Electronic Notes in Theoretical Computer Science 4 (1996)

A Formal Approach to Object-Oriented Software Engineering

Martin Wirsing and Alexander Knapp[1]

Ludwig–Maximilians–Universität München
Institut für Informatik
Oettingenstraße 67, D–80538 München, Germany
e-mail: {wirsing,knapp}@informatik.uni-muenchen.de

Abstract

The goal of this paper is to show how formal specifications can be integrated into one of the current pragmatic object-oriented software development methods. Jacobson's method OOSE ("Object-Oriented Software-Engineering") is combined with object-oriented algebraic specifications by extending object and interaction diagrams with formal annotations. The specifications are based on Meseguer's Rewriting Logic and are written in an extension of the language Maude by process expressions. As a result any such diagram can be associated with a formal specification, proof obligations ensuring invariant properties can be automatically generated, and the refinement relations between documents on different abstraction levels can be formally stated and proved. Finally, we provide a schematic translation of the specification to Java and thus an automatic generation of an object-oriented implementation.

1 Introduction

Current object-oriented design methods, such as those of Rumbaugh, Shlaer–Mellor, Jacobson and Booch use a combination of diagrammatic notations including object and class diagrams, state transition diagrams and scenarios. Other, academic approaches, such as Reggio's entity algebras, Meseguer's Maude and Ehrich/Sernadas' Troll propose fully formal descriptions for design specifications. Both approaches have their advantages and disadvantages: the informal diagrammatic methods are easier to understand and to apply but they can be ambiguous. Due to the different nature of the employed diagrams and descriptions it is often difficult to get a comprehensive view of all functional and dynamic properties. On the other hand, the formal approaches are more difficult to learn and require mathematical training. But they provide mathematical rigour for analysis and prototyping of designs.

[1] This research has been sponsored by the DFG-project OSIDRIS and the ESPRIT HCM-project MEDICIS.

To close partly this gap we propose a combination of formal specification techniques with pragmatic software engineering methods. Our specification techniques are well-suited to describe distributed object-oriented systems. They are based on Meseguer's Rewriting Logic and are written in an extension of the language Maude. The static and functional part of a software system is described by classical algebraic specifications whereas the dynamic behaviour is modeled by nondeterministic rewriting. The flow of messages is controlled by process expressions.

Jacobson's method OOSE ("Object-Oriented Software-Engineering") is combined with these object-oriented algebraic specifications in such a way that the basic method of Jacobson remains unchanged. As in OOSE the development process of our enhanced fOOSE method consists of five phases: use case analysis, robustness analysis, design, implementation and test. The only difference is that the OOSE diagrams can optionally be refined and annotated by formal text. Any annotated diagram can be semi-automatically translated into a formal specification, i.e. the diagram is automatically translated into an incomplete formal specification which then has to be completed by hand to a formal one.

Thus any fOOSE diagram is accompagnied 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, due to the choice of the executable specification language Maude early prototyping is possible during analysis and design. Moreover, in many situations we are able to provide a schematic translation of the specification to Java and thus an automatic generation of an object-oriented implementation.

Therefore the combination of algebraic specification with rewriting gives a coherent view of object-oriented design and implementation. Formal specification techniques are complementary to diagrammatic ones. The integration of both leads to an improved design and provides new techniques for prototyping and testing.

Several related approaches are known in the literature concerning the chosen specification formalism and also the integration of pragmatic software engineering methods with formal techniques. First, there is a large body of formal approaches for describing design and requirements of object-oriented systems (for an overview see [9]). Our approach is based on Meseguer's rewriting logic and Maude (cf. e.g. [20]) and was inspired by Astesiano's SMoLCS approach ([3], [23]) which can be characterized as a combination of algebraic specifications with transition systems instead of rewriting. Astesiano was the first integrating also process expressions in his framework. PCF [19] and LOTOS [7] also combine process expressions with algebraic specifications. A process algebra for controlling the flow of messages was introduced in a different way in Maude by [18]. By using an appropriate extension of the μ-calculus Lechner [17] presents a more abstract approach for describing object oriented requirements and designs on top of Maude. The use of strategies together with rewriting logic was introduced by Vittek et al. [6].

There are also several approaches for integrating pragmatic software engineering methods with formal techniques. Hußmann [13] gives a formal foundation of SSADM, the Syntropy method is based on Z and state charts, Dodani and Rupp [8] enhance the Fusion method by formal specifications written in COLD [15], Lano [16] presents a formal approach to object-oriented software development based on Z++ and VDM++. Very similar to our approach is the one of Futatsugi and Nakajima [22] who use OBJ for giving a formal semantics to interaction diagrams.

The paper is organized as follows: Section 2 gives a short introduction to our chosen specification language Maude and its extension with means for controlling the flow of messages. In section 3 an overview of our enhanced development method fOOSE is presented. Section 4 explains the details of our method for developing a formal specification out of an informal description of a use case and illustrates it by the example of a recycling machine which is the running example of Jacobson's book on OOSE (Object-Oriented Software Engineering, [14]). Section 5 ends with some concluding remarks.

2 Maude

This section gives a short introduction to our chosen specification language Maude (for more details see [21]).

Maude consists of two parts: a purely functional part and an object-oriented part. The functional part is the algebraic specification language OBJ3 [11]; it serves for specifying data types in an algebraic way by equations. The object-oriented part extends OBJ3 by notions of object, message and state, and allows one to describe the dynamic behaviour of objects in an operational style by rewrite rules.

2.1 Functional Part

Maude has two kinds of functional specifications: "modules" and "theories". A module (keyword `fmod` ... `endfm`) contains an import list (`protecting`, `extending`, or `using`), sorts (`sort`), subsorts (`<`), function (`op`) and variable declarations (`var`), and equations (`eq`) which provide the actual "code" of the module. Theories have different keywords (viz. `fth` ... `endft`) but have otherwise the same syntax. The real difference is a semantic one: the semantics of a module is the (isomorphism class of the) initial order-sorted algebra [10] whereas a theory is "loose", i.e. it denotes a class of (possibly non-isomorphic) algebras. A module is executable; a theory is not executable, it gives only a few characteristic properties ("requirements") the specified data type has to fulfill.

The following example specifies a trivial theory `TRIV` which introduces one sort `Elt`, and a module `LIST` for the data structure of lists with elements of sort `Elt`. `LIST` is parameterized by `TRIV`.

```
fth TRIV is
  sort Elt .
endft

fmod LIST[X::TRIV] is
  protecting NAT BOOL .
  sort List .
  subsort Elt < List .
  op _ _: List List -> List [assoc id: nil] .
  op length: List -> Nat .
  op _ in _: Elt List -> Bool .
  op _ ≤ _: List List -> Bool .
  var E E’: Elt .
  var L L’: List .
  eq length(nil) = 0 .
  eq length(E L) = (s 0) + length(L) .
  eq E in nil = false .
  eq E in (E’ L) = (E == E’) or (E in L) .
  eq (nil ≤ L) = true .
  eq (E L) ≤ (E’ L’) = (E in (E’ L’)) and (L ≤ (E’ L’)) .
endfm
```

For some of the explanations in the following we assume that the reader is familiar with the basic notions of algebraic specifications such as signature, term and algebra (for details see e.g. [24]).

2.2 Object-Oriented Specifications

The object-oriented concept in Maude is the object module. The declaration of an object module (`omod` ... `endom`) consists, additionally to functional modules, of a number of class declarations (`class`), message declarations (`msg`) and rewrite rules (`rl`).

```
omod BUFFER[X::TRIV] is
  protecting LIST[X] NAT .
  class Buffer | contents: List .
  msg put _ in _: Elt OId -> Msg .
  msg getfrom _ replyto _: OId OId -> Msg .
  msg to _ elt-in _ is _: OId OId Elt -> Msg .
  vars B I: OId .
  var E: Elt .
  var Q: List .
  rl [put] (put E in B) <B: Buffer | contents: Q> =>
             <B: Buffer | contents: E Q>
               if length(Q) < s(s(s(s(s(0))))) .
  rl [get] (getfrom B replyto I) <B: Buffer | contents: Q E> =>
             <B: Buffer | contents: Q>
             (to I elt-in B is E) .
endom
```

An (object) class is declared by an identifier and a list of attributes and their sorts. `OId` is the sort of Maude identifiers reserved for all object identifiers, `CId` is the sort of all class identifiers.

An object is represented by a term—more precisely by a tuple—comprising a unique object identifier, an identifier for the class the object belongs to and a set of attributes with their values, e.g. `<B: Buffer | contents: X Y Z nil>`.

A message is a term that consists of the message's name, the identifiers of the objects the message is addressed to, and, possibly, parameters (in mixfix notation), e.g. `(put W in B)`.

A Maude program makes computational progress by rewriting its global state, called "configuration" of Maude sort `Configuration` (in the following abstractly denoted by Γ). A configuration is a multiset of objects and messages:

$$\{\!|m_1, \dots, m_k|\!\} \cup \{\!|o_1, \dots, o_n|\!\} \quad \text{or, for short} \quad m_1 \dots m_k\ o_1 \dots o_n$$

where $\cup$ is a function symbol for multiset union (in Maude denoted by juxtaposition), $m_1, \dots, m_k$ are messages, and $o_1, \dots, o_n$ are objects.

A rewrite rule

$$t \xrightarrow{l} t' \Leftarrow \Pi \quad \text{denoted by} \quad \texttt{[}l\texttt{]}\ t\ \texttt{=>}\ t'\ \texttt{if}\ \Pi$$

transforms a configuration into a subsequent configuration, where t and t' are terms of sort `Configuration`, Π is a conjunction of equations and l is a label (or proof term) of the form $l\texttt{(}x_1\texttt{,}\dots\texttt{,}x_k\texttt{)}$ with $x_1, \dots, x_k$ being the variables occurring in t, t', and Π (we omit these variables). It accepts messages for some objects under a certain condition, possibly modifies these object, and emerges new ones and some additional messages.

In this paper we restrict rewrite rules to those used in Simple Maude of the form

$$m\ o \xrightarrow{l} o'\ o_1\ \dots\ o_n\ m_1\ \dots\ m_k \Leftarrow \Pi$$

where m, $m_1, \dots, m_k$ are messages ($k \geq 0$), m being optional, and o, o' (optional), $o_1, \dots, o_n$ are objects ($n \geq 0$) with o being (possibly) changed to o'.

