Case Studies with CafeOBJ*

Alexander Knapp
Ludwig–Maximilians–Universität München
knapp@informatik.uni-muenchen.de

Abstract

We describe two case studies involving the algebraic specification language CafeOBJ. The first case study uses CafeOBJ as a tool in formally based object-oriented software engineering. We explain the derivation process of a formal specification from informal but annotated object models and interaction diagrams by means of Jacobson's "Recycling Machine" example. On the specification side, we explore CafeOBJ's rewriting logic approach to object-orientation and discuss the implementation of a simple technique to constrain the non-determinism of rule applications used by our method. The second case study formalises a structural operational semantics of the object-oriented multi-threaded programming language Java in CafeOBJ. This case study provides a medium scale example of algebraic specification language's usage in rapid prototyping.

Introduction

CafeOBJ [8] is the product of a consequent further development of the executeable algebraic specification language OBJ3, the most prominent representative of the OBJ family; for a brief history of OBJ and related languages see [10]. The superior new aspect of this language and system is its tightly integrated support of both Goguen's behavioural specification technique [9] and Meseguer's rewriting logic [14] which provide two different, complementary approaches to object-orientation.

In this paper, we describe two case studies using CafeOBJ. We restrict ourselves to CafeOBJ's implementation of rewriting logic and leave a comparison to behavioural specification to future work.

The first case study uses CafeOBJ as a tool in formally based object-oriented software engineering. In [17], Wirsing and the author introduced the software development method fOOSE that enhances Jacobson's well-known informal object-oriented software development method OOSE [12] by formal annotations thus providing a mathematical rigorous software development process. Here, we exemplify this method by means of Jacobson's running example, the "Recycling Machine." We use the Unified Modeling Language (UML, [3]) and its stereotypes for OOSE in the modeling process and express (part of) our formal annotations in UML's Object Constraint Language. The specification

*This work has been partially supported by the CafeOBJ-project through the Information Technology Promotion agency (IPA), Japan.

uses CafeOBJ's implementation of rewriting logic for concurrency and object-orientation.

The second case study formalises a structural operational semantics of the object-oriented multi-threaded programming language Java described in [7]. CafeOBJ is used as a language for rapid prototyping. The specification formalising multi-threaded Java builds on two separate specifications: Sequential Java is extended to the multi-threaded case by parameterisation. A second specification formalises so-called event spaces, structures introduced in [7] to coordinate the communication of the different threads and the main memory.

All specifications are available at

`http://www.pst.informatik.uni-muenchen.de/personen/knapp/cafeobj.html`

Tests have been performed on a PentiumPro 200 with 64 megabytes running under Linux, kernel 2.0.30, using CafeOBJ version 1.4.0β5.

1 The Recycling Machine

Formal underpinnings of informal and diagrammatic modeling, analysis and design techniques have become one of the major research topics in software engineering. Three main approaches are discussed in the literature: On the one hand, well-known diagrammatic notations and methods are provided with different mathematically rigorous semantics; for example, Breu et. al. in [6] describe a unified system model for UML, Bourdeau and Cheng in [5] discuss a translation of the static diagrams used in Rumbaugh's Object Modeling Technique (OMT, [16]) to algebraic specifications. These foundational works mainly aim at checking the consistency of different views during the software development process and at allowing completeness tests even in the early phases of analysis and design. On the other hand, mathematically based development procedures are visualised by diagrammatic representations, helping the software developer to more easily grasp the specification's contents [13]. These two overall approaches focus either on purely informal or strongly mathematically oriented developement procedures. A third approach tries to combine the virtues of both, in particular, it enhances the informal, diagrammatic techniques by formal, mathematical annotations during the development process, see for example Nakajima and Futatsugi's enrichment of OMT by their Generic Interaction Language for Objects [15] or Achatz and Schulte's combination of Fusion with Object-Z [1].

In [17], Wirsing and the author also proposed such a complementation of Jacobson's OOSE method by annotations of the object models and the interaction diagrams in both the requirements and robustness analysis phase of OOSE's software development process. The resulting method, called fOOSE, provides a semi-automatic translation of the enriched diagrams to rewriting logic based, object-oriented concurrent algebraic specification languages such as Meseguer's Maude or, in particular, CafeOBJ. Additionally, an experimental translation to Java is stated.

In the following, we give a rough overview of fOOSE and describe the translation steps of the requirements and robustness analysis phase to CafeOBJ in more detail by means of Jacobson's recycling machine example. For a more detailed discussion of the theoretical basis, refinement issues, and translation to Java in the design step we refer to [17].

1.1 Enhanced OOSE Development Process

The development process of OOSE consists of five phases: use case analysis, robustness analysis, design, implementation and test [12] (see Figure 1).

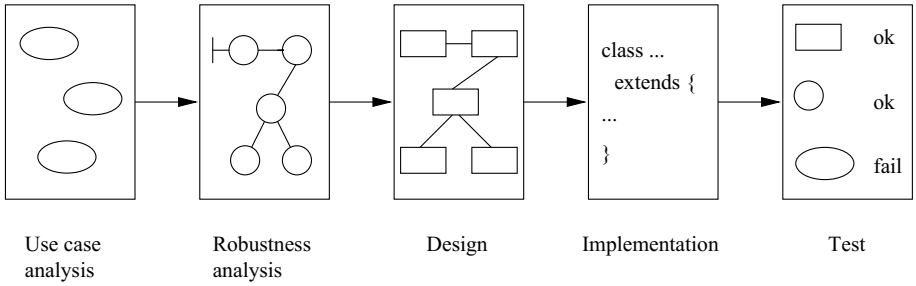

Figure 1: Development phases of OOSE

The use case analysis serves to establish a requirement document which describes the processes of the intended system in textual form. A use case is a sequence of transactions performed by actors (outside the system) and objects (of the system). During the robustness analysis the use cases are refined and the objects are classified in three categories: interaction, control and entity objects. Then in the design phase a system design is derived from the analysis objects and the objects of the reuse library. The design is implemented during the implementation phase and finally during test the implementation is tested with respect to the use case description.

As in all semi-formal approaches one problem is that testing can be done only at a very late stage of development; another problem is the fact that many important requirement and design details can neither be expressed by (the current) diagrams nor well described by informal text.

In the enhanced fOOSE method means are provided to overcome these deficiencies without changing the basic method. The enhanced development process consists of the same phases. The only difference is that the diagrams can optionally be refined and annotated by formal text. Any annotated diagram is semi-automatically translated into a formal CafeOBJ (or any other algebraic specification language implementing rewriting logic) specification, i.e. the diagram is automatically translated into an incomplete formal CafeOBJ specification which then has to be completed by hand to a formal one.

Thus any fOOSE diagram is accompanied by a formal specification so that every document has a formal meaning. In many cases the formal specification generates proof obligations which give additional means for validation of the current document. Further proof obligations are generated for the refinement of descriptions, e.g. from analysis to design. These proof obligations can serve as the basis for verification. Finally, since the executable specification language CafeOBJ is used, early prototyping is possible during analysis and design.

In the sequel the following method will be used for constructing a formal CafeOBJ specification (see Figure 2) from an informal description.

For any given informal description two diagrams are constructed: an object model with attributes and invariants, and an enhanced interaction diagram. The object model is used for describing the states of the objects and the (inheritance)

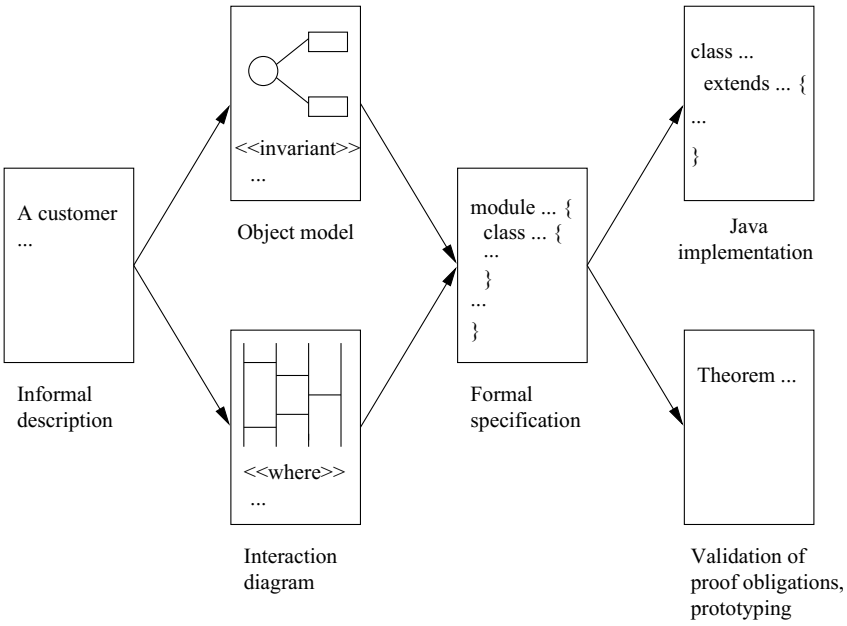

Figure 2: Construction and use of formal specifications

relationships, the interaction diagram describes the (data) flow of the messages the objects exchange. The object model directly translates to a specification; the interaction diagram yields an incomplete specification. The translation of both diagrams yields (after completion) a CafeOBJ specification together with some proof obligations. Moreover, refinement provides the information for tracing the relationship between use case descriptions and the corresponding design and implementation code, the induced proof obligations are the basis for verifying the correctness of designs and implementations.

1.2 Requirements Analysis

Use case. The informal description of the recycling machine consists of three use cases. One of them is the use case "returning items" which can be described in a slightly simplified form as follows (see also [12]):

> "A customer returns several items (such as cans or bottles) to the recycling machine. Descriptions of these items are stored and the daily total of the returned items of all customers is increased. The customer gets a receipt for all items he has returned before. The receipt contains a list of the returned items as well as the total return sum."

Object model and interaction diagram. We develop a first abstract representation of this use case with the help of an object diagram that describes the objects of the problem together with their attributes and interrelationships, and of an interaction diagram that describes the flow of exchanged messages.

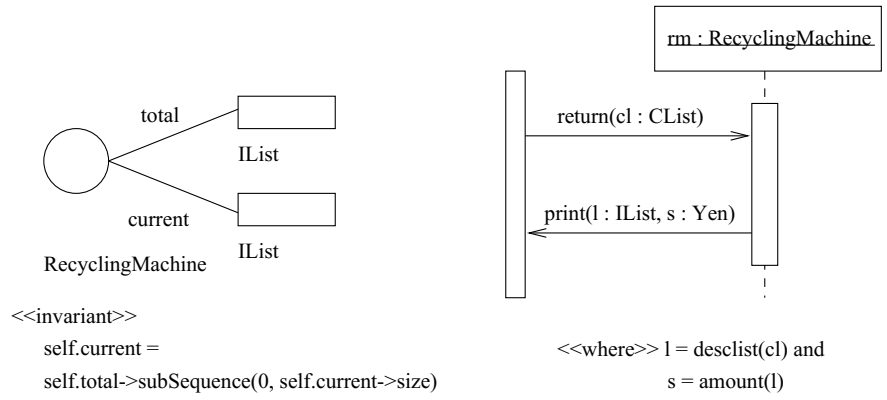

Figure 3: Object model and interaction diagram of the Recycling Machine

Additionally we extend the object model by invariants and refine the interaction diagrams with conditions on the messages.

