A formal approach to object-oriented software engineering ☆

Martin Wirsing*, Alexander Knapp

Institut für Informatik, Ludwig-Maximilians-Universität München, Oettingenstraße 67,
D-80538 München, Germany

Abstract

We show how formal specifications can be integrated into one of the current pragmatic object-oriented software development methods. Jacobson's "object-oriented software engineering" process 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 a meta-level 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 at different abstraction levels can be formally stated and proved. © 2002 Elsevier Science B.V. All rights reserved.

Keywords: Integrated formal software engineering; OOSE; Rewriting logic; Maude; Reflection; Process algebra

1. Introduction

Current object-oriented software design methods, such as those of Rumbaugh (OMT [39]), Shlaer-Mellor [40], Jacobson (OOSE [21]), and Booch [6], or the "unified process" (UP [20]) use a combination of diagrammatic notations including object and class diagrams, state transition diagrams, and scenarios to describe software models. Other, mostly academic, approaches such as Reggio's entity algebras [37], Meseguer's Maude [31], Ehrich's and Sernadas's TROLL (see e.g. [18]), or model-oriented specification techniques like Z++ or VDM++ (see [26]) propose fully formal descriptions

---

for software 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 of software models. 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 software designs.

To close partly this gap, we investigate the combination of formal specification techniques with pragmatic software engineering methods and notations. Our specification techniques are based on Meseguer's rewriting logic [31] and are written in a meta-level extension of the object-oriented algebraic specification language Maude [10]. The static and functional parts of a software system are described by classical algebraic specifications whereas the dynamic behaviour is modelled by non-deterministic rewriting of configurations of communicating objects; the communication flow is controlled by process expressions. These object-oriented algebraic specifications are integrated with Jacobson's "object-oriented software engineering" (OOSE [21]) process and the variants of the OOSE modelling notations as defined by the "unified modeling language" (UML [35]). Leaving unchanged the basic method of OOSE, the development process of our enhanced "formal object-oriented software engineering" (fOOSE) method consists of the traditional five phases: requirements analysis, robustness analysis, design, implementation, and test. However, in the fOOSE method, every diagram 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. Since every fOOSE diagram is thus accompanied by a formal specification, every document has a formal meaning. In many cases the formal specification generates proof obligations which give additional means for validating the current design 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, i.e., proof obligation discharge steps can be interweaved with traditional development steps. Moreover, due to the choice of the executable specification language Maude early prototyping is possible during analysis and design. Thus, the combination of algebraic specification with rewriting gives a coherent view of object-oriented design and implementation. Formal specification techniques complement diagrammatic ones. The integration of both leads to an improved design and provides new techniques for prototyping and testing.

1.1. Related work

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 [16]. Our approach, based on Meseguer's rewriting logic and Maude, was inspired by Astesiano's SMoLCS approach [3,37] which can be characterised as a combination of algebraic

specifications with transition systems instead of rewriting and which also has been investigated in the environment of the specification language CASL [2]. Astesiano was the first author integrating also process expressions in his framework; similarly, PCF [30] and LOTOS [8] combine process expressions with algebraic specifications. A process algebra for controlling the flow of messages has also been discussed for Maude [29]; by using an appropriate extension of the μ-calculus Lechner [28] presents a more abstract approach for describing object-oriented requirements and designs on top of Maude. The use of more general strategies together with rewriting logic has been studied for the specification language ELAN [7] as well as for Maude [9].

There are also several approaches for integrating pragmatic software engineering methods and notations with formal techniques [22]. Hußmann [19] gives a formal foundation of the structured analysis method SSADM. Coleman's object-oriented Fusion [12] introduces model annotations and object life-cycle expressions; Cook and Daniels' object-oriented Syntropy method [13] is based on Z and state charts. Turning Fusion and Syntropy into integrated formal software engineering methods, Lano et al. [27] formalise the notations and introduce explicit proof obligation discharge and refinement verification steps into these methods. Similarly, Dodani and Rupp [14] enhance the Fusion method by formal specifications written in the algebraic specification language COLD and Achatz and Schulte [1] enhance the semi-formal Fusion notations by formal annotations in Object-Z and also extend Fusion by validation steps. Nakajima and Futatsugi [34] integrate a scenario-based methodology, similar to the OOSE process, with algebraic specifications using the CafeOBJ language and a variant of Z, but neither address refinement steps nor proof obligations directly. Lano [26] presents a formal approach to object-oriented software development based on Z++ and VDM++. Reggio and Repetto [38] investigate a combination of CASL with state charts as an instance of a more general approach for composing languages in such a way that syntax, semantics and methodology of the components are preserved [2].

Our novel combination of object-oriented algebraic specification and rewriting logic techniques, the UML notation, and the OOSE process shows several advantageous features, technically as well as pragmatically: The Maude language allows for early rapid prototyping and executability. The UML notation has become the "lingua franca" of software engineering. The semantics of the UML, though not settled decisively, has been discussed to some extent; for class and object diagrams see, e.g., Kim and Carrington [23], for sequence diagrams and interactions Övergaard [36] and Knapp [25]. The UML also proposes the "Object Constraint Language" (OCL [41]) as a formal, simple, declarative language for side-conditions on diagrams; however, a formal semantics is under investigation. Finally, the core features of OOSE can be considered as paradigmatic for other current object-oriented software development methods such as the UP.

1.2. Synopsis

The paper is organised as follows: Section 2 gives a short introduction to our chosen specification language Maude. Section 3 describes a Maude meta-level extension with means for controlling the flow of messages. In Section 4 we present our enhanced

development method fOOSE; the details of our method for developing a formal specification out of an informal description of a use case is illustrated by the example of a recycling machine which is the running example of Jacobson's book on OOSE. Section 5 ends with some concluding remarks.

2. Maude

The object-oriented algebraic specification language Maude [10] consists of two parts: a data type oriented functional part and a state oriented system part. The functional part is based on membership equational logic [32]; it serves for specifying algebraic data types by sorts and conditional equations. The system part allows to interpret algebraically defined terms as states and to introduce transition rules on these states; it is based on rewriting logic [31]. Object-oriented concepts, such as classes, inheritance, objects, and messages are defined by means of the functional and the system part [15]. Moreover, Maude is reflective: specifications can be considered as terms on a meta-level and may thus be introspected and manipulated.

In the following, we briefly summarise the main features of Maude by means of some examples and recapitulate the foundations of Maude specifications; for a more detailed treatment see [10], for basic definitions and facts on algebraic specifications in general cf. [42]. The syntax of the sample Maude specifications has been beautified in order to avoid some idiosyncrasies of the Maude naming conventions.

2.1. Functional specifications

Maude provides two different kinds of functional specifications: "modules" and "theories", that may also be parameterised. A module (keyword `fmod ... endfm`) contains an import list (`protecting` or `including`), sorts (`sort`), subsorts (`subsort ... < ...`), function (`op`) and variable declarations (`var`), (conditional) membership axioms (`mb`, `cmb`), and (conditional) equations (`eq`, `ceq`), axiomatising data types. Theories have different keywords (viz. `fth ... endfth`) but have otherwise the same syntax. A functional Maude module is executable; its equations are assumed to be Church-Rosser and terminating. In contrast, a functional Maude theory is not executable; it gives only a few characteristic properties ("requirements") the specified data type has to fulfil.

The following example specifies a trivial theory `TRIV` which introduces a single sort `Elt`, and a module `LIST` parameterised by `TRIV` for the data structure of lists with elements of sort `Elt`:

```
fth TRIV is
  sort Elt .
endfth

fmod LIST[X :: TRIV] is
  protecting NAT .
  protecting BOOL .
```

```
    sort List[X] .
    subsort Elt.X < List[X] .
    op nil : -> List[X] .
    op __ : List[X] List[X] -> List[X] [assoc id: nil] .
    op length : List[X] -> Nat .
    op _in_ : Elt.X List[X] -> Bool .
    op _\_ : List[X] Elt.X -> List[X] .
    op _<=_ : List[X] List[X] -> Bool .
    vars E E' : Elt.X .
    vars L L' : List[X] .
    eq length(nil) = 0 .
    eq length(E L) = 1 + length(L) .
    eq E in nil = false .
    eq E in (E' L) = (E == E') or (E in L) .
    eq nil \ E = nil .
    eq (E L) \ E' = if (E == E') then L else (E (L \ E')) fi .
    eq (nil <= L) = true .
    eq (E L) <= (E' L') = (E in (E' L')) and (L <= ((E' L') \ E)) .
  endfm
```

A parameterised module can be instantiated by a view from the formal parameter theory to an actual parameter module, like

```
  view Nat from TRIV to NAT is
    sort Elt to Nat .
  endv
```

which allows to use a module expression like LIST[Nat] (see [10,15]).

2.2. Object-oriented specifications

Object-oriented concepts are supported in Maude by means of object-oriented modules. Extending the features of functional modules, the declaration of an object-oriented module (omod ... endom) may contain a number of class declarations (class), subclass declarations (subclass), message declarations (msg), and (conditional) rewrite rules (rl, crl). More precisely, object-oriented modules are a special kind of so-called system modules which only provide the use of equations and rewrite rules; object-oriented modules can be defined in terms of system modules [31,10,15].

The following example specifies a bounded buffer of natural numbers:

```
  omod BUFFER is
    protecting LIST[Nat] .
    class Buffer | contents : List[Nat] .
    msg put_in_ : Nat Oid -> Msg .
    msg getfrom_replyto_ : Oid Oid -> Msg .
    msg to_elt-in_is_ : Oid Oid Nat -> Msg .
    op k : -> Nat .
    op b : -> Oid .
    vars B I : Oid .
```

```
    var E : Nat .
    var Q : List[Nat] .
    crl [put] :
      (put E in B)
      < B : Buffer | contents : Q > =>
        < B : Buffer | contents : E Q >
      if length(Q) < k .
    rl [get] :
      (getfrom B replyto I)
      < B : Buffer | contents : Q E > =>
        < B : Buffer | contents : Q >
        (to I elt-in B is E) .
  endom
```

A class is declared by an identifier of sort `Cid` and a list of attributes and their corresponding sorts. An object is represented by a term comprising a unique object identifier of sort `Oid`, the identifier for the class the object belongs to, and a set of attributes with their corresponding values, e.g. `< b : Buffer | contents : nil >`. A message is a term of sort `Msg`, in general consisting of the message's name, the identifiers of the objects the message is addressed to, and, possibly, parameters (in mix-fix notation), e.g. `put 0 in b`.

A Maude program makes computational progress by rewriting its global state, called a configuration, which is represented as a term of sort `Configuration`. A configuration is a multiset of objects and messages; multiset union is denoted by juxtaposition. A rewrite rule

```
crl [l] :
    M₁ ⋯ Mₘ
    < O₁ : C₁ | atts₁ > ⋯ < Oₙ : Cₙ | attsₙ > =>
      < Oᵢ₁ : Cᵢ₁ | atts'ᵢ₁ > ⋯ < Oᵢₖ : Cᵢₖ | atts'ᵢₖ >
      < Q₁ : D₁ | atts''₁ > ⋯ < Qₚ : Dₚ | atts''ₚ >
      M'₁ ⋯ M'q
    if C .
```

transforms a configuration into a subsequent configuration. It accepts messages $M_1, \ldots, M_m$ for some objects $O_1, \ldots, O_n$ under a certain condition C, possibly modifies or deletes these objects, and may create new objects $Q_1, \ldots, Q_p$ and messages $M'_1, \ldots, M'_q$.

