduction Rules

Our reduction rules have judgements of the form \( S, b, e \vdash R S', b', e' \), where \( e \) is the expression in state \( S \) and box \( b \) which is reduced in one step to \( S', b', e' \). We give the reduction rules in Fig. 2.3 and shortly explain the semantics informally. Object creation is usual with the addition that it is recorded in the current box. To generate fresh object identifiers, we define the \textit{next} function which takes a set of object identifiers and returns the next object identifier available. There also exists a corresponding \textit{next}
function for box identifiers. Box creation creates a new object of the box implementation type, assigns this object to the box, and records the box signature and modified parent relation. The function implClass yields the implementation class associated to a box and intf the corresponding box interface. bsig(I) yields the box signature for an interface I as derived by Def. 2.1.

Field accesses are restricted as not to cross box boundaries. The same goes for field updates. The function fields yields the fields defined in a given class.

Method invocations are described by the CALL and RETURN rules. The CALL rule sets the new box (the box of the callee object), records the old box on the execution stack and adapts the expression, whereas the RETURN rule resets the environment and incorporates the newly calculated value into the saved expression. The partial function mbody yields the method body for a given method definition.

The cast rule is very elemental, as it restricts casts only to super-types of types which are in all the boundaries of the boxes crossed between the current box and the box the object upon which is casted belongs to (in reverse order).

Other rules are intentionally left out, as they give no new insight to the presentation.

2.5.2 Initial Configuration

For a program P, let $e_{\text{main}}$ be the main method body, $C$ some class in $P$, $O_0$ the initial object store capturing the input objects, $N_0$ mapping all these input objects to the global box globox (root of the box hierarchy), $j_0$ be of type Main and $j_1 \in O_0$ be of type $C$. The initial configuration is then defined as:

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

The standard semantics showed how boxes can be managed in a setting with a global state. Message semantics, which we will present now, partition this global state into box-local states, allowing box-local reasoning.

2.6 Message Semantics

References are not global anymore, each object reference’s value depends on the box it is referenced from. Thus references have to be adapted as messages cross box boundaries. Message semantics focuses on the interaction of different boxes, describing the adaptation of messages as they cross box boundaries. Method calls are partitioned into messages, depending on the number of boundaries they pass. Different type of messages are distinguished, inter- and intra-box messages, call and return messages, and for inter-box messages, their direction (in or out). We shortly present the definitions for the configurations of the messages semantics found in Fig. 2.4.

We have two different type of references, intra- and inter box references. This allows to reference the same object in different boxes by different relative references. Intra-box references are either local references (own), or pointing to an object in the environment of the box (env), or pointing to an object in an inner box (k). Inter-box references can be either in or out at a boundary, meaning that they point either to an object in the box (or an inner box), or to an object outside the box. As messages arrive at a box boundary, they are translated from inter- to intra-box messages and vice-versa.
2.6 Message Semantics

R-NEW-OBJ

\[
\begin{align*}
& j = \text{next}(\text{dom } O) \quad o = j : C \quad \text{fields}(C) = T \quad \mid \text{null} \mid c \mid \\
& (C, O, N, P, R), b, e \sqcap [\text{new } C] \overset{\text{R}}{\longrightarrow} (C, O[j \mapsto \text{null}], N[j \mapsto b], P, R), b, e \sqcap [a]
\end{align*}
\]

R-NEW-BOX

\[
\begin{align*}
& k = \text{next}(\text{dom } R) \\
& j = \text{next}(\text{dom } O) \quad \text{implClass}(B) = C \quad o = j : C \quad \text{fields}(C) = T \\
& \mid \text{null} \mid c \mid \mid \text{intf}(B) = I \quad P' = P[k \mapsto b] \quad R' = R[k \mapsto \text{bsig}(I)] \\
& (C, O, N, P, R), b, e \sqcap [\text{new } B] \overset{\text{R}}{\longrightarrow} (C, O[j \mapsto \text{null}], N[j \mapsto k], P', R'), b, e \sqcap [o]
\end{align*}
\]

R-FIELD-ACCESS

\[
\begin{align*}
& o = j : C \quad N(j) = b \quad \text{fields}(C) = T \quad O(j) = \overline{v} \\
& (C, O, N, P, R), b, e \sqcap [o.f_i = v] \overset{\text{R}}{\longrightarrow} (C, O, N, P, R), b, e \sqcap [v_i]
\end{align*}
\]

R-FIELD-UPDATE

\[
\begin{align*}
& o = j : C \quad N(j) = b \quad \text{fields}(C) = T \\
& (C, O, N, P, R), b, e \sqcap [o.f_i = v] \overset{\text{R}}{\longrightarrow} (C, O[j \mapsto v/v_i \overline{v}], N, P, R), b, e \sqcap [v]
\end{align*}
\]

R-CALL

\[
\begin{align*}
& o = j : C \quad \text{mbody}(C, m) = (\pi)e'' \quad b' = N(j) \quad e' = [o/\text{this}, v/\pi]e'' \\
& (C, O, N, P, R), b, e \sqcap [o.m(\pi)] \overset{\text{R}}{\longrightarrow} ((b, e \sqcap [\text{result}]) \cdot C, O, N, P, R), b', e'
\end{align*}
\]

R-RETURN

\[
\begin{align*}
& S = ((b, e \sqcap [\text{result}]) \cdot C, O, N, P, R) \\
& S, b, v \overset{\text{R}}{\longrightarrow} (C, O, N, P, R), b', e \sqcap [v]
\end{align*}
\]

R-CAST-OBJ

\[
\begin{align*}
& C <: T \quad b' = N(j) \quad T' = \sigma^+(\overline{R}(\text{crossedBoxes}_P(b', b)), C) \quad T' <: T \\
& (C, O, N, P, R), b, e \sqcap [(T)j : C] \overset{\text{R}}{\longrightarrow} (C, O, N, P, R), b', e \sqcap [j : C]
\end{align*}
\]

Figure 2.3: Reduction rules of the standard semantics with boxes
Figure 2.4: Definitions for the configurations of the message semantics
2.6 Message Semantics

We won’t describe in detail how the references are adapted. We will use the functions `forward` and `receive` to translate intra-box messages to inter-box messages and vice versa. These functions, besides adapting the references using the functions `import` and `export`, also do type adaptation (which was done in the standard semantics by the cast rule). The type adaptation ensures that the original type of an object can only be seen up to the types in the box signature.

Our box tree is represented by the currently executing box `b` and a box environment, capturing the state of all surrounding boxes and their children minus `b`. The state of a box consists of a local execution stack `C`, an object store `O` of the objects belonging to the current box, an inner box store `I`, and a mapping `R` which maps inter- to intra-box references and vice-versa. This mapping is build dynamically upon program execution. Box states can contain holes, which are denoted by □, acting mainly as a place holder.

Among the reduction rules, we only show the ones which are relevant to our specification extension. These are the rules which transfer messages across the external box boundaries, and the rule to create new boxes. One has to note however that there are also rules which handle messages going to and coming from inner boxes (which we call the internal boundary).

2.6.1 Small-Step Message Semantics

The small-step message semantics is expressed by judgements of the form

\[
B\square, B, r \xrightarrow{\text{RM}} B\square', B', r'
\]

which has the semantics of reducing an expression `r` or handling a message `r` in box state `B` and under the box environment `B\square` leading in one step to `B\square', B', r'`. The reduction rules can be found in Fig. 2.5 and Fig. 2.6. The main difference to the standard semantics is that the cast rule doesn’t have to do the type adaptation, as this is done when references pass boundaries. The adaptation is done through the `receive` and `forward` methods.

Initial Configuration

The initial configuration of the message semantics is similar to the one from the standard semantics. For a program `P` with a main method body `e_{main}`, and `C` some class in `P`, `O_0` some reference closed object store, the initial configuration is defined as:

\[
\square, (\bullet, O_0, \varnothing, \varnothing), [\text{own}.j_0:\text{Main}/\text{this}, \text{own}.j_1:C/\text{input}]e_{main}
\]

2.6.2 Mix-Step Message Semantics

The mix-step message semantics uses a small-step judgement to express box-local executions and a big-step judgement to relate message sent to inner boxes to their replies. As consequence, the box environment can be eliminated from the box-local judgement:

\[
B, r \xrightarrow{\text{RMM}} B', r'
\]

To express all the small steps between a message sent to an inner box and its reply as a big-step judgement, we can use the transitive closure of \(\xrightarrow{\text{RMM}}\) denoted by \(\xrightarrow{\text{RMM}^+}\).
20

\begin{align*}
\text{RM-NEW-OBJ} & \\
j = \text{next}(\text{dom}(\mathcal{O})) & \quad o_b = \text{own}.j:C & \quad \text{fields}(C) = T \bar{j} & \quad \text{null} \models \bar{j} \\
\mathcal{B}, \langle C, \mathcal{O}, I, R \rangle, e \square \[\text{new } C \] & \longrightarrow_{\text{RM}} \mathcal{B}, \langle C, \mathcal{O}[j \mapsto \text{null}], I, R \rangle, e \square [o_b]
\end{align*}

\begin{align*}
\text{RM-NEW-BOX} & \\
k = \text{next}(\text{dom}(\mathcal{I})) & \quad \text{fields}(C) = T \bar{j} & \quad \text{null} \models \bar{j} & \quad \text{impl}(B) = C & \quad \text{intf}(B) = I \\
(o_d, Q) = \text{export}(\text{env}, \text{own}.0:C, (\text{bsig}(I), \varnothing)) & \quad (o_b, Q') = \text{import}(k, o_d, (\text{bsig}(I), \varnothing)) & \\
I' = I[k \mapsto (\star, \{0 \mapsto \text{null}\}, \varnothing, \{\text{env} \mapsto Q\})] & \\
\mathcal{B}, \langle C, \mathcal{O}, I, R \rangle, e \square \[\text{new } B \] & \longrightarrow_{\text{RM}} \mathcal{B}, \langle C, \mathcal{O}, I', R[k \mapsto Q'] \rangle, e \square [o_b]
\end{align*}

\begin{align*}
\text{RM-CAST-OBJ} & \\
T' <: T & \\
\mathcal{B}, B, e \square [\langle T \rangle b.q:T'] & \longrightarrow_{\text{RM}} \mathcal{B}, B, e \square [b.q:T']
\end{align*}

\begin{align*}
\text{RM-FIELD-ACCESS} & \\
o_b = \text{own}.j:C & \quad \text{fields}(C) = T \bar{j} & \quad \mathcal{O}(j) = \overline{v_b} \\
\mathcal{B}, B, e \square [o_b.f_i] & \longrightarrow_{\text{RM}} \mathcal{B}, B, e \square [v_b]
\end{align*}

\begin{align*}
\text{RM-FIELD-UPDATE} & \\
o_b = \text{own}.j:C & \quad \text{fields}(C) = T \bar{j} & \quad \mathcal{O}(j) = \overline{v_b} & \quad \mathcal{O}' = \mathcal{O}[j \mapsto [v_b/v_b]]v_b \\
\mathcal{B}, \langle C, \mathcal{O}, I, R \rangle, e \square [o_b.f_i = v_b] & \longrightarrow_{\text{RM}} \mathcal{B}, \langle C, \mathcal{O}', I, R \rangle, e \square [v_b]
\end{align*}

Figure 2.5: Rules of the small-step message semantics (except messages)
2.6 Message Semantics

\[ B = \langle C, O, I, R \rangle \]

\[ B \Box, B, e\Box[ o_b, m(\overline{v})] \rightarrow_{RM} B \Box, e\Box[ result] \cdot C, O, I, R), \rightarrow o_b.m(\overline{v}) \]

**RM-HANDLE-CALL**

\[ B \Box, B, v_b \rightarrow_{RM} B \Box, B, \leftarrow v_b \]

**RM-HANDLE-RETURN**

\[ n_b \rightarrow o_b.m(\overline{v}) \quad o_b = \text{own.j:C} \quad \text{mbody}(C, m) = (\overline{v})e \]

\[ B \Box, B, n_b \rightarrow_{RM} B \Box, B, [o_b/this, \overline{v}/x]e \]