To do this we model the use case as an interactive system consisting of an object of class `RecyclingMachine` (Figure 3 on the left). This class represents the recycling machine and has two attributes storing the daily total and the current list of items. For simplicity of presentation both attributes are considered as lists of items. The interaction diagram (Figure 3 on the right) shows (abstractly) the interaction between the customer (represented by the system border) and the recycling machine (the system). The customer sends a return message containing a list of returned concrete items. The machine prints a receipt with the list of (descriptions of) the returned items as well as the total return sum in Yen. To distinguish between the concrete items and their descriptions in the machine we call the sort of lists of concrete items `CList` and the other `IList`.

The following invariant is appropriate for `RecyclingMachine`: All items of `current` have also to be in `total`. This invariant is easily expressed in the Object Constraint Language (OCL) of UML, but any other formal language that can be translated to CafeOBJ would do. We attach it to the object model.

In the same way we extend the interaction diagram in order to express semantic relationships of the message parameters. We specify not only the parameter sorts of messages, but introduce formal parameter names and state the relationships between the formal parameters in an additional "where clause". In our example, we add a formula requiring that the item list in parameter `l` and the amount `s` of the `print` message are a list of descriptions of the concrete items in parameter `cl` of message `return` and the sum of the prices in `l`, respectively. For this purpose, we introduce two abstract functions `desclist` and `amount` that have to be implemented later on.

Construction of a formal specification. Next we show how one can construct semi-automatically a formal specification of the use case from the diagrams. The object model generates the class declarations and invariants; by a combination of the object model with the interaction diagram one can construct automatically a set of (incomplete) rewrite rules which after completion

(by hand) define the dynamic behaviour of the use case. These rewrite rules work on multisets of objects and messages, the so-called `ACZ-Configurations` of CafeOBJ. Changes of configurations by rewrite rules form the computational model of the specification.

The first step is to provide functional specifications for all sorts, data types, and functions used in the diagrams. This has to be done by hand. The specifications may be constructed reusing predefined modules from a specification library such as `NAT` and `LIST` or designing a completely new specification.

The following specification of items is new. It introduces two sorts `CItem` and `Item` denoting the "concrete" items of the user and the descriptions of these items. The operation `desc` yields the description of a concrete item whereas the operation `price` computes the price whose value will be given in `Yen`.

```
module ITEM {
  imports {
    protecting (YEN)
  }

  signature {
    [ Item CItem ]

    op price : Item -> Yen
    op desc : CItem -> Item
  }
}
```

The specification of item lists is obtained by instantiating a list module (which we assume to be provided in a library) twice, once with concrete items for elements and once with items; in both cases we rename the sort `List`. Moreover, we need two more operations mentioned in the "where clause" of the interaction diagram: `amount(l)` calculates the sum of the prices of the elements of `l` and `desclist(cl)` converts any "concrete" list `cl` into a list of descriptions.

```
module ITEMLIST {
  import {
    protecting (YEN)
    protecting (ITEM)
    protecting (LIST[ITEM { sort Elt -> CItem }] *
                  { sort List -> CList })
    protecting (LIST[ITEM { sort Elt -> Item }] *
                  { sort List -> IList })
  }

  signature {
    op desclist : CList -> IList
    op amount : IList -> Yen
  }

  axioms {
    var c : CItem
    vars cl cl' : CList
    var i : Item
    vars il il' : IList
```

```
    eq desclist((nil):CList) = (nil):IList .
    eq desclist(c) = desc(c) .
    cq desclist(cl, cl') = desclist(cl), desclist(cl')
         if (cl =/= (nil):CList) and (cl' =/= (nil):CList) .
    eq amount((nil):IList) = 0 .
    eq amount(i) = price(i) .
    cq amount(il, il') = amount(il) + amount(il')
         if (il =/= (nil):IList) and (il' =/= (nil):IList) .
  }
}
```

As a second step we automatically construct class descriptions from the object models and interaction diagrams:

- Every object model and every interaction diagram induces a set of class declarations:
  - Each object name C with attributes $a_1, \ldots, a_n$ of types $s_1, \ldots, s_n$ of the diagram represents a class declaration. Such a class additionally has attributes of sort `ObjectId` for each object a class instance sends a message to in any interaction diagram.

    `class` C `{` a_1 `:` s_1 $\ldots$ a_n `:` s_n o_1 `:` `ObjectId` $\ldots$ `}`

  - Each inheritance relation from D to C corresponds to an extension of the class declaration of D by a subclass declaration

    `class` D `[` C $\ldots$ `]` `{` $\ldots$ `}`

- The interaction diagram induces a set of message declarations:

  Each message $m(v_1 : s_1, \ldots, v_n : s_n)$ from one object to another object induces the message declaration

  `op` m `:` `ObjectId` s_1 $\ldots$ s_n `ObjectId -> Message`

  The first argument of m indicates the sender object, the last argument the destination. Messages that involve only one object, i.e. messages to or from the system border, are treated analogously.

For the recycling machine we get:

```
class RecyclingMachine {
  total : IList
  current : IList
}

op return : CList ObjectId -> Message
op print : ObjectId IList Yen -> Message
```

- Both diagrams generate the skeleton of a rule:

For any message $m(\ldots v_j : s_j \ldots)$ from E_0 to C of the interaction diagram, let

$$\texttt{class}\ C\ \texttt{[}\ldots\texttt{]}\ \texttt{\{}\ \ldots a_i : s_i \ldots\ \texttt{\}}$$
$$\texttt{class}\ E_0\ \texttt{[}\ldots\texttt{]}\ \texttt{\{}\ \ldots b_i : s'_i \ldots\ \texttt{\}}$$

be the corresponding class declarations, $m_k(\ldots w_{kj} : s_{kj} \ldots)$ for $1 \leq k \leq n$, be the outgoing messages from C to class E_k of the same activity below m before another message is received by C (if any) and Π the "where clause" of the diagram. Then we obtain the following skeleton of a rewrite rule:

$$\begin{aligned}&\texttt{crl [}m\texttt{]}\ m(o_0\texttt{ , }\ldots\texttt{ , }v_j\texttt{ , }\ldots\texttt{ , }o)\\ &\qquad\texttt{< }o\texttt{ : }C\texttt{ | }\ldots\ a_i\texttt{ = }w_i\ \ldots\texttt{ > =>}\\ &\qquad\quad\texttt{< }o\texttt{ : }C\texttt{ | }\ldots\ a_i\texttt{ = ? }\ldots\texttt{ >}\\ &\qquad\quad m_1(o\texttt{ , }\ldots\texttt{ , }w_{1j}\texttt{ , }\ldots\texttt{ , }o_1)\ \ldots\\ &\qquad\quad m_n(o\texttt{ , }\ldots\texttt{ , }w_{nj}\texttt{ , }\ldots\texttt{ , }o_n)\\ &\qquad\quad\texttt{if }\Pi\texttt{ and }\Pi\texttt{? .}\end{aligned}$$

where $o_0, \ldots, o_n$ are object identifiers (for the classes $E_0, \ldots, E_n$). This schema has to be adapted for messages that come from or go to the system border.

Note that this rule schema introduces new variables on the right hand side of the rule, which is not allowed in CafeOBJ. In the actual implementation these new variables have to be substituted suitably by expressions from Π or Π? (by this substitution Π may vanish).

The rule skeleton expresses that if the object o receives the message m it sends the messages $m_1, \ldots, m_n$. The question marks ? on the right hand side of the rule indicate that the resulting state of o is not expressed in the diagram. Therefore the new values of the attributes have to be added by hand. Similarly, Π? states that the condition is perhaps under-specified.

For example, the diagrams of Figure 3 induce the following skeleton:

```
crl [return]:
  return(cl, rm)
  < rm : RecyclingMachine | (total = t), (current = c) > =>
    < rm : RecyclingMachine | (total = ?), (current = ?) >
    print(rm, desclist(l), amount(desclist(l), (nil):IList))
    if Π? .
```

To get the complete rule one has to fill the question marks with the appropriate values `(t, desclist(cl))` and `desclist(cl)`.

The third step of the specification construction process consists of translating the control strategy given by the interaction diagrams, that is the prescribed order of message flow. For this translation we use the intermediate language of process expressions [2] with message names as atomic processes. We provide a general scheme to constrain the applicability of rewrite rules by a given process expression.

This separation of logic and control yields an easy way to examine different views of interaction diagrams. In Jacobson's original interpretation every message generates a response. As suggested by the asynchronous arrows of UML we

used in our interaction diagrams, we prefer to state return messages explicitly. Both views are easily expressed as processes.

The abstract syntax of processes we use is given by:

$$\begin{aligned} &\mathrm{A} ::= \surd \mid \delta \mid m \\ &\mathrm{P} ::= \mathrm{A} \mid \mathrm{P} \,;\, \mathrm{P} \mid \mathrm{P} + \mathrm{P} \mid \mathrm{P} \parallel \mathrm{P} \mid \mathrm{P}^* \end{aligned}$$

An atomic process is a message name m (occuring in an interaction diagram), or a constant $\surd$ denoting successful termination, or a constant δ for deadlock.

A composite process may be an atomic process, sequential composition, nondeterministic choice, parallel composition of processes, or a repeat statement. We adopt the usual precedence conventions on process expressions.

Processes are assumed to satisfy the following laws (borrowed from process algebra, see [2]).

$$\begin{aligned} &\surd \,;\, p = p, \qquad p \,;\, \surd = p, \qquad \delta \,;\, p = \delta, \\ &p_1 \,;\, (p_2 \,;\, p_3) = (p_1 \,;\, p_2) \,;\, p_3, \\ &\delta + p = p, \\ &p_1 + p_2 = p_2 + p_1, \qquad p_1 + (p_2 + p_3) = (p_1 + p_2) + p_3, \\ &(p_1 + p_2) \,;\, p_3 = p_1 \,;\, p_3 + p_2 \,;\, p_3, \\ &\surd \parallel p = p, \qquad \delta \parallel p = \delta, \\ &p_1 \parallel p_2 = p_2 \parallel p_1, \qquad p_1 \parallel (p_2 \parallel p_3) = (p_1 \parallel p_2) \parallel p_3, \\ &(p_1 + p_2) \parallel p_3 = (p_1 \parallel p_3) + (p_2 \parallel p_3), \\ &(m_1 \,;\, p_1) \parallel (m_2 \,;\, p_2) = m_1 \,;\, (p_1 \parallel (m_2 ; p_2)) + m_2 \,;\, ((m_1 \,;\, p_1) \parallel p_2) \\ &p^* = (p \,;\, p^*) + \surd \end{aligned}$$

Note that the last equation for parallel composition induces an interleaving approach to concurrency: either m_1 or m_2 has to be executed first.

We say that a run of a specification satisfies a process expression if, roughly speaking, the sequence of consumed messages is a trace of the process expression. Informally this means that a rewrite rule consuming a message m may only be applied if m is accepted by the process expression. For example, an application of the rewrite rule `return` is only allowed for a given process expression if this process accepts the message `return` which is consumed by the rule `return`. For the details of the technically more involved precise definition see [17].

Interaction diagrams are easily translated to process expressions as follows:

- The interaction diagram defines a control strategy which is based on the assumption that the objects of the diagram are controlled by (sequential) processes which are composed in parallel:

  For each object the incoming messages are sequentially composed from top to bottom; if a message block is part of a loop, the translated block is surrounded by a repeat statement. These object behaviours are composed in parallel.

The control strategy interprets the vertical axis as time: the messages have to occur at one object in the defined order. The different objects may act in

parallel, controlled by this protocol. The emergence of new messages is left to the object.