2.3. Reflection

Maude specifications are also accessible to manipulation and reasoning in Maude itself by the reflective design of the language. The special Maude functional module `META-LEVEL` defines data structures to hold whole modules as well as single terms, provides functions to construct new modules and terms or changing existing ones, and allows for rewriting on these meta-representations.

For instance, the term

```
(put 0 in b) < b : Buffer | contents : nil >
```

of sort `Configuration` is represented at the meta-level as a term

```
'__['put_in_[{'O}'Nat, {'b}'Oid],
  '<_:_|_>[{'b}'Oid, {'Buffer}'Buffer,
             'contents:_[{'nil}'List[Nat]]]]
```

of sort `Term`. Constants are meta-represented as terms by a function `{_}_ : Qid Qid -> Term`, where the first quoted identifier is the name and the second quoted identifier the sort of the constant; these terms are atomic. The function `_[_] : Qid TermList -> Term` constructs more complex terms.

Functional and system modules (and thus also object-oriented modules) can be represented at the meta-level as terms of the sorts `FModule` and `Module`, respectively, where `FModule` is a subsort of `Module`. Maude provides a special operator up which, given a name of a module in a Maude specification, yields the meta-level representation of this module; and, given a name of a module in a Maude specification and a term of this module, yields the meta-level representation of this term.

Equational reduction and rewriting are reflected on the meta-level by so-called descent functions, most importantly

```
op meta-apply : Module Term
                 Qid Substitution MachineInt -> ResultPair .
```

A term `meta-apply`(M, t, q, σ, n) is evaluated as follows [10]: The meta-term t is converted to its concrete term with respect to the meta-module M and fully reduced using the equations in the module represented by M; all rewrite rules of the module represented by M which carry meta-label q are (partially) instantiated by substitution σ and the resulting rules are matched against the reduced term represented by t; the first n successful matches are discarded and, if there is an $(n+1)$th successful match, its match is applied, the resulting term is fully reduced using the equations in the module represented by M, and the meta-level representations of the final term and the matching substitution are returned as a result pair; if there is no $(n+1)$th successful matching, the result pair { `error*, none` } is returned.

For example, `meta-apply`(mod, term, 'put, none, 0), where the meta-term mod represents the module `BUFFER` with the constant k instantiated to, say, 5 and term is the meta-representation of the configuration (put 0 in b) < b : Buffer | contents : nil >, as shown above, yields a pair of the meta-term

```
'<_:_|_>[{'b}'Oid, {'Buffer}'Buffer, 'contents:_[{'O}'Nat]]
```

with the substitution

```
('E <- {'O}'Nat); ('B <- {'b}'Oid); ('Q <- {'nil}'List[Nat]); ...
```

(where we omit those parts of the substitution that only pertain to the internal Maude representation of object-oriented modules by system modules).

2.4. Membership equational theories and rewrite theories

2.4.1. Membership equational theories

Functional Maude modules and theories define equational theories in membership equational logic [15]. A membership equational theory is given by a pair (Ω, Γ) where Ω is a membership equational signature and Γ is a set of membership equational axioms. A membership equational signature is a triple (K, Σ, S) with K a set of kinds, (K, Σ) a many-kinded signature, and $S = (S_k)_{k \in K}$ a K-kinded set of sorts. An atomic membership equational formula is either an Ω-equation $t = t'$ where the terms t and t' have both the same kind, or an Ω-membership assertion $t : s$, where term t has kind k and $s \in S_k$. An Ω-membership equational formula is either a conditional membership equation

$$t = t' \Leftarrow \bigwedge_i (u_i = v_i) \wedge \bigwedge_j (w_j : s_j)$$

with terms over Ω, or a conditional membership axiom

$$t : s \Leftarrow \bigwedge_i (u_i = v_i) \wedge \bigwedge_j (w_j : s_j)$$

with terms over Ω.

A membership equational theory (Ω, Γ) entails an Ω-membership equational formula φ, written as $(\Omega, \Gamma) \vdash \varphi$, if φ can be deduced from Γ in membership equational logic [32].

2.4.2. Rewrite theories

Maude system modules, and hence object-oriented modules, define rewrite theories [15]. A rewrite theory is given by a pair $((\Omega, \Gamma), P)$ where (Ω, Γ) is a membership equational theory with a signature Ω and a set of membership equational formulas Γ, and P is a set of labelled rewrite rules. A rewrite rule

$$l : t \rightarrow t' \Leftarrow \Pi$$

is given by a label l, two terms t and t' such that t and t' have the same kind, and a condition Π which is a conjunction of atomic membership equational formulas.

For a rewrite theory $\Xi = ((\Omega, \Gamma), P)$, deduction, i.e. rewriting, takes place according to rewriting logic defined by the following four rules (cf. [31]):
(1) Reflexivity.

$$\overline{[t] \rightarrow [t]}$$

(2) Congruence. For each function symbol $f : s_1 \ldots s_n \rightarrow s$ in Ω

$$\frac{[t_1] \rightarrow [u_1], \ldots, [t_n] \rightarrow [u_n]}{[f(t_1, \ldots, t_n)] \rightarrow [f(u_1, \ldots, u_n)]}$$

(3) Replacement. For each rewrite rule $l : t \to u \Leftarrow \Pi$ in P

$$\frac{[t_1] \to [u_1], \ldots, [t_n] \to [u_n]}{[t(t_1, \ldots, t_n)] \to [u(u_1, \ldots, u_n)]}, \quad \text{if } \Pi(t_1, \ldots, t_n)$$

(4) Composition.

$$\frac{[t_1] \to [t_2], \; [t_2] \to [t_3]}{[t_1] \to [t_3]}$$

where $[t]$ denotes the congruence class of term t with respect to Γ.

   A rewrite theory $\Xi = ((\Omega, \Gamma), P)$ entails a sequent $t \to t'$, written as $\Xi \vdash t \to t'$, if $[t] \to [t']$ can be obtained from Ξ by finite application of the rules (1–4) above. Such a sequent is called a rewrite; a rewrite step is called a one-step concurrent rewrite if it can be derived from Ξ by a finite number of applications of the rules (1)–(3), with at least one application of the replacement rule (3); a one-step concurrent rewrite is called a sequential rewrite if it can be derived with exactly one application of (3). Every rewrite can be decomposed into an interleaving sequence of sequential rewrites [31]. A run of Ξ is a finite or infinite sequence

$$t_1 \to t_2 \to t_3 \to \cdots$$

of rewrites with $\Xi \vdash t_k \to t_{k+1}$ for every $k \geqslant 1$ and, if the sequence terminates with t_n, then $\Xi \vdash t_n \to t_{n+1}$ implies $(\Omega, \Gamma) \vdash t_n = t_{n+1}$; such a run is said to start in t_1.

2.4.3. Reflection

   Rewriting logic is reflective [9]: There is a finitely presented rewrite theory $Y = ((\Omega, \Gamma), P)$, i.e., Ω, Γ, and P are all finite, such that any pair of a finitely represented rewrite theory Ξ and a Ξ-term t can be represented as an Y-term $\langle \bar{\bar{\Xi}}, \bar{t} \rangle$ and

$$\Xi \vdash t \to t' \quad \text{if, and only if} \quad Y \vdash \langle \bar{\bar{\Xi}}, \bar{t} \rangle \to \langle \bar{\bar{\Xi}}, \overline{t'} \rangle.$$

3. Object theories and process control

   The reflective capabilities of the Maude language suggest to separate declaratively: logic, i.e. the rewrite rules, and control, i.e. the order of rewrite rule applications, in Maude specifications [11].

   We introduce a small control language that is based on process algebra [5] and that allows to monitor the messages that are consumed and produced by rule applications and thus allows to describe admissible rewrite runs. This simple control is mainly suited for object-oriented Maude specifications that only use flat configurations of objects and messages, that is, objects and messages do not contain further configurations; such Maude specifications define so-called object theories [33].

3.1. Object theories

An object theory is a rewrite theory $\Xi = ((\Omega, \Gamma), P)$ satisfying the following conditions: The signature Ω is an object signature, i.e., it contains a distinguished sort C of configurations and a distinguished sort G of messages such that G is a subsort of C; configurations are equipped with a constant $\emptyset : \to C$ and a binary operation $- - : C \times C \to C$ (written as juxtaposition); no other operation in Ω takes elements of sort C as arguments; and for any set X of variables, configurations form a free multiset under the operation $- -$ with neutral element $\emptyset$ in the free algebra over X [32]. The rewrite rules P all have the form $l : z \to z' \Leftarrow \Pi$, where Π is a conjunction of atomic membership equational formulas, $z, z' : C$, $z \neq \emptyset$, and z and z' contain no variables of sort C.

Let $\Xi = ((\Omega, \Gamma), P)$ be an object theory with configuration sort C and message sort G and let M be the set of function symbols with co-arity G. For two configurations c and c' in C and a sequential rewrite $\Xi \vdash c \to c'$ we write

$$\Xi \vdash c \overset{?I\ !O}{\to} c'$$

if I is the multiset of message function symbols in M that occur in c but not in c', and O is the multiset of message function symbols in M that occur in c' but not in c. More generally, for a non-sequential rewrite $\Xi \vdash c \to c'$ we write

$$\Xi \vdash c \overset{?I\ !O}{\twoheadrightarrow} c'$$

if there is a sequence $c = c_0, c_1, \ldots, c_n, c_{n+1} = c'$ of configurations in C such that $c_i \to c_{i+1}$ is a sequential rewrite and $\Xi \vdash c_i \overset{?I_i\ !O_i}{\longrightarrow} c_{i+1}$ for $0 \leqslant i \leqslant n$ and the multiset union of all I_i is I and the multiset union of all O_i is O.

Informally, I denotes the messages that are consumed in this rule application and O denotes the messages that are produced. For example, the specification BUFFER in Section 2.2 defines an object theory with configuration sort Configuration and message sort Msg. Thus, we have

$$\text{BUFFER} \vdash \text{(put 0 in b)} < \text{b : Buffer} \mid \text{contents : nil} > \overset{\text{?put\_in\_}}{\longrightarrow}$$
$$< \text{b : Buffer} \mid \text{contents : 0} >,$$

$$\text{BUFFER} \vdash \text{(getfrom b replyto i)} < \text{b : Buffer} \mid \text{contents : 0} >$$
$$\overset{\text{?getfrom\_replyto\_ !to\_elt-in\_is\_}}{\longrightarrow}$$
$$< \text{b : Buffer} \mid \text{contents : nil} > \text{(to i elt-in b is 0)}$$

A run of an object theory $\Xi = ((\Omega, \Gamma), P)$ with configuration sort C, message sort G, and M the set of functional symbols with co-arity G is a finite or infinite sequence

$$c_1 \overset{A_1}{\twoheadrightarrow} c_2 \overset{A_2}{\twoheadrightarrow} \cdots$$

Table 1
Operational semantics of processes

$$\frac{}{\mathrm{o}\sqrt{}}, \quad \frac{p_1\sqrt{},\ p_2\sqrt{}}{p_1;p_2\sqrt{}}, \quad \frac{p_1\sqrt{}}{p_1+p_2\sqrt{}}, \quad \frac{p_2\sqrt{}}{p_1+p_2\sqrt{}}, \quad \frac{p_1\sqrt{},\ p_2\sqrt{}}{p_1\|p_2\sqrt{}}, \quad \frac{}{p^*\sqrt{}}$$

$$\frac{}{\mu \xrightarrow{\mu} \mathrm{o}}, \quad \frac{p_1 \xrightarrow{\mu} p_1'}{p_1;p_2 \xrightarrow{\mu} p_1';p_2}, \quad \frac{p_1\sqrt{},\ p_2 \xrightarrow{\mu} p_2'}{p_1;p_2 \xrightarrow{\mu} p_2'}, \quad \frac{p \xrightarrow{\mu} p'}{p^* \xrightarrow{\mu} p';p^*}$$

$$\frac{p_1 \xrightarrow{\mu} p_1'}{p_1+p_2 \xrightarrow{\mu} p_1'}, \quad \frac{p_2 \xrightarrow{\mu} p_2'}{p_1+p_2 \xrightarrow{\mu} p_2'}$$

$$\frac{p_1 \xrightarrow{\mu} p_1'}{p_1\|p_2 \xrightarrow{\mu} p_1'\|p_2}, \quad \frac{p_2 \xrightarrow{\mu} p_2'}{p_1\|p_2 \xrightarrow{\mu} p_1\|p_2'}$$

where the multisets A_k consist of symbols $?m$ and $!m$ with $m \in M$ and $\Xi \vdash c_k \xrightarrow{?I_k\ !O_k} c_{k+1}$ with I_k the multiset of all message symbols m such that $?m$ occurs in A_k and O_k the multiset of all message symbols m such that $!m$ occurs in A_k and, if the sequence terminates with c_n, then $\Xi \vdash c_n \rightarrow c_{n+1}$ implies $(\Omega, \Gamma) \vdash c_n = c_{n+1}$.