Formally, we consider a Maude specification M as a quadruple (Σ, E, L, R) given by a signature $\Sigma = (S, F)$, a set E of conditional equations, a set L of labels (also called actions), and a set R of labeled conditional rewrite rules. We assume that for any label there is at most one rule.

Deduction, i.e. rewriting, takes place according to rewriting logic defined by the following four rules (cf. [20]):

(i) Reflexivity.

$$\frac{}{t \xrightarrow{t} t}$$

(ii) Congruence. For each function symbol $f : s_1 \dots s_n \to s \in F$

$$\frac{t_1 \xrightarrow{\alpha_1} u_1,\ \dots,\ t_n \xrightarrow{\alpha_n} u_n}{f(t_1, \dots, t_n) \xrightarrow{f(\alpha_1, \dots, \alpha_n)} f(u_1, \dots, u_n)}$$

(iii) Replacement. For each rewrite rule $t_0 \xrightarrow{l} u_0 \Leftarrow \Pi \in R$

$$\frac{t_1 \xrightarrow{\alpha_1} u_1,\ \dots,\ t_n \xrightarrow{\alpha_n} u_n}{t_0(t_1, \dots, t_n) \xrightarrow{l(\alpha_1, \dots, \alpha_n)} u_0(u_1, \dots, u_n)}, \quad \text{if } \Pi(t_1, \dots, t_n)$$

(iv) Composition.

$$\frac{t_1 \xrightarrow{\alpha_1} t_2,\ t_2 \xrightarrow{\alpha_2} t_3}{t_1 \xrightarrow{\alpha_1;\alpha_2} t_3}$$

where matching is defined modulo E. (In fact, Maude uses rewriting logic for both its functional and its object-oriented part; we use equational logic for the former one.)

We say that M entails a sequent $t \xrightarrow{\alpha} t'$ if $t \xrightarrow{\alpha} t'$ can be obtained by finite application of the rules above and write $M \vdash t \xrightarrow{\alpha} t'$.

Such a sequent is called one-step concurrent rewrite if it can be derived from R by finite application of the rules (i)–(iv), with at least one application of the replacement rule (iii). It is called a sequential rewrite if it can be derived with exactly one application of (iii).

Since every rewrite step can be decomposed in an (interleaving) sequence of sequential rewrites ([20]) we can restrict our attention to such simple rewrite steps. For any sequential rewrite, we abstract from the actual proof term α and consider only the label l of the unique application of the replacement rule (iii). Moreover, we omit parameters which are not necessary for the synchronisation; mostly this amounts to a statement of the sender and the receiver of a message. A run of M is a possibly infinite chain

$$t_1 \xrightarrow{l_1} t_2 \xrightarrow{l_2} t_3 \xrightarrow{l_3} \dots$$

of one-step sequential rewrites with $M \vdash t_n \xrightarrow{l_n} t_{n+1}$ for every $n \geq 1$.

A (Σ, L)-structure $\mathfrak{A} = ((A_s)_{s\in S}, (f^{\mathfrak{A}})_{f\in F}, (\xrightarrow{l}^{\mathfrak{A}})_{l\in L})$ is given by a family $(A_s)_{s\in S}$ of sets, a family $(f^{\mathfrak{A}})_{f\in F}$ of functions with $f^{\mathfrak{A}} : A_{s_1} \times \dots \times A_{s_n} \to A_s$ for $f : s_1 \dots s_n \to s \in F$ and a family $(\xrightarrow{l}^{\mathfrak{A}})_{l\in L}$ of relations with $\xrightarrow{l}^{\mathfrak{A}} \subseteq A_\Gamma \times A_\Gamma$ for a rewrite rule $t \xrightarrow{l} t' \Leftarrow \Pi \in R$.

$\mathfrak{A}$ is a model of $M = (\Sigma, E, L, R)$ if $\mathfrak{A}$ satisfies all equations of E and all conditional rules R. The semantics of M is defined to be the initial model $\mathfrak{I}$ of all models of M.

A run of $\mathfrak{A}$ is a possibly infinite chain

$$t_1^{\mathfrak{A}} \xrightarrow{l_1}^{\mathfrak{A}} t_2^{\mathfrak{A}} \xrightarrow{l_2}^{\mathfrak{A}} t_3^{\mathfrak{A}} \xrightarrow{l_3}^{\mathfrak{A}} \dots$$

of one-step rewrites.

2.3 Modules with Control

The one-step rewrites build the basis for a small language of processes describing the admissible chains of rewrite steps for particular computations.

Now, an atomic process is a sequential rewrite labeling l. Moreover, there are a constant “1” denoting reflexivity and a constant δ for deadlock which is used to denote a situation where none of the rules below can be applied.

A composite process may be an atomic process, sequential composition, nondeterministic choice, or parallel composition of processes, or a repeat statement. The abstract syntax of processes is given by

$$\mathrm{A} ::= 1 \mid \delta \mid l$$

$$\mathrm{P} ::= \mathrm{A} \mid (\mathrm{P};\mathrm{P}) \mid (\mathrm{P}+\mathrm{P}) \mid (\mathrm{P} \parallel \mathrm{P}) \mid (\mathrm{P}^*)$$

Processes are assumed to satisfy the following laws of Table 1 (borrowed from process algebra PA, see [5]). Note that the last equation for parallel composition induces an interleaving approach to concurrency: either l_1 or l_2 has to be executed first. This assumption simplifies our notion of refinement (cf. 2.4). Any process defines a set of traces it accepts where a trace is a finite or infinite sequence atomic processes.

$$\begin{array}{l}
1;p = p, \qquad p;1 = p, \qquad \delta;p = \delta,\\
p_1;(p_2;p_3) = (p_1;p_2);p_3,\\[4pt]
\delta + p = p,\\
p_1 + p_2 = p_2 + p_1, \qquad p_1 + (p_2 + p_3) = (p_1 + p_2) + p_3,\\
(p_1 + p_2);p_3 = p_1;p_3 + p_2;p_3,\\[4pt]
1 \parallel p = p, \qquad \delta \parallel p = \delta,\\
p_1 \parallel p_2 = p_2 \parallel p_1, \qquad p_1 \parallel (p_2 \parallel p_3) = (p_1 \parallel p_2) \parallel p_3,\\
(p_1 + p_2) \parallel p3 = (p_1 \parallel p_3) + (p_2 \parallel p_3),\\
(l_1;p_1) \parallel (l_2;p_2) = l_1;(p_1 \parallel (l_2;p_2)) + l_2;((l_1;p_1) \parallel p_2)\\[4pt]
p^* = (p;p^*) + 1
\end{array}$$

Table 1
Process algebra axioms

With the help of processes we can on the one hand constrain the set of possible runs of a Maude module; on the other hand, processes may trigger certain actions.

To actually incorporate process expressions into Maude specifications, we build up a hierarchy of process definitions $D = ((l_i, p_i))_{1 \leq i \leq n}$ over a given set of labels L, where $L' = \{l_1, \ldots, l_n\}$ is set of new labels disjoint from L and each process expression p_i uses only labels in $L \cup \bigcup_{1 \leq j < i}\{l_j\}$. $L \cup L'$ is called label set of D. Every such hierarchy defines a function $D : L' \to \mathrm{P}$ that maps a new label to a process expression over L; we will also denote its extension to process expressions over $L' \cup L$ by D.

Definition 2.1 A Maude specification with control (M, D, p, q_0) is a Maude specification $M = (\Sigma, E, R, L)$ together with a process definition D over L, a process expression p with labels in the label set of D and an initial configuration $q_0 \in \mathcal{T}(M)_\Gamma$ where $\mathcal{T}(M)_\Gamma$ denotes all terms built from the signature of M of Maude sort `Configuration`.

We say that $(M, D, p, q_0) \vdash t \xrightarrow{\alpha} t'$ with the labels of α in L, if t is reachable from the initial configuration q_0 and if t rewrites to t' via α, i.e. there is an α_0 such that $M \vdash q_0 \xrightarrow{\alpha_0} t$, $M \vdash t \xrightarrow{\alpha} t'$ and any trace of $\alpha_0;\alpha$ is a trace of $D(p)$. Analogously, $(M, D, p, q_0) \vdash t \xrightarrow{s} t'$ for a process expression s with labels in L and that of D, if $(M, D, p, q_0) \vdash t \xrightarrow{\alpha} t'$ for a trace α of $D(s)$. Finally, a run of M is a run of (M, D, p, q_0) if it starts in q_0 and its sequence of labels is a trace of $D(p)$.

A model $\mathfrak{A}$ of M is a model of (M, D, p, q_0) if every run in (M, D, p, q_0) is a run in $\mathfrak{A}$.

For the concrete syntax, we extend the Maude language by a new keyword `cntrl` to declare the message control that is to be used within `omod` ... `endom`; since it represents the global control, it is only meaningful in the uppermost module of a hierarchy. The `BUFFER` example could be extended by

```
cntrl [ppput] put(_,B)* .
cntrl [gget] getfrom(B,I); to(I,B,_); getfrom(B,I); to(I,B,_) .
cntrl ppput; gget .
```

The last label-less process declaration defines the global control.

The initial state is not regarded part of a module. It has to provided when opening (starting) a derivation in Maude.

Maude modules that use `cntrl` are called Maude modules with control.

Obviously, every Maude module M is equivalent to a module with control: let $l_1, \ldots, l_n$ be the rule labels of M. Then the process expression $p = (l_1 + \cdots + l_n)^*$ does not restrict the possible runs. Thus M and M extended by `cntrl` p `.` accept the same runs, if they start with the same configuration.

On the other hand, any Simple Maude module with control can easily be translated to a normal Maude module.

For this purpose, we define two functions $\mathrm{hd} : \mathrm{P} \to \wp(\mathrm{A})$ and $\mathrm{tl} : \mathrm{A} \times \mathrm{P} \to \mathrm{P}$ that compute the accepted atomic processes for an arbitrary process expression and its behaviour after an atomic process has been executed, respectively. These may be easily implemented, since every process expression has an equivalent head normal form (see [4]).