In the recycling machine example, the interaction diagram defines the following (trivial) control strategy

```
return
```

Constraints by a process expression can be implemented in CafeOBJ by a specification of processes with two functions on processes $hd : \mathrm{P} \to \wp((A)$ and $tl : \mathrm{A} \times \mathrm{P} \to \mathrm{P}$ that compute the accepted atomic processes for an arbitrary process expression and its behaviour after an atomic process has been executed, respectively.

```
module PROCESS[A :: ACTION] {
  protecting (SET[A { sort Elt -> Action }] *
               { sort Set -> Set<Action> })

  signature {
    [ Action < Process ]

    op terminate : -> Action
    op deadlock : -> Action
    op _;_ : Process Process -> Process
    op _+_ : Process Process -> Process
    op _||_ : Process Process -> Process
    op _* : Process -> Process
    op hd : Process -> Set<Action>
    op tl : Action Process -> Process
    ...
  }

  axioms {
    eq hd(terminate) = terminate .
    eq hd(deadlock) = empty .
    eq hd(a:Action) = a .
    cq hd(p1:Process ; p2:Process) = hd(p1)
       if (not (terminate in hd(p1))) .
    cq hd(p1:Process ; p2:Process) = (hd(p1) - terminate) hd(p2)
       if (terminate in hd(p1)) .
    eq hd(p1:Process + p2:Process) = hd(p1) hd(p2) .
    cq hd(p1:Process || p2:Process) = (hd(p1) hd(p2)) - terminate
       if (hd(p1) =/= empty) and (hd(p2) =/= empty) and
          (not ((terminate in hd(p1)) and (terminate in hd(p2)))) .
    cq hd(p1:Process || p2:Process) = hd(p1) hd(p2)
       if (terminate in hd(p1)) and (terminate in hd(p2)) .
    cq hd(p1:Process || p2:Process) = empty
       if (hd(p1) == empty) or (hd(p2) == empty) .
    eq hd(p:Process *) = hd(p) terminate .
    ...
  }
}
```

This specification of processes is used by a module `CONTROL` that provides a class `Control`. This class has an attribute `process` that holds the actual process

expression a run of a specification must satisfy. Since we can not use message names themselves as atomic processes we use strings instead.

```
module CONTROL {
  imports {
    protecting (ACZ-CONFIGURATION)
    protecting (PROCESS[STRING { sort Action -> String }])
  }

  signature {
    class Control {
      process : Process
    }

    op control : -> Process
  }
}
```

Each rewrite rule of the specification to be governed by a process expression is translated as follows: A rule

```
crl [r] m o => c if Π .
```

is replaced by

```
crl [r] m o < cntrl :  Control | process = p > =>
          c < cntrl :  Control | process = tl(m, Q) >
          if m in hd(p) and Π .
```

For the recycling machine, the full specification of the use case "return items" reads as follows:

```
module RM {
  imports {
    protecting (CONTROL)
    protecting (ITEMLIST)
  }

  signature {
    class RecyclingMachine {
      total : IList
      current : IList
    }

    op return : CList ObjectId -> Message
    op print : ObjectId IList Yen -> Message
  }

  axioms {
    var p : Process
    vars cntrl rm : ObjectId
    var cl : CList
    vars t c : IList
    var s : Yen
```

```
    crl [return]:
      return(cl, rm)
      < cntrl : Control | process = p >
      < rm : RecyclingMachine | (total = t), (current = c) > =>
        < cntrl : Control | process = tl("return", p) >
        < rm : RecyclingMachine | (total = t, desclist(cl)),
                                  (current = desclist(cl)) >
          print(rm, desclist(cl), amount(desclist(cl), (nil):IList))
        if ("return" in hd(p)) .

    eq control = "return" .
  }
}
```

A sample run. We test the specification with a start configuration where the user returns three bottles.

```
module RM1 {
  imports {
    protecting (RM)
  }

  signature {
    op ret1 : -> CItem
    op ret2 : -> CItem
    op ret3 : -> CItem

    op bottle : -> Item
  }

  axioms {
    eq price(bottle) = 100 .
    eq desc(ret1) = bottle .
    eq desc(ret2) = bottle .
    eq desc(ret3) = bottle .
  }
}

select RM1

exec makeControl(Cntrl, (process = control)) .
exec makeRecyclingMachine(Rm, (total = nil), (current = nil)) .

exec < Cntrl > < Rm > return(ret1, ret2, ret3, (nil):CList, Rm) .
```

CafeOBJ answers

```
-- execute in RM1 : < Cntrl > < Rm > return(ret1 , ret2 , ret3 ,
    nil,Rm)
< Cntrl : Control | (process = terminate) > < Rm : RecyclingMachine
  | (total = bottle , bottle , bottle , current = bottle , bottle
  , bottle) > print(Rm,bottle , bottle , bottle,300) : ACZ-Configuration
(0.010 sec for parse, 113 rewrites(0.230 sec), 424 match attempts)
```

1.3 Robustness Analysis

Use case. The use case "return items" is refined in two aspects: instead of returning a list of items the customer returns the items one by one; the machine itself is decomposed into several objects. Accordingly, the informal description consists of a refinement of the use case description of Section 1.2 and a description of the objects of the machine:

> "A recycling machine receives returning items (such as cans or bottles) from a customer. Descriptions of these items and the daily total of the returned items of all customers are stored in the machine. If the customer presses the start button he can return the items one by one. If the customer presses the receipt button he gets a receipt for all items he has returned before. The receipt contains a list of the returned items as well as the total return sum."

The second phase of OOSE, called "robustness analysis", deals with such a refinement.

Object model. Objects are classified in three categories: interface, control and entity objects. Interface objects build the interface between the actors (the system border) and the system, the entity objects represent the (storable) data used by the system and the control objects are responsible for the exchange of information between the interface and the entity objects.

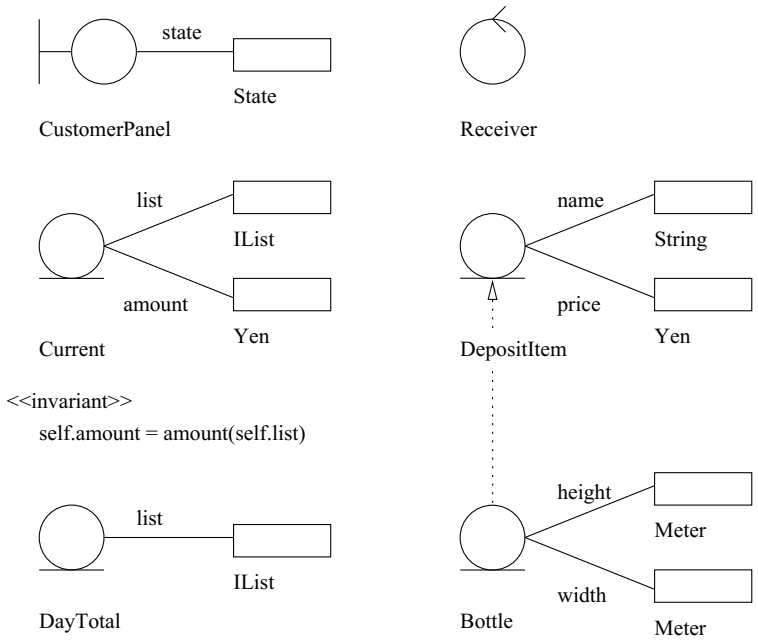

Figure 4: Object model of the robustness analysis for the Recycling Machine

The recycling machine consists of five objects (sorts): the interface object `CustomerPanel`, a control object `Receiver` and the entity objects `Current`, `DayTotal` and `DepositItem`. `CustomerPanel` and `Receiver` communicate the

data concerning the returned items, the `Receiver` uses `Current` and `DayTotal` for storing and computing the list of current items and the daily total. `DepositItem` stands for all kinds of returned items, in particular for the class of bottles which is modeled as its heir (see Figure 4).

The objects have the following attributes: the `CustomerPanel` has a state attribute to record whether the start button was pressed; the `Receiver` has no attributes; `DepositItem` has a name and a price, `Bottle` has additionally a height and a width; the class `Current` has a list (of `DepositItem`) and an amount as attributes, `DayTotal` a list of deposit items.

The attributes of `Current` satisfy the invariant that the amount is the sum of the prices of the items of the list.

Interaction diagram. From the informal description one can derive three kinds of messages which are sent from the system border (i.e. from the customer) to the `CustomerPanel`: a `start` message, a `return` message for returning one concrete item and a `receipt` message for requiring a receipt. Each of these messages begins a new activity of the customer panel. On the other hand, the customer panel sends a print message to the system border.

The `start` message concerns only the `CustomerPanel`. After receipt of the `return` message the customer panel sends a message, say `new(i)`, with the description `i` of the concrete item to the receiver. Then the receiver forwards this information to `Current` and `DayTotal` by two messages, called `add` and `conc` respectively; the end of such a return process is to be acknowledged by a message `ack`. In the third activity the `CustomerPanel` sends a `printreceipt` request to the `Receiver` which in turn sends a standard `get` message to `Current`. After getting the answer the `Receiver` forwards the answer to the `CustomerPanel` (by a message called `send`) which prints the result (see Figure 5).

The "where clause" of the diagram shows that the description `i` of a returned item `ci` is not changed and that the amount of the print message is compatible with the prices of the returned items.

Construction of a Formal Specification. We add a functional specification for the states of the customer panel.

```
module STATE {
  signature {
    [ State ]

    ops on, off, wait : -> State
  }
}
```

The refined diagram generates automatically six class and twelve message declarations according to our method in Section 1.2, e.g.

```
    class Receiver {
      customerpanel : ObjectId
      current : ObjectId
      daytotal : ObjectId
    }
```

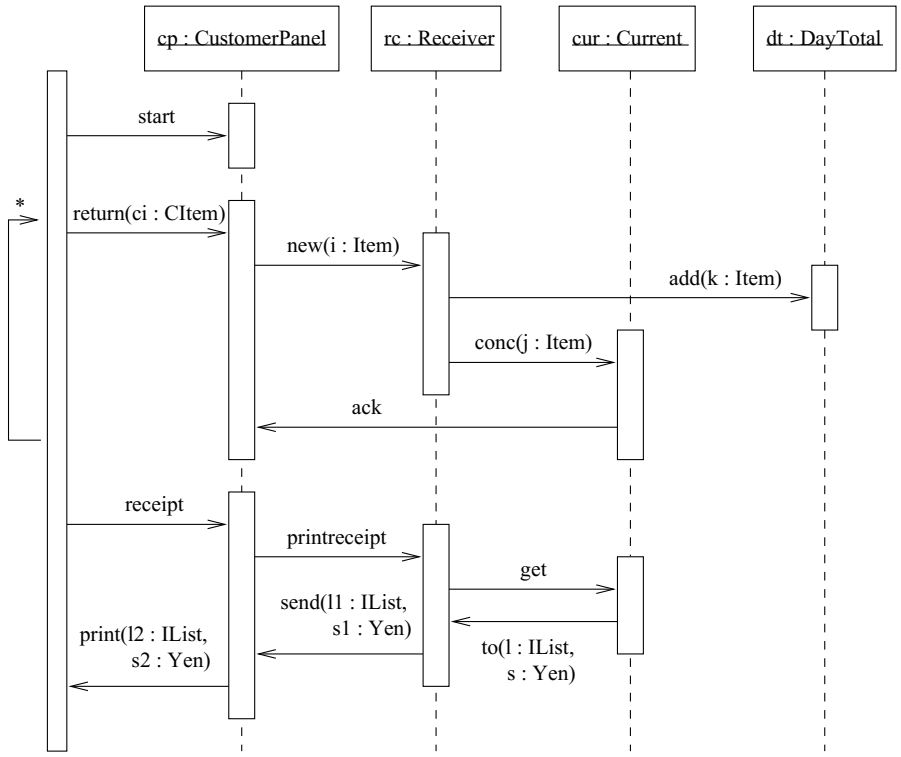

Figure 5: Interaction diagram of the robustness analysis for the Recycling Machine

```
op start : ObjectId -> Message
op return : ObjectId ObjectId -> Message
op new : ObjectId ObjectId ObjectId -> Message
```