3.2. Process control

A process control expression for an object theory defines constraints on the messages that have to be produced and consumed in all runs of the object theory.

Let $\Xi = ((\Omega, \Gamma), P)$ be an object theory with message sort G and let M be the set of function symbols with co-arity G. An action for Ξ is a symbol of the form $?m$ or $!m$ with $m \in M$, meaning that some message m has to be consumed or to be produced. A process for Ξ is either a constant o denoting the successfully terminated process, or a constant δ for deadlock, or an action, or a composite process. A composite process may be formed by sequential composition, nondeterministic choice, parallel composition, or iteration of processes. The abstract syntax of processes over a set of message symbols M is given by

$$\mathrm{Act} ::= ?M \mid !M$$
$$\mathrm{Pr} ::= \mathrm{o} \mid \delta \mid \mathrm{Act} \mid \mathrm{Pr};\mathrm{Pr} \mid \mathrm{Pr}+\mathrm{Pr} \mid \mathrm{Pr}\|\mathrm{Pr} \mid \mathrm{Pr}^*$$

where $*$ has highest precedence followed by $;, \|$, and $+$.

The operational semantics of processes is inductively defined in Table 1 (see e.g. [4]): Processes that may terminate immediately are captured by a predicate $\sqrt{}:\mathrm{Pr}$, written in postfix notation. A process p accepts a single atom μ and results thereby in a process p' if $p \xrightarrow{\mu} p'$ is derivable from the axioms and rules of the operational semantics of processes. Note that the rules for parallel composition induce an interleaving approach to concurrency.

More generally, a trace of a process p is a sequence of actions $\mu_1, \mu_2, \ldots$ such that $p \xrightarrow{\mu_1} p_2, p_2 \xrightarrow{\mu_2} p_3$, etc. holds for some processes p_k. We say that a process p accepts a multiset of actions A thereby resulting in a process p', written as $p \xrightarrow{A} p'$, if $p \xrightarrow{\mu_1} p_2$, $p_2 \xrightarrow{\mu_2} p_3, \ldots, p_{n-1} \xrightarrow{\mu_n} p'$, for some processes $p_2, \ldots, p_{n-1}$ and some actions $\mu_1, \ldots, \mu_n$ such that the multiset of these actions is A.

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

Definition 1. An object theory with control $Z = (\Xi, p, q_0)$ is an object theory $\Xi = ((\Omega, \Gamma), P)$, with configuration sort C and message sort G, together with a process definition p for Ξ, and an initial configuration q_0.

The object theory with control $Z = (\Xi, p, q_0)$ entails a run $q_0 \xrightarrow{A_0} q_1 \xrightarrow{A_1} \cdots$, written as $Z \vdash q_0 \xrightarrow{A_0} q_1 \xrightarrow{A_1} \cdots$, if $q_0 \xrightarrow{A_0} q_1 \xrightarrow{A_1} \cdots$ is a run of Ξ and there are processes p_k with $p_0 = p$ such that $p_k \xrightarrow{A_i} p_{k+1}$ for all k.

For instance, the object theory Ξ defined by the module BUFFER in Section 2.2 defines an object theory with control if equipped with process

```
?put_in_*; ?getfrom_replyto_; !to_elt-in_is_
```

and initial configuration

```
< b : Buffer | contents : nil >
(put 0 in b) (put 1 in b) (put 2 in b)
(getfrom b replyto i)
```

3.3. Maude implementation

Object theories with control can be implemented in Maude by reflection. First of all, the operational semantics of processes may be captured by

```
fmod ACTION is
  including QID .
  sort Action .

  op act : Qid -> Action .
endfm

view Action from TRIV to ACTION is
  sort Elt to Action .
endv

mod PROCESS is
  protecting SET[Action] .  - sets of actions
  protecting BAG[Action] .  - multisets of actions
```

```
    op nil : -> Process .
    op deadlock : -> Process .
    op _;_ : Process Process -> Process .
    op _+_ : Process Process -> Process .
    op _|_ : Process Process -> Process .
    op _* : Process -> Process .

    op terminates : Process -> Bool .
    op hd : Process -> Set[Action] .
    op tl : Action Process -> Process .

    op accept : Bag[Action] Process -> Process .

    ...
endm
```

where the operation `terminates` of PROCESS corresponds to $\sqrt{}$, `hd` computes all actions accepted by a process, `tl` yields the resulting process after accepting an action, and `accept` returns the resulting process after accepting a multiset of actions in an arbitrary order.

Thus, a general meta-rewrite engine may be defined that, given a Maude object-oriented module inducing an object theory (meta-represented by a `Module` term), a control process (of sort `Process`), and an initial configuration (meta-represented by a term of sort `Term`), computes all pairs of meta-configurations and processes such that a meta-configuration of a resulting pair is obtained by rewriting the initial meta-configuration (using `meta-apply`) according to the control process and no rewrite rule of the object-oriented module is applicable:

```
  fth MOD-CONTROL is
    including META-LEVEL .
    protecting PROCESS .

    op mod : -> Module .
    op control : -> Process .
    op term : -> Term .
  endfth

  fmod ENGINE[M :: MOD-CONTROL] is
    protecting SET[Qid] .   - sets of quoted identifiers
    protecting SET[Pair[Term, Process]] .
                                    - sets of pairs of terms and processes
    ...

    op getLabels : Module -> Set[Qid] .
                                    - labels of rewrite rules in a module
    ...
```

```
op inoutMsg : Term Term -> Bag[Action] .
                              - messages consumed and produced
...

op allRewrites : Set[Pair[Term, Process]] ->
                    Set[Pair[Term, Process]] .
op allRewrites : Pair[Term, Process] Set[Pair[Term, Process]] ->
                    Set[Pair[Term, Process]] .
op allRewrites : Term Process -> Set[Pair[Term, Process]] .
op allRewrites : Set[Qid] Term Process ->
                    Set[Pair[Term, Process]] .
op allRewrites : Qid Set[Qid] Term Process MachineInt ->
                    Set[Pair[Term, Process]] .

var P : Process .
var QS : Set[Qid] .
var N : MachineInt .
var TP : Pair[Term, Process] .
var TPS : Set[Pair[Term, Process]] .


eq allRewrites(TPS) =
    if (TPS == empty)
    then empty
    else allRewrites(choose(TPS), TPS) fi .
eq allRewrites(TP, TPS) =
    allRewrites(fst(TP), snd(TP)) union
      allRewrites(TPS \ { TP }) .
eq allRewrites(T, P) = allRewrites(getLabels(mod), T, P) .
eq allRewrites(QS, T, P) =
    if (QS == empty)
    then empty
    else allRewrites(choose(QS), QS, T, P, 0) fi .
eq allRewrites(Q, QS, T, P, N) =
    if (fst(meta-apply(mod, T, Q, none, N)) == error*)
    then allRewrites(QS \ { Q }, T, P)
    else (if (accept(
                inoutMsg(T, fst(meta-apply(mod, T, Q, none, N))),
              P) == deadlock)
          then allRewrites(Q, QS, T, P, N + 1)
          else { < fst(meta-apply(mod, T, Q, none, N)),
                   accept(inoutMsg(T,
                   fst(meta-apply(mod, T, Q, none, N))), P) > }
              union allRewrites(Q, QS, T, P, N + 1) fi) fi .

op accepting : -> Set[Pair[Term, Process]] .
op accepting : Set[Pair[Term, Process]] ->
                 Set[Pair[Term, Process]] .
```

```
    eq accepting =
        accepting({ < meta-reduce(mod, term), control > }) .
    eq accepting(TPS) =
        if (TPS == empty)
        then empty
        else union(final(TPS),
                    accepting(allRewrites(reduce(TPS)))) fi .

    ...
  endfm
```

The function `allRewrites` on `Set[Pair[Term, Process]]` repeatedly chooses some pair TP of a meta-configuration term and its adhering process, tries to apply all rewrite rules to this meta-configuration term (`getLabels(mod)`) and checks whether such a rewrite rule is currently allowed by the control process (`accept(...)`); if a rewrite rule succeeds for a given meta-configuration term and its process the rewritten term and the changed process are added to the current set of pairs of meta-configuration terms and processes (`union`). The function `accepting` returns all final pairs of configurations and processes (`final(...)`) from the meta-term `term` with respect to the module `mod` (`meta-reduce(...)`) and the process term `control`, removing all those pairs configurations and processes that are deadlocked (`reduce(...)`).

This general engine is instantiated as follows: Given an object-oriented Maude module M, a control module specifies the initial configuration and the control process:

```
  omod M-CONTROL is
    protecting M .
    protecting PROCESS .

    op initial : -> Configuration .
    eq initial = ... .

    op control : -> Process .
    eq control = ... .
  endom
```

The control module is transferred to the meta-level that can be used as the actual parameter for the engine:

```
  mod META-M-CONTROL is
    including META-LEVEL .
    protecting M-CONTROL .

    op mod : -> Module .
    eq mod = up(M-CONTROL) .
```

```
  op term : -> Term .
  eq term = up(M-CONTROL, initial) .
endm

view M-CONTROL from MOD-CONTROL to META-M-CONTROL is
  op mod to mod .
  op term to term .
  op control to control .
endv
```

Finally, all accepting rewrite sequences can be calculated by

```
reduce-in ENGINE[M-CONTROL] : accepting .
```

The complete Maude specification can be found in [24].

3.4. Refinement

The principal notion for expressing the correctness of a system wrt. its requirements is the notion of refinement. Object theories with control may be refined both at the level of object theories and at the control level.

Processes are refined by substituting complex processes for actions:

Definition 2. Let p and p' be processes over the message symbols M and M', respectively, and let A be the actions of p. The process p' is a process refinement of p if there is a function $\rho : A \to \mathrm{Pr}$ such that every trace of $\rho^*(p)$ is a trace of p' where ρ^* is the homomorphic extension of ρ to processes.

A refinement of object theories with control is based on a process refinement and an abstraction relation of the underlying object theories: Let $\varXi = ((\varOmega, \varGamma), P)$ and $\varXi' = ((\varOmega', \varGamma'), P')$ be object theories. A surjective, partial function α from the configurations of the object theory $\varXi'$ to the configurations of $\varXi$ is an abstraction from $\varXi'$ to $\varXi$ if it respects $\varGamma'$ and $\varGamma$, i.e., if $u' = v'$ with respect to $\varGamma'$ then $\alpha(u') \simeq \alpha(v')$ with respect to $\varGamma$ (where $c \simeq c'$ means that if c is defined then c' is also defined and both are equal, and vice versa).