**RM-EXEC-CALL**

\[ B = \langle e\Box[ result] \cdot C, O, I, R \rangle \]

\[ B \Box, B, n_b \rightarrow_{RM} B \Box, B, [o_b/this, \overline{v}/x]e \]

**RM-EXEC-RTRN**

\[ n_b \rightarrow o_b.m(\overline{v}) \quad o_b = \text{env.j:T} \quad (n_d, Q) = \text{forward}(env, n_b, R(env)) \]

\[ B \Box, (\text{env.c}, O, I, R), n_b \rightarrow_{RM} B \Box, (\text{env.c}, O, I, R[env \rightarrow Q]), n_d \]

**RM-FWD-CALL-ENV**

\[ n_b \rightarrow v_b \quad (n_d, Q) = \text{forward}(env, n_b, R(env)) \]

**RM-FWD-RTRN-ENV**

\[ n_d \rightarrow [v_d, m(\overline{v})] \quad o_d = \text{in.j:T} \quad (n_d, Q) = \text{receive}(env, n_d, R(env)) \]

\[ B \Box, (\text{env.c}, O, I, R), n_d \rightarrow_{RM} B \Box, (\text{env.c}, O, I, R[env \rightarrow Q]), n_b \]

**RM-RCV-CALL-ENV**

\[ n_d \rightarrow [v_d, m(\overline{v})] \quad o_d = \text{in.j:T} \quad (n_d, Q) = \text{receive}(env, n_d, R(env)) \]

**RM-RCV-RTRN-ENV**

\[ n_d \rightarrow [v_d, m(\overline{v})] \quad o_d = \text{in.j:T} \quad (n_d, Q) = \text{receive}(env, n_d, R(env)) \]

\[ B \Box, (\text{env.c}, O, I, R), n_d \rightarrow_{RM} B \Box, (\text{env.c}, O, I, R[env \rightarrow Q]), n_b \]

Figure 2.6: Rules of the small-step message semantics (messages)
As we do not care in this presentation about messages passing the internal boundary of boxes, the changes to the small-step message semantics rules we present are minimal and can be found in Fig. 2.7, where we only picked out the most important ones.

The initial configurations are similar to the small-step message semantics, with only difference that we can leave out the empty environment.

2.7 Equivalence of Standard and Message Semantics

We won’t go into detail for the proof of the equivalence of the standard and the different message semantics. As we will reuse the results in the next chapter, we give a short outline how the proof goes. The idea is to show first that both standard and short-step message semantics are bisimilar to each other. In order to achieve strong bisimulation, yet another intermediate semantics (called the summarised message semantics) must be defined which remedies the problem of the short-step message semantics that it needs several steps to reduce calls and returns. Thus every chain of steps of the form HANDLE-CALL (FWD-CALL ★ RCV-CALL ★)+ EXEC-CALL is replaced by a single step CALL and every chain of steps of the form HANDLE-RTRN (FWD-RTRN ★ RCV-RTRN ★)+ EXEC-RTRN is replaced by a single step RETURN, where ★ ∈ {ENV, INNER}. The number of steps is of course dependent on the distance of the initial and final box of the call or return in the box hierarchy tree.

From here, a bisimulation relation ≈ can be formulated between the configurations
of the standard and the summarised message semantics. In order to formulate this relation concisely, some additional functions and constraints have to be specified. We will give these definitions in chapter 4 as we enhance them to incorporate our changes.

2.8 Conclusion

The technical report [20] presented in this chapter continues to derive a fully abstract semantics, which certainly is an interesting result, but the way this is achieved is of no direct interest for this thesis.

As we have presented all these different semantics, the reader should remember that these different semantics can be proven to be equivalent. This allows us to base our specification technique on one of the semantics, in our case the mix-step message semantics. We will see that this will lead to a very natural notion of box specification.
In this chapter, we describe our implementation of the OOBox language concepts in Java\cite{11}. As we cover some internal aspects of Java, we assume the reader to have a good knowledge of the Java platform. We had to make a lot of design decisions, for which we always give a motivation in the next sections. We will first present the enhancement to the Java language, shortly describe the implementation, and then present encountered problems and open issues.

3.1 Requirements and Goals

Resulting from the presentation in the previous chapter, we must be able to do the following things in JBox. We must be able to declare boxes with their corresponding interface and implementation. The checks we must be able to perform follow directly from the standard semantics of our OOBox language:

- recognise new box instances and manage the box hierarchy
- map newly created objects to their respective boxes
- know in which box the program currently is
- restrict field access across box boundaries
- restrict casts according to cast rule

The main goal behind our implementation is that we want it to be as non-invasive as possible, so that it fits well into an existing build chain. We thus make no changes to the actual Java language, but use meta-facilities to achieve our goals. We use the annotation facilities of Java 5, whereas we ensure that valid JBox programs remain valid Java programs. This allows for JBox programs to be executed as regular Java programs, even if the advanced checking facilities are not enabled. One important thing to note is that OOBox is a sequential language, whereas Java allows for programs with concurrency (although there are no concurrency constructs in the language). We will also have to cope with that aspect.
3 JBox, Program Components for Java

3.2 The Language

We provide a library for annotating the existing code. The design decision which we took here was strongly restricted by our implementation strategy of the runtime checks, as we will explain later. Box declarations and implementation bindings are defined using delegating methods and annotations. We therefore provide the `@newBox` annotation, which can be found in the `jbox.runtime.spec` package.

This annotation is used on a delegating method the following way:

```java
@newBox
public SomeBoxInterface create() {
    return new SomeBoxImplementation();
}
```

Listing 3.1: A box declaration example using the `@newBox` annotation

The box interface is encoded in the method signature. The return type of the method is the box interface from which we derive the box signature. The implementation type of the box is the instance we create in the method body. Our box instance remains unnamed, as it is marked by the method. To create a box with interface `SomeBoxInterface`, we only have to invoke the annotated method which can be either static or not, allowing for different patterns to declare boxes.

The aforementioned solution allows also to declare boxes for library classes which can not be modified, like declaring a box with box interface `List` and box implementation `LinkedList`. Although there might appear other ways to the reader to declare boxes in an interesting way using annotations, e.g. direct annotation of the types, we selected the given approach for reasons of implementation ease, as we have all the type information at one location. However, we can imagine different possibilities to declare boxes in future implementations.

3.3 Runtime

We target the Java(TM) 2 Platform Standard Edition 5.0 which is widely used nowadays. There are several possibilities to check the box restrictions at runtime:

- Adapting a Java Virtual Machine (JVM)
- Adapting a compiler to insert checks into existing annotated code, starting the analysis at the source or bytecode level
- Rewrite annotated bytecode at load time to insert checks

Adapting the JVM has the advantage that we can optimise our data structures and algorithms as much as we want. We evaluated different ways for fulfilling our purposes. One possibility is to modify the source code of an existing JVM, for example the Java HotSpot virtual machine of the OpenJDK\(^1\) project. Because it is quite an amount of work to get familiarised with the existing architecture as the project has about 250.000

\(^1\)http://openjdk.java.net
lines of code, it is very risky to break some existing working modules. Furthermore, with each new version of the project, we would have to re-adapt our existing changes.

Another less invasive possibility is using the Java Virtual Machine Tools Interface (JVMTI)\(^2\) of the Java Platform Debugger Architecture (JPDA). It is a native programming interface implemented by the JVM which provides both a way to inspect the state and to control the execution of applications running in the Java Virtual Machine. It is two-way as it allows to observe and control the application, either in response to events or in a direct fashion. It allows the tagging of objects, in our case mapping to boxes, in a very efficient way. A drawback of this method is that JVMTI may not be available in all the different virtual machine implementations for Java. Using the native interface also lets us develop our runtime checker for different platforms. In our opinion, for a serious industrial strength solution of the box model, this is the solution to choose.

Adapting a Java source compiler is yet another way, as there are some good open-source Java compilers out there. However this leads to similar issues as explained above with the JVMs. The advantage is that we can make language changes to allow for nicer syntactic constructs. As we want to be as non-invasive as possible, this does not lead to our goals. We would also possibly break existing tool chains using one particular compiler and its API.

We implemented the third possibility by rewriting byte code at load time to insert checks, as this is the most portable and non-intrusive way to implement the checks. The infrastructure we constructed can in fact very easily be transformed to a bytecode compiler, as bytecode transformation is already done at load time. The Java 2 Platform SE 5.0 defines the `java.lang.instrument`\(^3\) package which provides services that allow agents in bytecode format to instrument programs running on the JVM, thus being platform independent. The mechanism for instrumentation is modification of the bytecode of method bodies, as, according to the spec, the redefinition may change method bodies, the constant pool and attributes but not add, remove or rename fields or methods, change the signatures of methods, or change inheritance. Since the changes are purely additive, they should not modify application state or behaviour. Because the inserted agent code is standard bytecode, the VM can run at full speed, optimising not only the target program but also the instrumentation.

We thus have a standard API for retrieving the bytecode of class files and, according to the specs, we are allowed to make changes to the method bytecode. The only drawback is that this must be supported by the current JVM configuration, which can be checked using the `isRedefineClassesSupported()` method from the `Instrumentation` interface.

To make the actual bytecode introspection and transformation, the following tools could be used:

- BCEL : Bytecode Engineering Library \(^4\)
- SERP \(^5\)
- Javaassist \(^6\)

\(^2\)[http://java.sun.com/j2se/1.5.0/docs/guide/jvmti/jvmti.html]
\(^3\)[http://java.sun.com/j2se/1.5.0/docs/api/java/lang/instrument/package-summary.html]
\(^4\)[http://jakarta.apache.org/bcel/]
\(^5\)[http://serp.sourceforge.net/]
\(^6\)[http://labs.jboss.com/javassist/]
ASM 7

For a more complete list we refer to [1]. We only need basic type information, a small runtime, and fast execution. After a short evaluation of the different tools, and a quick search on performance comparisons, e.g. http://www.mail-archive.com/bcel-dev@jakarta.apache.org/msg00612.html, we chose ASM as the way to go. This runtime library weighs about 33 KB in its lightest form with a SAX like API only. It lacks a full verifier, but that’s not even needed as our transformations are very basic e.g. no changes to stack frames are made. The ASM framework is described on its webpage as an all purpose Java bytecode manipulation and analysis framework. It can be used to modify existing classes or dynamically generate classes, directly in binary form. Provided common transformations and analysis algorithms allow to easily assemble custom complex transformations and code analysis tools.

It is highly optimised for performance, making it therefore highly attractive for its use in dynamic systems.

The bytecode transformations we make are directly resulting from our checking requirements in Section 3.1. In order to keep work simple and extensible, we opt to keep the actual bytecode transformation minimal and do most of the checking in a library. This is of course a point where optimisation is possible in the future. We thus have two packages, jbox.agent and jbox.runtime. The agent package represents the bytecode transformation part whereas the runtime package provides the actual checking algorithms. In the agent component we also have a main handler which receives all event calls from the modified bytecode program and delegates them to the appropriate handler. A simple handler implements methods like onMethodEnter(), onFieldAccess() etc. which then does the aforementioned checks. The events to be generated are woven into the bytecode by the agent.

We also generate events for box creation, casts and object creation which can then be mapped to a box. In the package jbox.agent we defined a handler interface IHandler which must be implemented by handler implementations, which can be arbitrarily complex, e.g. provide facilities to attach specifications etc. The IHandler interface looks as follows and directly reflects the bytecode transformations which have been made:

```java
public interface IHandler {
    public abstract void onMethodEnter(
        Object newthis ,
        String className, String methodName, String sig);

    public abstract void onMethodExit(
        Object oldthis ,
        String className, String methodName, String sig);

    public abstract void onCast(Object o, String type);

    public abstract void onNewObject(Object o);

    public abstract void nextObjectNewBox(
```

7 http://asm.objectweb.org/
3.4 Conceptual Issues

In this section we present some of the challenges we encountered and discuss the issues of JBox mainly due to Java having more concepts than can be found in OOBox minus components. We then present how we adapted JBox regarding the issues. Among the issues are multiple subtyping, static members, generics and bridge methods.