Now, let M be such a module with process definition D, control p and rules $r_1, \ldots, r_n$. First, we flatten p to $D(p)$ by replacing all labels of L' by their corresponding bodies, thus making D superfluous. Next, we construct another module M' which extends M by the import of an implementation of `hd` and `tl` and sorts `A` and `P` for atomic and composed processes (such that `A` is a subsort of `P`). Moreover, M' declares a class `Control` of synchronization objects which have a process expression as attribute:

```
class Control | process: P .
```

Now we define two reductions to Maude, a general one with a global control which works for all process expressions and a more specific one with a distributed control which is well defined only for a set of parallel processes.

In the case of global control, we declare one control object `C0: Control` and initialize it with p. Moreover, for $i = 1, \ldots, n$ we translate any rule

```
rl [r_i] m o => c if Π .      to
rl [r_i] m o <C0: Control | process: Q> =>
  c <C0: Control | process: tl(m,Q)>
```

```
          if m ∈ hd(Q) and Π .
```

Note that M' admits only "interleaving concurrency"; in contrast to M (and thus in contrast to (M, D, p, q_0)) no concurrent rewrite steps are possible in M' among $r_1, \ldots, r_n$.

Note also that using methods as advocated e.g. in [18] M' can further be translated to a module in Simple Maude.

In the case of distributed control we assume that we have k objects $o_1, \ldots, o_k$ and that p is of the form $q_1 \parallel \cdots \parallel q_k$ such that all atomic labels (different from 1 and δ) in q_i denote messages received by o_i $(i = 1, \ldots, k)$. For each o_i we declare a control object c_i: `Control` and initialize it with q_i. Moreover, we translate any rule

```
rl [r] m o_i => c if Π .      to
rl [r] m o_i <c_i: Control | process: Q> =>
   c <c_i: Control | process: tl(m,Q)>
      if m ∈ hd(Q) and Π .
```

Here M' admits "true concurrency": using the replacement rule (iii) of rewriting logic, rules corresponding to different objects can be applied ("fire") concurrently.

Fact 2.2 *Let (M, D, p, q_0) be a module with control, M' its translation to Maude with global control and under the assumptions above M'' its translation to Maude with distributed control. Then (M, D, p, q_0), M', and M'' admit the same runs.*

2.4 Refinement

The principal notion for expressing the correctness of a system wrt. its requirements is the notion of refinement.

Definition 2.3 Let $\mathfrak{A}$ be a (Σ, L)-structure and $\mathfrak{C}$ a (Σ', L')-structure with $\Sigma = (S, F) \subseteq (S', F') = \Sigma'$ and $L \subseteq L'$. Then $\mathfrak{C}$ is a refinement of $\mathfrak{A}$ if there exists a (Σ, L)-substructure $\mathfrak{R} = ((R_s)_{s \in S}, (f^{\mathfrak{R}})_{f \in F}, (\xrightarrow{l}{}^{\mathfrak{R}})_{l \in L})$ of $\mathfrak{C}$ and a Σ–homomorphism $\bar{\cdot} : \mathfrak{R} \to \mathfrak{A}$ which induces a bisimulation, i.e. for any $s \in S$, $r \in R_s$, and $l \in L$ the following holds:

$$\forall a \in A_s : \Big((\bar{r} \xrightarrow{l}{}^{\mathfrak{A}} a) \;\Rightarrow\; \exists r' \in R_s : \bar{r'} = a \wedge r \xrightarrow{l}{}^{\mathfrak{R}} r' \Big)$$
$$\forall r' \in R_s : \Big((r \xrightarrow{l}{}^{\mathfrak{R}} r') \;\Rightarrow\; \exists a \in A_s : \bar{r'} = a \wedge \bar{r} \xrightarrow{l}{}^{\mathfrak{A}} a \Big)$$

Let M and M' be two Maude modules (both possibly with control). M' is called a (semantic) refinement of M if the initial model of M' is a refinement of the initial model of M.

This refinement relation is obviously transitive.

The control may be refined in the standard way (see [1] and [2]) by substituting complex process expressions for atomic ones. In our context, a hierarchy of process definitions is enlarged at the lower end by new process expressions for labels the hierarchy is based on.

Definition 2.4 Let $D = ((l_i, p_i))_{1 \leq i \leq n}$ and $D' = ((l'_i, p'_i))_{1 \leq i \leq n'}$ be process definitions over label sets L and L' respectively, with $L = L_1 \uplus L_2$, $L_1 \cap L' = \emptyset$, and $L_2 \subseteq L'$. Then, D' is called a process definition refinement of D, if it is of the form $((l'_1, p'_1), \ldots, (l'_k, p'_k), (l_1, p_1), \ldots, (l_n, p_n))$ with $\{l'_1, \ldots, l'_k\} \subseteq L_1$.

We distinguish a special refinement relation that will play a major rôle in the sequel.

Definition 2.5 Let $M = ((\Sigma, E, L, R), D, p, q_0)$ and $M' = ((\Sigma', E', L', R'), D', p', q'_0)$ be two Maude specifications with control. We call M' an object control refinement of M if (Σ', E') is a persistent extension of (Σ, E), D' is a process definition refinement of D, $p = p'$, and the following holds:

There is a $C' \subseteq \mathcal{T}(M')_\Gamma$ with $q'_0 \in C'$ and a surjective abstraction function $\tilde{\ } : C' \to \mathcal{T}(M)_\Gamma$ of configurations compatible with the equational axioms such that $C'/\,\tilde{}$ is isomorphic to $\mathcal{T}(M)_\Gamma$ with respect to multiset union (on equivalence classes), $\tilde{q'_0} = q_0$ and

(i) For any sequent $M \vdash c_1 \xrightarrow{l} c_2$ and for any $\tilde{c'_1} = c_1$ there is a corresponding sequent $M' \vdash c'_1 \xrightarrow{l} c'_2$ such that $\tilde{c'_2} = c_2$.

(ii) For any sequent $M' \vdash c'_1 \xrightarrow{l} c'_2$ with $\tilde{c'_1} = c_1$ there is a corresponding sequent $M \vdash c_1 \xrightarrow{l} c_2$ such that $\tilde{c'_2} = c_2$.

Lemma 2.6 *Let M and M' be two Maude modules with control. If M' is an object control refinement of M then M' is a refinement of M.*

Proof. Let $\tilde{\ } : \mathcal{T}(M')_\Gamma \supseteq C' \to \mathcal{T}(M)_\Gamma$ be an abstraction function with the required properties. Let $\mathfrak{A}$ be the initial model of M and $\mathfrak{C}$ that of M'. Let $R_\Gamma = \mathfrak{C}(C')$ (where $\mathfrak{C}(C')$ denotes the interpretation of C' in $\mathfrak{C}$), $R_s = C_s$ for any other sort s, and $\bar{\ } : \mathfrak{R} \to \mathfrak{A}$ be $\tilde{\ }$ transferred to $\mathfrak{R}$ such that $\overline{\mathfrak{R}(r)} = \mathfrak{A}(\tilde{r})$. Then $\mathfrak{R}$ and $\bar{\ }$ fulfill the requirements of the refinement definition. □

3 Enhanced OOSE Development Process

The development process of OOSE consists of five phases: use case analysis, robustness analysis, design, implementation and test [14] (see Figure 1).

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.

The use case analysis is the particular feature which distinguishes OOSE

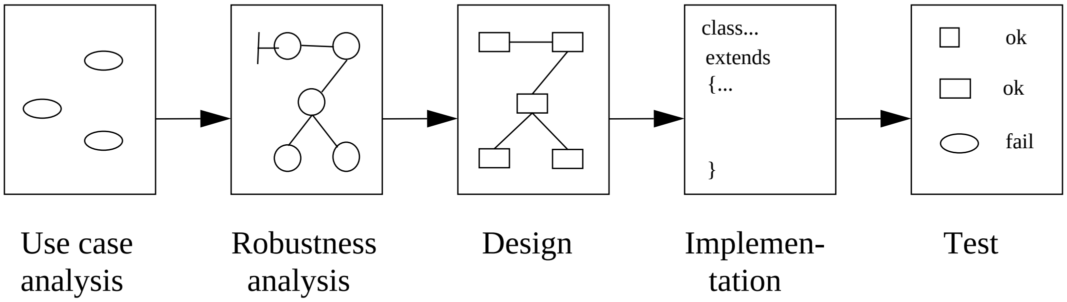

Fig. 1. Development phases of OOSE

from other development methods and which shall be integrated e.g. in the new versions of OMT and Booch's method. Use cases have the advantage to provide a requirement document which is the basis for testing and which can serve as a reference during the whole development.

As in all semiformal 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 our enhanced fOOSE method we provide means 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 can be semi-automatically translated into a formal specification, i.e. the diagram is automatically translated into an incomplete formal specification which then has to be completed by hand to a formal one.

Thus any fOOSE diagram is accompagnied 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, due to the choice of the executable specification language Maude early prototyping is possible during analysis and design. Moreover, in many situations we are able to provide a schematic translation of the specification to Java and thus an automatic generation of an object-oriented implementation.

In the sequel we will use the following method for constructing a formal Maude specification (see Figure 2) from an informal description.

For any given informal description we construct two diagrams: 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) 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 Maude specification with control together with some proof obligations. Moreover, object control 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.

Further schematic translation to Java provides a direct implementation in an object-oriented language. Our current translation is well-suited to systems composed of a set of concurrently running sequential objects; it might be slow and cumbersome in more complex situations. A further precondition is that the basic data types have efficient implementations. This may not be the case for specifications of the requirements or analysis phase.

Note that in contrast to OOSE we use interaction diagrams also in the analysis phases, and not only in the design phase. We believe that interaction diagrams are good for illustrating the interactions of the objects also at abstract levels.