Definition 3. Let $Z = (\varXi, p, q_0)$ and $Z' = (\varXi', p', q_0')$ be object theories with control. Then Z' is an object control refinement of Z if p' is a process refinement of p, and there is an abstraction α from $\varXi'$ to $\varXi$ with $\alpha(q_0) = q_0'$ such that the following holds:

(1) For every run $Z \vdash q_0 \xrightarrow{A_1} q_1 \xrightarrow{A_2} \cdots$ there is a run $Z' \vdash q_0' \xrightarrow{A_1'} q_1' \xrightarrow{A_2'} \cdots$ and a strictly increasing sequence $(k_i)_{i \geqslant 1}$ of natural numbers with $k_1 = 0$ such that $\alpha(q_{k_i}') = q_i$.

(2) For every run $Z' \vdash q_0' \xrightarrow{A_1'} q_1' \xrightarrow{A_2'} \cdots$ there is a run $Z \vdash q_0 \xrightarrow{A_1} q_1 \xrightarrow{A_2} \cdots$ and a strictly increasing sequence $(k_i)_{i \geqslant 1}$ of natural numbers with $k_1 = 0$ such that $\alpha(q_{k_i}') = q_i$.

Thus an object theory with control Z is refined into an object theory with control Z′ if every run of Z′ can be abstracted into a run of Z such that the messages consumed and produced during such a run in Z′ weakly correspond to the messages consumed and produced during the abstracted run in Z, and vice versa, thus establishing a weak form of bisimulation between Z and Z′.

4. fOOSE

We extend Jacobson's OOSE software development process and its diagrammatic modelling techniques by formal specifications and formal proof steps, based on object theories with control. This extended method will be called fOOSE ("formal object-oriented software engineering").

4.1. OOSE

The OOSE development process [21] consists of five phases: requirements analysis, robustness analysis, design, implementation, and test (see Fig. 1).

The requirements analysis, also called use case analysis, serves to establish a requirements document which describes the functionality 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. In the design phase, a system design is derived from the analysis objects and the objects of reuse libraries. The design is implemented during the implementation phase and finally, during the test phase, the implementation is tested with respect to the use case description.

Use cases are a particular feature which distinguishes OOSE from other development methods. Use cases have the advantage of providing a requirements document which is the basis for testing and which can serve as a reference during the whole development; a use case analysis step has also been integrated into the UP [20].

4.2. fOOSE extensions

As with all semiformal approaches, the most prominent problem of OOSE is that testing can be performed only at a very late stage of development; another problem is
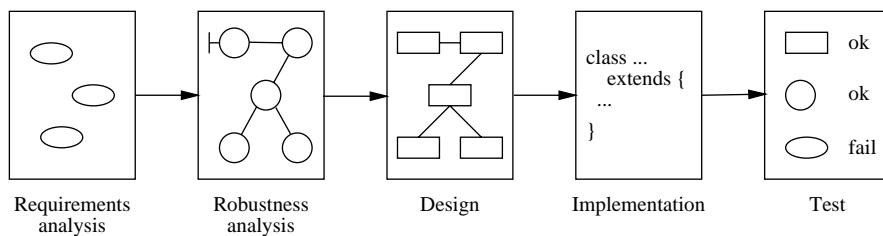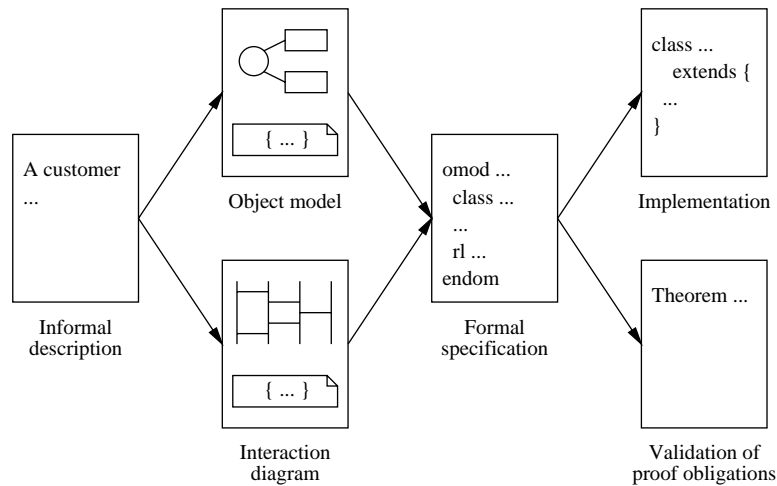

Fig. 1. Development phases of OOSE.

Fig. 2. Construction and use of formal specifications.

the fact that many important requirement and design details can neither be expressed by (the current) diagrams nor can they be well described by informal text.

In our enhanced "formal object-oriented software engineering" (fOOSE) method we provide means to overcome these deficiencies without changing the basic method. The enhanced development process consists of the same phases. However, the models produced in each phase are accompanied by formal specifications and we add formal validation steps. Technically, we use UML notation for the OOSE diagrams [35]; these diagrams can optionally be annotated by formal text. Any annotated diagram can be translated semi-automatically into a formal Maude specification, i.e., the diagram is automatically translated into an incomplete Maude specification which then has to be completed by hand. Additionally, the annotations automatically generate proof obligations.

In the sequel, we propose the following method for fOOSE developers for constructing a formal Maude specification from an informal use case description (see Fig. 2). First, for any given informal description the fOOSE developer has to produce a semiformal description consisting of an object model and an interaction diagram in the usual OOSE style. The object model is used for describing the attributes and states of objects and generalisation and association relationships, the interaction diagram describes the flow of the messages the objects exchange. Note that, in contrast to OOSE, interaction diagrams are also used in the analysis phases, and not only in the design phase; we believe that interaction diagrams are useful for illustrating the interactions of the objects also at abstract levels. Second, the object model has to be enhanced with invariants, and the interaction diagram has to be extended by constraints on the messages. These enhanced diagrams on the one hand give rise to functional specifications of occurring data types; on the other hand the enhanced object model directly translates into a Maude specification and the enhanced interaction diagram yields an incomplete Maude specification and a process control. The translation of both diagrams yields, after completion, a Maude specification with control together with some proof

obligations generated from the formal annotations which have to be discharged by the fOOSE developer.

Any refined specification is constructed in the same way. Moreover, for relating the refined "concrete" specification to the more abstract specification the fOOSE developer has to give the relationship between the "abstract" and the "concrete" specifications as an object control refinement. This generates proof obligations which have to be verified to guarantee the correctness of the refinement. Moreover, object control refinements provide the information for tracing the relationship between use case descriptions and the corresponding Maude specifications. The induced proof obligations are the basis for verifying the correctness of designs and implementations.

We illustrate our fOOSE method for developing and refining a formal specification from an informal use case description by the example of a recycling machine which is the running example of Jacobson's book on OOSE [21]. We show the specification and refinement activities for the recycling machine example at the level of requirements analysis and robustness analysis in Sections 4.3 and 4.5.

4.3. Requirements analysis

The informal description of the recycling machine consists of three use cases: "returning items", "generate daily report", and "change item". The use case "returning items" 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. The receipt contains a list of the returned items as well as the total return sum".

4.3.1. Object and interaction diagrams

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

The use case is modelled as an interactive system consisting of a single object with name RM (see Fig. 3 on the left) and the customer as an actor. The object RM represents the recycling machine and has two attributes storing the daily total and the current items. The interaction diagram (Fig. 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 recycling machine prints a receipt with the list of (descriptions of) the returned items as well as the total return sum (in Euro).

More generally, an object model consists of several objects, represented by circles, with their attributes, represented by lines from circles to rectangles, and the relationships between the objects, represented by arrows. Objects are labelled with their name and
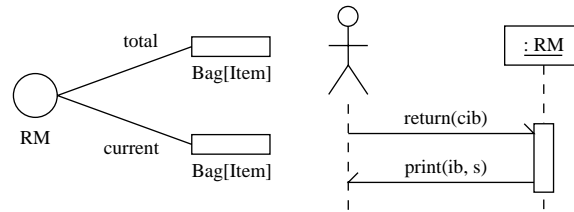
Fig. 3. Object model and interaction diagram of the recycling machine.

attributes with their corresponding name on the line and the sort of the attribute below the rectangle. Generalisations of objects are represented by a dotted arrow from the heir to the parent; (bidirectional) associations of objects are represented by solid line arrows (with two heads). (For an example see Fig. 5(a); note that we use the OOSE name "objects" for object diagram entities.)

An interaction diagram consists of several objects represented by vertical lines and messages represented by horizontal arrows. Objects are labelled with the object name and messages with their name and variable names for their arguments. Each message arrow leads from the sender object to the receiving object; filled arrow heads represent synchronous messages, i.e., the sender waits until the message has been received, whereas half-stick arrow heads represent asynchronous messages, i.e., the sender does not wait. Activations of objects by incoming messages are represented by rectangles. 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, like loops, can be given on the left-hand side of the diagram for describing the control flow. (For an example, see Fig. 6.)

Object and interaction diagrams give an abstract view of the informal description. But several important relationships which will be expressed by the formal specifications are not represented. For example in the use case "return items" there is a connection between the current list and the daily total; moreover, the printed items are descriptions of the returned items. The formal specification will be able to express these semantic dependencies. It will also be used to fix the basic data types.

4.3.2. Enhanced diagrams

The second fOOSE step consists of two activities: the extension of the object models by invariants and the extension of the interaction diagrams by message constraints.

An invariant is a relation between the attributes of an object or between the objects of a configuration which has to be preserved during system execution. We extend an object diagram with formal invariants by adding an annotation.

For example, the attributes `total` and `current` of the recycling machine have to satisfy the property that all items of `current` have also to be in `total` (see Fig. 4 to the left).

Interaction diagrams are refined in order to express semantic relationships of the parameters of the messages. We add the parameter sorts of messages and state the relationships between the parameters in an additional annotation: any message expression
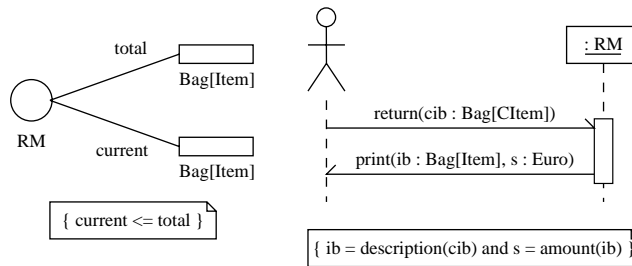
Fig. 4. Object model with invariant and refined interaction diagram of the recycling machine.

$m(v_1,\ldots,v_n)$ is replaced by an expression $m(v_1 : s_1, \ldots, v_n : s_n)$ where $v_1,\ldots,v_n$ are variables of sorts $s_1,\ldots,s_n$. The annotation is a conjunction of equations of the form $t_1 = u_1$ and ... and $t_k = u_k$ such that t_j and u_j are terms containing only (free) variables of messages in the interaction diagram.

For example, the message expressions `return(cib)` and `print(ib, s)` of the interaction diagram in Fig. 3 are replaced by `return(cib : Bag[CItem])` and `print(l : Bag[Item], s : Euro)`. The intended meaning of the message constraint annotation `ib = description(cib) and s = amount(ib)` is that `ib` is the multiset of descriptions of the elements of `cib` and that `s` is the sum of the prices of `ib` (see Fig. 4 on the right).