3.4.1 Static Members

Java has static fields and methods, which don’t fit very naturally in the box concepts. There are different possibilities to assign semantics to these fields and methods. In our opinion, each of the given semantics only feels natural for some given cases.

One possibility to look at static fields and methods is that they are part of a given class object. We can then assign given class objects to boxes, where this leads to the issue, which box we assign them too. If we respect the JVM semantics where the class is loaded on first use, this can confuse the user as it is quite non transparent. Another possibility is to allocate class objects always in the global box.

Yet another possibility to look at static methods is that these are executed in the current context, which is how we implemented it as this posed no additional overhead. We don’t check any access restrictions to static fields. Another possibility is to prohibit access to static fields and/or static methods but from the global box.
Another issue worthwhile discussing would be static initialisers which are even more problematic. We currently refrain the JBox programmer from using these, as we have not studied the behavioural impact yet. As conclusion, all the mentioned problems result from the fact that Java is not purely object-oriented in contrast to OOBox.

### 3.4.2 Multiple Subtyping

In our JBox implementation we have to respect the fact that we have multiple subtyping in Java. As a consequence, there are sets of messages which can appear at the boundary of boxes. An example of this would be a message from a method \( m \) on an object of class \( C \) implementing two interfaces \( A \) and \( B \), both declaring \( m \). Now if only \( A \) and \( B \) are in the box signature, both method \( m \) declarations of \( A \) and \( B \) are the most specific ones in the box signature, thus resulting in a compound message.

While this is not directly an issue, it will have an impact on the concrete specification language for JBox which we will describe later e.g. we have to be able to specify such occurrences.

### 3.4.3 Generics

Java generics fit very well into the concept of box signatures. If we want to expose a list of objects of type \( \text{SomeType} \), we can use the given syntax:

```java
java.util.List<SomeType> doSomething()
{
  ...
}
```

Listing 3.4: Generics example 1

Thus we have to extend box signatures with the types found in parametrized types of method signatures. In the given example, if \( \text{doSomething} \) is a method of a type appearing in the box signature, then \( \text{SomeType} \) is also in the box signature. So far we have only revealed half of the truth. Generics allow far more complicated scenarios as presented above.

```java
<T> void doSomething(T x)
{
  ...
}
```

Listing 3.5: Generics example 2

In this case, the instantiation of \( T \) depends on the scenario it is used in. Thus, it does not directly have any effect on the box signature, which is known at compile time. We did not implement the box signature inference for instantiated generic types, but claim that this would only be a small addition.

As the developers of Java always had a very strong interest in backwards compatibility, old bytecode (pre 1.5) must be able to run on new JVMs and new code be able to compile and run on old JVMs. This has lead to the so-called concept of erasure, where at bytecode level, parametrised types are mapped to raw types, and cast operations are inserted. However, even as we perform our analysis at a bytecode level, the type
3.4 Conceptual Issues

parameter information, if it is available, is still stored in the bytecode and can even be queried with the reflection facility.

The issue we encounter here are with pre 1.5 versions of Java, where there was no possibility to specify type parameters. If, for example, in the given example, List would not be parametrised to SomeType, and SomeType would not appear elsewhere in the box signature, than an external user of the box would not be able to access the objects of SomeType by a downcast from Object. In this scenario, we would have to give the user the possibility to manually extend the box signature or the user could give a dummy method with SomeType in its signature just for the sake of the box signature.

3.4.4 Primitive Types and Arrays

So far we only considered types defined by classes or interfaces. However Java also has primitive and array types. For primitive types (int, boolean, float,...), parameter values are copied upon method invocation. They fall under the category of the null value in OOBox. For array types, if they appear in a box signature, their respective content type also has to be in the box signature and vice-versa. Otherwise, no special rules apply.

3.4.5 Bridge Methods

Bridge methods are created by the Java compiler to preserve the semantics of Java 5 while being backwards compatible with the JVM bytecode format. They are used for example for covariant return types and type parameter instantiations as we can see in the following example:

```java
class Super<T> {
    public void foo(T t) {
        System.out.println("super");
    }
}

class Sub extends Super<String> {
    @Override
    public void foo(String t) {
        System.out.println("sub");
    }
}
```

Listing 3.6: Bridge method example

Since foo in the subclass has a different erasure, the compiler creates the following bridge method in Sub:

```java
public void foo(Object t) {
    foo((String)t);
}
```

Listing 3.7: Bridge method example cont’d

These bridge methods are marked as synthetic in the bytecode. We have to take care about synthetic methods in our implementation of JBox. In our concrete case, where
a call on foo (Sub) from the exterior of the box foo belongs to happens, the signature of the call contains as parameter Object, not what was intended semantically at the source level. As bridge methods are marked in the bytecode by the *synthetic* attribute, we have the possibility to detect them and act accordingly.

We have not implemented solutions for the issue, but only want to highlight the issues here for future better implementations. This is a classical example where we can see that the Java language semantics are not nicely preserved across translation to bytecode due to backwards compatibility. Other semantic entities in Java to be discussed are inner, local and anonymous classes. This all leading again to bridge methods, we don’t want to discuss them in detail. For future implementations, these have to be taken into consideration too.

### 3.5 Implementation Issues

We encountered several issues while implementing the given semantics.

#### 3.5.1 New Object Detection

We implemented the object to box mapping by rewriting the bytecode of the constructor of the `java.lang.Object` class, which is called upon each object creation. This design decision was based upon the following principles:

- This location is the first possible location we can do an external call to a library, as specified by the JVM specification[16]. The reasoning is the following one. If we have an object creation of class `myClass`, e.g. `new myClass()`, bytecode can look like this:

```
1 ...  
2 new #1 // Allocate uninitialised space for myClass
3 dup   // Duplicate object on the operand stack
4 ...   // code can appear here
5 ...   // but not use the newly created object
6 invokespecial #5 // Invoke myClass.<init>
```

Listing 3.8: First possible location: Example

There are however some restrictions, namely the following one according to section 4.9.4 of the spec:

The verifier rejects code that uses the new object before it has been initialised.

We can thus access the object at the earliest in the constructor of the object. Furthermore, Section 2.12 states that

if a constructor body does not begin with an explicit constructor invocation and the constructor being declared is not part of the primordial class Object, then the constructor body is implicitly assumed by the compiler to begin with a superclass constructor invocation "super();"....

Consequently, we can access the object at the earliest in the constructor of the Object class, root of the class hierarchy.
3.5 Implementation Issues

- This location is the last possible location we want to do an external call to a library to associate the object to a box. The reasoning behind this can be easily explained by the following scenario:

```java
class A {
    A() {
        super();
        new F();
    }
}
class B extends A {
    @newBox static B create() {
        return new B();
    }
    private B() {
        super();
        new G();
    }
}
```

**Listing 3.9: Last possible location: Example**

We have a box with box interface B, which upon creation, creates a new object of class G. But we also want the object of class F in class A to be assigned to the newly created box. In the given scenario, it becomes clear we must register the newly created box right at the latest after the `super()` call in A.

As no objects are created in the constructor of the `Object` class and we want to keep track of objects of class `Object` too, we insert a call to our library with `this` as parameter in the constructor of the `Object` class which then registers the object in the current active box.

New boxes are registered in a somewhat hack-like fashion. We analyse methods marked with the `@newBox` annotation, search for a compatible object to be created in that method body and, before the constructor of the box is called, make a call into our library that the next object to be created is a box object of the box type extracted from the method signature.

The aforementioned methods keep the bytecode changes very low, and helps us to focus on the implementation of the utility library. It also dodges the problem that our bytecode analyser is in a different class-path than the application, leading to the issue that inter-class analysis is not possible in the bytecode analyser. This was an issue we had in prior versions of our bytecode rewriter and where we then aimed for an easier solution.

3.5.2 Other Open Issues

As we retransformed certain basic library classes of the `java.lang` and `java.io` package, we encountered problems that the JVM would not accept the changed base library classes, probably because for certain classes e.g. `String` it has more knowledge on the interna of the class. This was at least the case for the Sun JVM which we used for testing.
We suppose somehow fixed offsets etc. are used in the implementation, but have no concrete proof but the crashes of the JVM as we fed it the modified String class.

Another issue we didn’t address was that our algorithm to find overridden methods was sometimes not intelligent enough e.g. we didn’t respect auto-boxing etc. Perfecting this algorithm, mainly by reading the Java language specification, was out of scope for this thesis.

As compound values have to be represented in Java as sets of objects, it remained unclear for us to which box these set of objects must belong. We would have liked that these objects be treated as values, thus not registering read methods calls to these objects as messages. This issue is mainly due to the Java concept of nearly everything being an object. One remedy would be to introduce immutable objects which could then be used as values. While there certainly already exists (better or worse) solutions for this problem e.g. in form of more constrained type systems, we did not look further into this problem as this would lead to a more complicated implementation of JBox, which is not the focus of this thesis.

Another nontrivial task was to make sure that, due to the aforementioned library calls in the Object constructor, we would disable analysis for the time our program was executing bytecode transformation, which could occur at a program point where a new class is loaded into the JVM, e.g. when a new class is encountered in the executing program. Analysis was also disabled during the time we were actually already analysing, avoiding infinite recursive calls. This issue was resolved by setting global flags.

The advantage of our package separation approach is however that we did this in the jbox.agent package, and the actual Handler in jbox.runtime doing the analysis would not have to care about these issues.

### 3.6 Conclusion

Implementing JBox was an interesting yet challenging task. It took several partial rewrites as we encountered several difficulties implementing our concepts, as Java is far more complicated than the small theoretic sandbox language we used to describe the concepts.

We have achieved a simple yet powerful implementation of OOBox in Java. This is mainly due to the Java agent facility which we believe to be very useful to add small language changes in an non-invasive manner. While there are many open issues, most of them can be fixed for a more serious implementation. The problems we encountered were of various nature, technical as conceptual but always challenging.
As we have introduced the OOBox language with its various semantics in Chapter 2, we are now able to formally present our extension. Our concept of box specifications describe the external boundary behaviour of a box. As explained in Chapter 2 we define the external boundary as the boundary between a box and its enclosing box. Based on the message semantics from OOBox and the notion of box denotation, we describe a box as how it reacts to a series of incoming messages. What we want to do is formulate restrictions on the valid message sequences. We abstract from a concrete specification technique in this chapter, and provide a way to plug-in custom specification techniques, which then have the following in common:

- They put restrictions on the external boundary message behaviour of a box
- They are modular in the sense that they only describe the behaviour of one box at a time
- They relate the state space of the implementation to the specification

In our opinion, this serves as a firm basis for various interesting specification techniques. Allowing different concrete specification techniques allows the programmers to select the one fitting his needs best for a particular component.

In this chapter we first talk about the semantics of our extension to OOBox, and introduce new syntactic constructs as needed. In order to keep the presentation simple and not to put even more burden on the reader regarding the number of definitions and function symbols, we refrain from defining too many helper functions, and thus ask the reader to forgive us the lack of formality in some parts of this presentation.

### 4.1 Notions and Syntax Enhancements

In a first step, we must somehow relate boxes to concrete specification techniques and specifications. We call concrete specification handler a specification technique (with its appropriate language) and box specification the specification of a box for which there
must be some concrete specification handler which can interpret it. We introduce a box
specification declaration as follows:

\[ \text{map spec } H < S > \text{ to } B \]

which maps a concrete specification handler \( H \) initiated with a box specification \( S \)
to a box \( B \). We describe an initiated specification handler \( H < S > \) as a pair of
transition relation \( \rightarrow_{H_S} \) and a value \( \text{initState}_{H_S} \). The transition relation \( \rightarrow_{H_S} \) maps
the abstract and concrete specification states \( S \) and \( S_I \) of a box to the next states \( S' \)
and \( S'_I \) of the box, depending on the message \( \mu \) occurring at the boundary:

\[ \langle \mu, \langle S, S_I \rangle \rangle \rightarrow_{H_S} \langle S', S'_I \rangle \]

The exact form of the abstract and concrete specification states are left open to a
particular instance of a concrete specification handler. The concrete specification han-
der is thus like an interpreter for a certain specification language, whereas the initiated
concrete specification handler is like an interpreter with a given program. As we will
later see, we require the transition relation \( \rightarrow_{H_S} \) to be deterministic, e.g. behave as a
function. As, later in this chapter, we want to refer to the concrete specification handler
of a certain box type \( B \), we write \( \text{handler}(B) \), yielding a pair of transition relation and
initial state, which we will sometimes simply call the specification of \( B \).