In order to define the rule skeletons for the interaction diagram of the recycling machine we use the attributes defined in the object model (see Figure 4). We obtain the following skeletons for e.g. the `start`, `return` and `new` message:

```
crl [start] start(cp)
            < cp : CustomerPanel | state = a > =>
              < cp: CustomerPanel | state = ? >
              if Π?
crl [return] return(ci, cp)
               < cp : CustomerPanel | state = a > =>
                 < cp : CustomerPanel | state = ? >
                 new(cp, i, rc)
               if i = desc(ci) and Π?
[new] new(cp, i, rc)
      < rc : Receiver > =>
        < rc : Receiver >
        conc(rc, i, cur)
```

```
add(rc, i, dt)
  if Π?
```

The behaviour of the interaction diagram is represented by the following strategy:

$$\begin{array}{l} \texttt{start} ; (\texttt{return})^* ; \texttt{ack} ; \texttt{receipt} ; \texttt{send} \\ \| \, (\texttt{new})^* ; \texttt{printreceipt} ; \texttt{to} \\ \| \, (\texttt{conc})^* ; \texttt{get} \\ \| \, (\texttt{add})^* \end{array}$$

To get the full rules one has to add the state changes and the necessary preconditions. We require preconditions only for the behaviour of the customer panel: pressing the start button should actually start the machine only if it is in state off, returning an item and requiring a receipt should be possible only if the machine is on. By filling in values also for the other question marks we obtain the following rules:

```
rl [start]: start(cp)
            < cp : CustomerPanel | (state = off) > =>
              < cp : CustomerPanel | (state = on) > .
rl [return]: return(ci, cp)
             < cp : CustomerPanel | (state = on) > =>
               < cp : CustomerPanel | (state = on) >
               new(cp, desc(< ci : CItem >), rc) .
rl [new]: new(cp, i, rc)
          < rc : Receiver | (current = cur), (daytotal = dt) > =>
            < rc : Receiver | (current = cur), (daytotal = dt) >
            conc(rc, i, cur)
            add(rc, i, dt) .
```

A sample run. The whole specification with integrated process control can now be tested. We give a run for the returning of three bottles.

```
exec makeCustomerPanel(Cp, (state = off), (receiver = Rc)) .
exec makeReceiver(Rc, (customerpanel = Cp), (current = Cur),
                      (daytotal = Dt)) .
exec makeCurrent(Cur, (list = nil), (amount = 0),
                      (receiver = Rc), (customerpanel = Cp)) .
exec makeDayTotal(Dt, (list = nil)) .
exec makeItem(Bottle, (price = 100)) .
exec makeCItem(Ret1, (desc = Bottle)) .
exec makeCItem(Ret2, (desc = Bottle)) .
exec makeCItem(Ret3, (desc = Bottle)) .

exec makeControl(Cntrl,
       (process =    ("start" ; (("return" ; "ack") * ;
                      ("receipt" ; "send")))
                  || ((("new") * ; ("printreceipt" ; "to"))
                  || ((("conc") * ; "get")
                  || ("add" *))))) .

exec < Cntrl > < Cp > < Rc > < Cur > < Dt >
     return(Ret3, Cp) return(Ret2, Cp) return(Ret1, Cp) start(Cp) .
```

```
exec < Cntrl > < Cp > < Rc > < Cur > < Dt >
     receipt(Cp) .
```

CafeOBJ answers:

```
-- execute in RM : < Cntrl > < Cp > < Rc > < Cur > < Dt > return(
    Ret3,Cp) return(Ret2,Cp) return(Ret1,Cp) start(Cp)
< Rc : Receiver | (current = Cur , daytotal = Dt , customerpanel
  = Cp) > < Cur : Current | (list = Bottle , Bottle , Bottle , amount
  = 300 , customerpanel = Cp , receiver = Rc) > < Dt : DayTotal |
   (list = Bottle , Bottle , Bottle) > < Cntrl : Control | (process
  = ((("return" ; "ack") *) ; ("receipt" ; "send")) || ((("new" *
  ) ; ("printreceipt" ; "to")) || ((("conc" *) ; "get") || ("add"
  *)))) > < Cp : CustomerPanel | (state = on , receiver = Rc) > :
  ACZ-Configuration
(23.350 sec for parse, 418988 rewrites(374.290 sec),
                                                  3343231 match attempts)

-- execute in RM : < Cntrl > < Cp > < Rc > < Cur > < Dt > receipt(
    Cp)
< Dt : DayTotal | (list = Bottle , Bottle , Bottle) > < Cur : Current
  | (list = nil , amount = 0 , receiver = Rc , customerpanel = Cp)
   > < Rc : Receiver | (customerpanel = Cp , current = Cur , daytotal
  = Dt) > < Cntrl : Control | (process = "add" *) > < Cp : CustomerPanel
  | (state = off , receiver = Rc) > print(Cp,l,300) : ACZ-Configuration
(1.430 sec for parse, 40301 rewrites(51.250 sec), 311025 match attempts)
```

Another interpretation of interaction diagrams. Because of our separation of logic and control we can switch between different interpretations of the control flow of messages given by the interaction diagrams. The following is an alternative view: The vertical axis is interpreted causally: A message m to an object o is called the reason of all messages emerged by o between the reception of m and the reception of any other message after (below) m. A message must occur before any other message it is the reason for.

For the recycling machine this yields the following process expression:

$$\begin{array}{l}\mathtt{start}\,;\\ (\mathtt{return}\,;\mathtt{new}\,;(\mathtt{add}\parallel\mathtt{conc})\,;\mathtt{ack})^*\\ \mathtt{receipt}\,;\mathtt{printreceipt}\,;\mathtt{get}\,;\mathtt{to}\,;\mathtt{send}\end{array}$$

In CafeOBJ we get the following behaviour:

```
exec makeControl(Cntrl, (process = "start" ;
                                  ((("return" ; ("new" ;
                                   (("add" || "conc") ; "ack"))) *) ;
                                  ("receipt" ; ("printreceipt" ;
                                   ("get" ; ("to" ; "send")))))))) .

-- execute in RM : < Cntrl > < Cp > < Rc > < Cur > < Dt > receipt(
    Cp)
< Dt : DayTotal | (list = Bottle , Bottle , Bottle) > < Cur : Current
  | (list = nil , amount = 0 , receiver = Rc , customerpanel = Cp)
   > < Rc : Receiver | (customerpanel = Cp , current = Cur , daytotal
 = Dt) > < Cntrl : Control | (process = terminate) > < Cp : CustomerPanel
```

```
  | (state = off , receiver = Rc) > print(Cp,1,300) : ACZ-Configuration
(1.420 sec for parse, 533 rewrites(6.670 sec), 3305 match attempts)
```

Specification metrics. The result for the recycling machine example are two specifications with about 100 lines in four modules and 200 lines in five modules, respectively. An additional library containing `PROCESS` and `CONTROL` comprises about 150 lines in five modules. The parsing time for the specification of the requirements analysis is about four seconds, for the specification of the robustness analysis about seven seconds.

2 Formalising Java Semantics

Java offers simple and tightly integrated support for concurrent programming. A concurrent program consists of multiple tasks that are or behave as if they were executed all at the same time. In Java tasks are implemented using *threads*, which are sequences of instructions that run independently within the encompassing program.

A structural operational semantics of a non-trivial sublanguage of Java including threads and their synchronisation was given in [7]. The interaction between threads via shared memory is described in terms of structures called *event spaces*. These provide an abstract, declarative description of the Java thread model which is an exact formal counterpart of the informal language description [11, Chapter 17]. The semantics is presented in two steps: First a simple operational description of the sequential part of Java is introduced. Then the thread model is developed.

In this section we are formalising this semantics in CafeOBJ following the structure of the original presentation in [7].

The following part of the concrete Java syntax is covered by the semantics:

Block	::=	{ *BlockStatement** }
BlockStatement	::=	*LocalVariableDeclaration* | *Statement*
LocalVariableDeclaration	::=	*Type VariableDeclarator*$^+$
VariableDeclarator	::=	*Identifier* = *Expression*
Statement	::=	; | *Block* | *ExpressionStatement*;
		| `synchronized(`*Expression*`)` *Block*
ExpressionStatement	::=	*Assignment* | `new` *ClassType* `()`
Assignment	::=	*LeftHandSide* = *AssignmentExpression*
LeftHandSide	::=	*Name* | *FieldAccess*
Name	::=	*Identifier* | *Name* . *Identifier*
FieldAccess	::=	*Primary* . *Identifier*
AssignmentExpression	::=	*Assignment* | *UnaryExpression*
UnaryExpression	::=	*UnaryOperator UnaryExpression*
		| *Primary* | *Name*
Primary	::=	*Literal* | `this` | *FieldAccess*
		| `(` *Expression* `)` | `new` *ClassType* `()`
Expression	::=	*AssignmentExpression*

2.1 Sequential Java

We first briefly summarise the semantic domains used in the operational semantics of sequential Java and their formalisation. Then we introduce the abstract syntax for Java terms. Finally, the operational rules and their translation to CafeOBJ are presented. We illustrate some terminology and our approach by means of the following Java program:

```
class Point {
  int x, y;
  Point() { }
}

class Main {
  public static void main(String[] argv) {
    Point p = new Point();
    p.x = 1;  p.y = 2;
    p.x = p.y;
  }
}
```

Semantic domains. Objects, for example an instance of class `Point`, are kept in the main memory. This store can be thought of as mapping addresses of instance variables (left-values, from a semantic domain *LVal*), e.g. of instance variable `p.x`) of objects (from a semantic domain *Obj*, e.g. referenced by `p`) to values or object references (right-values, from a semantic domain *RVal*, e.g. 2). A semantic domain *Store* is assumed equiped with five semantic functions: upd : $LVal \times RVal \times Store \to Store$ updates a given store; $upd(l, v, \mu)$ is written $\mu[l \mapsto v]$. The function $lval$: $Obj \times Identifier \times Store \rightharpoonup LVal$ retrieves the left-value of an instance variable (such as of `p.x`). Analogously, $rval$: $LVal \times Store \rightharpoonup RVal$ retrieves the right-value of a left-value (for example 2 for `p.y`); $rval(l, \mu)$ is written $\mu(l)$. New objects are allocated by new_C : $Store \to Obj \times Store$. This family of functions is indexed by class types $C \in ClassType$. For a given store μ, $new_C(\mu)$ yields an object o and a store μ' such that μ' extends μ where $\mu'(lval(o, i, \mu'))$ is defined for any identifier i ranging over all instance variables of the class C.