4.3.3. Functional specifications

For any data type occurring in the enhanced diagrams a functional Maude specification has to be constructed, either by reusing predefined modules from a specification library such as `NAT`, `LIST`, or `BAG`, or by designing a completely new specification. In particular, these specifications provide a formal meaning for the annotations of the enhanced diagrams.

The following new specification of items introduces two sorts `CItem` and `Item` denoting the "concrete" items of the customer 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 Euro.

```
fmod CITEM is
sort CItem .
op bottle : -> CItem .
endfm
fmod ITEM is
protecting CITEM .
protecting EURO .   - currency specification
sort Item .
op price : Item -> Euro .
op desc : CItem -> Item .
eq price(desc(bottle)) = 10 E .
endfm
```

The functional specification for multisets (bags) of items and concrete items has to contain a suitable axiomatisation of the multiset containment `<=` for items, a function `description` that extends `desc` to multisets, and a function `amount` yielding the total sum of a multiset of items. We may assume that a parameterised specification `BAG` exists as a library component.

```
fmod ITEMS is
  protecting BAG[CItem] .
  protecting BAG[Item] .

  op description : Bag[CItem] -> Bag[Item] .
  op amount : Bag[Item] -> Euro .

  var I : Item .
  var C : CItem .
  var Ib : Bag[Item] .
  var Cb : Bag[CItem] .

  eq description(empty) = empty .
  eq description(C Cb) = desc(Cb) description(Cb) .
  eq amount(empty) = 0 E .
  eq amount(I Ib) = price(I) + amount(Ib) .
endfm
```

4.3.4. Construction of a formal specification

In this step, we show how a Maude specification and a control process can be constructed semi-automatically from the enhanced diagrams for a use case; the specification and the process define an object theory with control which is executable in Maude (see Section 3). The object model automatically generates class declarations; by combining the object model with the interaction diagram a set of (incomplete) rewrite rules can be constructed automatically which after completion (by hand) define the dynamic behaviour of the use case.

The automatic part of the construction is as follows:

- The object model induces a set of Maude class declarations:
  - Each object with name C, with attributes $a_1, \ldots, a_m$ of types $s_1, \ldots, s_m$, and with associated objects named $C_1, \ldots, C_n$ becomes a class declaration

    class C | a_1 : s_1, ..., a_m : s_m, C_1 : Oid, ..., C_n : Oid .

  - Each inheritance relation from an object named D to an object named C corresponds to a subclass declaration

    subclass D < C .

- The interaction diagram induces a set of message declarations:
  - Each message $m(v_1 : s_1, \ldots, v_n : s_n)$ from some object to another object of the interaction diagram becomes a message declaration

    msg m : Oid s_1 ... s_n Oid -> Msg .

The first argument of message m indicates the sender object, the last argument
the target object.

○ Each message $m(v_1:s_1,\ldots,v_n:s_n)$ from the actor to some object of the interaction
diagram becomes a message declaration

```
msg m : s₁ ... sₙ Oid -> Msg .
```

○ Each message $m(v_1:s_1,\ldots,v_n:s_n)$ from some object to the actor of the interaction
diagram becomes a message declaration

```
msg m : Oid s₁ ... sₙ -> Msg .
```

• Both diagrams generate rule skeletons:
  ○ For any message $m(v_1:s_1,\ldots,v_n:s_n)$ from an object C to an object D of the
  interaction diagram: Let $a_1,\ldots,a_k$ be the attributes of C with corresponding sorts
  $s_1,\ldots,s_k$, let $C_1,\ldots,C_l$ be the objects associated to C; let $m_1,\ldots,m_p$ be the out-
  going messages from C to objects $C_{t_1},\ldots,C_{t_p}$ (which have to be in $\{C_1,\ldots,C_l\}$)
  and $m'_1,\ldots,m'_q$ the outgoing messages from C to the actor of the same ac-
  tivation block below m. Then we obtain the following skeleton of a rewrite
  rule:

```
crl [m] :
    m(O’, v₁, ..., vₙ, O)
    < O  : C | a₁ : V₁, ..., aₖ : Vₖ, C₁ : O₁, ..., Cₗ : Oₗ > =>
      < O  : C | a₁ : ?, ..., aₖ : ?, C₁ : ?, ..., Cₗ : ? >
      m₁(O, ?, ..., ?, O_{t₁}) ... mₚ(O, ?, ..., ?, O_{tₚ})
      m'₁(?, ..., ?) ... m'_q(?, ..., ?)
    if ? .
```

where $O, O’, O_1,\ldots O_l$, and $O_{t_1},\ldots,O_{t_p}$ are Maude variables of sort `Oid` and $V_1,\ldots,V_k$
are variables of sorts $s_1,\ldots,s_k$, respectively. If any of the messages above is sent to
or received from the actor the receiver or sender object identifier is omitted.

• The interaction diagram defines a control process:
  ○ For each object, the names of incoming asynchronous messages m form actions
  $?m$, the names of outgoing asynchronous messages m form actions $!m$, and the
  names of outgoing synchronous messages form a process $!m; ?m$. Each activation
  block of an object for an incoming asynchronous message forms a sequential
  process of the action for the incoming message and the parallel composition of
  its activated message processes; each activation block of an object for an incoming
  synchronous message forms a parallel process of its outgoing message processes.
  For each object these activation block processes are composed sequentially from
  top to bottom; if an activation block is part of a loop, the translated activation
  block is surrounded by an iteration. These object behaviours are composed in
  parallel.
  ○ For the actor, the names of incoming asynchronous messages form actions $?m$,
  the names of outgoing messages m form actions $!m$. The message processes are

composed sequentially; if a message is part of a loop, the corresponding message process is surrounded by an iteration. The resulting actor process is composed in parallel with the object behaviours.

For our example, the diagrams of Fig. 4 induce the following class declaration:

```
class RM | total : Bag[Item], current : Bag[Item] .
```

and the following message declarations:

```
msg return : Bag[CItem] Oid -> Msg .
msg print : Oid Bag[Item] Euro -> Msg .
```

In particular, the actor of a use case, as being external, is assumed to be anonymous.

The rule skeleton expresses that if the object $< O : C| \ldots >$ receives a message m it sends the messages $m_1, \ldots, m_p$. The question marks ? on the right hand side of the rule indicate that the resulting state of $< O : C | \ldots >$ is not expressed directly in the diagram; therefore the new values of the attributes have to be added by hand, based on the object invariants. Similarly, the actual parameters of the outgoing messages are not given explicitly, but have to be inferred by the message constraints. The question mark in the `if`-clause states that the condition is possibly under-specified.

For example, the diagrams of Fig. 4 induce the following skeleton:

```
crl [return] :
  return(cib, Rm)
  < Rm : RM | total : V1, current : V2 > =>
    < Rm : RM | total : ?, current : ? >
    print(Rm, ?, ?)
if ? .
```

In order to obtain a complete rule, the question marks have to be filled in with appropriate values: The parameters for the `print` message may be inferred immediately from the enhanced diagram annotation: `description(cib)` and `amount (description(cib))`. The new state of the object `< Rm : RM | ... >` after rewriting has to satisfy the object invariant `current <= total` for `RM`; each completion generates a proof obligation (see Section 4.4). A possible new value for `total` is `description(cib) V1`, for `current` we may choose `description(cib)`; however, also `description(cib) V1` for `total` and `description(cib) V2` for `current`, or `V1` and `V2`, respectively, would be possible, though not desired, since `current` should represent the currently returned items by a single customer and `total` the daily total of all returned items. Finally, the condition of the rewrite rule may be chosen as `true`, and thus be omitted.

This completion leads to the following Maude module for the use case "return items":

```
omod RRM is
  protecting ITEMS .

  class RM | total : Bag[Item], current : Bag[Item] .
```

```
    msg return : Bag[CItem] Oid -> Msg .
    msg print : Oid Bag[Item] Euro -> Msg .

    var Rm : Oid .
    var cib : Bag[CItem] .
    vars V1 V2 : Bag[Item] .

    rl [return] :
      return(cib, Rm)
      < Rm : RM | total : V1, current : V2 > =>
        < Rm : RM | total : description(cib) V1,
                    current : description(cib) >
        print(Rm, description(cib), amount(description(cib))) .
  endom
```

The process control 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.

In the example, the interaction diagram defines the following control process

!return ; ?print ∥ ?return ; !print

A use case may be tested by providing a specification for the actor and a suitable initial configuration containing the actor and the set of cooperating objects of the interaction diagram. The test specification can be directly simulated in Maude (see Section 3.3).

A possible test specification for the use case "return items" is as follows:

```
    omod RRM-CONTROL is
      including RRM .

      class User | items : Bag[CItem], rm : Oid .

      var U : Oid .
      var Rm : Oid .
      var Cb : Bag[CItem] .
      var Ib : Bag[Item] .
      var S : Euro .

      crl [returning]:
        < U : User | items : Cb, rm : Rm > =>
          < U : User | items:empty, rm : Rm >
          return(Cb, Rm)
        if Cb =/= empty .
      rl [receipt] :
        print(Rm, Ib, S)
```

```
          < U : User | items : empty, rm : Rm > =>
          < U : User | items : empty, rm : Rm > .

       op u : -> Oid .
       op rm : -> Oid .
       op initial : -> Configuration .
       eq initial = < u : User | items : icb, rm : rm >
                    < rm : RM | total : itb, current : empty > .

       op control : -> Process .
       eq control =  (act('!return) ; act('?print))
                   | (act('?return) ; act('!print)) .
    endom
```

where *icb* is a multiset of concrete items and *itb* is a multiset of items.

Choosing *icb* = `bottle bottle bottle` and *itb* = `empty` the Maude engine for object theories with control of Section 3.3 produces

```
   rewrites: 2498 in 540ms cpu (550ms real) (4625 rewrites/second)
   reduce in ENGINE [ RRM-CONTROL ] : accepting .
   result Set[Pair[Term, Process]] :
   ‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾
   < < rm : RM | current : desc(bottle) desc(bottle) desc(bottle),
   ‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾
                 total : desc(bottle) desc(bottle) desc(bottle) >
   ‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾
     < u : User | rm : rm, items : empty > ,
   ‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾
     empty >
```

where the overlined term represents actual meta-term output.

The complete, executable Maude specification can be found in [24].

4.4. Proof obligations

An invariant *I* for an object *C* of an enhanced object model has to be satisfied by all instances of *C* and of its heirs. As a consequence an invariant generates a proof obligation on the rewrite rules that have been generated from an enhanced interaction diagram and which have been completed by hand: Every rule of the form

```
   crl [m] :
     m(...)
     < O : D | a₁ : V₁, ..., aₖ : Vₖ, C₁ : O₁, ..., Cₗ : Oₗ > =>
       < O : D | a₁ : t₁, ..., aₖ : tₖ, C₁ : o₁, ..., Cₗ : oₗ >
       ...
       if Π
```

where D is C or any of its subclasses has to satisfy the correctness condition

$$I\{a_1 \mapsto t_1, \ldots, a_k \mapsto t_k, C_1 \mapsto o_1, \ldots, C_l \mapsto o_l\} \Leftarrow$$

$$(\Pi \wedge I)\{a_1 \mapsto \mathtt{V}_1, \ldots, a_k \mapsto \mathtt{V}_k, C_1 \mapsto \mathtt{O}_1, \ldots, C_l \mapsto \mathtt{O}_l\}$$

where $\varphi\{\ldots\}$ means simultaneous substitution.