The messages we consider are of the following form:

\[ \mu = \langle T, m, \delta \in \{ic, oc, ir, or\} \rangle \]

where \( \mu \) is a tuple consisting of a type, method name and direction. \( ic \) and \( oc \)
stand for incoming and outgoing call, and \( ir \) and \( or \) stand for incoming and outgoing
return. We won’t reuse the syntax for messages in the message semantics as we want
to describe these messages without relating to a concrete semantics (e.g. message or
standard semantics). We will clarify the reasoning behind this later.

We provide another addition to the OOBox language, introducing the \textit{assert} expres-
sion which allows the programmer to state programmatically the specification state the
implementation is in. As the programmer wants to incrementally define this based on
program flow, we introduce the \textit{assert} expression as follows

\texttt{assert } x \texttt{ before } e

which states that the specification of the current box has property \( x \) just before the
next external message call happens in expression \( e \). The \( x \) property is specified in a form
that is to be understood by the concrete specification handler of the current executing
box. In a concrete programming language like Java, one would probably use the \texttt{String}
type for \( x \) which allows for arbitrary complex value content. Whereas the abstract
specification state \( S \), which denotes the specification state the box implementation is
currently in, is derived by the message behaviour, the concrete specification state \( S_I \) is
also determined by the assertions the programmer formulates in the implementation.

In the next chapters, we will give the reader a better idea about the aforementioned
constructs and explain the detailed semantics. We describe the relation between box
implementations and specifications first using the message semantics, as these provide a
more \textit{natural} view. This will mainly motivate why we introduced the different semantics
in chapter 2. We will start by the high-level view of the behavioural semantics, and will refine our ideas down to the various semantics encountered for OOBx, resulting in an adapted standard semantics.

4.2 Specification-Constrained Behavioural Semantics

As stated in the introduction, our specifications put restrictions on the valid boundary message sequences of a box (also described as boundary behaviours). We will introduce the notion of a specification correct denotation of a box (partial function \([\text{BI}_{H<\mathcal{S}>}]\)). The definition is given later, and can be derived from the adapted mix-step semantics similar as it was done for the box denotation \([\text{BI}]\) in [20]. We could as easily have chosen the denotation-based semantics to describe this impact, but we will use the mix-step semantics, as we will later draw the relation to the standard semantics, which it is nearer to. The transformation of the enhanced mix-step semantics rules to enhanced denotation-based message semantics rules is trivial as the changes which have to be made are all described in [20].

4.2.1 Specification-Constrained Mix-Step Semantics

We introduce the transition relation \(\mapsto_{\text{RMM}_{S}}\), which is based on the transition relation \(\mapsto_{\text{RMM}}\) of the mix-step semantics.

The rules of the mix-step semantics from Fig. 2.7 which are related to the external boundary behaviour of a box are the following ones:

- RMM-FWD-CALL-ENV
- RMM-FWD-RTRN-ENV
- RMM-RCV-CALL-ENV
- RMM-RCV-RTRN-ENV

They describe messages transferring from and to the environment which are either call or return messages. Besides the RMM-NEW-BOX rule, only the aforementioned rules are modified and the remaining rules are kept by and assimilated to \(\mapsto_{\text{RMM}_{S}}\). The modifications are as follows.

To the box state \(\langle \mathcal{C}, \mathcal{O}, \mathcal{I}, \mathcal{R} \rangle\), we add the abstract specification state \(\mathcal{S}\), which denotes the specification state the box implementation is currently in, and the concrete specification state \(\mathcal{S}_{I}\), which denotes the specification state the box implementation is programmatically asserted to be currently in. The concrete specification state is determined by the box implementation with the corresponding \texttt{assert} expressions. As a message appears at the external boundary of a box, the (concrete) specification handler which is associated with the box has to be able to do a step with the given message. As types in the message, we select the currently visible type at the box boundary which has been adapted so far. To ease the presentation, we write \(\texttt{cbt}\) when we mean the type of the currently executing box.
If a new box (instance) is created, we have to initialise the abstract and concrete specification state to the correct values. For the specification state of a box \( B \), we use the state given by \( \text{initState}_H \). For the implementation state, we use the empty stack \( \bullet \). The rest is left unchanged.

\[
\text{RMMS-new-box} \\
\frac{k = \text{next}(\text{dom}(\mathcal{I}))}{\langle \mathcal{C}, \mathcal{O}, \mathcal{T}, \mathcal{R}, \langle \mathcal{S}, \mathcal{S}_I \rangle \rangle, \langle e \sqsupset \text{new } B \rangle \leftarrow \text{RMMS} \langle \mathcal{C}, \mathcal{O}, \mathcal{T}, \mathcal{R} | \langle e \Rightarrow Q \rangle \rangle, \langle \mathcal{S}, \mathcal{S}_I \rangle \rangle, \langle e \sqsupset \mathcal{O}_B \rangle}
\]

As we also added the possibility for a programmer to specify the state the implementation is in, the following expression was added to the syntax of the language:

\[
e ::= \text{assert } x \text{ before } e
\]

The assertion \( x \) is formulated in the specification language and not further processed by our host language OOBBox\textsubscript{spec}. We leave it to the specification handler to process the formulated assertions upon box transition and just ensure that they are available at that time by storing them in our stack structure.

\[
\text{RMMS-assert} \\
\frac{S'_I = x \cdot S_I}{\langle \mathcal{C}, \mathcal{O}, \mathcal{T}, \mathcal{R}, \langle \mathcal{S}, \mathcal{S}_I \rangle \rangle, \text{assert } x \text{ before } e \leftarrow \text{RMMS} \langle \mathcal{C}, \mathcal{O}, \mathcal{T}, \mathcal{R}, \langle \mathcal{S}, \mathcal{S}'_I \rangle \rangle, e}
\]
4.2.2 Conformance of a Box Implementation to a Specification

As we have seen before, checking if a box implementation satisfies a box specification is simply to ensure that whenever a transition of a certain kind occurs in the box implementation, then a transition in the box specification handler must occur. As a denotation was given for box implementations, we can now give the denotation for specification correct implementations. It is simply derived in the same way as the aforementioned box denotation using our modified transition rules. We write the specification valid box implementation denotation as \([ BI_{H<S>} ]\). Similar to \([ B I ]\), it maps an incoming message sequence and an incoming message to an outgoing message but furthermore enforces that the message sequence is valid according to a specification \( H < S \).

We can then formulate the conditions on conformance of a box implementation to a specification as follows:

**Definition 4.1.** A box implementation \( B I \) satisfies a box specification \( H < S \) iff \([ BI_{H<S>} ] = [ B I ]\).

For a box implementation not satisfying its specification, we have \([ BI_{H<S>} ] \subset [ B I ]\). We call a specification \( H < S \) satisfiable if there exists a box implementation \( B I \) such that \([ BI_{H<S>} ] = [ B I ]\). We call a specification \( H < S \) trivial if for any box implementation \( B I \) we have \([ BI_{H<S>} ] = [ B I ]\). We are interested in specifications which are satisfiable but not trivial. It also becomes obvious why the specification transition relations \( \rightarrow_H \) have to be deterministic, as otherwise we would not be able to derive such a strong notion of specification valid box implementation denotation.

4.3 Towards a Correct Implementation

It would be sufficient to explain our specification technique with the use of one of the semantics e.g. the aforementioned message semantics. But as we want to make some claims about the correctness of our implementation, we will port the specification concepts to the standard semantics and give a reason for its equivalence to the message semantics. The advantage is that we can reuse huge parts of the proof results in [20].

4.3.1 Specification-Constrained Summarised Message Semantics

To show the equivalence between message and standard semantics as seen in Section 2.7, we would first have to describe the summarised message semantics as an intermediate step. As the rules for the summarised message semantics can be trivially derived from our modified big-step message semantics according to [20], we won’t repeat it here.

4.3.2 Specification-Constrained Standard Semantics

Based on Section 2.5 we now describe the modifications to the standard semantics and then later provide the necessary definition changes to show the equivalence to the summarised message semantics. The global state \( \langle C, O, N, P, R \rangle \) is enhanced by \( T \), which is a mapping from boxes to tuples of abstract and concrete box specification states.

The following rules are then modified:
RS-NEW-BOX

\[ k = \text{next}(\text{dom} \, \mathcal{R}) \]
\[ j = \text{next}(\text{dom} \, \mathcal{O}) \]
\[ \text{implClass}(B) = C \quad o = j : C \quad \text{fields}(C) = T' \]
\[ | \text{null} | = \langle B \rangle \quad \text{inf}(B) = I \]
\[ P' = \mathcal{P}[k \mapsto b] \quad \mathcal{R}' = \mathcal{R}[k \mapsto \text{bsig}(I)] \]
\[ T' = T[k \mapsto \langle \text{initState}_H, \bullet \rangle] \]
\[ \text{handler}(B) = \langle \rightarrow H, \text{initState}_H \rangle \]

\[ \langle C, \mathcal{O}, \mathcal{N}, \mathcal{P}, \mathcal{R}, T \rangle, b, e[\text{null}] \mapsto \langle C, \mathcal{O}[j \mapsto \text{null}], \mathcal{N}[j \mapsto k], P', R', T', e[\text{null}] \rangle \]

The definitions of \textit{crossedBoxesIn} and \textit{crossedBoxesOut} used in the adapted CALL and RETURN rules are from Def. 2.2. We use an array syntax to access values of \( M_{\text{OUT}} \) and \( M_{\text{IN}} \). What we do here is basically adapting the message type, as done previously by the message semantics, in order to give the correct box message to the appropriate specification handler of each traversed box. Each handler must be able to do a step using the appropriate message. As described earlier, the path from one box to another in the box hierarchy has the form of an arc first going up and then down, where \( M_{\text{OUT}} \) represents the part of the path going upwards and \( M_{\text{IN}} \) the part coming downwards.