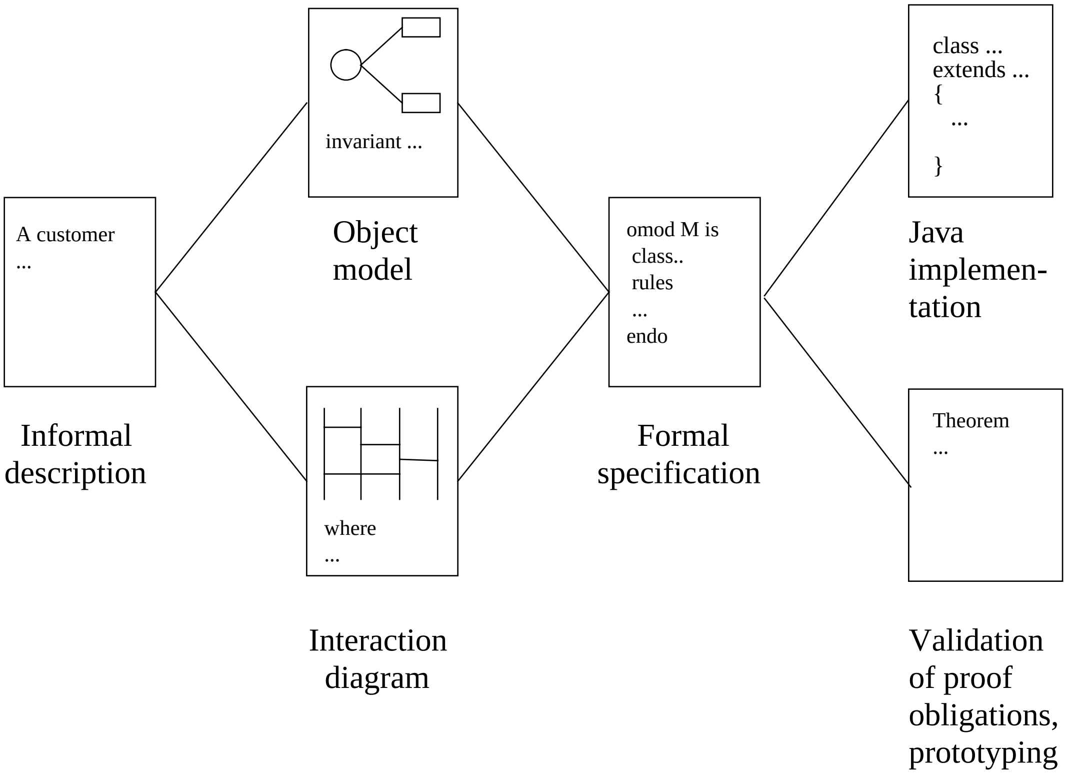

Fig. 2. Construction and use of formal specifications

4 The fOOSE Method in More Detail

In this section we present our method fOOSE (formal Object-Oriented Software Engineering) for developing and refining a formal specification of an informal description of a use case and illustrate it by the example of a recycling machine which is the running example of Jacobson's book on OOSE (Object-Oriented Software Engineering, [14]).

For the construction of a formal specification of a use case we proceed in three steps:

(i) A semi-formal description consisting of an object model and an interaction diagram are developed in the usual OOSE style from the informal (textual) description.

(ii) Functional specifications are constructed for all data types occurring in the diagrams.

(iii) The object diagram is (if necessary) extended by invariants and the interaction diagram is refined. Then a Maude object module is semi-automatically generated from both refined diagrams.

Any refined specification is constructed in the same way. Moreover, for relating the refined "concrete" specification with the more abstract specification one has to give the relationship between the "abstract" and the "concrete" configurations and to define the process definitions for the refined labels. This generates proof obligations (see Section 2.4) which have to be verified to guarantee the correctness of the refinement.

Finally, if the specification is concrete enough, it is schematically translated to a Java program.

We show the specification and refinement activities for the recycling machine example on the level of requirements analysis and robustness analysis in Sections 4.1 and 4.2. In Section 4.3 the generation of the Java code is presented.

4.1 Requirements Analysis

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:

> "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."

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.

To do this we model the use case as an interactive system consisting of two objects of classes SB and RM (Figure 3 on the left). The class SB stands for "system border" representing the customer, i.e. the actor of this use case. It is modeled without any attributes. The class RM repesents 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.[2] The interaction diagram (Figure 3 on the right) shows (abstractly) the interaction between the customer and the recycling machine. 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 DM). 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.

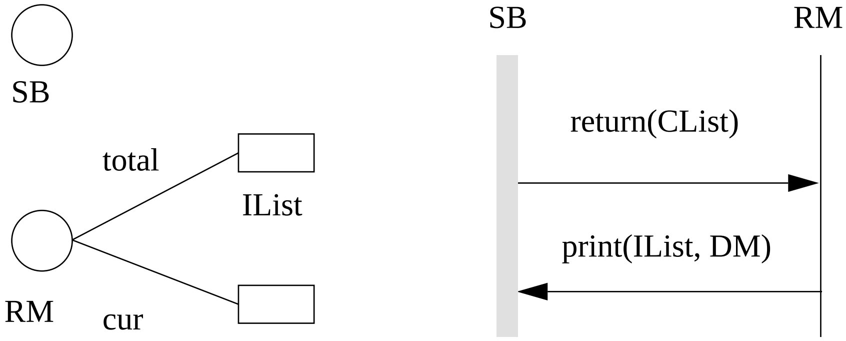

Fig. 3. Object model and interaction diagram of the recycling machine

More generally, an object model consists of several objects (represented by cycles) with their attributes (represented by lines from circles to rectangles) and the relationships between the objects (represented by arrows). Objects are labeled with their class name and attributes with their name (on the line) and the sort of the attribute (below the rectangle). There are several kinds of relationships. In this paper we consider only the inheritance relationship represented by a dotted arrow from the heir to the parent (for an example see Figure 4).

An interaction diagram consists of several objects represented by vertical lines and messages represented by horizontal arrows. Each arrow leads from the sender object to the receiving object. Objects are labeled with the class name and messages with their name and the sorts of their arguments. Progress in time is represented by a time axis from top to bottom: a message below another should be handled later in time. Moreover, an abstract algorithm can be given at the left hand side of the diagram for describing the control flow (for an example see Figure 7).

There are different ways of interpreting interaction diagrams: Jacobson focuses on sequential systems where every message generates a response. Since we aim at asynchronous distributed systems, in our approach we prefer to state the return messages explicitly. Obviously, it would not be difficult to formalize also Jacobson's interpretation.

Object and interaction diagrams give an abstract view of the informal description. But several important relationships are not represented which will

[2] The choice of lists for the daily total here is a premature design decision we make for simplicity of presentation. It would be better to choose an abstract container type.

be expressed by the formal specifications. For example in the use case "return items" there is a connection between the current list and the daily total; moreover, the list of printed items is a description of the list of returned items. The formal specification will be able to express these semantic dependencies. It will also be used to fix the basic data types.

4.1.1 Functional Specifications for Data Types

The functional specifications are written in the functional style of Maude. For any data type occurring in the diagrams a specification is constructed either by 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 any concrete item whereas the operation `price` computes the price whose value will be given in `DM`.

```
fth ITEM is
  protecting DM .
  sorts CItem Item .
  op price: Item -> DM .
  op desc: CItem -> Item .
endft
```

The specification of lists is obtained by instantiating the list module twice, once with concrete items for elements and once with items; in both cases we rename also the sort `List`.

```
make CLIST is LIST[CItem] * (sort List to CList) endmk
make ILIST is LIST[Item] * (sort List to IList) endmk
```

Moreover, we need two more operations: `amount(l)` calculates the sum of the prices of the elements of `l` and `desclist(cl)` converts any "concrete" list `cl` in a list of descriptions.

```
fmod LIST1[I::ITEM] is
  protecting CLIST ILIST .
  op desclist: CList -> IList .
  op amount: IList -> DM .
  var I: Item .
  var Ci: CItem .
  var L: IList .
  var Cl: CList .
  eq desclist(nil) = nil .
  eq desclist(Ci Cl) = desc(Ci) desclist(Cl) .
  eq amount(nil) = 0 .
  eq amount(I L) = price(I) + amount(L) .
endfm
```

4.1.2 Refining The Diagrams

The third step consists of two activities: the extension of the object models by invariants and the refinement of the interaction diagrams.

An invariant is a relation between the attributes of an object or between the objects of a configuration which has to be preserved by all rewriting steps.

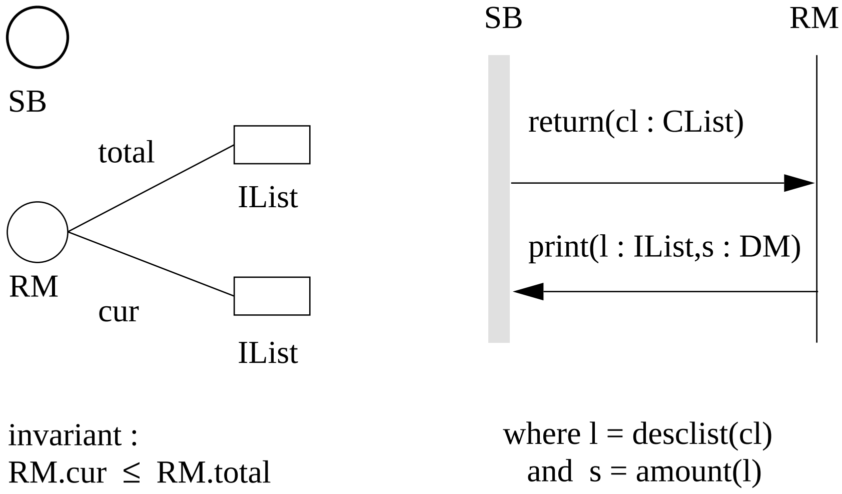

Fig. 4. Object model with invariant and refined interaction diagram of the recycling machine

For example, the attributes total and cur of the recycling machine satisfy the property that all items of cur have also to be in total, i.e. the value of the cur is in the $\leq$ relation (see the specification LIST on Section 2.1) w.r.t the value of total. We express this formally in the object diagram by using a dot notation for selecting the values of the attributes (see Figure 4).

Interaction diagrams are refined in order to express semantic relationships of the parameters of the messages.

We replace the parameter sorts of messages by variables of the appropriate sorts and state the relationships between the variables in an additional "where clause": any message expression $m(s_1, \dots, s_n)$ is replaced by an expression $m(v_1 : s_1, \dots, v_n : s_n)$ where $v_1, \dots, v_n$ are variables of sorts $s_1, \dots, s_n$; the "where clause" is a conjunction of equations of the form $t_1 = u_1 \wedge \dots \wedge t_k = u_k$ such that t_j, u_j are terms containing at most the variables $v_1, \dots, v_n$.