In our example, the completed rule with label `return` from the use case "return items" induces the correctness condition

```
(description(cib)) <= (description(cib) V1) <= V2 <= V1
```

Obviously, the multiset containment `M <= M M'` holds for all multisets `M` and `M'`; in this case the preconditions are irrelevant.

4.5. Robustness analysis

In the second phase of OOSE, called "robustness analysis", the sample use case "return items" is refined in two aspects: Instead of returning all items at the same time, the customer returns the items one by one; he has to press a button to request a receipt; the machine itself is decomposed into several objects.

For simplicity of presentation, we omit the object and interaction diagrams as constructed by the original OOSE method; instead, we directly present the enhanced fOOSE diagrams.

4.5.1. Enhanced diagrams

The object model of the requirements analysis is refined and the resulting objects are classified in three categories: interface, control and entity objects. Interface objects build the interface between the actors (at the system boundary) 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.

In the object model, interface objects are represented by hooked circles, control objects by circles with an arrow, and entity objects by underlined circles. For convenience, object models are given in two parts, one showing the attributes of the objects and the other showing the relationships between the objects.

The recycling machine is represented by five objects: the interface object `Customer Panel`, a control object `Receiver`, and the entity objects `Current`, `DayTotal`, and `DepositItem`. The objects `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. The object `DepositItem` stands for all kinds of returned items; in particular the object `Bottle` is modelled as its heir (see Fig. 5(a)).

These objects have the following attributes (see Fig. 5(b)): `CustomerPanel` and `Receiver` have no attributes; `DepositItem` has a name and a price, `Bottle` has additionally a height and a width; `Current` has a list of deposit items and a sum as
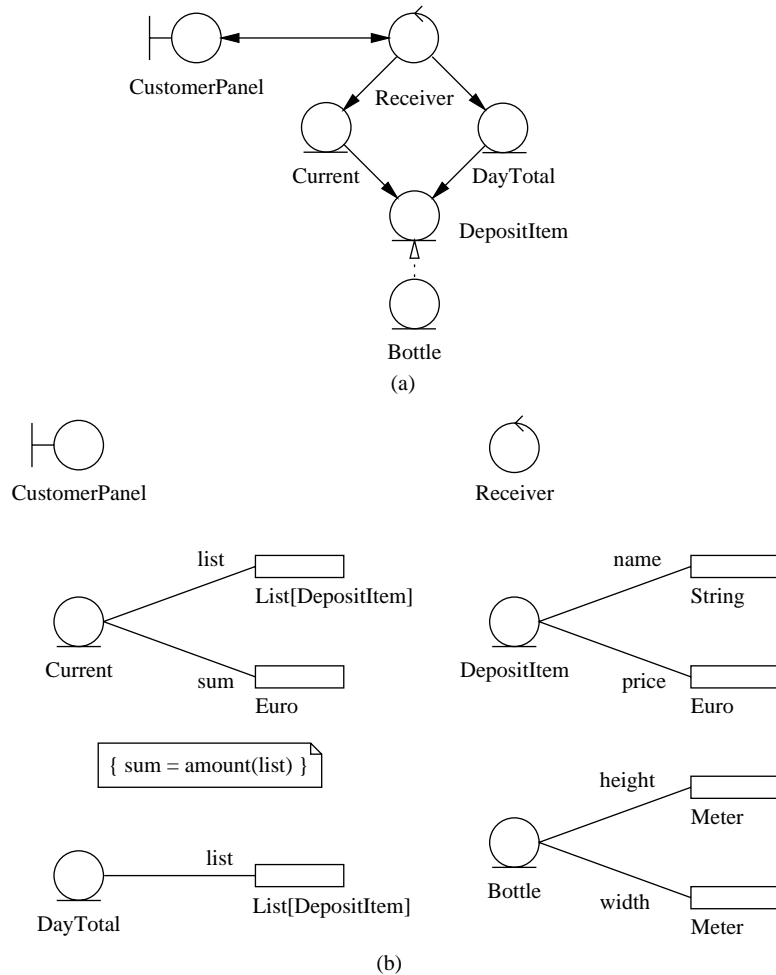
Fig. 5. Object model of the robustness analysis of the recycling machine: (a) generalisations and associations; (b) attributes and invariants.

attributes; and `DayTotal` has a list of deposit items. Note that the choice of lists as storage structure for `Current` and `DayTotal` accounts for the fact that the customer returns his items one by one.

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

The interaction diagram has to take into account these object model refinements. From the informal use case description we derive two kinds of messages which are sent from the actor, i.e. the customer, to the `CustomerPanel`: a `return` message for returning a single concrete item and a `receipt` message for requesting a receipt. Each
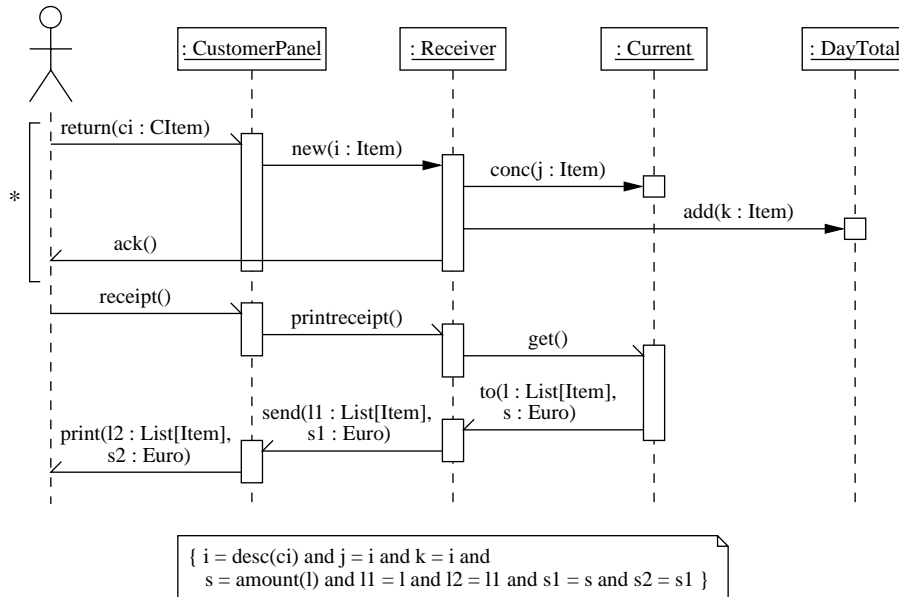
Fig. 6. Refined interaction diagram of the robustness analysis of the recycling machine.

of these messages begins a new activity of the customer panel. On the other hand, the customer panel sends an acknowledgement and a print message to the actor.

After receiving a `return` message carrying a concrete item, the customer panel sends a message, say `new`, with the description of the concrete item to the receiver. Then the receiver forwards this information to `Current` and `DayTotal` by two messages, called `conc` and `add` respectively; the end of such a return process is 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 this answer to the `CustomerPanel` by a message called `send`, which prints the result.

From this text, a conventional OOSE interaction diagram is derived and, in the next step, enhanced by inserting sorts for the parameters of messages and by stating semantic properties of the parameters (see Fig. 6). 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.

4.5.2. Functional specifications

We have chosen a more concrete representation of deposit items. The enhanced object and interaction diagrams suggest to model this representation as a functional data type, since deposit items may be identified by value:

```
fmod DEPOSITITEM is
  protecting CITEM .
```

```
    protecting METER . - measurements
    protecting EURO .

    sort DepositItem .
    sort Bottle .
    subsort Bottle < DepositItem .

    op name : DepositItem -> Qid . - simple strings
    op price : DepositItem -> Euro .
    op height : Bottle -> Meter .
    op width : Bottle -> Meter .
    op desc : CItem -> DepositItem .

    eq name(desc(bottle)) = 'Bottle .
    eq price(desc(bottle)) = 10 E .
    eq height(desc(bottle)) = 290 MM .
    eq width(desc(bottle)) = 85 MM .
  endfm

  view DepositItem from TRIV to DEPOSITITEM is
    sort Elt to DepositItem .
  endv

  fmod DEPOSITITEMS is
    protecting LIST[DepositItem] .

    op amount : List[DepositItem] -> Euro .

    var I : DepositItem .
    var Ib : List[DepositItem] .

    eq amount(nil) = 0 E
    eq amount(I Ib) = price(I) + amount(Ib) .
  endfm
```

4.5.3. Construction of a formal specification

By the general procedure in Section 4.3.4, from the enhanced object diagram automatically six classes are generated: CustomerPanel, Receiver, Current, DayTotal, and DepositItem and Bottle, which, as explained above, we have replaced by data types:

```
  class CustomerPanel | rc : Oid .
  class Receiver | cp : Oid, cur : Oid, dt : Oid .
  class Current | list : List[DepositItem], sum : Euro,
                  rc : Oid, cp : Oid .
  class DayTotal | list : List[DepositItem] .
```

(where we slightly changed the names for the associated attributes).

The enhanced interaction diagram automatically generates eleven message declarations:

```
msg return : CItem Oid -> Msg .
msg new : Oid DepositItem Oid -> Msg .
msg conc : Oid DepositItem Oid -> Msg .
...
msg send : Oid List[DepositItem] Euro Oid -> Msg .
msg print : Oid List[DepositItem] Euro -> Msg .
```

Additionally, we obtain the following rule skeletons (where we slightly change the automatically generated variable names):

```
crl [return] :
  return(Ci, Cp)
  < Cp : CustomerPanel | rc : Rc > =>
    < Cp : CustomerPanel | rc : ? >
    new(Cp, ?, Rc)
  if ? .
crl [new] :
  new(Cp, I, Rc)
  < Rc : Receiver | cp : Cp, cur : Cur, dt : Dt > =>
    < Rc : Receiver | cp : ?, cur : ?, dt : ? >
    conc(Rc, ?, Cur)
    add(Rc, ?, Dt)
    ack
  if ? .
crl [conc] :
  conc(Rc, I, Cur)
  < Cur : Current | list : L, sum : S, rc : Rc', cp : Cp > =>
    < Cur : Current | list : ?, sum : ?, rc : ?, cp : ? >
  if ? .
...
crl [send] :
  send(Rc, L, S, Cp)
  < Cp : CustomerPanel | rc : Rc > =>
    < Cp : CustomerPanel | rc : ? >
    print(Cp, ?, ?)
  if ? .
```

In order to obtain full rules, on the one hand we add state changes and preconditions. This completion generates proof obligations stating that the completed rules preserve the object invariants. On the other hand we add values for the message parameters which are guided by the annotations of the enhanced interaction diagram.

```
rl [return] :
  return(Ci, Cp)
```

```
      < Cp : CustomerPanel | rc : Rc > =>
       < Cp : CustomerPanel | rc : Rc >
       new(Cp, desc(Ci), Rc) .
 rl [new] :
   new(Cp, I, Rc)
   < Rc : Receiver | cp : Cp, cur : Cur, dt : Dt > =>
       < Rc : Receiver | cp : Cp, cur : Cur, dt : Dt >
       conc(Rc, I, Cur)
       add(Rc, I, Dt)
       ack .
 rl [conc] :
   conc(Rc, I, Cur)
   < Cur : Current | list : L, sum : S, rc : Rc, cp : Cp > =>
       < Cur : Current | list : I L, sum : price(I) + S,
                         rc : Rc, cp : Cp > .
 ...
 rl [send] :
   send(Rc, L, S, Cp)
   < Cp : CustomerPanel | rc : Rc > =>
       < Cp : CustomerPanel | rc : Rc >
       print(Cp, L, S) .
```

All preconditions can be chosen to be true; the proof obligations justifying our choices are treated below. The full specification can be found in Appendix A.