RS-CALL

\[ o = j : C \quad \text{mbody}(C, m) = (\pi)e \]
\[ b' = N(j) \quad e' = [o/\text{this}, \nu]\pi e \quad M_{\text{IN}} = \text{crossedBoxesIn}(b, b') \]
\[ M_{\text{OUT}} = \text{crossedBoxesOut}(b, b') \]
\[ \forall i \in \{1.. | M_{\text{OUT}}[i] \} : M_{\text{OUT}}[i] \text{ is of type } B_i \]
\[ \forall i \in \{1.. | M_{\text{IN}}[i] \} : M_{\text{IN}}[i] \text{ is of type } B_{i+1} | M_{\text{OUT}}[i] \]
\[ \forall i \in \{1.. | M_{\text{OUT}}[i] \} : \text{handler}(B_i) = \langle \rightarrow H_i, \bullet \rangle \]
\[ C_0 = C \quad \forall i \in \{1.. | M_{\text{OUT}}[i] \} : C_i = \sigma(\mathcal{R}(M_{\text{OUT}}[i]), C_{i-1}) \]
\[ \forall i \in \{1.. | M_{\text{IN}}[i] \} : C_{i+1} = \sigma(\mathcal{R}(M_{\text{IN}}[i]), C_{i+1} | M_{\text{OUT}}[i-1]) \]
\[ \forall i \in \{1.. | M_{\text{OUT}}[i] \} : \langle \langle C_i, m, \text{oc}, T(M_{\text{OUT}}[i]) \rangle \mapsto H_i, \diamond_i \]
\[ \forall i \in \{1.. | M_{\text{IN}}[i] \} : \langle \langle C_{i+1} | M_{\text{OUT}}[i], m, \text{inc}, T(M_{\text{IN}}[i]) \rangle \mapsto H_{i+1} | M_{\text{OUT}}[i], \diamond_{i+1} | M_{\text{OUT}}[i] \]

\[ T_0 = T \quad T' = T[M_{\text{OUT}}[i] \mapsto H_i | M_{\text{OUT}}[i]] \quad \forall i \in \{1.. | M_{\text{OUT}}[i] \} : T_i = T_{i-1}[M_{\text{OUT}}[i] \mapsto H_i] \]
\[ \forall i \in \{1.. | M_{\text{IN}}[i] \} : T_{i+1} | M_{\text{OUT}}[i] = T_{i+1} | M_{\text{OUT}}[i] = M_{\text{IN}}[i] \mapsto \diamond_{i+1} | M_{\text{OUT}}[i] \]

Method \( m \) is defined for class \( C_{|M_{\text{OUT}}[i]|} \)
\[ \langle C, \mathcal{O}, \mathcal{N}, \mathcal{P}, \mathcal{R}, T \rangle, b, e[\text{null}] \mapsto \langle \langle b, e \mapsto \text{result}, C, m \rangle \cdot C, \mathcal{O}, \mathcal{N}, \mathcal{P}, \mathcal{R}, T \rangle, b', e' \]

We need to enhance the execution stack \( C \) from Fig. 2.2 in order to know upon return from which method we are returning. Thus, as

\[ C = \pi \]

the following change is made:

\[ r ::= \langle b, e, C, m \rangle \]
4.4 Equivalence of Standard and Summarised Message Semantics

The given rules are also a first hint at how an implementation of the rules might look like in an interpreter based on the standard semantics. In a next step we adapt the definitions for the proof of equivalence between standard and message semantics.

4.4 Equivalence of Standard and Summarised Message Semantics

We first need to define the helper functions, which are given formally in [20]. The function \( bp \) returns for a given box environment the path of box identifiers which leads from the root to the box hole. It basically acts as a global position function in the box hierarchy tree. The function \( \pi \) maps box-local object or box identifiers from the semantic domain of the message semantics to the semantic domain of the standard semantics, where \( \boxi \) is used as the box identifier of the current box. Using the helper functions, we can define a mapping \( \text{glob}_x \) from \(<\text{box environment} - \text{box state}>\) pairs of the message semantics to \(<\text{state} - \text{box}>\) pairs of the standard semantics:

\[
\text{glob}_x(\mathcal{B}_i, \mathcal{B}) = (\langle C_R, \mathcal{O}_R, \mathcal{N}_R, \mathcal{P}_R, \mathcal{R}_R, \mathcal{T}_R \rangle, b_R)
\]

with

\[
\mathcal{T}_R = \text{glob}_x^T(\boxi, \mathcal{B}_i[\mathcal{B}])
\]

where \( \text{glob}_x^T \) is defined by:
\[ \text{glob}_{\pi}^T(\square, \langle C, O, I, R, T \rangle) \]
\[ = \bigcup_{k \in \text{dom} I} \text{glob}_{\pi}^T\left(\langle C, O, I[k \mapsto \square], R, T \rangle, I(k)\right) \]

(root case) and
\[ \text{glob}_{\pi}^T\left(B\square[\langle C, O, I[k \mapsto \square], R, T \rangle], \langle C', O', I', R', T' \rangle\right) \]
\[ = \{ \pi(bp(B\square[\langle C, I[k \mapsto \square], R, T \rangle]), \square) \mapsto T' \} \]
\[ \bigcup \bigcup_{k' \in \text{dom} I'} \text{glob}_{\pi}^T\left(B\square[\langle C, O, I[k \mapsto \langle C', O', I'[k' \mapsto \square], R', T' \rangle], R, T \rangle], I(k')\right) \]

What we described here is the composition of the box local specification handlers into a map from box identifiers to their respective specification handlers. Using this modified definition of \( \text{glob}_{\pi} \), we can shortly describe the influence on \( \approx \), the strong bisimulation relation between the standard and summarised message semantics. As seen in the message semantics, the types of the messages used as parameter for the specification handler transitions \( \mapsto \) are adapted at the box boundaries. We preserve this by explicitly doing this in the standard semantics using the \text{crossedBoxesIn} and \text{crossedBoxesOut} functions.

### 4.5 Conclusion

The main idea behind this chapter is that we now have a formal foundation to plug in concrete specification languages into our extended OOBbox language. For the formal presentation, we left some details out, namely parameter and return values. In a concrete specification language, these are of a very high importance, as they allow for a far more fine-granular specification technique. Another important result is that we can derive an implementation strategy from the specification-constrained standard semantics.
JBox\textsubscript{spec}, Pluggable Component Specification Techniques for Java

This chapter presents the adaption of the extensions to OOBox described in the previous chapter to JBox. To adapt JBox with the option to plug-in custom specification techniques, only some minor changes have to be made. In addition to the requirements mentioned in Section 3.1, we must be able to detect method calls passing box boundaries in order to trigger our box specification transitions.

5.1 Syntax Changes

In a first step, we must provide a facility for assigning box specifications and handlers to boxes as described in section 4.1. We therefore enhance the @newBox annotation of listing 3.1 with two attributes, the specification handler and the specification:

```
public @interface newBox {
    Class<? extends java.lang.Class> specHandler()
    default EmptySpec.class;
    String spec()
    default "";
}
```

Listing 5.1: Definition of the modified @newBox annotation

The spec attribute describes the location of a specification, which must be locatable by the designated specification handler. An example can look like the following:

```
@newBox(specHandler = jcs.JCSHandler.class, spec = "SomeBox.spec")
public SomeBoxInterface create() {
    return new SomeBoxImplementation();
}
```

Listing 5.2: A box specification declaration example using the modified @newBox annotation

The assert expression is implemented as a simple library call using a single value of type String as value.
5.2 Pluggable Specification Handlers

As described in the previous chapter, we need to provide a facility to plug in concrete specification techniques. We thus define an interface which the pluggable specification techniques must implement. A facility is then implemented to discover and initialise specification handlers which can load the given specifications. Our abstract interface for pluggable specification handlers looks like the following:

```java
public interface SpecificationHandler {
  public abstract SpecificationState initState();
  public abstract SpecificationState transition(
    MsgSet messages, SpecificationState previousState)
  throws NoTransitionException;
}
```

Listing 5.3: Definition of the SpecificationHandler interface

where SpecificationState is only a marker interface for the specification state, which is managed by its corresponding box handler according to the standard semantics of the previous chapter.

As our specifications need to have access to the parameter or return values of the messages, we have to slightly modify the IHandler interface from chapter 3. The signature of the methods onMethodEnter and onMethodExit is adapted the following way:

```java
public interface IHandler {
  public abstract void onMethodEnter(
    Object newthis, Object[] args,
    String className, String methodName, String sig);
  
  public abstract void onMethodExit(
    Object returnVal, Object oldthis, Object[] args,
    String className, String methodName, String sig);
  
  public abstract void onCast(Object o, String type);
  
  public abstract void onNewObject(Object o);
  
  public abstract void nextObjectNewBox(
    String intfClass, String specHandler, String spec);
  
  public abstract void onFieldAccess(Object field);
}
```

Listing 5.4: Modified IHandler interface

The other method signatures remain identical.

5.3 Conclusion

The implementation of JBox\textsubscript{spec} did not cost much effort, as most of the ground work was covered in the previous chapter.
The OOCS Specification Technique

The specification language OOCS (Object-Oriented Component Specifications), which we describe in this chapter, is a concrete specification language for OOBox\textsubscript{spec}. Our goal is to describe a simple, yet powerful specification language, while keeping checkability in mind. We first present the underlying ideas and concepts of the OOCS language and then present how these are integrated into our specification framework OOBox\textsubscript{spec}.

Our specification language describes the order in which messages are allowed to occur at an external box boundary. We reuse the concept of type states \cite{8}, where we associate a set of states with each box. In reaction to incoming messages, the specification language allows for an operational specification of outgoing messages with state restrictions. The specification language is very close to an actual object-oriented language. We keep the syntax as simple as possible, although a lot of syntactic sugar building on the presented material is possible and should be introduced in a day-to-day use setting.

In the rest of this chapter, we will give a formal syntax and semantics of OOCS. We will only give an abstract syntax, as a concrete syntax is not needed for theoretic foundations. We additionally try to illustrate the semantics of OOCS using a simple specification example.

6.1 Introduction with an Example

Our simple example models a restriction on a subject in the Observer pattern where we describe the subject as a box. The \textit{Subject} interface has two methods \texttt{register} and \texttt{update}, where the first one registers objects of type \textit{Observer}, and the second notifies the registered observer using their \texttt{notify} method.

As result from the box implementation denotation regarding correct sequencing of message types, we know that only a restricted chain of messages can occur, e.g. to each call message, a corresponding return message must occur. This information helps us to structure our specification language. For a more thorough description of the sequencing of messages, we refer back to chapter 4 and the presented transition rules. A valid message trace which arrives at the specification handler would be for example (where we abbreviate \texttt{O} for \textit{Observer}, \texttt{S} for \textit{Subject} and \texttt{n, u, r} for \texttt{notify}, \texttt{update} and \texttt{register}):
6 The OOCS Specification Technique

\[(S, u, ic) \rightarrow (O, n, oc) \rightarrow (S, r, ic) \rightarrow (S, r, or) \rightarrow (O, n, ir) \rightarrow (S, u, or)\]

Traces which do not occur are for example those where an incoming message appears right after another incoming message:

\[(S, u, ic) \rightarrow (O, n, ic) \rightarrow \ldots\]

The restriction on valid message sequences we want to impose using our specification is the following one. During a notification call, which is represented as an outgoing followed by an incoming message to the box, we don’t allow call-backs. What we want to do is put restrictions on the valid traces for our box. A specification correct trace is for example:

\[(S, u, ic) \rightarrow (O, n, oc) \rightarrow (O, n, ir) \rightarrow (O, n, oc) \rightarrow (O, n, ir) \rightarrow (S, u, or)\]

where, upon call of the update method, two notifications are sent out. A trace which is not correct w.r.t. the specification is for example:

\[(S, u, ic) \rightarrow (O, n, oc) \rightarrow (S, r, ic) \rightarrow \ldots\]

where a callback occurs during a notification. During the rest of the presentation, we will always refer back to the given example and illustrate our formal definitions.

6.2 Syntax

We give the syntax of OOCS in Fig. 6.1. A box specification describes how a box should react (e.g. with which messages) when a certain type of messages arrives at its boundary. A box specification thus consists of a set of state specifications and so-called method specifications, which capture the behaviour of the box upon external call of a method. A state specification has a name denoted by \(s\) and consists of a set of values denoted by \(v\). A method specification has a type signature, pre-condition, method body and a post-condition. As we do not have method overloading, it is sufficient to describe a method by its name and the class or interface it belongs to. A method body describes and structures the possible outgoing calls. Sequence, choice and loop constructs are similar to those in existing programming or specification languages. A sequence construct states that an action (namely an outgoing message) must happen before another action. The choice construct states that either of two actions can occur. A call construct describes an external method call (and thus outgoing message) with its appropriate return (incoming message). An outgoing message is tagged by a guarantee that the box specification is in a certain state. Ingoing messages are tagged by a condition which must be checked. We also have a loop construct which allows for an action to happen repeatedly. There is a corresponding break construct to leave a directly enclosing loop. If there is no break construct in a loop, looping occurs infinitely often. In order to describe the semantics of a specification instance, we assign a distinct label to each node instance of the specification AST. We write \(x : a\) if we want to denote a label \(x\) which is assigned to a node of kind \(a\).

Conditions are of the following form, which we give in a more concrete-like syntax:
6.3 Transformation to Automata

Before we give a formal semantics for OOCS, we define the transformation of the specifications to automata and then describe the semantics of the automata. We assume that all of the following definitions are instantiated for one particular specification instance. We first need to describe some helper functions.