For example, the message expressions return(CList) and print(List, DM) of the interaction diagram in Figure 3 are replaced by return(cl : CList) and print(l : IList, s : DM) where cl, l, and s are variables of sorts CList, List, and DM. Then the equation l = desclist(cl) states that l is a list of descriptions of the elements of cl and the equation s = amount(l) that s is the sum of the prices of l.

The right part of Figure 4 shows the refined interaction diagram.

4.1.3 Construction of a Formal Specification

In this step 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.

The automatic part of the construction is as follows:

- Every object model induces a set of Maude class declarations:
  - · Each object name C with attributes $a_1, \dots, a_n$ of types $s_1, \dots, s_n$ of the diagram represents a class declaration

    `class` C `|` a_1`:` s_1`,` ... `,`a_n`:` s_n `.`

  - · Each inheritance relation from D to C corresponds to a subclass declaration

    `subclass` D `<` C `.`

  - · Each invariant I of the attributes $C.a_1, \dots, C.a_n$ of an object C is translated to the sort constraint

    `sct <O: <`C `|` a_1`:` v_1`,` ... `,`a_n`:` v_n`,` a`>:` C
    `if` $I[a_1 \leftarrow v_1] \dots [a_n \leftarrow v_n]$

    where the constraint condition $I[a_1 \leftarrow v_1] \dots [a_n \leftarrow v_n]$ is obtained from I by substituting the variables $v_1, \dots, v_n$ for $C.a_1, \dots, C.a_n$.
- The interaction diagram induces a set of message declarations:
  - · Each message $m(v_1 : s_1, \dots, v_n : s_n)$ induces the message declaration

    `msg` m`: OId` s_1 ... s_n `OId -> Msg .`

    The first argument of m indicates the sender object, the last argument the destination.
- Both diagrams generate the skeleton of a rule:
  - · For any message $m(\dots v_j : s_j \dots)$ from E_0 to C of the interaction diagram, let

    `class` C `|` ... $a_i : s_i$ ... `.`
    `class` E_0 `|` ... $b_i : s'_i$ ... `.`

    be the corresponding class declarations, $m_k(\dots w_{kj} : s_{kj} \dots)$ 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:

```
[$m$] $m$($o_0$, ... ,$v_j$, ... ,$o$) <$o$: $C$ | ... $a_i$: $w_i$ ... > =>
        <$o$: $C$ | ... $a_i$: ? ... >
        $m_1$($o$, ... ,$w_{1j}$, ... ,$o_1$) ...
        $m_n$($o$, ... ,$w_{nj}$, ... ,$o_n$)
          if $\Pi$ and $\Pi$?
```

where $o_0, \ldots, o_n$ are object identifiers (for the classes $E_0, \ldots, E_n$).[3]

- 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 initial state contains a concrete example of the use case, i.e. the set of objects derived from the object model that are concerned by the interaction diagrams and some messages occuring there.

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. Similarily, Π? states that the condition is perhaps under-specified.

For example, the diagrams of Figure 4 induce the following skeleton:

```
[ret] return(OO,Items,Rm)
      <Rm: RM | total: W1, cur: W2> =>
        <Rm: RM | total: ?, cur: ?>
        print(Rm,L,S,OO)
          if L = desclist(Items) and S = amount(L) and Π?
```

To get the complete rule one has to fill the question marks with the appropriate value `(l d)` and `l`.

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 example, the interaction diagram defines the following control strategy

```
cntrl ret .
```

The full specification of the use case "return items" is as follows:

```
omod RM is
```

[3] Note that in practice only the relevant part of the "where clause" is taken as the condition for the rule, not the full "where clause".

```
  protecting LIST1 .
  class RM | total: IList, cur: IList .
  sct <O: <RM | total: D, cur: L>: RM if L ≤ D .
  msg return: OId IList OId -> Msg .
  class User .
  msg print: OId IList DM OId -> Msg.
  var Items: CList .
  vars Rm Usr: OId .
  vars L0 L D: Ilist .
  var S: DM .
  rl [ret] return(Usr,Items,Rm)
           <Rm: RM | total: D, cur: L0> =>
             <Rm: RM | total: L D, cur: L>
             print(Rm,L,S,Usr)
               if L = desclist(Items) and S = amount(L) .
  rl [print] print(Rm,L,S,Usr)
             <Usr: User> =>
               <Usr: User> .
  cntrl ret .
endom
```

An invariant I for a class C has to be satisfied by all objects of C and of its subclasses. As a consequence it generates a proof obligation on rewrite rules: every axiom of the form

$$\begin{array}{l} m\ \texttt{<}o\texttt{:}\ Y\ \texttt{|}\ a_1\texttt{:}\ t_1\texttt{,}\ \ldots\ \texttt{,}a_n\texttt{:}\ t_n\texttt{,}\ a\texttt{> =>}\\ \quad \texttt{<}o\texttt{:}\ Y\ \texttt{|}\ a_1\texttt{:}\ u_1\texttt{,}\ \ldots\ \texttt{,}a_n\texttt{:}\ u_n\texttt{,}\ a'\texttt{>}\ c\\ \qquad \texttt{if}\ \Pi \end{array}$$

(where Y is C or any of its subclasses) has to satisfy the correctness condition

$$I[a_1 \leftarrow u_1]\ldots[a_n \leftarrow u_n] \Leftarrow \Pi \wedge I[a_1 \leftarrow t_1]\ldots[a_n \leftarrow t_n]$$

For example the rule `[ret]` induces the correctness condition

```
L ≤ (L D) if L0 ≤ D and L = desclist(Items)
  and S = amount(L)
```

Obviously, `L ≤ (L D)` holds for any list `L` and `D`. In this case the preconditions are irrelevant.

A possible initial configuration of `RM` can be defined as follows:

$$\begin{array}{l} q_0\ =\ \texttt{<Usr: User> <Rm: RM | total: nil, cur: nil>}\\ \qquad\quad \texttt{return(Usr,(}ci_1\ ci_2\ ci_3\ \texttt{nil),Rm)} \end{array}$$

4.2 Robustness Analysis

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 4.1 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."

4.2.1 Object Model With Invariants

To cope with these refinements, in the second phase of OOSE, called "robustness analysis", the 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.

Now, the recycling machine consists of five objects (sorts): the interface object Customer_Panel, a control object Receiver and the entity objects Current, Day_Total and Deposit_Item. Customer_Panel and Receiver communicate the data concerning the returned items, the Receiver uses Current and Day_Total for storing and computing the list of current items and the daily total. Deposit_Item stands for all kinds of returned items, in particular for the class of bottles which is modeled as its heir (see Figure 5).

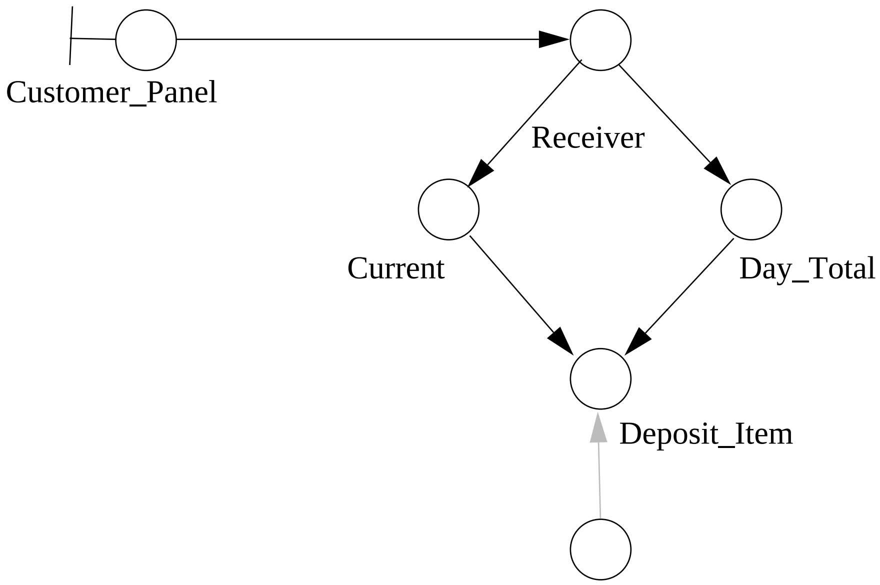

Fig. 5. Object model of the robustness analysis of the recycling machine

In the object model interface objects are represented by hooked circles, control objects by circles with an arrow, and entity objects by full circles.

Additionally, object models are given in two parts, one showing the attributes of the objects and the other showing the relationships between the objects.

In our case, the objects of the robustness analysis have the following attributes (see Figure 6): the Customer_Panel and the Receiver have no attributes; Deposit_item has a name and a price, Bottle has additionally a height and a width; the class Current has a list (of Deposit_item) and an amount as attributes, Day_Total a list of deposit items.

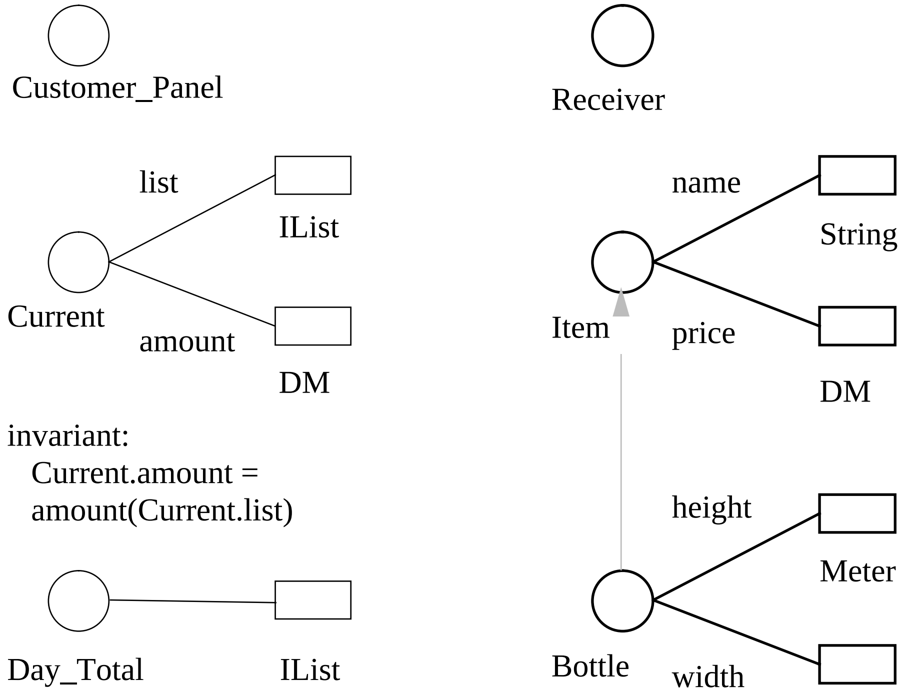

Fig. 6. Object model with attributes and invariants of the robustness analysis of the recycling machine

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

4.2.2 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 Customer_Panel: 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 Customer_Panel. 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 Day_Total by two messages, both being called add; the end of such a return process is to be acknowledged by a message ack. In the third activity the Customer_Panel sends a print_receipt 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 Customer_Panel (by a message called send) which prints the result.