The behaviour of the interaction diagram is represented by the following control process:

$$((!\mathrm{return};?\mathrm{ack})^*;!\mathrm{receipt};?\mathrm{print})$$
$$\| \; ((?\mathrm{return};!\mathrm{new};?\mathrm{new})^*;?\mathrm{receipt};!\mathrm{printreceipt};?\mathrm{send};!\mathrm{print})$$
$$\| \; (((!\mathrm{add};?\mathrm{add}) \| (!\mathrm{conc};?\mathrm{conc}) \| !\mathrm{ack})^*;$$
$$\quad ?\mathrm{printreceipt};!\mathrm{get};?\mathrm{to};!\mathrm{send})$$
$$\| \; (?\mathrm{get};!\mathrm{to})$$

We provide a suitable test environment by specifying the actor and giving an initial configuration:

```
 omod ARM-CONTROL is
   including ARM .
   protecting LIST[CItem] .
   protecting PROCESS .

   class User | items : List[CItem], receipt : Bool, cp : Oid .

   vars U Cp : Oid .
   var C : CItem .
```

```
    var Cl : List[CItem] .
    var Ib : List[DepositItem] .
    var S : Euro .

    rl [returning] :
      < U : User | items : C Cl, cp : Cp > =>
        < U : User | items : Cl, cp : Cp >
        return(C, Cp) .
    rl [ack] :
      ack
      < U : User | > =>
        < U : User | > .
    rl [request] :
      < U : User | items : nil, receipt : false, cp : Cp > =>
        < U : User | items : nil, receipt : true, cp : Cp >
        receipt(Cp) .
    rl [receipt] :
      print(Cp, Ib, S)
      < U : User | items : nil, cp : Cp > =>
        < U : User | items : nil, cp : Cp > .

    op u : -> Oid .
    op cp : -> Oid .
    op rc : -> Oid .
    op cur : -> Oid .
    op dt : -> Oid .
    op initial : -> Configuration .
    eq initial =
      < u : User | items : icl, cp : cp >
      < cp : CustomerPanel | rc : rc >
      < rc : Receiver | cp : cp, cur : cur, dt : dt >
      < cur : Current | list : nil, sum : 0 E, rc : rc, cp : cp >
      < dt : DayTotal | list : itl > .

    op control : -> Process .
    eq control =
          ((act('!return) ; act('?ack)) * ;
           act('!receipt) ; act('?print))
        | ((act('?return) ; act('!new) ; act('?new)) * ;
           act('?receipt) ; act('!printreceipt) ; act('?send) ;
                                                    act('!print))
        | (((act('!add) ; act('?add)) |
            (act('!conc) ; act('?conc)) | act('!ack)) * ;
           act('?printreceipt) ; act('!get) ; act('?to) ;
                                                    act('!send))
        | (act('?get) ; act('!to)) .
  endom
```

where *icl* is a list of concrete items and *itl* is a list of deposit items.

Choosing $icb =$ bottle bottle bottle and $itb =$ empty the Maude engine for object theories with control of Section 3.3 produces

```
rewrites: 1053762 in 50220ms cpu (52080ms real)
                                        (20982 rewrites/second)
reduce in ENGINE [ ARM-CONTROL ] : accepting .
result Set[Pair[Term,Process]] :
  < < rc : Receiver | dt : dt, cur : cur, cp : cp >
    < cp : CustomerPanel | rc : rc >
    < cur : Current | rc : rc, cp : cp,
                      sum : 0 E, list : nil >
    < dt : DayTotal | list : desc(bottle) desc(bottle)
                             desc(bottle) >
    < u : User | cp : cp, items :  nil, receipt : true >,
    empty >
```

The complete, executable Maude specification can be found in [24].

4.5.4. Proof obligations

The invariant of `Current` automatically generates the following proof obligations for the completed rules with labels `conc` and `get`:

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

The first proof obligation is discharged by using the definition of `amount` in `DEPOSIT-ITEMS`, while the second proof obligation is trivial.

It remains to discharge the proof obligation that the specification of the robustness analysis step is a refinement of the specification of the requirements analysis.

When instantiating *icl* in `ARM-CONTROL` by some non-empty list of concrete items and *itl* by some list of deposit items such that *icl* `<=` *itl*; and instantiating *icb* of `RRM-CONTROL` by a multiset of concrete items representing the instantiation of *icl* and *itb* by a multiset of items representing the instantiation of *itl*, the object theory with control $Z' = (((\Omega', \Gamma'), P'), p', q'_0)$, defined by `ARM-CONTROL`, its control process `control`, and its instantiated initial configuration `initial`, in fact is an object control refinement of the object theory with control $Z = (((\Omega, \Gamma), P), p, q_0)$, defined by `RRM-CONTROL`, its control process `control` and its instantiated initial configuration `initial`.

Indeed, p' is a process refinement of p by choosing the refinement function

$$\rho(!\text{return}) = (!\text{return}; ?\text{ack})^*; !\text{receipt}$$
$$\rho(?\text{return}) = (?\text{return}; !\text{new}; ?\text{new})^*$$
$$\qquad\qquad ||((!\text{add}; ?\text{add})||(!\text{conc}; ?\text{conc})||!\text{ack})^*$$
$$\rho(!\text{print}) = ?\text{receipt}; ?\text{printreceipt}; !\text{get}; ?\text{to}; !\text{send}; !\text{print}$$
$$\rho(?\text{print}) = ?\text{print}$$

Moreover, note that the object `RM` of the requirements analysis is represented in the robustness analysis by the four objects `CustomerPanel`, `Receiver`, `Current`, and

DayTotal, i.e., each instance of RM is replaced by one instance of each of the mentioned objects. We may thus define an abstraction function α from Ξ' to Ξ in the following way: Let $\{|-|\}_c$ be a function that converts lists of concrete items into multisets of concrete items and let $\{|-|\}_i$ be a function that converts lists of deposit items into multisets of items. Let C be a configuration of Ξ' of the form

```
c = < u : User | items : cl, cp : cp >
    < cp : CustomerPanel | rc : rc >
    < rc : Receiver | cp : cp, cur : cur, dt : dt >
    < cur : Current | list : l, sum : s, rc : rc >
    < dt : DayTotal | list : dl > c′
```

If c does not contain any messages and $(\Omega', \Gamma') \vdash cl \neq \texttt{nil}$ and $(\Omega', \Gamma') \vdash l = \texttt{nil}$, we define

```
α(c) = < u : User | items : {|cl|}_c, rm : rm >
       < rm : RM | total : {|dl|}_i, current : empty >
```

If c does not contain any messages and $(\Omega', \Gamma') \vdash cl = \texttt{nil}$ and $(\Omega', \Gamma') \vdash l = \texttt{nil}$, we define

```
α(c) = < u : User | items : empty, rm : rm >
       < rm : RM | total : {|dl|}_i, current : {|icl|}_i >
```

If c contains some message but no `print` message, we define

```
α(c) = < u : User | items : empty, rm : rm >
       < rm : RM | total : {|idl|}_i, current : empty >
       return({|icl|}_c, rm)
```

If c contains a `print` message, we define

```
α(c) = < u : User | items : empty, rm : rm >
       < rm : RM | total : {|dl|}_i, current : empty >
       print(rm, {|l|}_i, s)
```

We leave α undefined for all other configurations of (Ω', Γ').

The first clause for α abstracts all configurations of Z' in which the customer has not yet started returning his items to the recycling machine; in particular the initial configuration q_0' of Z' is abstracted into the initial configuration q_0 of Z. The second clause for α abstracts all configurations of Z' in which the customer has returned all his items and has received the receipt. The third clause for α delays sending the `return` message in Z. Finally, the fourth clause for α synchronises the `print` messages of Z' and Z.

Since every trace of p' induced by a run of Z' is of the form

```
!return, ..., !ack, !return, ..., !receipt, ..., !print, ?print
```

and hence corresponds to a trace

```
!return, ?return, !receipt, ..., !print, ?print
```

of p induced by a run of Z, we have

Fact 4. *The robustness analysis specification of this section is an object control refinement of the requirements specification of Section* 4.3.

5. Concluding remarks

We 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 that can be used for validation purposes.
- The refinement relation gives the information for tracing the relationships between use case descriptions and the corresponding design and implementation code, and 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 early simulation and rapid prototyping.
- 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 iteration statements; means to deal with conditional and exception statements should be added as well. Interaction diagrams as in Jacobson's OOSE are inherently sequential, since the whole OOSE method is designed for the development of sequential systems. When aiming at the description of distributed concurrent systems, we would also need means for describing the concurrent behaviour in our diagrams, not only in interaction diagrams but also in other kinds of behavioural diagrams. Another problem is that our notion of refinement is defined at the level of specifications. For software engineers it would be easier if we could also define a refinement relation at the level of diagrams which ensures the validity of an object control refinement.

Acknowledgements

Appendix Complete Maude specification

```
fmod EURO is
  protecting MACHINE-INT .
```

```
  sort Euro .

  op _E : MachineInt -> Euro .
  op _+_ : Euro Euro -> Euro [assoc comm] .

  vars X Y : MachineInt .

  eq (X E) + (Y E) = (X + Y) E .
endfm

fmod METER is
  protecting MACHINE-INT .

  sort Meter .

  op _MM : MachineInt -> Meter .
endfm

fmod CITEM is
  sort CItem .

  op bottle : -> CItem .
endfm

view CItem from TRIV to CITEM is
  sort Elt to CItem .
endv

fmod DEPOSITITEM is
  protecting QID .
  protecting CITEM .
  protecting METER .
  protecting EURO .

  sort DepositItem .
  sort Bottle .
  subsort Bottle < DepositItem .

  op name : DepositItem -> Qid .
  op price : DepositItem -> Euro .
  op height : Bottle -> Meter .
  op width : Bottle -> Meter .
  op desc : CItem -> DepositItem .
```

```
  eq name(desc(bottle)) = 'Bottle .
  eq price(desc(bottle)) = 10 E .
  eq height(desc(bottle)) = 290 MM .
  eq width(desc(bottle)) = 85 MM .
endfm

view DepositItem from TRIV to DEPOSITITEM is
  sort Elt to DepositItem .
endv

fmod DEPOSITITEMS is
  protecting LIST[DepositItem] .

  op amount : List[DepositItem] -> Euro .

  var I : DepositItem .
  var Ib : List[DepositItem] .

  eq amount(nil) = 0 E
  eq amount(I Ib) = price(I) + amount(Ib) .
endfm

omod ARM is
  protecting LIST[DepositItem] .

  class CustomerPanel | rc : Oid .
  class Receiver | cp : Oid, cur : Oid, dt : Oid .
  class Current | list : List[DepositItem], sum : Euro,
                  rc : Oid, cp : Oid .
  class DayTotal | list : List[DepositItem] .

  msg return : CItem Oid -> Msg .
  msg new : Oid DepositItem Oid -> Msg .
  msg add : Oid DepositItem Oid -> Msg .
  msg conc : Oid DepositItem Oid -> Msg .
  msg ack : -> Msg .
  msg receipt : Oid -> Msg .
  msg printreceipt : Oid Oid -> Msg .
  msg get : Oid Oid -> Msg .
  msg to : Oid List[DepositItem] Euro Oid -> Msg .
  msg send : Oid List[DepositItem] Euro Oid -> Msg .
  msg print : Oid List[DepositItem] Euro -> Msg .

  vars Cp Rc Cur Dt : Oid .
  var Ci : CItem .
```

```
var I : DepositItem .
var L : List[DepositItem] .
var S : Euro .

rl [return] :
  return(Ci, Cp)
  < Cp : CustomerPanel | rc : Rc > =>
    < Cp : CustomerPanel | rc : Rc >
    new(Cp, desc(Ci), Rc) .
rl [new] :
  new(Cp, I, Rc)
  < Rc : Receiver | cp : Cp, cur : Cur, dt : Dt > =>
    < Rc : Receiver | cp : Cp, sum : S, cur : Cur, dt : Dt >
    conc(Rc, I, Cur)
    add(Rc, I, Dt)
    ack .
rl [conc] :
  conc(Rc, I, Cur)
  < Cur : Current | list : L, sum : S, rc : Rc, dt : Dt > =>
    < Cur : Current | list : I L, sum : price(I) + S,
                      rc : Rc, dt : Dt > .
rl [add] :
  add(Rc, I, Dt)
  < Dt : DayTotal | list : L > =>
    < Dt : DayTotal | list : I L > .
rl [receipt] :
  receipt(Cp)
  < Cp : CustomerPanel | rc : Rc > =>
    < Cp : CustomerPanel | rc : Rc >
    printreceipt(Cp, Rc) .
rl [printreceipt] :
  printreceipt(Cp, Rc)
  < Rc : Receiver | cp : Cp, cur : Cur, dt : Dt > =>
    < Rc : Receiver | cp : Cp, cur : Cur, dt : Dt >
    get(Rc, Cur) .
rl [get] :
  get(Rc, Cur)
  < Cur : Current | list : L, sum : S, rc : Rc, dt : Dt > =>
    < Cur : Current | list : nil, sum : 0 E, rc : Rc, dt : Dt >
    to(Cur, L, S, Rc) .
rl [to] :
  to(Cur, L, S, Rc)
  < Rc : Receiver | cp : Cp, cur : Cur, dt : Dt > =>
    < Rc : Receiver | cp : Cp, cur : Cur, dt : Dt >
    send(Rc, L, S, Cp) .
```

```
  rl [send] :
    send(Rc, L, S, Cp)
    < Cp : CustomerPanel | rc : Rc > =>
      < Cp : CustomerPanel | rc : Rc >
      print(Cp, L, S) .
endom
```