The store is straightforwardly formalised in module `STORE`. Note that, since the semantics does not deal with class declarations, classes (in our example `Point` and `Main`) have to be coded in the specification by hand.

```
module* DECLARATIONS {
  signature {
    [ Bool Int < Literal,
      ObjectId < Obj,
      Literal Obj < RVal,
      Ident, ClassType < Type ]

    op void : -> RVal
    op null : -> Obj

    class Main {
    }
```

```
    class Point {
      x : Int
      y : Int
    }
    ...
  }
  ...
}

module* STORE {
  imports {
    protecting (DECLARATIONS)
    ...
  }

  signature {
    [ LVal, Store ]

    op new : ClassType Store -> Pair<Obj;Store>
    op upd : LVal RVal Store -> Store
    op lval : Obj Ident Store -> LVal
    op rval : LVal Store -> RVal
    op init : Obj Set<Ident> -> Obj { strat: (1 2 0) }

    op _[_|->_] : Store LVal RVal -> Store
    op __ : Store LVal -> RVal
    ...
  }
  ...
}
```

The local variables of a block are kept in a stack of environments. Stacks of the domain *S-Stack* are ranged over by σ, environements of the domain *Env* are ranged over by ρ. We omit the straightforward definitions for these domains and their formalisation.

Abstract Syntax. The operational semantics works on a set *S-Term* of *abstract terms*. Let the metavariable t range over *S-Term*. To each syntactic category of Java a homonymous category of abstract terms is associated. The well-typed terms of Java are mapped to abstract terms of corresponding category by a translation $(\_)^\circ$, which is left implicit when no confusion arises. Abstract blocks are terms of the form $\{\, t \,\}_{(I,\rho)}$ where the source I of the environment (I, ρ) contains the local variables of the block.

In our formalisation in CafeOBJ, the abstract syntax for the the block of the `main()` method of the example above looks as follows:

```
{ (((Point (p =' (new Point ( )))) ;)
   ((((p . x) = 1) ;) ((((p . y) = 2) ;)
   ((((p . x) = (p . y)) ;) nil)))) } ^ (undefined [ p |-> null ])
```

The different syntactical categories are modeled by sorts and subsorting. For example, the grammar rule

$$\mathit{LeftHandSide} ::= \mathit{Name} \mid \mathit{FieldAccess}$$

is modeled by the sort declaration

```
[ Name FieldAccess < Name^FieldAccess < LeftHandSide ] .
```

The symbol `='` is the operator for variable declarations. The operator `undefined` represents the empty environment. A block $\{\ldots\}_\rho$ is written `{ ... }^`ρ. We use an explicit terminator `nil` for lists of block statements.

Operational judgements. The *configurations* of sequential Java are triples (t, σ, μ) consisting of an S-term t, an S-stack σ, and a store μ. The operational semantics is the binary relation $\longrightarrow$ on configurations inductively defined by the rules in Tables 1–3. In the rule schemes, the metavariables range as follows: $i \in$ *Identifier*, $k \in$ *Identifier* $\cup$ *LVal*, $l \in$ *LVal*, $o \in$ *Obj*, $e \in$ *Expression*, $v \in$ *RVal*, $s \in$ *Statement*, $b \in$ *BlockStatement*, $B \in$ *BlockStatement**, $q \in$ *Block*, and $t \in$ *S-Term*. Stacks and stores are *omitted* when they are not relevant.

Configurations can be represented directly in CafeOBJ:

```
make S-TERM (ABSTRACTSYNTAX *  sort AbstractSyntax -> S-Term )

module! S-CONFIGURATION {
  imports {
    protecting (TRIPLE(S-TERM { sort Elt -> S-Term },
                       S-STACK { sort Elt -> S-Stack },
                       STORE { sort Elt -> Store }) *
                  { sort Triple<X;Y;Z> -> Conf })
  }

  signature {
    [ Conf ]
    ...
```

For the formalisation of the rules, observe that the language specification requires sequential Java to be a deterministic language, i.e. for each configuration there is at most one legal successor configuration. We can therefore operationalise the semantics by introducing a successor function that calculates the immediate or the n-th successor of a given configuration.

```
    op step : Conf -> Conf
    op step : Conf Nat -> Conf
  }
```

There is a minor obstacle in directly formalising the rules with such a step function. Suppose we had a rule like

```
    var ss : S-Stack
    var mm : Store
    vars h h1 : LeftHandSide
    vars ae ae1 : AssignmentExpression

    eq [assign1]:
      step(<< (h1 = ae), ss, mm >>) =
        assignment-left(step(<< (h1), ss, mm >>), ae) .

    eq assignment-left(<< h, ss, mm >>, ae) = << (h = ae), ss, mm >> .
```

[assign1]	$\dfrac{e_1 \longrightarrow e_2}{e_1 = e \longrightarrow e_2 = e}$	[assign2]	$\dfrac{e_1 \longrightarrow e_2}{k = e_1 \longrightarrow k = e_2}$
[assign3]	$l = v, \mu \longrightarrow v, \mu[l \mapsto v]$	[assign4]	$i = v, \sigma \longrightarrow v, \sigma[i \mapsto v]$
[unop1]	$\dfrac{e_1 \longrightarrow e_2}{\texttt{op}(e_1) \longrightarrow \texttt{op}(e_2)}$	[unop2]	$\texttt{op}(v) \longrightarrow op(v)$
[access1]	$\dfrac{e_1 \longrightarrow e_2}{e_1 \,.\, i \longrightarrow e_2 \,.\, i}$	[access2]	$o.i \,, \mu \longrightarrow lval(o, i, \mu), \mu$
[this]	$\texttt{this} \longrightarrow \sigma(this)$	[pth]	$(e) \longrightarrow e$
[new]	$\texttt{new}\ C\ (\,) \,, \mu \longrightarrow new_C(\mu)$	[val]	$l \,, \mu \longrightarrow \mu(l), \mu$
[var]	$i \,, \sigma \longrightarrow \sigma(i), \sigma$		

Table 1. Expressions

[decl1]	$\dfrac{e_1 \longrightarrow e_2}{i = e_1 \longrightarrow i = e_2}$	[decl2]	$i = v, \sigma \longrightarrow \sigma[i = v]$
[declseq1]	$\dfrac{d_1 \longrightarrow d_2}{d_1\, D \longrightarrow d_2\, D}$	[declseq2]	$\dfrac{d, \sigma_1 \longrightarrow \sigma_2}{d\, D, \sigma_1 \longrightarrow D, \sigma_2}$
[locvardecl1]	$\dfrac{D_1 \longrightarrow D_2}{\tau\ D_1 \longrightarrow \tau\ D_2}$	[locvardecl2]	$\dfrac{D, \sigma_1 \longrightarrow \sigma_2}{\tau\ D \,, \sigma_1 \longrightarrow \sigma_2}$

Table 2. Local variable declarations

[statseq1]	$\dfrac{s_1 \longrightarrow s_2}{s_1\, S \longrightarrow s_2\, S}$	[statseq2]	$\dfrac{s, \mu_1 \longrightarrow \mu_2}{s\, S, \mu_1 \longrightarrow S, \mu_2}$
[expstat1]	$\dfrac{e_1 \longrightarrow e_2}{e_1\ ; \longrightarrow e_2\ ;}$	[expstat2]	$\dfrac{e, \mu_1 \longrightarrow v, \mu_2}{e\ ; \,, \mu_1 \longrightarrow \mu_2}$
[skip]	$;\ , \sigma \longrightarrow \sigma$	[block1]	$\{\ \}_\rho \,, \sigma \longrightarrow \sigma$
[block2]	$\dfrac{S_1, push(\rho_1, \sigma_1) \longrightarrow S_2, push(\rho_2, \sigma_2)}{\{S_1\}_{\rho_1}, \sigma_1 \longrightarrow \{S_2\}_{\rho_2}, \sigma_2}$		

Table 3. Statements

where `assignment-left` is used to put the result of the evaluation in the premise of the rule back into the context. Due to this context, the result of `step(<< (h1), ss, mm >>)` has again to be in `LeftHandSide` which must be ensured by an additional condition.

We decided to analyse the left-hand sides of the rules' conclusions in more detail and to separate those subsorts the rules can safely apply to. For example, [assign1] should only apply if e_1 is neither in `Ident` nor in `LVal`. We avoid the introduction of new sorts like in a declaration

```
Ident^LVal LeftHandSide-Ident-LVal < LeftHandSide
```

(where `LeftHandSide-Ident-LVal` would represent left-hand sides of assignments that are not identifiers nor left-values). Instead, we define membership predicates like for example

```
pred is-ident^lval : AbstractSyntax
var k : Ident^LVal
eq is-ident^lval(k) = true .
```

The reduction of `is-ident^lval(t) == true` for a `t : AbstractSyntax` yields `true` if and only if `t : Ident^LVal`. With such predicates the rules can be formalised straightforwardly:

```
axioms {
  var ss : S-Stack
  var mm : Store
  var i : Ident
  var l : LVal
  var k : Ident^LVal
  var v : RVal
  vars h h1 : LeftHandSide
  vars ae ae1 : AssignmentExpression

  ceq [assign1]:
    step(<< (h1 = ae), ss, mm >>) =
        assignment-left(step(<< (h1), ss, mm >>), ae)
      if not (is-lefthandsidenf(h1) == true) .
  ceq [assign2]:
    step(<< (k = ae1), ss, mm >>) =
        assignment-right(step(<< (ae1), ss, mm >>), k)
      if not (is-assignmentexpressionnf(ae1) == true) .
  eq [assign3]:
    step(<< (l = v), ss, mm >>) = << (v), ss, mm [ l |-> v ] >> .
  eq [assign4]:
    step(<< (i = v), ss, mm >>) = << (v), ss [ i |-> v ], mm >> .

  eq assignment-left(<< h, ss, mm >>, ae) = << (h = ae), ss, mm >> .
  eq assignment-right(<< ae, ss, mm >>, h) = << (h = ae), ss, mm >> .

  eq is-lefthandsidenf(h) = is-ident^lval(h) == true .
  eq is-assignmentexpressionnf(ae) = is-rval(ae) == true .
  ...
}
}
```

The main block of our running example can now be executed:

```
let test = << ({ (((Point (p =' (new Point ( )))) ;)
                 ((((p . x) = 1) ;)
                 ((((p . y) = 2) ;)
                 ((((p . x) = (p . y)) ;) nil))))
               } ^ (undefined [ p |-> null ])),
               empty, empty >> .

exec step(test, 15) .
```

CafeOBJ answers

```
-- execute in S-CONFIGURATION : step(<< ({ (((Point (p =' new Point
    ( ))) ;) ((((p . x) = 1) ;) ((((p . y) = 2) ;) ((((p . x) = (
    p . y)) ;) nil)))) } ^ (undefined [ p |-> null ])) , empty ,
    empty >>,15)
<< * , empty , point-0 >> : Conf
(0.000 sec for parse, 973 rewrites(0.140 sec), 2539 match attempts)
```

The program has created a new instance `point-0` of class `Point`. We may inspect it with `red < point-0 >`:

```
-- reduce in S-CONFIGURATION : < point-0 >
< point-0 : Point | (x = 2 , y = 2) > : Point
(0.230 sec for parse, 1 rewrites(0.000 sec), 1 match attempts)
```

2.2 Event Spaces

The execution of a non-idealised, non-sequential Java program comprises many *threads* of computation running in parallel. Threads exchange information by operating on values and objects residing in a shared *main memory*. As explained in the Java language specification [11], each thread also has a private *working memory* in which it keeps its own working copy of variables that it must use or assign. As the thread executes a program, it operates on these working copies. The main memory contains the master copy of each variable. There are rules about when a thread is permitted or required to transfer the contents of its working copy of a variable into the master copy or vice versa. Moreover, there are rules which regulate the *locking* and *unlocking* of objects, by means of which threads synchronise with each other. These rules are given in [11, Chapter 17] and have been formalised in [7] as "well-formedness" conditions for structures called *event spaces*. Event spaces are included in the configurations of multi-threaded Java to constrain the applicability of certain operational rules. Additionally, they are used to model the working memories of all threads. We summarise the definition of event spaces and simultaneously formalise them in a CafeOBJ specification.