Our automata are basically pushdown automata with some additional guards. In order to represent these guards which result from our conditions, we transform the formulated conditions into (boolean) propositional formulas. This symbolic representation of the state allows for very compact automata\(^1\) as we shall see later. For each

\(^1\)We could expand the state space of the automata with the state specifications but this would lead to an exponential blow-up
state value we introduce a variable which we denote by the same name. If we have a state specification with state name \( s \) and values \( v_i \), where \( i \in \mathcal{V} = \{1..n\} \), the following transformations apply to our atomic condition expressions:

\[
s \equiv v_i \text{ transforms to } v_i \land \bigwedge \neg v_j , \text{ where } v_j \in \mathcal{V} - \{i\}
\]

If we denote the previous transformation by the function \( \tau \), we can then enhance the definition of \( \tau \) for the other cases:

\[
\begin{align*}
\tau(c_1 \& \& c_2) & := \tau(c_1) \land \tau(c_2) \\
\tau(c_1 \| c_2) & := \tau(c_1) \lor \tau(c_2) \\
\tau(!c) & := \neg \tau(c) \\
\tau(\text{true}) & := T \\
\tau(\text{false}) & := F
\end{align*}
\]

As said earlier, we have assigned a distinct label to each node (instance). We then define the function \( \text{next} \) modelling the control flow. To do this, we first need to define some helper entities. We define the set \( \text{INSET} \) as the set of all the labels which mark nodes of kind \( \text{In} \) in the specification AST. We then define the parent function \( p \) which, for a label of a node yields the label of the parent node. There are two special labels \( \text{start} \) and \( \text{break} \). Our function \( \text{next} \) then maps a label to a successor label denoting the next point in the execution:

\[
\text{next} : \text{Label} \rightarrow \text{Label}
\]
In order to give the definition of \textit{next} in Fig. 6.3, we need yet another helper function, which we will overload by the same name \textit{next}, yet having one more parameter, a context label, which we use to route the control flow.

To demonstrate how the \textit{next} function works, we apply it to the \textit{in1} label of our example specification in Fig. 6.2:

\[
\begin{align*}
\text{next}(\text{in1}) &= \text{next}(\text{in1} : \text{In}(\ldots), \text{start}) \\
&= \text{next}(\text{ms1} : \text{MSpec}(\text{ms1}, \text{in1}, \text{loop}, \text{out1}), \text{in1}) \\
&= \text{next}(\text{loop} : \text{Loop}(\text{or}), \text{loop}) \\
&= \text{next}(\text{or} : \text{Or}(\text{brk}, \text{call}), \text{or}) \\
&= \text{next}(\text{brk} : \text{Break}(), \text{brk}) \cup \text{next}(\text{call} : \text{Call}(\text{m2}, \text{out2}, \text{in2}), \text{call}) \\
&= \text{next}(\text{or} : \text{Or}(\text{brk}, \text{call}), \text{break}) \cup \text{next}(\text{out2} : \text{Out}(\ldots), \text{out2}) \\
&= \text{next}(\text{loop} : \text{Loop}(\text{or}), \text{break}) \cup \{\text{out2}\} \\
&= \text{next}(\text{ms1} : \text{MSpec}(\text{ms1}, \text{in1}, \text{loop}, \text{out1}), \text{loop}) \cup \{\text{out2}\} \\
&= \text{next}(\text{out1} : \text{Out}(\ldots), \text{out1}) \cup \{\text{out2}\} \\
&= \{\text{out1}\} \cup \{\text{out2}\} \\
&= \{\text{out1}, \text{out2}\}
\end{align*}
\]

This yields the set of labels containing \textit{out1} and \textit{out2}.

### 6.4 Transition Rules

We can now define our initiated specification handler, given by its transition relation and value \textit{initState}. We assume again that our transition relation is a mapping of an abstract and concrete specification state \(S\) and \(S_I\) to their respective successor states, depending on a message. Our \(S\) states are set of triples \((x, \psi, s)\) where \(x \in \text{Label}\) is a label, \(\psi\) a guard, and \(s\) a stack of labels. More precisely, \(x\) is either the root label or a label of an \textit{In} or \textit{Out} node. These denote the program points in the specification where synchronisation with the implementation must occur. The stack of labels \(s\) only contains labels of \textit{MSpec} and \textit{Call} nodes. This stack contains the current context (execution stack) of the specification. \(S_I\) is a stack of guards and represents the state the programmer intends the specification to be in by stating it explicitly in the implementation using the assert constructs. We assume there exists a function \(\tau'\) which transforms the assert constructs into our abstract syntax, which are nodes of kind \textit{Cond}.

There are four transition rules, one for each type of message, which can be found in Fig. 6.4. In order to define less helper methods we, again, use a semi-formal notation for our transition rules.

We also have to describe how our method descriptions, which are nodes of kind \textit{Method}, match to messages. We therefore describe a matching function which takes a type and method name denoting the message type and the same kind of parameters for our method descriptions. We require that method names are identical, as we have no method overloading in our formalism, and that the type of the message is a subtype of type described by the method description.

\[
\text{matches}(T, m, T', m') = (m == m') \land (T <: T')
\]

We also define a function \textit{accumCond} which takes a stack of conditions and yields the
\[
\text{next}(x) = \text{next}(x, \text{start})
\]

\[
\text{next} (\text{root} : \text{BSpec}(\_ , \_)) = \bigcup \text{next}(y_i, y_i), \\
\text{where } y_i : \text{In}_i \in \text{INSET}
\]

\[
\text{next}(w : \text{MSpec}(\_ , x , \_), w) = \text{next}(x, x) \\
\text{next}(w : \text{MSpec}(\_ , x , y , \_), x) = \text{next}(y, y) \\
\text{next}(w : \text{MSpec}(\_ , y , z , \_), y) = \text{next}(z, z) \\
\text{next}(w : \text{MSpec}(\_ , \_ , z , \_), z) = \text{next}(\text{root}, w)
\]

\[
\text{next}(x : \text{Seq}(y , \_), x) = \text{next}(y, y) \\
\text{next}(x : \text{Seq}(\_ , \_), \text{break}) = \text{next}(p(x), \text{break}) \\
\text{next}(x : \text{Seq}(y, z), y) = \text{next}(z, z) \\
\text{next}(x : \text{Seq}(\_ , z), z) = \text{next}(p(x), x)
\]

\[
\text{next}(x : \text{Or}(y , z), x) = \text{next}(y, y) \cup \text{next}(z, z) \\
\text{next}(x : \text{Or}(\_ , \_), \text{break}) = \text{next}(p(x), \text{break}) \\
\text{next}(x : \text{Or}(y , \_), y) = \text{next}(p(x), x) \\
\text{next}(x : \text{Or}(\_ , z), z) = \text{next}(p(x), x)
\]

\[
\text{next}(x : \text{Loop}(y), x) = \text{next}(y, y) \\
\text{next}(x : \text{Loop}(y), \text{break}) = \text{next}(p(x), x) \\
\text{next}(x : \text{Loop}(y), y) = \text{next}(y, y)
\]

\[
\text{next}(x : \text{Call}(\_ , y , \_), x) = \text{next}(y, y) \\
\text{next}(x : \text{Call}(\_ , y , \_), y) = \text{next}(\text{root}, x) \\
\text{next}(x : \text{Call}(\_ , \_ , z), z) = \text{next}(p(x), x)
\]

\[
\text{next}(x : \text{Out}(\_), \text{start}) = \text{next}(p(x), x) \\
\text{next}(x : \text{Out}(\_), x) = \{x\}
\]

\[
\text{next}(x : \text{In}(\_), \text{start}) = \text{next}(p(x), x) \\
\text{next}(x : \text{In}(\_), x) = \{x\}
\]

\[
\text{next}(x : \text{Empty}(\_), \_ ) = \text{next}(p(x), \text{break})
\]

Figure 6.3: Definition of the control flow function \textit{next}
6.5 Extension to Value Restrictions

In this presentation, we omitted restrictions on parameter and return values for our method descriptions. As our toy language OOBox only supports reference types with the special null value, which is member of each type, the only meaningful restrictions which can be formulated on message parameters and return values are reference comparisons to null or other references. In particular, this would lead to the following changes in our abstract syntax. Parameters and return values of method specifications are named. Message calls should specify a pre- and post-condition on these name. We thus enhance the In and Out branches with conditions on the parameter and return values (see Fig. 6.6).

\footnote{We have left out the additional conditions which result from the assert declarations in the implementation}
The OOCS Specification Technique

STEP-OUT-CALL
\[ \forall (y, \phi, r) \in S' : \exists (x, \psi, s) \in S : \]
y \in next(x) 
p(y) = \_ : Call(Method(T', m'), y : Out(\text{cond}), \_) 
matches(T, m, T', m') 
\exists \xi : \xi(\text{accumcond}(\tau'(S_I)) \land \tau(\text{cond})) 
\]
r = p(y) \cdot s 
\]
\[ \langle (T, m, oc), (S, S_I) \rangle \rightarrow_{S} \langle S', S_I \rangle \]

STEP-IN-RETURN
\[ \forall (y, \phi, r) \in S' : \exists (x, \psi, s) \in S : \]
y \in next(x) 
p(y) = \_ : Call(Method(T', m'), \_, y : In(\text{cond})) 
matches(T, m, T', m') 
\exists \xi : \xi(\text{accumcond}(\tau'(S_I)) \land \tau(\text{cond}) \land \phi) 
\]
s = p(y) \cdot r 
\]
\[ \langle (T, m, ir), (S, S_I) \rangle \rightarrow_{S} \langle S', \bullet \rangle \]

STEP-IN-CALL
\[ \forall (y, \phi, r) \in S' : \exists (x, \psi, s) \in S : \]
y \in next(x) 
p(y) = \_ : MSpec(Method(T', m'), y : In(\text{cond}), \_) 
matches(T, m, T', m') 
\exists \xi : \xi(\text{accumcond}(\tau'(S_I)) \land \tau(\text{cond}) \land \phi) 
\]
r = p(y) \cdot s 
\]
\[ \langle (T, m, ic), (S, S_I) \rangle \rightarrow_{S} \langle S', \bullet \rangle \]

STEP-OUT-RETURN
\[ \forall (y, \phi, r) \in S' : \exists (x, \psi, s) \in S : \]
y \in next(x) 
p(y) = \_ : MSpec(Method(T', m'), \_, \_ : Out(\text{cond})) 
matches(T, m, T', m') 
\exists \xi : \xi(\text{accumcond}(\tau'(S_I)) \land \tau(\text{cond})) 
\]
s = p(y) \cdot r 
\]
\[ \langle (T, m, or), (S, S_I) \rangle \rightarrow_{S} \langle S', S_I \rangle \]

Figure 6.4: Transition rules of the automaton
### 6.5 Extension to Value Restrictions