The resulting interaction diagram derived from this text is, in the next step, refined by inserting variables for the parameters of messages and by stating semantic properties of the parameters (see Figure 7).

In particular 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.

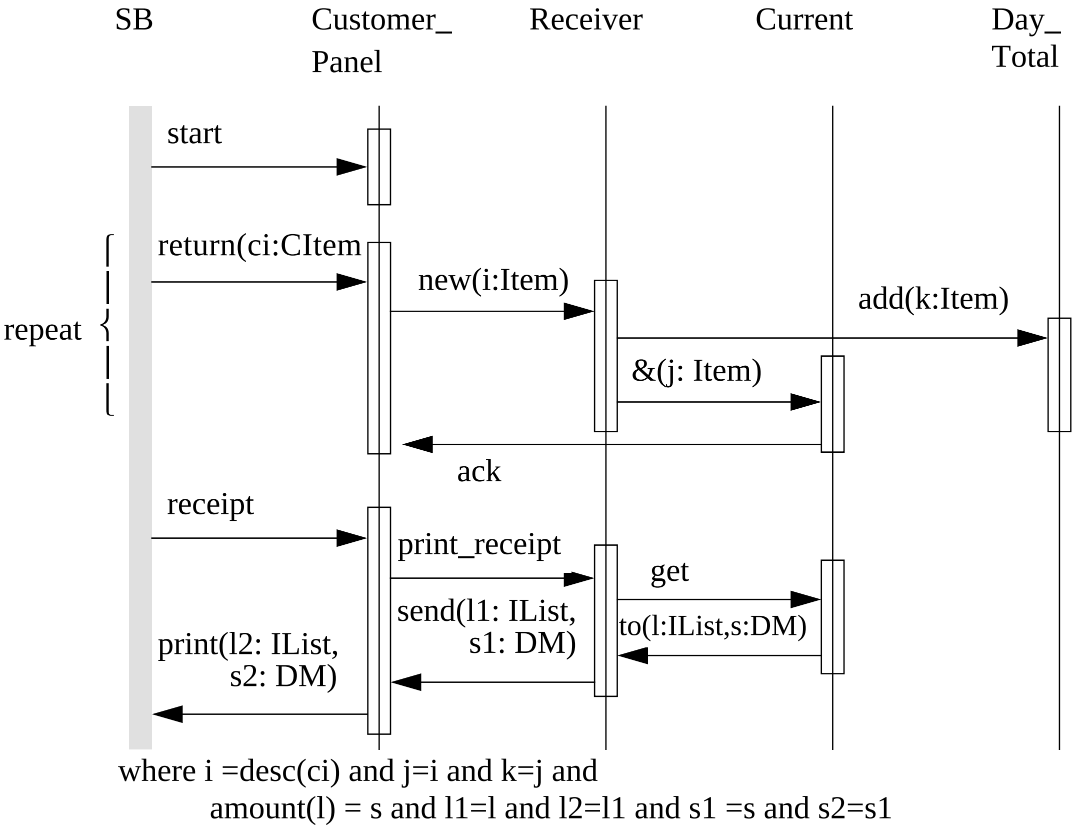

Fig. 7. Refined interaction diagram of the robustness analysis of the recycling machine

4.2.3 Construction of a Formal Specification

The refined diagram generates automatically eleven message declarations according to our method in Section 4.1.3, e.g.

```
msg start: OId OId -> Msg .
msg return: OId CItem OId -> Msg .
msg new: OId Item OId -> Msg .
```

To define the rule skeletons for the interaction diagram of the recycling machine we use the attributes defined in the object model (Figure 5). Then we get the following skeletons for e.g. the start, return and new message:

```
[start] start(Sb,Cp)
        <Cp: Customer_Panel | state: A> =>
          <Cp: Customer_Panel | state: ?>
            if Π?
```

```
[return] return(Sb,Ci,Cp)
           <Cp: Customer_Panel | state: A> =>
             <Cp: Customer_Panel | state: ?>
             new(Cp,I,Rc)
               if I = desc(Ci) and Π?
[new] new(Cp,I,Rc)
        <Rc: Receiver> =>
          <Rc: Receiver>
          &(Rc,I,Cur)
          add(Rc,I,Dt)
            if Π?
```

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

```
  (start; (return)*; ack; receipt; send)
‖ ((new)*; print_receipt; to)
‖ ((&)*; get)
‖ (add)*
```

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(Sb,Cp)
             <Cp: Customer_Panel | state: A> =>
               <cp: Customer_Panel | state: on>
                 if A = off .
rl [return] return(Sb,Ci,Cp)
              <Cp: Customer_Panel | state: A> =>
                <Cp: Customer_Panel | state: on>
                new(Cp,I,Rc)
                  if I = desc(Ci) and A = on .
rl [new] new(Cp,I,Rc)
           <Rc: Receiver> =>
             <Rc: Receiver>
             &(Rc,I,Cur)
             add(Rc,I,Dt) .
```

Moreover, the following proof obligation is automatically created:

```
price(I)+S = amount(I L) if S = amount(L)
```

(whose proof follows trivially from the definition of `amount`).

A possible initial configuration of this specification can be defined as follows:

```
q_0 = <Usr: User>
       <Cp: Customer_Panel | state: off>
       <Rc: Receiver>
       <Cur: Current | list: nil, amount: 0>
       <Dt: Day_Total | list: nil>
```

```
start(Sb,Cp)
return(Sb,ci1,Cp) return(Sb,ci2,Cp) return(Sb,ci3,Cp)
receipt(Sb,Cp)
```

The text of the full specification can be found in Appendix A.

It remains to prove, that the specification of the robustness analysis step is a refinement of the specification of the requirements analysis. Actually, we will prove that it is an object control refinement: The former class `RM` is now represented by the four classes `Customer_Panel`, `Receiver`, `Current`, and `Day_Total`, i.e. each instance of `RM` is replaced by one instance of the mentioned classes each. More precisely, we set

```
<Cp: Customer_Panel | state: A>
  <Rc: Receiver>
  <Cur: Current | list: L, amount: S>
  <Dt: Day_Total | list: L> =
<Rm: RM | total: Dt.L, cur: Cur.L>
```

The control expression is trivially refined by

```
cntrl [ret]   (start; (return; ack)*; receipt; send)
               || ((new)*; print_receipt; to)
               || ((&)*; get)
               || (add)*
```

Fact 4.1 *The robustness analysis specification of this section is an object control refinement of the requirements specification of Section 4.1.*

4.3 Design and Implementation

In the design step, the analysis model of the system is transformed and refined in the light of the actual implementation environment. In our case, this will be the programming language Java with its extensive class libraries ([12]).

In general, there are three ways to proceed: First, the robustness analysis model can be directly implemented using ad-hoc-methods, but guided by some heuristics; that would amount to the original OOSE method. Second, the Maude module with control that was the result of the specification process can be implemented, either by using the translation with global control of Section 2.3 or by re-implementing the rewriting and the control mechanism. Third, one can make use of the special simple structure of this resulting specification which satisfies the assumptions of our translation with local control (in Section 2.3). Moreover, since the behaviour of any object of the interaction diagram is sequential the computations of hd and tl are particularly simple and can be represented by a finite automaton. Thus we can consider an interaction diagram as a set of asynchronously running concurrent automata which run in parallel to the method calls defined by the rewrite rules.

We will follow this third option. However, it seems worth trying to extend steadily the transformable constructs of Maude in order to make the analysis–

design-step more natural.

In our Java implementation, every object is provided with control, organized as a finite state machine. This makes use of the fact, that inside an object there is only sequentiality. The only branching states of the controlling automaton are loop starting points where a decision is to be taken whether the body of the loop is entered or not. (For this decision we must require that the first action inside the loop and the first action after the loop are different.)

Every method checks if the object is in a state to accept the message called. If this is not the case the call is refused. The sending object—which is obliged to make this specific call—has to wait for a state change of the object called.

Now, a Maude module with control that is the result of the specification process shown is implemented in Java as follows:

- The underlying equational theory is translated to suitable Java functions. (We omit this translation.)
- Every class defines a separate Java class; all attributes of the Maude class are taken over by the Java class.
- Every message to a class, i.e. every message on the left hand side of a rule that occurs together with that class, defines an instance method of the corresponding Java class. Only the formal parameters that do not concern the sending and the accepting class are taken over.
- The control part of the Maude module defines an automaton for every object: Each class extends a special `Foose` class that itself extends the Java class `Thread`.