Actions and events. In accord with [11], the terms *Use*, *Assign*, *Load*, *Store*, *Read*, *Write*, *Lock*, and *Unlock* are used to name actions which describe the activity of the memories during the execution of a Java program. *Use* and *Assign* denote the above mentioned actions of the private working memory. *Read* and *Load* are used for a loosely coupled copying of data from the main

memory to a working memory and dually *Store* and *Write* are used for copying data from a working memory to the main memory.

Let *Thread_id* be a set of thread identifiers. An *action* is either a 4-tuple of the form (A, θ, l, v) where $A \in \{Assign, Store, Read\}$, $\theta \in$ *Thread_id*, $l \in$ *LVal* and $v \in$ *RVal*, or a triple (A, θ, l), where θ and l are as above and $A \in \{Use, Load, Write\}$, or a triple (A, θ, o), where $A \in \{Lock, Unlock\}$ and $o \in$ *Obj*.

Events are instances of actions, which can be thought of as happening at different times during execution. The same tuple notation for actions and their instances is used.

In CafeOBJ, events are most easily modeled as elements of `Object`; only their object identifiers will be kept in the formalised event spaces.

```
module! ACTION {
  imports {
    protecting (THREAD)
    protecting (STORE)
  }

  signature {
    [ Actions,
      LVal Obj < LocObjs ]

    class Action {
      action : Actions
      thread : ThreadId
    }

    class ASR-Action [ Action ] {
      location : LVal
      value : RVal
    }

    class ULW-Action [ Action ] {
      location : LVal
    }

    class ON-Action [ Action ] {
      object : Obj
    }

    ops read load use write store assign lock unlock : -> Actions

    op event : Actions ThreadId LVal -> ObjectId
    op event : Actions ThreadId LVal RVal -> ObjectId
    op event : Actions ThreadId Obj -> ObjectId
    ...
  }

  axioms {
    var a : Actions
    var t : ThreadId
    var l : LVal
    var v : RVal
    var o : Obj
```

```
    cq event(a, t, l, v) = oid(makeASR-Action(action = a, thread = t,
                                              location = l, value = v))
      if (a == assign) or (a == store) or (a == read) .
    cq event(a, t, l) = oid(makeULW-Action(action = a, thread = t,
                                           location = l))
      if (a == use) or (a == load) or (a == write) .
    cq event(a, t, o) = oid(makeON-Action(action = a, thread = t,
                                          object = o))
      if (a == lock) or (a == unlock) .
    ...
  }
}
```

An *event space* is a poset of events (thought of as occurring in the given order) in which every chain can be enumerated monotonically with respect to the arithmetical ordering $0 \leq 1 \leq 2 \leq \ldots$ of natural numbers, and which satisfies the conditions (17.2.1–17.6.2') of Table 4. These conditions, which formalise directly the rules of [11, Chapter 17], are expressed by clauses of the form:

$$\forall \vec{a} \in \eta \,.\, (\Phi \Rightarrow ((\exists \vec{b}_1 \in \eta \,.\, \Psi_1) \vee (\exists \vec{b}_2 \in \eta \,.\, \Psi_2) \vee \ldots \ (\exists \vec{b}_n \in \eta \,.\, \Psi_n)))$$

where $\vec{a}$ and $\vec{b}_i$ are lists of events, η is an event space and $\forall \vec{a} \in \eta \,.\, \Phi$ means that Φ holds for all tuples of events in η matching the elements of $\vec{a}$ (and similarly for $\exists \vec{b} \in \eta \,.\, \Psi$).

Such statements are abbreviated by adopting the following conventions: quantification over $\vec{a}$ is left implicit when all events in $\vec{a}$ appear in Φ; quantification over $\vec{b}_i$ is left implicit when all events in $\vec{b}_i$ appear in Ψ_i. Moreover, a rule of the form $\forall \vec{a} \in \eta \,.\, (\mathit{true} \Rightarrow \ldots)$ is written $\vec{a} \Rightarrow (\ldots)$. Furthermore, B ranges over the set of thread actions and C over the set of memory actions, that is: $B \in \{\mathit{Use}, \mathit{Assign}, \mathit{Load}, \mathit{Store}, \mathit{Lock}, \mathit{Unlock}\}$, $C \in \{\mathit{Read}, \mathit{Write}, \mathit{Lock}, \mathit{Unlock}\}$. Components of an action or event may be omitted: e.g. (Read, l) is written for $(\mathit{Read}, \theta, l, v)$ when θ and v are not relevant. The term $(A, \theta, x)_n$ denotes the n-th occurrence of (A, θ, x) in a given space, if such an event exists, and is undefined otherwise.

The origin of each rule from [11, Chapter 17] is included; for more details refer to [11] and [7]. For instance, rule (17.2.1) says that actions performed by any thread are totally ordered and (17.2.2) that so are the actions performed by the main memory for any variable or object. Similarly, rules (17.6.2) and (17.6.2') say that a lock action acts as if it flushes all variables from the thread's working memory, i.e. before use they must be assigned or loaded from main memory.

The formula scheme of the event space rules can be imitated in CafeOBJ by parameterisation. The module `RULE-BASE` provides the parameter specification with predicates `all-pred` representing Φ and `exists-pred` representing $\Psi_1, \ldots, \Psi_n$. The type of these predicates depends on whether quantification runs over event space elements, such as for example in (17.3.2), or over ordered pairs in event spaces, such as for example in (17.3.4). These dependent types could be avoided by specifying several different `RULE-BASE` and `RULE` modules.

```
module* RULE-BASE {
```

$$(B,\theta),(B',\theta) \Rightarrow (B,\theta) \leq (B',\theta) \vee (B',\theta) \leq (B,\theta) \quad (17.2.1)$$

$$(C,x),(C',x) \Rightarrow (C,x) \leq (C',x) \vee (C',x) \leq (C,x) \quad (17.2.2)$$

$$(Assign,\theta,l) \leq (Load,\theta,l) \Rightarrow (Assign,\theta,l) \leq (Store,\theta,l) \leq (Load,\theta,l) \quad (17.3.2)$$

$$(Store,\theta,l)_m < (Store,\theta,l)_n \Rightarrow (Store,\theta,l)_m \leq (Assign,\theta,l) \leq (Store,\theta,l)_n \quad (17.3.3)$$

$$(Use,\theta,l) \Rightarrow (Assign,\theta,l) \leq (Use,\theta,l) \vee (Load,\theta,l) \leq (Use,\theta,l) \quad (17.3.4)$$

$$(Store,\theta,l) \Rightarrow (Assign,\theta,l) \leq (Store,\theta,l) \quad (17.3.5)$$

$$(Assign,\theta,l,v)_n \leq (Store,\theta,l,v') \Rightarrow v = v' \vee (Assign,\theta,l,v)_n < (Assign,\theta,l)_m \leq (Store,\theta,l,v') \quad (17.1.5)$$

$$(Load,\theta,l)_n \Rightarrow (Read,\theta,l)_n \leq (Load,\theta,l)_n \quad (17.3.6)$$

$$(Write,\theta,l)_n \Rightarrow (Store,\theta,l)_n \leq (Write,\theta,l)_n \quad (17.3.7)$$

$$(Store,\theta,l)_m \leq (Load,\theta,l)_n \Rightarrow (Write,\theta,l)_m \leq (Read,\theta,l)_n \quad (17.3.8)$$

$$(Lock,\theta,o)_n \leq (Lock,\theta',o) \wedge \theta \neq \theta' \Rightarrow (Unlock,\theta,o)_n \leq (Lock,\theta',o) \quad (17.5.1)$$

$$(Unlock,\theta,o)_n \Rightarrow (Lock,\theta,o)_n \leq (Unlock,\theta,o)_n \quad (17.5.2)$$

$$(Assign,\theta,l) \leq (Unlock,\theta) \Rightarrow (Assign,\theta,l) \leq (Store,\theta,l)_n \leq (Write,\theta,l)_n \leq (Unlock,\theta) \quad (17.6.1)$$

$$(Lock,\theta) \leq (Use,\theta,l) \Rightarrow (Lock,\theta) \leq (Assign,\theta,l) \leq (Use,\theta,l) \vee (Lock,\theta) \leq (Read,\theta,l)_n \leq (Load,\theta,l)_n \leq (Use,\theta,l) \quad (17.6.2)$$

$$(Lock,\theta) \leq (Store,\theta,l) \Rightarrow (Lock,\theta) \leq (Assign,\theta,l) \leq (Store,\theta,l) \quad (17.6.2')$$

Table 4. Event space axioms

```
imports {
  protecting (PARTIALORDER(ACTION  sort Elt -> ObjectId ) * { ... }
  protecting (SET(TRIV) * { sort Elt -> AllRange
                            sort Set<X> -> Set<AllRange> })
  protecting (SET(TRIV) * { sort Elt -> ExistsRange
                            sort Set<X> -> Set<ExistsRange> })
}

signature {
  [ AllRange ExistsRange ]

  op all-range : PartialOrder<ObjectId> -> Set<AllRange>
  op exists-range : PartialOrder<ObjectId> -> Set<ExistsRange>
  pred all-pred : AllRange
  pred exists-pred : PartialOrder<ObjectId> AllRange ExistsRange
```

```
    ...
  }
}

module* RULE(X :: RULE-BASE) {
  signature {
    pred rule : PartialOrder<ObjectId>
    pred rule : PartialOrder<ObjectId> Set<AllRange>
    op filter : Set<AllRange> -> Set<AllRange>
    pred exists : PartialOrder<ObjectId> AllRange Set<ExistsRange>
    ...
  }

  axioms {
    var po : PartialOrder<ObjectId>
    var x : AllRange
    var m : Set<AllRange>
    var y : ExistsRange
    var n : Set<ExistsRange>

    eq rule(po) = rule(po, filter(all-range(po))) .

    eq rule(po, empty) = true .
    eq rule(po, x m) = exists(po, x, exists-range(po)) and rule(po, m) .

    eq filter(empty) = empty .
    eq filter(x m) = if all-pred(x)
                     then x filter(m)
                     else filter(m)
                     fi .

    eq exists(po, x, empty) = false .
    eq exists(po, x, y n) = exists-pred(po, x, y) or exists(po, x, n) .
    ...
  }
}
```

A separate module `RULES` provides all the necessary predicates. For rules (17.3.2) and (17.3.4) these are

```
    -- Rule 3
    eq assign<load(<< a, l >>) = action(< a >) == assign and
                                 action(< l >) == load and
                                 thread(< a >) == thread(< l >) and
                                 location(< a >) == location(< l >) .
    eq <store<(po, << x1, x2 >>, s) = action(< s >) == store and
                                      thread(< s >) == thread(< x1 >) and
                                      location(< s >) ==
                                                     location(< x1 >) and
                                      (po . x1 <= s) and (po . s <= x2) .

    -- Rule 5
    eq use(u) = (action(< u >) == use) and true .
    eq assign<-or-load<(po, x1, x2) = (action(< x2 >) == assign or
```

```
                                        action(< x2 >) == load) and
                                       thread(< x2 >) ==
                                                      thread(< x1 >) and
                                       location(< x2 >) ==
                                                    location(< x1 >) and
                                       (po . x1 <= x2) .
```