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
<th>Message</th>
<th>Stack</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>in1</td>
<td>out1</td>
<td>(S, u, or)</td>
<td>ms1 · ○ ⇝ ○</td>
<td>(Yes ∧ ¬No)</td>
</tr>
<tr>
<td></td>
<td>out2</td>
<td>(S, n, oc)</td>
<td>○ ⇝ call · ○</td>
<td>(No ∧ ¬Yes)</td>
</tr>
<tr>
<td>in2</td>
<td>out1</td>
<td>(S, u, or)</td>
<td>ms1 · ○ ⇝ ○</td>
<td>(Yes ∧ ¬No)</td>
</tr>
<tr>
<td></td>
<td>out2</td>
<td>(S, n, oc)</td>
<td>○ ⇝ call · ○</td>
<td>(No ∧ ¬Yes)</td>
</tr>
<tr>
<td>in3</td>
<td>out3</td>
<td>(S, r, or)</td>
<td>ms2 · ○ ⇝ ○</td>
<td>(Yes ∧ ¬No)</td>
</tr>
<tr>
<td>root</td>
<td>in1</td>
<td>(S, u, ic)</td>
<td>○ ⇝ ms1 · ○</td>
<td>T ∧ (Yes ∧ ¬No)</td>
</tr>
<tr>
<td></td>
<td>in3</td>
<td>(S, r, ic)</td>
<td>○ ⇝ ms2 · ○</td>
<td>T ∧ (Yes ∧ ¬No)</td>
</tr>
<tr>
<td>out1</td>
<td>in1</td>
<td>(S, u, ic)</td>
<td>ms1 · ○ ⇝ ○</td>
<td>(Yes ∧ ¬No) ∧ (Yes ∧ ¬No)</td>
</tr>
<tr>
<td></td>
<td>in2</td>
<td>(S, n, ir)</td>
<td>○ ⇝ call · ○</td>
<td>(Yes ∧ ¬No) ∧ (No ∧ ¬Yes)</td>
</tr>
<tr>
<td></td>
<td>in3</td>
<td>(S, r, ic)</td>
<td>ms2 · ○ ⇝ ○</td>
<td>(Yes ∧ ¬No) ∧ (Yes ∧ ¬No)</td>
</tr>
<tr>
<td>out2</td>
<td>in1</td>
<td>(S, u, ic)</td>
<td>ms1 · ○ ⇝ ○</td>
<td>(No ∧ ¬Yes) ∧ (Yes ∧ ¬No)</td>
</tr>
<tr>
<td></td>
<td>in2</td>
<td>(S, n, ir)</td>
<td>○ ⇝ call · ○</td>
<td>(No ∧ ¬Yes) ∧ (No ∧ ¬Yes)</td>
</tr>
<tr>
<td></td>
<td>in3</td>
<td>(S, r, ic)</td>
<td>ms2 · ○ ⇝ ○</td>
<td>(No ∧ ¬Yes) ∧ (Yes ∧ ¬No)</td>
</tr>
<tr>
<td>out3</td>
<td>in1</td>
<td>(S, u, ic)</td>
<td>ms1 · ○ ⇝ ○</td>
<td>(Yes ∧ ¬No) ∧ (Yes ∧ ¬No)</td>
</tr>
<tr>
<td></td>
<td>in2</td>
<td>(S, n, ir)</td>
<td>○ ⇝ call · ○</td>
<td>(Yes ∧ ¬No) ∧ (No ∧ ¬Yes)</td>
</tr>
<tr>
<td></td>
<td>in3</td>
<td>(S, r, ic)</td>
<td>ms2 · ○ ⇝ ○</td>
<td>(Yes ∧ ¬No) ∧ (Yes ∧ ¬No)</td>
</tr>
</tbody>
</table>

Figure 6.5: Derived automaton for the observer specification

<table>
<thead>
<tr>
<th>Method</th>
<th>(T, m, n, π)</th>
<th>method description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Out</td>
<td>(Cond, VCond)</td>
<td>condition called assert (guarantee)</td>
</tr>
<tr>
<td>In</td>
<td>(Cond, VCond)</td>
<td>condition called assume (check)</td>
</tr>
</tbody>
</table>

Figure 6.6: Modified abstract syntax for OOCS
In our conditions we can check for reference equality and equality to the null value. Composition of more complex conditions is done similar to the afore-mentioned conditions on box states:

\[
VCond ::= (n == n) \mid (n == \text{null}) \mid (VCond \&\& VCond) \mid ...
\]

Checking of these conditions is like running the expression fragment in the host language with appropriate substitutions. We won't refine these ideas, as all this has been done in specification languages like JML[15].

\section*{6.6 Conclusion}

The presented specification technique allows for component specifications in an object-oriented setting, which, as we know, has not existed before in this form. As we kept the specification very basic and lightweight, it was possible to give a simple formal description of the transformation rules to automata and the corresponding transition relations. So far, we have not considered the checking of these specifications but done ground work with this by the transformation to automata. We will consider the checking of these techniques in the next chapter, which will even more reflect the usefulness of the transformation to automata in this chapter.
In this chapter we describe JCS, the Java Component Specification technique, which is based upon the ideas of OOCS in the previous chapter. The JCS specification language is used to specify JBox programs and integrates into the JBoxspec technique. We will shortly describe the language using the previous observer example, present our implementation and cover the issues concerning checking of our specifications.

We will only shortly outline the syntax and semantics of the language with respect to the differences to OOBox. As seen in section 3.4.2, our specification language has to be able to respect multiple subtyping, resulting in intersection types which can appear at the boundary of boxes. We thus introduce the possibility to specify such message occurrences by so-called intersection types. Apart from this conceptual difference, all the concepts of the previous chapter remain valid.

7.1 Example Cont’d

To ease the transition from OOBox to JBox for the reader, we specify the observer example of the previous chapter using the JBox specification language. To show how the type adaption at box boundaries works, we first define interfaces for the subject and observers:

```java
public interface Subject {
    public void register(Observer o);
    public void update();
}

public interface Observer {
    public void notify(Subject s);
}
```

Listing 7.1: Subject and Observer interfaces

We then give the box specification which, as in the previous chapter, prohibits callbacks during notification calls. The syntax is Java-oriented and very straight-forward.
We have a matching declaration (line 4) which allows to group a set of incoming call messages for a specific type. Looping is introduced by the loop command and choice between a and b represented by the $<a>$ or $<b>$ statement. assume commands establish the conditions for incoming messages, and assert commands do the same for outgoing messages. assert statements can be put at (nearly) arbitrary places, which allows to incrementally specify the state conditions, e.g. if they directly precede the choice construct, they count for all branches of the choice.

```java
box Subject {
    state cbAllowed = { Yes, No } ;

    match Subject {
        void update() {
            assume cbAllowed == Yes;
            loop
                $<break;$
                $>$ or $<$
                assert cbAllowed == No;
                call Observer with notify(Subject s);
                assume cbAllowed == No;
                $>$
                assert cbAllowed == Yes;
            }
            void register(Observer o) {
                assume cbAllowed == Yes;
                assert cbAllowed == Yes;
            }
        }
    }
}
```

Listing 7.2: Subject box specification

We then give an implementation of the box which matches the specification:

```java
public class SubjectImpl implements Subject {
    final private List<Observer> obs = new ArrayList<Observer>();

    public void register(Observer o) {
        obs.add(o);
    }

    public void update() {
        for (Observer o : obs) {
            o.notify(this);
        }
    }
}
```

Listing 7.3: Subject implementation

The previous specification is respected as we have no calls going out of the box in the implementation of the register method. The update method makes as much outgoing notify calls as there are observers registered. This also conforms to the specification as
it allows for an arbitrary number of outgoing notification calls after an incoming update call.

We then link the box implementation to the specification and hide it behind the Subject interface. This is stated somewhere in our program code where we need to be able to instantiate subject boxes. The handler which we use is called JCSHandler and is the handler which understands JCS specifications.

```java
@newBox(specHandler = jcs.JCSHandler.class, spec = "Subject.spec")
public static Subject create() {
    return new SubjectImpl();
}
```

Listing 7.4: Specification to box mapping

Finally we define an example run:

```java
Observer o1 = ... // from somewhere
Observer o2 = ... // from somewhere
Subject s = create(); // subject box creation
s.register(o1);
s.register(o2);
s.update();
```

Listing 7.5: Example run

We have now the possibility to check if the run is a valid run. This is done automatically at runtime by our JBoxspec environment and the handler for our JCS specification language. In case of a run which is not specification conformant, the run is aborted with an exception. Whether the presented run is a valid one or not, depends on the concrete implementation of the observers. We give an implementation which leads to invalid runs:

```java
public class ObserverImpl implements Observer {
    public void notify(Subject s) {
        s.register(this);
    }
}
```

Listing 7.6: Observer implementation which is not conformant to Subject box specification

As the observer implementation does a callback into the subject box which is prohibited by the specification, the run presented above, where the observer implementations are of type ObserverImpl, is not a valid one.

To visualize what valid and invalid runs are, the JCS implementation is able to generate dot files, graphical descriptions of our automata. The generated graphics file of the automaton defined by the observer specification is shown in Fig. 7.1. The state N0 corresponds to state root of Fig. 6.2 and Fig. 6.5, N2 to in1, N6 to in3, N4 to in2, N3 to out1, N5 to out2 and N7 to out3. Transitions denoting incoming messages are marked in red, outgoing messages in blue. Transitions denoting call messages are marked in bold in contrast to return messages. Stack changes are marked by a label pre-fixed by a plus or minus sign, whereas the first stands for pushing the pre-fixed
label onto the stack, and the latter popping the label off. Some states have a condition
associated to it. These are the states denoting nodes of kind Out (as seen in the previous
chapter). Transition conditions thus have to be read like the following: The condition
of a transition with condition $a$ where the source node has a condition $b$ is the logical
conjuntion of both conditions $a$ and $b$. As can be seen for the transition from the node
labelled $N5$, which corresponds to the node labelled out2 in Fig. 6.2, there is only one
transition possible, namely the one with the notify incoming return message. The other
two are not possible as the condition, resulting from the logical conjunction of the one
on the transition and the one specified in $N5$, as described earlier, is not satisfiable. It
becomes obvious that some transitions, as seen in the automaton, can never be taken.
In a next step, it is thus possible to discard these transitions.

7.2 Implementation

In this section we shortly describe our implementation. As a parser module, we used
Rats! ¹, an extensible parser generator which is part of xtc, the extensible compiler
toolkit. The benefits of using Rats! are the following ones:

- It organizes grammars into modules[12], which allowed us to reuse parts of the
existing Java grammar written for Rats!.

- It builds on parsing expression grammars instead of context-free grammars and in-
tegrates lexing with parsing, thus being scannerless. Parsing expression grammars[10]
(PEGs) are an alternative to context free grammars (CFGs) for formally speci-
fying syntax. Where CFGs express nondeterministic choice between alternatives,
PEGs instead use prioritized choice. Packrat parsers are parsers for PEGs that op-
erate through the use of memoization. Memoization ensures that packrat parsers
have linear time performance even in the presence of unlimited lookahead and
backtracking.

- It seems to be relatively mature, as it is being used for the project Fortress ²,
Sun’s ³ new language for high-performance computing.

Another feature of Rats! is that it can automatically generate abstract syntax trees.
In particular, it supports productions that return no semantic values (such as those
recognizing spaces or comments), string values (such as those recognizing literals or
identifiers), and generic tree nodes (potentially all other productions). We only used
this feature partially (mostly for the scanner part), as we also wanted to evaluate the
interaction between the Rats! tool and Katja[18], a small and light-weight specification
language for order-sorted, recursive term and position datatypes for Java. Katja allows
us to concisely describe the AST of JCS. The grammar and AST specifications defined
for Rats! and Katja can be found in the appendix.

To check the boolean formulas for satisfiability, we used the SAT4J ⁴ satisfiability
solver which has special support for small problem instances. It also gave us support to

¹http://www.cs.nyu.edu/rgrimm/xtc/rats.html
²http://projectfortress.sun.com/
³http://www.sun.com
⁴http://www.sat4j.org

58
transform the conditions into conjunctive normal form, which is a prerequisite to feed them to a SAT solver.

As how to check our automata, we are not yet sure what the most efficient way is. It really depends on the size and complexity of the specification, which is statically decidable. Message matching, condition checking and label stack manipulation determine the checking process. What we are concerned with are the order in which to do these different checks, and, for each of the checks, determine a fast checking technique. In our example implementation we have not implemented more efficient data structures.

The presented JCS language has been implemented in a purely prototypical way. We did not strive for completeness nor correctness, the main goal was only to get some easy examples running, which we achieved to do.
Figure 7.1: Auto-generated graphics of the automaton defined by the observer example specification
Conclusion and Future Work

In this master thesis we have developed a formal framework for describing and checking grey-box specifications for object-oriented program components. We have then presented one particular instance of such a specification language. We also mapped all the theoretical concepts onto the popular programming language Java.

8.1 Conclusion

The program components used as a starting point were the ones presented in [20]. We then defined the interaction of programs with their box specifications. This allows to give future box specification languages a precise semantics. We exemplarily illustrated the semantics of such a specification language which, in our opinion, provides a very natural feel as to the syntax. A method for checking the specifications was derived once for box specifications in general and once for our concrete specification language. As we have borrowed ideas from different domains (e.g. process semantics or automata theory), we have all integrated into a unique outcome. To our knowledge, there exists no comparable specification technique for object-oriented program components which has been described in such a formal setting.

