Electronic Notes in Theoretical Computer Science 5 (1996) *to appear*

Object-Oriented Specifications of Distributed Systems in the μ-Calculus and Maude

Ulrike Lechner [1]

Fakultät für Mathematik und Informatik
Universität Passau
D–94030 Passau, Germany

Abstract

We refine an abstract property-oriented specification in the μ-calculus to a specification in Maude. As an intermediate step, we use a structured specification in the μ-calculus blended with propositions on states appropriate for object-oriented specification. We use the loose approach in refinement and refine data types as well as behavior. Throughout, our example is the bounded buffer.

1 Introduction

Specification languages provide a level of abstraction from implementation details in the design of complex systems. Specifications are property-oriented descriptions, while programs are executable descriptions of an algorithm. We use two specification formalisms, the μ-calculus [5,19,29] and Maude [25,28] for the object-oriented specification of concurrent systems.

Maude has been developed especially for the object-oriented specification of concurrent systems. The rewriting calculus and the underlying rewriting logic make Maude to very powerful and general specification language [25,27]. Maudes advantages are its object model, its ability to combine the two paradigms of inheritance and concurrency in a sensible way [21,26] and, particularly, its abstract way of specifying synchronization and communication between objects [22]. But, on the other hand, Maude's semantics is operational, and thus not really property-oriented, and the transition rules specifying the behavior of objects are not powerful enough to express, e.g., safety properties. The μ-calculus is property-oriented, i.e., it is able to express safety and liveness properties of the behavior of a concurrent system [5,19,29]. We enrich the μ-calculus with basic propositions on states that make it possible to reason also about the properties of states, not only about the properties of the dynamic behavior.

[1] The author is supported by the DFG under project OSIDRIS and received travel support from the ARC program of the DAAD.

There exist several approaches to specify or describe objects, especially objects in a concurrent setting and to give a formal semantics. We follow the basic concepts of [16,17,31], where the semantics of a class specification is given by a coalgebraic construction. Such a coalgebra specifies the observations one can make of an object or a class, it does not specify how a class is built up and which data it contains. (In the algebraic approach constructors describe which properties an object has, which data it contains [6]). This coalgebraic construction reflects the idea of an observable properties rather than the properties of the implementation of a class. We adopt this idea and use the μ-calculus with greatest fixpoint as our construct of specification.

We use three levels of specification with different degrees of abstraction:

(i) At the *abstract* level, we use the language of modal μ-formulas for specification. A typical specification would describe, e.g., the sequences of messages an object or a collection of objects accepts. Invariants on states restrict the transition system such that all states obey certain wellformedness conditions.

(ii) At the *intermediate* level, we use again the language of μ-formulas. The propositions on states we introduce describe the states of the objects. At this level, the formulas have a very rigid structure. The behavior of the objects belonging to a common class is a conjunction of five formulas specifying (1) that objects are persistent (2) the consistent state of an object (3) the synchronization code determining which messages are accepted depending on the local state of an object (4) the state changes of the objects and (5) the answer messages generated.

(iii) At the *concrete* level, we use Maude as our specification language. At this point, Maude itself provides us with a choice of the degree of abstraction, using, e.g., implicit synchronous or explicit asynchronous communication.

This refinement from an abstract to a concrete specification is reflected at the semantic level by the loose approach to refinement. In this approach, the semantics of a specification is the set of all models that satisfy all formulas of the specification. In each refinement step, the set of models of the specification becomes smaller. The last, most concrete step in such a refinement typically yields a singleton set of models – a program. In this paper all specifications are at a very high level of abstraction: our abstract, property-oriented specification language is the μ-calculus, our concrete, executable language the specification language Maude.

In the process of refinement the properties that shall be preserved determine the kind of relation between the different levels of specification. Particularly the refinement of concurrent systems offers a large variety of relations between transition systems. From [7,20,23] it is known that only a bisimulation relation between finite transition systems preserves all properties, which the μ-calculus can express. Coarser relations between transition systems preserve only certain classes of μ-formulas.

The paper is organized as follows: In Sect. 2 we give an introduction to Maude and the μ-calculus. The refinement relation is defined in Sect. 3. In

Sect. 4 we give specifications at different levels of abstraction and relate the three different levels. Throughout, our example is the bounded buffer. In Sect. 5 we relate our approach of specification and refinement to other work.