Thus we get the formal counterpart to (17.3.2) and (17.3.4) by a fairly complicated instantiation:

```
make RULE3 (RULE(RULES { sort AllRange -> Pair<ObjectId;ObjectId>,
                         sort ExistsRange -> ObjectId,
                         sort Set<AllRange> ->
                                          Set<Pair<ObjectId;ObjectId>>,
                         sort Set<ExistsRange> -> Set<ObjectId>,
                         op all-range -> relation,
                         op exists-range -> underlying,
                         op all-pred -> assign<load,
                         op exists-pred -> <store<
                         ... }))

make RULE5 (RULE(RULES { var x : AllRange,
                         sort AllRange -> ObjectId,
                         sort ExistsRange -> ObjectId,
                         sort Set<AllRange> -> Set<ObjectId>,
                         sort Set<ExistsRange> -> Set<ObjectId>,
                         op all-range -> underlying,
                         op exists-range -> underlying,
                         op all-pred(x) -> use(x),
                         op exists-pred -> assign<-or-load<
                         ... }))
```

The module `EVENTSPACE` imports all the instantiated rule schemes. It checks for a given poset of object identifiers whether it fulfills all event space rules through function `eventspace`.

```
module! EVENTSPACE {
  imports {
    protecting (RULE1 * { op rule -> rule1, ... })
    ...
    protecting (RULE15 * { op rule -> rule15, ... })
  }

  signature {
    [ PartialOrderErr<ObjectId> < EventSpaceErr < EventSpace,
      PartialOrderNull<ObjectId> < EventSpaceNull < EventSpaceOK,
      EventSpaceOK < PartialOrderOK<ObjectId>,
      EventSpaceOK < EventSpace,
      EventSpace < PartialOrder<ObjectId>,
      ObjectId Pair<ObjectId;ObjectId> < EventSpace ]

    op eventspace : PartialOrder<ObjectId> -> EventSpace
    ...
  }
  ...
}
```

Event space extensions. An event space η may also be extended by a new event $a = (A, \theta, x)$ as follows: if A is a thread action, then $b \leq a$ for all instances b of (B, θ) in η; if a is a main memory action, then $c \leq a$ for all instances c of (C, x) in η. Moreover, if A is *Load* then $c \leq a$ for all instances c of $(Read, \theta, x)$ in η, and if A is *Write* then $c \leq a$ for all instances c of $(Store, \theta, x)$ in η. The term $\eta \oplus a$ denotes the space thus obtained, provided it obeys the above rules, and it is otherwise undefined.

This behaviour is formalised by a function `+<`. We omit its straightforward but lengthy definition.

```
op _+<_ : EventSpace ObjectId -> EventSpace
```

2.3 Multi-Threaded Java

Stores assume in multi-threaded Java a more active role than they have in sequential Java because of the way the main memory interacts with the working memories: a "silent" computational step changing the store may occur without the direct intervention of a thread's execution engine. Changes to the store are subject to the previous occurrence of certain events which affect the state of computation. Event spaces are included in the configurations to record such historical information.

We first state the necessary extensions for the notions of terms, stacks, and configurations from the single-threaded to the multi-threaded case. We include a brief overview of their formalisation. Then we give the operational rules for multi-threaded Java and discuss their pendants in CafeOBJ.

Multi-threaded terms, stacks, and configurations. A multi-threaded Java configuration may include multiple S-terms, one for each running thread. An abstract term T of multi-threaded Java is a set of pairs (θ, t), where $\theta \in$ *Thread_id*, $t \in$ *S-Term* and no distinct elements of T bear the same thread identifier. The set of abstract terms of multi-threaded Java is called *M-Term*. M-terms $\{(\theta_1, t_1), (\theta_2, t_2), \dots\}$ are written as lists $(\theta_1, t_1) \mid (\theta_2, t_2) \mid \dots$ and pairs (θ, t) are written t when θ is irrelevant.

We formalise these requirements simply as sets of elements of a sort `T-Term`. The restriction that the thread identifiers in an M-term must be distinct is omitted, since it is never used in the semantics.

```
make T-TERM (PAIR(THREAD { sort Elt -> ThreadId },
                 S-TERM { sort Elt -> S-Term }) *
             { sort Pair<X;Y> -> T-Term })

make M-TERM (SET(T-TERM { sort Elt -> T-Term }) *
             { sort Set<X> -> M-Term })
```

Each thread of execution of a Java program has its own stack. Let *M-Stack* = *Thread_id* $\rightharpoonup$ *S-Stack* be the domain of multi-threaded stacks, ranged over by σ. Given $\sigma \in$ *M-Stack*, the multi-threaded stacks $push(\theta, \rho, \sigma)$, $\sigma[\theta, i \mapsto v]$ and $\sigma[\theta, i = v]$ map θ' to $\sigma(\theta')$ when $\theta \neq \theta'$, and otherwise map θ respectively to $push(\rho, \sigma(\theta))$, $\sigma(\theta)[i \mapsto v]$ and $\sigma(\theta)[i = v]$.

```
make M-STACK (PARTIALFUNCTION(THREAD { sort Elt -> ThreadId },
```

```
                              S-STACK { sort Elt -> S-Stack,
                                        op bottom ->
                                                (bottom):S-StackErr }) *
                { sort PartialFunction<X;Y> -> M-Stack,
                  sort Set<X> -> Set<ThreadId>,
                  sort Set<Y> -> Set<S-Stack> } )
```

The configurations of multi-threaded Java are 4-tuples (T, η, σ, μ) consisting of an M-term T, an event space η an M-stack σ and a store μ.

```
module! M-CONFIGURATION {
  imports {
    protecting (QUADRUPLE(M-TERM { sort Elt -> M-Term },
                          EVENTSPACE { sort Elt -> EventSpace },
                          M-STACK { sort Elt -> M-Stack },
                          STORE { sort Elt -> Store }) *
                  { sort Quadruple<W;X;Y;Z> -> Conf })
    ...
  }
```

Multi-threaded rules. The operational rules make use of the following notation: $store_\eta(\theta, l)$ denotes the oldest unwritten value of l stored by θ in η and $rval_\eta(\theta, l)$ denotes the latest value of l assigned or loaded by θ in η.

These functions are easily specified in CafeOBJ; their lengthy definitions are therefore omitted here.

The operational semantics for multi-threaded Java is given in Table 5. There is a "primed" version [x'] for of each rule [x] of Section 2.1; [x'] is omitted if it reads as [x] by the notational conventions.

The high non-determinism of the "silent action"-rules [read], [load], [store], and [write] is hard to model in CafeOBJ, since for example the [read] rule is always applicable. We therefore do not use event spaces for our formalised rules, but restrict ourselves to a simple implementation that provides the same interface as `EVENTSPACE`. In particular, for sake of simplicity, a thread reads from the main memory whenever it is using a variable and writes through to main memory whenever it is assigning a variable.

```
  axioms {
    var th : ThreadId
    var ee : EventSpace
    var ss : M-Stack
    var mm : Store
    var l : LVal
    var v : RVal

    ceq [assign3']:
      step(<< << th, (l = v) >>, ee, ss, mm >>) =
        << << th, (v) >>, ee +< event(assign, th, l, v)
                             +< event(store, th, l, v)
                             +< event(write, th, l),
                          ss, mm [ l |-> v ] >>
        if (ee +< event(assign, th, l, v)
               +< event(store, th, l, v)
               +< event(write, th, l)) =/= (bottom):EventSpaceErr .
```

[assign3']	$(\theta, l = v), \eta \longrightarrow (\theta, v), \eta \oplus (Assign, \theta, l, v)$
[assign4']	$(\theta, i = v), \sigma \longrightarrow (\theta, v), \sigma[\theta, i \mapsto v]$
[val']	$(\theta, l), \eta \longrightarrow (\theta, rval_\eta(\theta, l)), \eta \oplus (Use, \theta, l)$
[var']	$(\theta, i), \sigma \longrightarrow (\theta, \sigma(\theta, i)), \sigma$
[block2']	$\dfrac{(\theta, S_1), push(\theta, \rho_1, \sigma_1) \longrightarrow (\theta, S_2), push(\theta, \rho_2, \sigma_2)}{(\theta, \{S_1\}_{\rho_1}), \sigma_1 \longrightarrow (\theta, \{S_2\}_{\rho_2}), \sigma_2}$
[synchro1]	$\dfrac{e_1 \longrightarrow e_2}{\texttt{synchronized}(e_1)\ q \longrightarrow \texttt{synchronized}(e_2)\ q}$
[synchro2]	$\dfrac{q_1 \longrightarrow q_2}{\texttt{synchronized}(o)\ q_1 \longrightarrow \texttt{synchronized}(o)\ q_2}$
[lock]	$\dfrac{(\theta, e), \eta_1 \longrightarrow (\theta, o), \eta_2}{(\theta, \texttt{synchronized}(e)\ q), \eta_1 \longrightarrow (\theta, \texttt{synchronized}(o)\ q), \eta_2 \oplus (Lock, \theta, o)}$
[unlock]	$(\theta, \texttt{synchronized}(o)\ \{\ \}_\rho), \eta \longrightarrow \eta \oplus (Unlock, \theta, o)$
[read]	$T, \eta, \mu \longrightarrow T, \eta \oplus (Read, \theta, l, \mu(l)), \mu$
[load]	$T, \eta \longrightarrow T, \eta \oplus (Load, \theta, l)$
[store]	$T, \eta \longrightarrow T, \eta \oplus (Store, \theta, l, v)$
[write]	$T, \eta, \mu \longrightarrow T, \eta \oplus (Write, \theta, l), \mu[l \mapsto store_\eta(\theta, l)]$
[par]	$\dfrac{t_1 \longrightarrow t_2}{t_1 \mid T \longrightarrow t_2 \mid T}$

Table 5. Multi-threaded Java

```
ceq [val']:
  step(<< << th, (l) >>, ee, ss, mm >>) =
    << << th, rval(ee +< event(read, th, l, (mm l))
                     +< event(load, th, l), th, l) >>,
       ee +< event(read, th, l, (mm l))
          +< event(load, th, l)
          +< event(use, th, l), ss, mm >>
    if (ee +< event(read, th, l, (mm l))
           +< event(load, th, l)
           +< event(use, th, l)) =/= (bottom):EventSpaceErr .
```

Additionally we now use a different approach to guide the reduction and to avoid wrong reduction branches as in [assign1]: we introduce priorities of rewrite rules. This requires to implement a certain amount of reflection capabilities in CafeOBJ in order to decide which rules can be applied to a given term. This is done by a function `match`. The priorities are fixed in a partial order.

```
  op priorities : -> PartialOrder<String>
  pred match : Conf String

    eq match(<< << th, (l = v) >>, ee, ss, mm >>, "assign3'") = true .
    eq match(<< << th, (l) >>, ee, ss, mm >>, "val'") = true .

    eq priorities = make-partialorder(<< "assign2", "assign1" >>
                                      << "assign3'", "assign2" >>
                                      << "assign4'", "assign2" >>
                                      << "unop2!", "unop1" >>
                                      << "access2", "access1" >>
                                      << "pth2", "pth1" >>
                                      << "decl2", "decl1" >>
                                      << "block1", "block2" >>
                                      << "synchro2", "synchro1" >>
                                      << "unlock", "synchro2" >>) .

    ceq [assign1]:
      step(<< << th, (h1 = ae) >>, ee, ss, mm >>) =
        assignment-left(step(<< << th, (h1) >>, ee, ss, mm >>), ae)
        if priority(<< (<< th, (h1 = ae) >>), ee, ss, mm >>,
                    "assign1") == true .
    ceq [assign2]:
      step(<< << th, (k = ae1) >>, ee, ss, mm >>) =
        assignment-right(step(<< << th, (ae1) >>, ee, ss, mm >>), k)
        if priority(<< (<< th, (k = ae1) >>), ee, ss, mm >>,
                    "assign2") == true .
```