Mapping all of the previously mentioned concepts onto a programming language which is fairly popular today was the next step to do. Along with all the technical challenges, we managed to build a prototype which would allow to interpret and check our specifications in a fairly straight-forward manner. The different software packages built were divided into different Eclipse projects, which allowed us to explicitly state their dependencies. As consequence, our software packages allow for easy extensibility and reconfiguration.

All in all, we have laid ground work for a lot of interesting future work.

8.2 Future Work

There is still space for a lot of improvements of our approach. As result of our implementation in Java, we are not happy with the concept of pure data belonging to
8 Conclusion and Future Work

A certain box. This unnecessarily complicates our specifications, as we have to specify method calls just to traverse these object graphs representing purely abstract data types. In a future version of the component model, we would like to see a distinction between pure value types and reference types. For a mapping to Java, appropriate modelling constructs have to be found.

Our framework for specifications can also be improved. We would like to incorporate inter-box messages which go to inner boxes, e.g. respect the inner boundary behaviour of boxes. As a consequence, we would like to see the same kind of composability for the specifications. We want specifications to be more modular in the sense that specifications can be composed of specifications. The state of a box specification box will then consist of its own states plus the states which result from inner boxes. Moreover, a concept for box sub-typing has to be found.

Besides these improvements on the current concepts, we also think that a large amount of work can be done on verification and model checking for the presented specification technique. As pointed out earlier, an integration of our work into other existing specification techniques for object-oriented languages like JML[15] or Spec♯[2] seems to be a viable option.

Finally, our language and checking tools (see appendix A.1) have to be further improved as to allow for more complex specifications.


Bibliography


Appendix

A.1 Software Installation and Use

The software fragments we provide are partitioned into four different Eclipse projects:

- **jbox-agent**: contains the software fragments described in chapter 3: Bytecode modification fragments and event interception.

- **jbox-runtime**: contains the software fragments described in chapter 3 and 5: Handler enforcing box semantics and support for pluggable specifications.

- **jcs-runtime**: contains the software fragments described in chapter 7: Concrete specification handler for JCS (Status: highly prototypical)

- **jbox-test**: contains some basic test cases

The dependencies are the following. **jbox-runtime** depends on **jbox-agent** as its handler implements the generic handler interface. **jcs-runtime** depends on **jbox-runtime** as it implements a concrete specification handler. **jbox-test** depends on both of the previously mentioned packages.

Each of the Eclipse projects contains also an Ant file to build the project into a Java archive (.jar file). To activate the JBox agent for a program run, it has to be passed as argument to the java interpreter as follows:

```
--javaagent: jboxagent.jar
```

The **jbox-test** package contains further instructions about running the test examples. The software packages are available on the accompanying CD-ROM and at

Appendix

A.2 JCS Language Grammar Specification

```java
/**
 * JCS grammar
 * @version 1.0
 */

module jcs.ast.gen.JCS;

import xtc.util.Symbol(xtc.util.Spacing);
import xtc.lang.JavaSymbol(xtc.util.Symbol);
import xtc.util.Spacing;

body {
  static {
    add(JAVA_KEYWORDS, new String[] {
      "abstract", "continue", "for", "new",
      "assert", "default", "if", "package",
      "boolean", "do", "goto", "private",
      "break", "double", "implements", "protected",
      "byte", "else", "import", "public",
      "case", "instanceof", "return",
      "catch", "extends", "int", "short",
      "char", "final", "interface", "static",
      "class", "finally", "long", "strictfp",
      "const", "float", "native", "super",
      "switch", "this", "throw", "throws",
      "volatile", "try", "transient", "void",
      "while", "synchronized",
    });
  }
  option withLocation,
  constant,
  parser(jcs.ast.gen.JCSRecognizer),
  setOfString(JAVA_KEYWORDS);

  public BoxSpecification =
    Spacing
    "package":Word QualifiedIdentifier ";":Symbol
    ImportDeclaration*
    BoxDeclaration
    '\u001a'?
    EndOfFile
  ;

  ImportDeclaration =
    "import":Word QualifiedIdentifier
    ( ";":Symbol "+":Symbol )?
    ";":Symbol
  ;

  BoxDeclaration =
    "box":Word Ident
    ( "with":Word yyValue:Type )?

"{" : Symbol
StateDeclaration* 
MatchDeclaration* 
"}" : Symbol
;

StateDeclaration = 
"state" : Word Ident "=" : Symbol ArrayInitializer " ; " : Symbol
;

ArrayInitializer = 
"{" : Symbol
Ident
(" ", " : Symbol Ident )*
"}" : Symbol
;

MatchDeclaration =
"match" : Word Type ( ", " : Symbol Type )* 
"{" : Symbol
MethodDeclaration* 
"}" : Symbol
;

MethodDeclaration =
IntersectionType Ident FormalParameters Block
;

Block =
"{" : Symbol Statement* "}" : Symbol
;

Statement =
Block
/ "<<" : Symbol Statement* ">>" : Symbol
"or" : Word
"<<" : Symbol Statement* ">>" : Symbol
("or" : Word "<<" : Symbol Statement* ">>" : Symbol)*
/ "loop" : Word Statement
/ "break" : Word " ; " : Symbol
/ "assume" : Word Expression "; " : Symbol
/ "assert" : Word Expression "; " : Symbol
/ "call" : Word IntersectionType
"with" : Word (Ident ", " : Symbol)?
Ident FormalParameters ("returns" : Word Ident)? " ; " : Symbol
/ " ; " : Symbol

112 ParExpression =
"(" : Symbol Expression ")" : Symbol
115 ;

Expression =
ConditionalOrExpression
;

118 FormalParameter =

A.2 JCS Language Grammar Specification
Appendix

Type Ident

FormalParameters =
  "(" : Symbol FormalParameter
  ( "." : Symbol FormalParameter )* ")" : Symbol
  / "(" : Symbol ")" : Symbol

ConditionalOrExpression =
  ConditionalAndExpression ( ConditionalOrExpressionTail )*

ConditionalOrExpressionTail =
  ";

ConditionalAndExpression =
  EqualityExpression ( ConditionalAndExpressionTail )*

ConditionalAndExpressionTail =
  "||" : Symbol ConditionalAndExpression

EqualityExpression =
  UnaryExpressionNotPlusMinus ( EqualityExpressionTail )*

EqualityExpressionTail =
  <Equal> "==" : Symbol UnaryExpressionNotPlusMinus
  / <NotEqual> "!=" : Symbol UnaryExpressionNotPlusMinus

UnaryExpressionNotPlusMinus =
  <LogicalNot> "!" : Symbol PrimaryExpression
  / <Base> PrimaryExpression

PrimaryExpression =
  Literal
  / Ident
  / "(" : Symbol Expression ")" : Symbol

IntersectionType =
  "{" : Symbol Type ("&" : Symbol Type )* "}" : Symbol
  / Type

Type =
  "void" : Word
  / "+" : Symbol
  / "?" : Symbol
  / TypeName Dimensions
A.2 JCS Language Grammar Specification

TypeName =
    BasicType
    / QualifiedIdentifier

BasicType =
    "byte": Word
    / "short": Word
    / "char": Word
    / "int": Word
    / "long": Word
    / "float": Word
    / "double": Word
    / "boolean": Word

Dimensions =
    ( '"\[": Symbol '"\]": Symbol )* ;

Ident =
    Identifier

// -------------------------- Literals

Literal =
    FloatingPointLiteral Spacing
    / IntegerLiteral Spacing
    / CharacterLiteral Spacing
    / StringLiteral Spacing
    / BooleanLiteral
    / NullLiteral

// ______ Integer literals

IntegerLiteral =
    HexNumeral IntegerTypeSuffix?
    / OctalNumeral IntegerTypeSuffix?
    / DecimalNumeral IntegerTypeSuffix?

DecimalNumeral = "0" / NonZeroDigit Digit* ;
NonZeroDigit = [1−9] ;
Digit = [0−9] ;

HexNumeral = "0" [xX] HexDigit+ ;
HexDigit = [0−9a−fA−F] ;

OctalNumeral = "0" OctalDigit+ ;
OctalDigit = [0−7] ;

IntegerTypeSuffix = [IL] ;
Appendix

// Floating point literals

FloatingPointLiteral =
  Digit+ '.' Digit* Exponent? FloatTypeSuffix?
/ '.' Digit+ Exponent? FloatTypeSuffix?
/ Digit+ Exponent FloatTypeSuffix?
/ Digit+ Exponent? FloatTypeSuffix
;

Exponent = [eE] [+-]? Digit+ ;

FloatTypeSuffix = [fFdD] ;

// Character and string literals

CharacterLiteral = ['\' (EscapeSequence / ['\'] ) ] '\';

StringLiteral = ['"' (EscapeSequence / ['"]""""""""""""""""""""""""""""""""""""""""

EscapeSequence =
  '\\' [0-3] OctalDigit OctalDigit
/ '\\' OctalDigit OctalDigit
/ '\\' OctalDigit
;

OctalEscape =
  '\\' [0-3] OctalDigit OctalDigit
/ '\\' OctalDigit OctalDigit
/ '\\' OctalDigit
;

UnicodeEscape =
  '\\' 'u' HexDigit HexDigit HexDigit HexDigit ;

// Boolean literals

BooleanLiteral =
  "true":Word
/ "false":Word
;

// Null literals

NullLiteral =
  "null":Word
;

// QualifiedIdentifier

QualifiedIdentifier =
  Ident ( void":"Symbol Ident )*
;

Identifier =
  yyValue:Word &{ ! contains(JAVAKEYWORDS, yyValue) } ;

Word = WordCharacters Spacing ;

WordCharacters =
  start:_ &{ Character.isJavaIdentifierStart(start) }
Listing A.1: PEG grammar of JCS for the *Rats!* parser generator
Appendix

A.3 JCS AST Specification

```java
package jcs.ast.gen
import java.lang.String
import java.lang.Integer
import java.lang.Boolean

public class JCS
{
    backend java {
        package jcs.ast.gen
        import java.lang.String
        import java.lang.Integer
        import java.lang.Boolean
    }

    root BoxSpecification Nd
    external String
    external Integer
    external Boolean

    BoxSpecification ( Identifier* identPackage ,
                      ImportDeclaration* imports ,
                      BoxDeclaration box )

    ImportDeclaration ()

    BoxDeclaration ( Identifier ident ,
                     [ Type type ],
                     StateDeclaration* states ,
                     MatchDeclaration* matches )

    StateDeclaration ( Identifier ident ,
                       Identifier* idents )

    MatchDeclaration ( Type* types ,
                       MethodDeclaration* methods )

    MethodDeclaration ( Type type ,
                        Identifier id ,
                        Parameter* params ,
                        Block block )

    Parameter ( Type type ,
                [ Identifier ident ] )

    Block * Statement

    Statement = Or
               | Loop
               | Block
               | Assume
               | Assert
               | Call
               | EmptyStatement
               | Break

    Or * Statement
    Loop ( Statement statement )
    Assume ( Expression expr )
    Assert ( Expression expr )
    Call ( Type type ,
```
[Identifier identCallee],
Identifier methodName,
Parameter* params,
[Identifier identReturn]

EmptyStatement ()
Break ()

Expression = OrExpression
| AndExpression
| EqExpression
| NotExpression
| Literal
| Identifier
| BoolValue
| IntValue

OrExpression (Expression e1, Expression e2)
AndExpression (Expression e1, Expression e2)
EqExpression (Expression e1, Expression e2)
NeqExpression (Expression e1, Expression e2)
NotExpression (Expression e)

Identifier (String)
Literal (String)
BoolValue (Boolean)
IntValue (Integer)

Type = WildcardOne
| WildcardAny
| Identifier
| Identifier*
| IntersectionType
| BasicType

IntersectionType * Type

WildcardOne ()
WildcardAny ()
BasicType (String)

Listing A.2: Katja specification of the JCS abstract syntax