References

[1] K. Achatz, W. Schulte, A formal OO method inspired by Fusion and Object-Z, in: J.P. Bowen, M.G. Hinchey, D. Till (Eds.), Proc. 10th Internat. Conf. on Z Users, Lecture Notes in Computer Science, Vol. 1212, Springer, Berlin, 1997, pp. 92–111.

[2] E. Astesiano, M. Cerioli, G. Reggio, Plugging data constructs into paradigm-specific languages: towards an application to UML, in: T. Rus (Ed.), Proc. 8th Internat. Conf. on Algebraic Methodology and Software Technology, Lecture Notes in Computer Science, Vol. 1816, Springer, Berlin, 2000, pp. 273–292.

[3] E. Astesiano, G. Mascari, G. Reggio, M. Wirsing, On the parameterized algebraic specification of concurrent processes, in: H. Ehrig, C. Floyd, M. Nivat, J.W. Thatcher (Eds.), TAPSOFT'85, Vol. 1, Lecture Notes in Computer Science, Vol. 185, Springer, Berlin, 1985, pp. 342–358.

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

[5] J.A. Bergstra, J.W. Klop, Algebra of communicating processes with abstraction, Theoret. Comput. Sci. 37 (1985) 77–121.

[6] G. Booch, Object-Oriented Analysis and Design: With Applications, 3rd ed., Addison-Wesley, Reading, MA, 1999.

[7] P. Borovansky, C. Kirchner, H. Kirchner, P.-E. Moreau, M. Vittek, ELAN: a logical framework based on computational systems, in: J. Meseguer (Ed.), Proc. 1st Internat. Workshop on Rewriting Logic and Its Applications, Electronic Notes in Theoretical Computer Science, Vol. 4, Elsevier, Amsterdam, 1996, pp. 35–50.

[8] E. Brinksma (Ed.), LOTOS: a formal description technique based on the temporal ordering of observational behaviour, Technical Report Dis 8807, ISO, 1987.

[9] M. Clavel, Reflection in general logics and in rewriting logic with applications to the Maude language, Ph.D. Thesis, Universidad de Navarra, 1998.

[10] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, J. Quesada, Maude: Specification and Programming in Rewriting Logic. Manual, Computer Science Laboratory, SRI, 1999. URL: http://maude.csl.sri.com/manual.

[11] M. Clavel, J. Meseguer, Internal strategies in a reflective logic, in: B. Gramlich, H. Kirchner (Eds.), Proc. CADE 14 Workshop on Strategies in Automated Deduction, Townsville, 1997, pp. 1–12.

[12] D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, P. Jeremaes, Object-Oriented Development — The Fusion Method, Prentice-Hall, Englewood Cliffs, NJ, 1994.

[13] S. Cook, J. Daniels, Designing Object Systems — Object-Oriented Modelling With Syntropy, Prentice-Hall, New York, 1994.

[14] M. Dodani, R. Rupp, Integrating formal methods with object-oriented methodologies, in: M. Wirsing (Ed.), ICSE-17 Workshop on Formal Methods Applications in Software Engineering Practice, Seattle, 1995, pp. 212–219.

[15] F. Durán, A reflective module algebra with applications to the Maude language, Ph.D. Thesis, Universidad de Málaga, 1999.

[16] H.-D. Ehrich, M. Gogolla, A. Sernadas, Objects and their specification, in: M. Bidoit, C. Choppy (Eds.), Recent Trends in Data Type Specification, Lecture Notes in Computer Science, Vol. 655, Springer, Berlin, 1993, pp. 40–65.

[17] R.B. France, B. Rumpe (Eds.), Proc. 2nd Internat. Conf. on UML, Lecture Notes in Computer Science, Vol. 1723, Springer, Berlin, 1999.

[18] P. Hartel, G. Denker, M. Kowsari, M. Krone, H.-D. Ehrich, Information systems modelling with TROLL: formal methods at work, Inform. Systems 22 (2/3) (1997) 79–99.

[19] H. Hußmann, Formal Foundations for Software Engineering Methods, Lecture Notes in Computer Science, Vol. 1322, Springer, Berlin, 1997.

[20] I. Jacobson, G. Booch, J. Rumbaugh, The Unified Software Development Process, Addison-Wesley, Reading, MA, 1999.

[21] I. Jacobson, M. Christerson, P. Jonsson, G. Övergaard, Object-Oriented Software Engineering — a Use Case Driven Approach, 4th ed., Addison-Wesley, Wokingham, England, 1993.

[22] R.A. Kemmerer, Integrating formal methods into the development process, IEEE Software 7 (5) (1990) 37–50.

[23] S.-K. Kim, D. Carrington, Formalizing the UML class diagram using object-Z, in: R.B. France, B. Rumpe (Eds.), Proc. 2nd Internat. Conf. on UML, Lecture Notes in Computer Science, Vol. 1723, Springer, Berlin, 1999, pp. 83–98.

[24] URL: http://www.pst.informatik.uni-muenchen.de/~knapp/foose.html.

[25] A. Knapp, A Formal Semantics for UML Interactions, in: R.B. France, B. Rumpe (Eds.), Proc. 2nd Internat. Cont. on UML, Lecture Notes in Computer Science, Vol. 1723, Springer, Berlin, 1999, pp. 116–130.

[26] K. Lano, Formal Object-Oriented Development, Formal Approaches to Computing and Information Technology, Springer, London, 1995.

[27] K. Lano, R.B. France, J.-M. Bruel, A semantic comparison of Fusion and Syntropy, Object Oriented Systems 6 (1999) 43–72 (URL: http://compscinet.dcs.kcl.ac.uk/OO.).

[28] U. Lechner, Object-oriented specifications of distributed systems in the μ-calculus and Maude, in: J. Meseguer (Ed.), Proc. 1st Internat. Workshop on Rewriting Logic and Its Applications, Electronic Notes in Theoretical Computer Science, Vol. 4, Elsevier, Amsterdam, 1996, pp. 384–403.

[29] U. Lechner, C. Lengauer, F. Nickl, M. Wirsing, (Objects + concurrency) & reusability — a proposal to circumvent the inheritance anomaly, in: P. Cointe (Ed.), Proc. European Conf. on Object-Oriented Programming '96, Lecture Notes in Computer Science, Vol. 1098, Springer, Berlin, 1996, pp. 232–247.

[30] S. Mauw, An algebraic specification of process algebra, including two examples, in: M. Wirsing, J.A. Bergstra (Eds.), Algebraic Methods: Theory, Tools and Applications, Lecture Notes in Computer Science, Vol. 394, Springer, Berlin, 1989, pp. 507–554.

[31] J. Meseguer, A Logical theory of concurrent objects and its realization in the Maude language, in: G.A. Agha, P. Wegner, A. Yonezawa (Eds.), Research Directions in Concurrent Object-Oriented Programming, MIT Press, Cambridge, MA and London, 1991, pp. 341–389.

[32] J. Meseguer, Membership algebra as a logical framework for equational specifications, in: F.P. Presicce (Ed.), Select. Papers 12th Internat. Workshop on Recent Trends in Algebraic Development Techniques, Lecture Notes in Computer Science, Vol. 1376, Springer, Berlin, 1998, pp. 18–61.

[33] J. Meseguer, C.L. Talcott, A partial order event model for concurrent objects, in: J.C.M. Baeten, S. Mauw (Eds.), Proc. 10th Internat. Conf. on Concurrency Theory, Lecture Notes in Computer Science, Vol. 1664, Springer, Berlin, 1999, pp. 415–430.

[34] S. Nakajima, K. Futatsugi, An object-oriented modeling method for algebraic specifications in CafeOBJ, in: W.R. Adrion (Ed.), Proc. 19th Internat. Conf. on Software Engineering, 1997, pp. 34–44.

[35] Object Management Group, Unified modeling language specification, Version 1.3, Technical Report, OMG, 1999, URL: http://cgi.omg.org/cgi-bin/doc?ad/99–06–08.

[36] G. Övergaard, A formal approach to collaborations in the unified modeling language, in: R.B. France, B. Rumpe (Eds.), Proc. 2nd Internat. Cont. on UML, Lecture Notes in Computer Science, Vol. 1723, Springer, Berlin, 1999, pp. 99–115.

[37] G. Reggio, Entities: an institution for dynamic systems, in: H. Ehrig, K.P. Jantke, F. Orejas, H. Reichel (Eds.), Recent Trends in Data Type Specification, Lecture Notes in Computer Science, Vol. 534, Springer, Berlin, 1991, pp. 244–265.

[38] G. Reggio, L. Repetto, CASL-CHART: a combination of statecharts and the algebraic specification language CASL, in: T. Rus (Ed.), Proc. 8th Internat. Conf. on Algebraic Methodology and Software Technology, Lecture Notes in Computer Science, Vol. 1816, Springer, Berlin, 2000, pp. 243–272.

[39] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, Object-Oriented Modeling and Design, Prentice-Hall, Englewood Cliffs, NJ, 1991.

[40] S. Shlaer, S.J. Mellor, Object Lifecycles: Modelling the World in States, Yourdon, Englewood Cliffs, NJ, 1992.
[41] J. Warmer, A. Kleppe, The Object Constraint Language, Addison-Wesely, Reading, MA, 1999.
[42] M. Wirsing, Algebraic specification, in: J. van Leeuwen (Ed.), Handbook of Theoretical Computer Science, Vol. B: Formal Models and Semantics, Elsevier, Amsterdam, 1990, pp. 675–788.