Although this approach is much more complex than the one described in Section 2.1 it seems to be easier to reason about. Logic, i.e. the operational rules, and control, i.e. the priorities, are separated more clearly. The additional complexity can be read from a run of the example in Section 2.1. Now the block is executed in a thread with thread identifier `thd1`.

```
let test = << << thd1, ({ (((Point (p =' (new Point ( )))) ;)
                           ((((p . x) = 1) ;)
                           ((((p . y) = 2) ;)
                           ((((p . x) = (p . y)) ;) nil))))
                         } ^ (undefined [ p |-> null ])) >>,
              null, undefined [ thd1 |-> empty ], empty >> .

exec step(test, 15) .
```

CafeOBJ answers

```
-- execute in M-CONFIGURATION : step(<< (<< thd1 , ( (((Point (p
    =' new Point ( ))) ;) ((((p . x) = 1) ;) ((((p . y) = 2) ;) (
    (((p . x) = (p . y)) ;) nil))))  ^ (undefined [ p |-> null ]
    )) >>) , null , (undefined [ thd1 |-> empty ]) , empty >>,15)

<< (<< thd1 , * >>) , ((((((((((((null +< asr-action-0) +< asr-action-1)
   +< ulw-action-0) +< asr-action-2) +< asr-action-3) +< ulw-action-1)
   +< asr-action-5) +< ulw-action-3) +< ulw-action-4) +< asr-action-6)
   +< asr-action-7) +< ulw-action-5) , ((((((((((((((((((((((((((
```

```
  ((((undefined [ thd1 |-> empty ]) [ thd1 |-> push(undefined [ p
  |-> null ],empty) ]) ... ) , point-0 >> : Conf
(0.000 sec for parse, 699059 rewrites(596.660 sec),
                                                1876858 match attempts)
```

We inspect the event space and the main memory.

```
M-CONFIGURATION> red < asr-action-0 > .
-- reduce in M-CONFIGURATION : < asr-action-0 >
< asr-action-0 : ASR-Action | (action = assign , thread = thd1 ,
  location = << point-0 , x >> , value = 1) > : ASR-Action
(0.000 sec for parse, 1 rewrites(0.000 sec), 1 match attempts)

M-CONFIGURATION> red < point-0 > .
-- reduce in M-CONFIGURATION : < point-0 >
< point-0 : Point | (x = 2 , y = 2) > : Point
(0.000 sec for parse, 1 rewrites(0.000 sec), 1 match attempts)
```

The non-determinism in the [par] rule is implemented using ACI-matching of sets. However, we can not be sure that each thread can make a computational step at any time; for example, a thread waiting for a lock can not move. Thus, we perform a (non-deterministic) search for a thread that is able to proceed. The predicate `is-config` tests whether a the `step` function was applied successfully.

```
var st : S-Term
var tt : T-Term
var mt : M-Term

eq [par]:
  step(<< (<< th, st >> tt mt), ee, ss, mm >>) =
    if (is-config(step(<< << th, st >>, ee, ss, mm >>)) == true)
    then par(step(<< << th, st >>, ee, ss, mm >>), tt mt)
    else par(step(<< (tt mt), ee, ss, mm >>), << th, st >>)
    fi .
```

We illustrate the specification by the so-called "Possible Swap" example detailed in [11] and [7]. Two threads θ_1 and θ_2 running in parallel want to manipulate the coordinates of the same point object o referenced in both threads by the local variable `p`. These manipulations are to run under mutual exclusion.

$$(\theta_1, \texttt{synchronized(p)}\,\{\,\texttt{p.x} = \texttt{p.y};\,\}) \mid (\theta_2, \texttt{synchronized(p)}\,\{\,\texttt{p.y} = \texttt{p.x};\,\})$$

We assume an instance `< point-0 : Point | (x = 1 , y = 2) >` of `Point` in the main memory. The following describes a possible run in CafeOBJ:

```
exec new(Point, empty) .
exec set-x(< point-0 >, 1) .
exec set-y(< point-0 >, 2) .

let test = << (<< thd1, (synchronized (p) (
                        ((((p . x) = (p . y)) ;) nil)  ^ undefined)) >>
               << thd2, (synchronized (p) (
                        ((((p . y) = (p . x)) ;) nil)  ^ undefined)) >>),
```

```
              (null),
              ((undefined [ thd1 |-> push(undefined [ p |-> point-0 ],
                                                      empty) ])
                          [ thd2 |-> push(undefined [ p |-> point-0 ],
                                                      empty) ]),
              point-0 >> .

exec step(test, 16) .

-- execute in M-CONFIGURATION : step(<< (<< thd1 , (synchronized
    ( p ) ( ((((p . x) = (p . y)) ;) nil)  ^ undefined)) >> <<
    thd2 , (synchronized ( p ) ( ((((p . y) = (p . x)) ;) nil)
     ^ undefined)) >>) , null , ((undefined [ thd1 |-> push(undefined
    [ p |-> point-0 ],empty) ]) [ thd2 |-> push(undefined [ p |->
     point-0 ],empty) ]) , point-0 >>,16)
<< (<< thd1 , * >> << thd2 , * >>) , ((((((((((((((((null +< on-action-1)
   +< asr-action-1) +< ulw-action-1) +< ulw-action-2) +< asr-action-2)
   +< asr-action-3) +< ulw-action-3) +< on-action-2) +< on-action-4)
   +< asr-action-5) +< ulw-action-5) +< ulw-action-6) +< asr-action-6)
   +< asr-action-7) +< ulw-action-7) +< on-action-5) , ((((((((((
  ((((((((((((((((undefined [ thd1 |-> push(undefined [ p |-> point-0
  ],empty) ]) ... ) , point-0 >> : Conf
(0.000 sec for parse, 624990 rewrites(515.410 sec),
                                                  1676076 match attempts)

M-CONFIGURATION> red < on-action-1 > .
-- reduce in M-CONFIGURATION : < on-action-1 >
< on-action-1 : ON-Action | (action = lock , thread = thd2 , object
  = point-0) > : ON-Action
(0.000 sec for parse, 1 rewrites(0.000 sec), 1 match attempts)

M-CONFIGURATION> red < point-0 > .
-- reduce in M-CONFIGURATION : < point-0 >
< point-0 : Point | (x = 1 , y = 1) > : Point
(0.000 sec for parse, 1 rewrites(0.000 sec), 1 match attempts)
```

Specification metrics. The formalisation of multi-threaded Java takes about 1800 lines in four packages and 40 modules. The maximum nesting level we use is five for imports and three for parameterisation. The parsing time for the whole specification is about half an hour (sic!).

Conclusions and Future Work

We provided two case studies with CafeOBJ. In the first case study we developed a recycling machine specification as an example of the formally based object-oriented software engineering method fOOSE. The second case study formalised a Java semantics in CafeOBJ.

The system proved to be fast and stable enough to specify medium-scale examples like the Java formalisation. However, it could be a little bit faster and a little bit more robust. In particular, parsing seems to be slow and becomes tedious when it comes to complex mixfix expressions. Also the interpretation of parameterised modules and only average complex rewrite rules turns out be

rather slow. Nevertheless, it must be conceded that these objections can be easily compensated by a small expense in hardware.

Some features we would definitely like to see in future version of CafeOBJ: A let- or where-construct would simplify many rules, as e.g. for the schematic translation of interaction diagrams to rewrite rules or rules in the Java formalisation. An implementation of membership equational logic [4] would greatly support useability; we only simulate similar features in our Java formalisation. Parameterisation as used in the event space would benefit from a more sophisticated type inference algorithm for views. And last not least any kind of meta-reasoning over modules and rules would certainly simplify CafeOBJ's translation from and to other formalism.

We did not investigate one major feature of CafeOBJ: behavioural specification. A detailed comparison with the rewriting logic approach to object-orientation and concurrency is left to future work.

References

[1] K. Achatz and W. Schulte. A Formal OO Method Inspired by Fusion and Object-Z. In J. P. Bowen, M. G. Hinchey, and D. Till, editors, *Proc. 10^{th} Int. Conf. Z Users*, volume 1212 of *Lect. Notes Comp. Sci.*, pages 92–111, Berlin, 1997. Springer.

[2] J. C. M. Baeten and W. P. Weijland. *Process Algebra*. Cambridge University Press, Cambridge, 1990.

[3] G. Booch, I. Jacobson, and J. Rumbaugh. The Unified Modeling Language (Version 1.1). Technical report, Rational Software Corp., 1997.

[4] A. Bouhoula, J.-P. Jouannaud, and J. Meseguer. Specification and Proof in Membership Equational Logic. In M. Bidoit and M. Dauchet, editors, *Proc. 7^{th} Int. Conf. Theory and Practice of Software Development*, volume 1214 of *Lect. Notes Comp. Sci*, pages 67–92, Berlin, 1997. Springer.

[5] R. H. Bourdeau and B. H. C. Cheng. A Formal Semantics for Object Model Diagrams. *IEEE Trans. Softw. Eng.*, 21(10):799–821, 1995.

[6] R. Breu, U. Hinkel, C. Hofmann, C. Klein, B. Paech, B. Rumpe, and V. Thurner. Towards a Formalization of the Unified Modeling Language. In M. Akşit and S. Matsuoka, editors, *Proc. 11^{th} Europ. Conf. Object-Oriented Programming*, number 1241 in Lect. Notes Comp. Sci., pages 344–366, 1997.

[7] P. Cenciarelli, A. Knapp, B. Reus, and M. Wirsing. From Sequential to Multi-Threaded Java: An Event-Based Operational Semantics. In M. Johnson, editor, *Proc. 6^{th} Int. Conf. Algebraic Methodology and Software Technology*, volume 1349 of *Lect. Notes Comp. Sci.*, Berlin, 1997. Springer.

[8] R. Diaconescu and K. Futatsugi. *CafeOBJ Report*, volume 6 of *AMAST Series in Computing*. World Scientific, Singapore, etc., 1998. To appear.

[9] J. A. Goguen and G. Malcolm. A Hidden Agenda. Technical Report CS97-538, University of California, San Diego, 1997.

[10] J. A. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.-P. Jouannaud. Introducing OBJ. In J. A. Goguen and G. Malcolm, editors, *Software Engineering with OBJ: Algebraic Specification in Practice*. Cambridge, 1998. To appear.

[11] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison–Wesley, Reading, Mass., 1996.

[12] I. Jacobson, M. Christerson, P. Jonsson, and G. Övergaard. *Object-Oriented Software Engineering*. Addison–Wesley, Wokingham, England, 4th edition, 1993.

[13] K. Lano. *Formal Object-Oriented Development*. Springer, London, 1995.

[14] J. Meseguer. A Logical Theory of Concurrent Objects and Its Realization in The Maude Language. In G. A. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 314–389. MIT Press, Cambridge, Massachusetts–London, 1991.

[15] S. Nakajima and K. Futatsugi. An Object-Oriented Modeling Method for Algebraic Specifications in CafeOBJ. In W. R. Adrion, editor, *Proc. 19^{th} Int. Conf. Softw. Eng.*, pages 34–44, 1997.

[16] J. Rumbaugh, M. Blaha, W. Premerlani, and F. Eddy. *Object-Oriented Modeling and Design*. Prentice-Hall, Englewood Cliffs, New Jersey, etc., 1991.

[17] M. Wirsing and A. Knapp. A Formal Approach to Object-Oriented Software Engineering. In J. Meseguer, editor, *Proc. 1^{st} Int. Wsp. Rewriting Logic and Its Applications*, volume 4 of *Electr. Notes Theo. Comp. Sci.*, pages 321–359. Elsevier, 1996.