```
class Foose extends Thread
{
  private protected int acc,got;
  private protected Object[] env;

  private protected void notifyenv()
  {
    for (int i=0;i<env.length;i++)
    { if (env[i]!=this)
      { synchronized (env[i])
        { env[i].notify(); } } }
  }

  private protected void accept(int s)
  {
    synchronized (this)
    { got=0; acc=s; }
    notifyenv();
    while ((got&acc)==0)
    { Thread.yield(); }
    acc=0;
```

```
      }
    }
```

This Foose class provides the attributes and methods necessary to implement the control automaton: There is an attribute acc which represents the state. If a message was accepted this is stored in another attribute got. A method accept serves for the manipulation of the state and the acknowledgement of the other participating objects (in env) of a change in state.

The state itself is coded as a disjunction of the allowed messages. For this purpose, additionally, each class contains class constants M for the possible messages m. Finally, the run()-method of Thread is re-defined by an implementation of the control-automaton using the accept()-method.

The automaton is constructed in the standard way by calculating (using for example hd and tl) the accepted traces of the control part that belongs to the object. For the repeat statement a while-loop is constructed that stops if one of the possible first messages after the repeat statement has arrived; the body of the while loop must—because its first message will have arrived before it is executed—accept the messages of the control expression in the order rotated one to the left.

- Every rule defines (different parts of) a body of an Java instance method. The translation is performed in a natural way which we omit. Merely the methods are enriched by synchronization code:

```
public boolean m()
{
  if (M&acc!=M)
    return false;
  else
  { synchronized (this)
    { got=M;
      notifyenv(); }
    ...
    return true; }
}
```

A method call m of another object o is replaced by

```
while (!o.m( ... ))
{ synchronized (this)
  { try { wait(); }
    catch (InterruptedException ignored) { } } }
```

Note, that the automaton in the run()-method largely corresponds to the distributed control expressions of the Maude implementation (see Section 2.3).

For the recycling machine this means for example:

```
class Receiver extends Foose
{
  private final static int NEW=1, PRINTRECEIPT=2, TO=4;
  private CustomerPanel cp;
  private Current cur;
  private DayTotal dt;

  public Receiver()
  {
    System.err.println("Receiver");
  }

  public void reg(CustomerPanel c,Current u,
                  DayTotal d,Object[] e)
  {
    cp=c; cur=u; dt=d; env=e;
  }

  public boolean mynew(Item i)
  {
    if ((NEW&acc)!=NEW)
      return false;
    else
    { System.err.println("new()");
      synchronized (this)
      { got=NEW;
        notifyenv(); }
      while (!dt.add(i))
      { synchronized (this)
        { try { wait(); }
          catch (InterruptedException ignored) { } } }
      while (!cur.conc(i))
      { synchronized (this)
        { try { wait(); }
          catch (InterruptedException ignored) { } } }
      return true; }
  }

  public boolean printreceipt()
  {
    if ((PRINTRECEIPT&acc)!=PRINTRECEIPT)
      return false;
    else
    { System.err.println("printreceipt()");
      synchronized (this)
      { got=PRINTRECEIPT;
        notifyenv(); }
      while (!cur.get())
      { synchronized (this)
        { try { wait(); }
          catch (InterruptedException ignored) { } } }
      return true; }
  }

  public boolean to(IList l,int s)
  {
    if ((TO&acc)!=TO)
      return false;
    else
    { System.err.println("to()");
      synchronized (this)
      { got=TO;
        notifyenv(); }
      while (!cp.send(l,s))
      { synchronized (this)
        { try { wait(); }
          catch (InterruptedException ignored) { } } }
      return true; }
  }

  public void run()
  {
    System.err.println("runrc");
    while ((got&PRINTRECEIPT)!=PRINTRECEIPT)
      accept(PRINTRECEIPT|NEW);
    accept(TO);
  }
}
```

The complete Javs program and a test run of the initial configuration can be found in Appendix B. It runs with the so-called "appletviewer" program.

5 Concluding Remarks

In this paper we have presented an extension of OOSE by formal specifications which has several advantages:

- The formal meaning of diagrams provides possibilities for prototyping and generates systematically proof obligations for which can serve for validation activities.
- The refinement relation gives the information for tracing the relationships between use case descriptions and the corresponding design and implementation code, the generated proof obligations form the basis for the verification of the correctness of designs and implementations.
- The operational nature of our specification formalism allows one to generate directly Java code from design specifications.
- Traditional OOSE development can be used in parallel with fOOSE since all OOSE diagrams and development steps are valid in fOOSE.

However, there remain several open problems and issues. Our formal annotations of the interaction diagrams cover only repeat statements, means for if and while statements should be added as well. Interaction diagrams as in Jacocson's OOSE are inherently sequential, the whole OOSE method is designed for the development of sequential systems. In contrast to this we focus on the description of distributed concurrent systems. Therefore we need also means for describing the concurrent behaviour in our diagrams, not only in the interaction diagrams but also in other kinds such as use case diagrams.

Another problem is that our notion of refinement is defined on the level of specifications. For software engineers it would be easier if we could define also a refinement relation on the level of diagrams which ensures the validity of an object control refinement.

Finally, our Java implementation has two drawbacks: until now we do not have any formal semantics of Java which makes it impossible to prove the correctness of our translation to Java. To compensate this we plan to define a rewriting logic semantics of central parts of Java which would allow us to perform correctness proofs. The second drawback concerns the style of our implementation which uses heavily the "synchronization code" of the process expressions. In many cases this code is superfluous since the control is already induced by the "natural" data flow of the messages (e.g. in the recycling machine example it would be enough to construct an automaton for the customer panel; all other synchronization code could be omitted). We are investigating data flow analysis methods for eliminating unnecessary synchronizations.

References

[1] L. Aceto and M. Hennessy. Towards action-refinement in process algebras. *Inf. Comp.*, 103:204–269, 1993.

[2] L. Aceto and M. Hennessy. Adding action-refinement to a finite process algebra. *Inf. Comp.*, 115:179–247, 1994.

[3] E. Astesiano, G. Mascari, G. Reggio, and M. Wirsing. On the parameterized algebraic specification of concurrent processes. In H. Ehrig, C. Floyd, M. Nivat, and J. Thatcher, editors, *TAPSOFT'85, Vol. 1*, volume 185 of *LNCS*, pages 342–358, Berlin, 1985. Springer.

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

[5] J. A. Bergstra and J. W. Klop. Algebra of communicating processes with abstraction. *Theo. Comp. Sci.*, 37:77–121, 1985.

[6] P. Borovansky, C. Kirchner, H. Kirchner, P.-E. Moreau, and M. Vittek. ELAN: A logical framework based on computational systems. This volume.

[7] E. Brinksma, editor. LOTOS: A formal description technique based on the temporal ordering of observational behaviour. Technical Report Dis 8807, ISO, 1987.

[8] M. Dodani and R. Rupp. Integrating formal methos with object-oriented methodologies. In M. Wirsing, editor, *ICSE–17 Workshop on Formal Methods Applications in Software Engineering Practice*, Seattle, 1995.

[9] H. D. Ehrich, M. Gogolla, and A. Sernadas. Objects and their specification. In M. Bidoit and C. Choppy, editors, *Recent Trends in Data Type Specification*, volume 655 of *LNCS*, pages 40–65, Berlin, 1993. Springer.

[10] J. Goguen and J. Meseguer. Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theo. Comp. Sci.*, 105:217–273, 1992.

[11] J. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.-P. Jouannaud. Introducing OBJ3. Technical Report SRI-CSL-92-03, SRI, 1992.

[12] J. Gosling and H. McGilton. *The Java Language Environment: A White Paper.* Sun Microsystems, Mountain View, Oct. 1995.

[13] H. Hußmann. Formal foundations for pragmatic software engineering methods. In B. Wolfinger, editor, *Innovationen bei Rechnern und Kommunikationssystemen*, pages 1–50, Berlin, 1994. Springer.

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

[15] H. B. M. Jonkers. An introduction to cold-k. In J. A. B. M. Wirsing, editor, *Algebraic methods: theory, tools and applications*, volume 394 of *LNCS*, pages 139–206, Berlin, 1989. Springer.

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

[17] U. Lechner. Object-oriented specifications of distributed systems in the μ-calculus and Maude. This volume.

[18] U. Lechner, C. Lengauer, F. Nickl, and M. Wirsing. (Objects + Concurrency) & Reusability — A Proposal to Circumvent the Inheritance Anomaly. In *Proc. Europ. Conf. Object-Oriented Programming '93*, LNCS, Berlin, 1996. Springer. To appear.

[19] S. Mauw. An algebraic specification of process algebra. In J. A. B. M. Wirsing, editor, *Algebraic methods: theory, tools and applications*, volume 394 of *LNCS*, Berlin, 1989. Springer.

[20] J. Meseguer. A logical theory of concurrent objects and its realization in the Maude language. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 314–389. MIT Press, Cambridge, Massachusetts–London, 1991.

[21] J. Meseguer and T. Winkler. Parallel programming in Maude. In J. Banatre and D. le Metayer, editors, *Research Directions in High-Level Parallel Languages*, volume 574 of *LNCS*, pages 253–293, Berlin, 1992. Springer.

[22] S. Nakajima and K. Futatsugi. Constructing OBJ specifications with object-oriented design methodoly, 1996. To appear.

[23] G. Reggio. Entities: an institution for dynamic systems. In H. Ehrig, K. P. Jantke, F. Orejas, and H. Reichel, editors, *Recent Trends in Data Type Specification*, volume 534 of *LNCS*, pages 244–265, Berlin, 1991. Springer.

[24] M. Wirsing. Algebraic specification. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. B: Formal Models and Semantics*, pages 675–788. Elsevier, Amsterdam, 1990.

A Complete Maude Specification

```
fth STATE is
  sort State .
  op on: -> State .
  op off: -> State .
  op wait: -> State .
endft

fth DMM is
  protecting NAT .
  sort DM < NAT .
endft

omod RM is
  protecting DMM, STATE, CLIST, ILIST .

  class Usr .
  class Customer_Panel | state: State .
  class Receiver .
  class Current | list : IList, amount : DM .
  class Day_Total | list : IList .

  msg start: OId OId -> Msg .
  msg return: OId CItem OId -> Msg .
  msg new: OId Item OId -> Msg .
  msg add: OId Item OId .
  msg &: OId Item OId .
  msg ack: OId OId .
  msg receipt: OId OId .
  msg print_receipt: OId OId .
  msg get: OId OId .
  msg to: OId IList DM OId .
  msg send: OId IList DM OId .
  msg print: OId IList DM OId .

  vars Sb,Cp,Rc,Cur,Dt: OId .
  var A: State .
  var CI: CItem .
  var I: Item .
  var L: IList .
  var S: DM .

  rl [start] start(Sb,Cp)
             <Cp: Customer_Panel | state: A> =>
               <Cp: Customer_Panel | state: on>
                 if A = off .
  rl [return] return(Sb,Ci,Cp)
              <Cp: Customer_Panel | state: A> =>
                <Cp: Customer_Panel | state: on>
                new(Cp,I,Rc)
                  if I = desc(Ci) and A = on .
  rl [new] new(Cp,I,Rc)
           <Rc: Receiver> =>
             <Rc: Receiver>
             &(Rc,I,Cur)
             add(Rc,I,Dt) .
  rl [&] &(Rc,I,Cur)
```

```
          <Cur: Current | list: L, amount: S> =>
            <Cur: Current | list: I L,
                            amount: price(I)+S> .
            ack(Cur,Cp) .
  rl [add] add(Rr,I,Dt)
           <Dt: Day_Total | list: L> =>
             <Dt: Day_Total | list: I L> .
  rl [ack] ack(Cur,Cp)
           <Cp: Customer_Panel> =>
             <Cp: Customer_Panel>
  rl [receipt] receipt(Sb,Cp)
               <Cp: Customer_Panel | state: A> =>
                 <Cp: Customer_Panel | state: wait>
                 print_receipt(Cp,Rc)
                   if A = on .
  rl [print_receipt] print_receipt(Cp,Rc)
                     <Rc: Receiver> =>
                       <Rc: Receiver>
                       get(Rc,Cur) .
  rl [get] get(Rc,Cur)
           <Cur: Current | list: L, amount: S> =>
             <Cur: Current | list: nil, amount: 0>
             to(Cur,L,S,Rc) .
  rl [to] to(Cur,L,S,Rc)
          <Rc: Receiver> =>
            <Rc: Receiver>
            send(Rc,L,S,Cp) .
  rl [send] send(Rc,L,S,C)
            <Cp: Customer_Panel | state: A> =>
              <Cp: Customer_Panel | state: off>
              print(CP,L,S,SB)
                if A = wait .
endom
```

B Complete Java Program

```
class Foose extends Thread
{
  private protected int acc,got;
  private protected Object[] env;

  private protected void notifyenv()
  {
    for (int i=0;i<env.length;i++)
    { if (env[i]!=this)
      { synchronized (env[i])
        { env[i].notify(); } } }
  }

  private protected void accept(int s)
  {
    synchronized (this)
    { got=0; acc=s; }
    notifyenv();
    while ((got&acc)==0)
    { Thread.yield(); }
    acc=0;
  }
}

class CustomerPanel extends Foose
{
  private final static int START=1, RETURN=2, ACK=4,
                           RECEIPT=8, SEND=16;
  private Receiver rc;
  private State state;

  public CustomerPanel()
  {
    System.err.println("CustomerPanel");
  }

  public void reg(Receiver r,Object[] e)
  {
    rc=r; env=e;
  }

  public boolean mystart()
  {
    if ((START&acc)!=START)
      return false;
    else
    { if (state==OFF)
      { System.err.println("start()");
        synchronized (this)
        { got=START;
          notifyenv(); }
        state=ON;
        return true; }
      else
        return false; }
  }

  public boolean myreturn(CItem ci)
  {
    if ((RETURN&acc)!=RETURN)
      return false;
    else
    { if (state==ON)
      { Item i=ci.desc();
        System.err.println("return()");
        synchronized (this)
        { got=RETURN;
          notifyenv(); }
        while (!rc.mynew(i))
        { synchronized (this)
          { try { wait(); }
            catch (InterruptedException ignored) { } } }
        return true; }
      else
        return false; }
  }
```

```
public boolean ack()
{
  if ((ACK&acc)!=ACK)
    return false;
  else
  { System.err.println("ack()");
    synchronized (this)
    { got=ACK;
      notifyenv(); }
    return true; }
}

public boolean receipt()
{
  if ((RECEIPT&acc)!=RECEIPT)
    return false;
  else
  { if (state==ON)
    { System.err.println("receipt()");
      synchronized (this)
      { got=RECEIPT;
        notifyenv(); }
      state=WAIT;
      while (!rc.printreceipt())
        synchronized (this)
        { try { wait(); }
          catch (InterruptedException ignored) { } }
      return true; }
    else
      return false; }
}

public boolean send(IList l,int s)
{
  if ((SEND&acc)!=SEND)
    return false;
  else
  { if (state==WAIT)
    { System.err.println("send()");
      synchronized (this)
      { got=SEND;
        notifyenv(); }
      state=OFF;
      System.out.println(l);
      System.out.println(s);
      return true; }
    else
      return false; }
}

public void run()
{
  System.err.println("runcp");
  accept(START);
  while (true)
  { accept(RETURN|RECEIPT);
```

```
   if ((got&RECEIPT)==RECEIPT)
    break;
   accept(ACK); }
  accept(SEND);
 }
}
class Receiver extends Foose
{
 private final static int NEW=1, PRINTRECEIPT=2, TO=4;
 private CustomerPanel cp;
 private Current cur;
 private DayTotal dt;

 public Receiver()
 {
  System.err.println("Receiver");
 }

 public void reg(CustomerPanel c,Current u,
         DayTotal d,Object[] e)
 {
  cp=c; cur=u; dt=d; env=e;
 }

 public boolean mynew(Item i)
 {
  if ((NEW&acc)!=NEW)
   return false;
  else
  { System.err.println("new()");
   synchronized (this)
   { got=NEW;
    notifyenv(); }
   while (!dt.add(i))
   { synchronized (this)
    { try { wait(); }
     catch (InterruptedException ignored) { } } }
   while (!cur.conc(i))
   { synchronized (this)
    { try { wait(); }
     catch (InterruptedException ignored) { } } }
   return true; }
 }

 public boolean printreceipt()
 {
  if ((PRINTRECEIPT&acc)!=PRINTRECEIPT)
   return false;
  else
  { System.err.println("printreceipt()");
   synchronized (this)
   { got=PRINTRECEIPT;
    notifyenv(); }
   while (!cur.get())
   { synchronized (this)
    { try { wait(); }
```

```
     catch (InterruptedException ignored) { } } }
   return true; }
 }
 public boolean to(IList l,int s)
 {
  if ((TO&acc)!=TO)
   return false;
  else
  { System.err.println("to()");
   synchronized (this)
   { got=TO;
    notifyenv(); }
   while (!cp.send(l,s))
   { synchronized (this)
    { try { wait(); }
     catch (InterruptedException ignored) { } } }
   return true; }
 }
 public void run()
 {
  System.err.println("runrc");
  while ((got&PRINTRECEIPT)!=PRINTRECEIPT)
   accept(PRINTRECEIPT|NEW);
  accept(TO);
 }
}
class Current extends Foose
{
 private final static int CONC=1, GET=2;
 private CustomerPanel cp;
 private Receiver rc;
 private IList list;
 private int amount;
 public Current()
 {
  System.err.println("Current");
 }
 public void reg(CustomerPanel c,Receiver r,Object[] e)
 {
  cp=c; rc=r; env=e;
 }
 public boolean conc(Item i)
 {
  if ((CONC&acc)!=CONC)
   return false;
  else
  { System.err.println("conc()");
   synchronized (this)
   { got=CONC;
    notifyenv(); }
   list.cons(i);
   amount+=i.price();
```

```
      while (!cp.ack())
      { synchronized (this)
        { try { wait(); }
          catch (InterruptedException ignored) { } } }
      return true; }
  }

  public boolean get()
  {
    if ((GET&acc)!=GET)
      return false;
    else
    { System.err.println("get()");
      synchronized (this)
      { got=GET;
        notifyenv(); }
      list=null;
      amount=0;
      while (!rc.to())
      { synchronized (this)
        { try { wait(); }
          catch (InterruptedException ignored) { } } }
      return true; }
  }

  public void run()
  {
    System.err.println("runcur");
    while ((got&GET)!=GET)
      accept(CONC|GET);
  }
}

class DayTotal extends Foose
{
  private final static int ADD=1;
  private IList list;

  public DayTotal()
  {
    System.out.println("DayTotal");
  }

  public void reg(Object[] e)
  {
    env=e;
  }

  public boolean add(Item i)
  {
    if ((ADD&acc)!=ADD)
      return false;
    else
    { System.out.println("add()");
      synchronized (this)
      { got=ADD;
        notifyenv(); }
      list.cons(i);
      return true; }
  }

  public void run()
  {
    System.out.println("rundt");
    while (true)
      accept(ADD);
  }
}

class User extends Thread
{
  private CustomerPanel cp;
  private Receiver rc;
  private Current cur;
  private DayTotal dt;
  private Object[] env;

  public void run()
  {
    cp=new CustomerPanel();
    rc=new Receiver();
    cur=new Current();
    dt=new DayTotal();

    env=new Object[5];
    env[0]=cp;
    env[1]=rc;
    env[2]=cur;
    env[3]=dt;
    env[4]=this;

    cp.reg(rc,env);
    rc.reg(cp,cur,dt,env);
    cur.reg(cp,rc,env);
    dt.reg(env);

    dt.start();
    System.out.println("dt");
    cp.start();
    System.out.println("cp");
    rc.start();
    System.out.println("rc");
    cur.start();
    System.out.println("cur");

    while (!cp.mystart())
    { System.out.println("start?");
      synchronized (this)
      { try { wait(); }
        catch (InterruptedException ignored) { } } }
    while (!cp.myreturn(new CItem("Item 1")))
    { System.out.println("return1?");
      synchronized (this)
      { try { wait(); }
        catch (InterruptedException ignored) { } } }
    while (!cp.myreturn(new CItem("Item 2")))
    { System.out.println("return2?");
```

```
      synchronized (this)
      { try { wait(); }
        catch (InterruptedException ignored) { } } }
    while (!cp.myreturn(new CItem("Item 3")))
    { System.out.println("return3?");
      synchronized (this)
      { try { wait(); }
        catch (InterruptedException ignored) { } } }
    while (!cp.receipt())
    { System.out.println("receipt?");
      synchronized (this)
      { try { wait(); }
        catch (InterruptedException ignored) { } } }
  }
}

public class RecyclingMachine extends java.applet.Applet
{
  public void start()
  {
    System.out.println("Main");

    new Thread(new User()).start();
  }
}
```

The following is a sample debugging output of this Java program (on System.err):

```
Main           return()       ack()
CustomerPanel  new()          receipt?
Receiver       add()          receipt()
Current        conc()         printreceipt()
DayTotal       ack()          get()
dt             return2?       to()
cp             return()       send()
rc             new()
cur            add()
start?         conc()
rundt          ack()
runcp          return3?
runrc          return()
runcur         new()
start()        add()
return1?       conc()
```