Our mathematical notation follows Dijkstra [10]. Quantification over a dummy variable x is written $(Q\,x : R(x) : P(x))$. Q is the quantifier, R a predicate in x representing the range of the dummy and P a term that depends on x. E.g., $(\cup x : x \in X \wedge \phi(x) : x)$ is the set of all elements of X for which $\phi(x)$ holds. Formal logical deductions are written:

$$\begin{array}{ll} & \textit{formula}_1 \\ \text{op} & \{\text{ comment explaining the validity of this relation }\} \\ & \textit{formula}_2 \end{array}$$

2 The specification languages

2.1 Maude

Maude [25–28] is an object-oriented specification language for the specification of distributed systems. In this section we assume prior knowledge of Maude and explain only the aspects of Maude relevant in our work. Let us give the specification of a bounded buffer, `BDBUFFER`, and explain it later.

```
omod BDBUFFER is
  protecting OIDLIST .
  extending CONFIGURATION .

  class BdBuffer | in: Nat, out: Nat, max: Nat, cont: OIdList .
  msg (to _ : get)  : OId -> Msg .
  msg (answer to get is _ ) :  OId -> Msg .
  msg (to _ : put _ ) : OId OId -> Msg .

  vars B U E : OId .
  vars I O M : Nat .
  var L : OIdList .

  [get] rl  (to B: get)
            < B:BdBuffer | in:I, out:O, max:M, cont:L E >
            => < B:BdBuffer | out:O+1, cont:L >
               (answer to get is E)
            if I - O > 0 and I - O =< M
               and length(L E) = I - O .

  [put] rl  (to B: put E)
            < B:BdBuffer | in:I, out:O, max:M, cont:L >
            => < B:BdBuffer | in:I+1, cont:E L >
            if I - O < M and  I-O >= 0
               and length(L) = I - O .
endom
```

In the specification `BDBUFFER` we declare one class, `BdBuffer` with four attributes. The behavior of a bounded buffer is specified by two transition rules, with label `get` and `put`. They specify whether and how a bounded buffer reacts to a `put` or `get` message: If the pattern at the left hand side of a rule matches a buffer and a message in a configuration, and if the precondition of the rule holds, then a state transition *may* happen such that the message is removed from the configuration, the buffer changes its state, i.e., the values of attributes, according to the rule and, possibly, an `answer` message is generated and part of the resulting configuration.

`BDBUFFER` imports two specifications: `CONFIGURATION` contains the basic data types like objects, messages and configurations (see, e.g., [28]) and `OIDLIST` the specification of lists of objects identifiers (see App. A).

Important for us is that Maude employs asynchronous message passing and, thus, the rewrite rules specify the possible state transitions that *may* happen. Important is also that Maude abstracts from the implementations of methods, what we specify here is the communication between objects and the state changes of objects and the overall system and not how the state changes are performed.

A rewriting calculus applies these rules to configurations. We use as in [21] a simplified version of a rewriting calculus. Let us introduce some notation. A *specification* $Sp = (\Sigma, E, T)$ consists of a *signature* Σ, a set of *equations* E and a set of *transition rules* T. A signature $\Sigma = (S, C, \leq, F, M)$ consists of a set of (ordinary) sort names S, a set of class names C, a subclass relation $\leq$, a set of function symbols F and a set of messages M. $T(\Sigma, X)$ denotes the terms with variables from X of a signature Σ. We use `Cf` as an abbreviation for `Configuration`, the sort of the states.

The *rewriting calculus*, given below in three rules, defines Maude's semantics in the form of a *transition system*.[2] In the following, let m, m' denote messages, a_i attribute names, v_i and w_i values, o_i object identifiers, C_i, C'_i, D_i and D'_i class identifiers, $atts_i$ sets of pairs of attributes together with their variables, and σ a substitution. An expression e adorned with an overbar, $\overline{e}$ stands for a set whose elements are of the form e (with the exception that $\overline{m}$ is a multiset of messages).

A transition

$$\begin{array}{l} \overline{m}[\sigma] \\ \overline{\texttt{<}\ \sigma(o_i) : D_i \mid \overline{a_i : v_i}[\sigma],\ atts_i\ \texttt{>}} \\ \to \overline{\texttt{<}\ \sigma(o_i) : D'_i \mid \overline{a_i : w_i}[\sigma],\ atts_i\ \texttt{>}} \\ \quad \overline{m'}[\sigma] \end{array} \tag{Inst}$$

is possible if T contains a transition rule (in which all attributes of classes C_i together with their values are stated)

[2] In contrast to [25] we neither have a reflexivity nor a transitivity rule nor parallel composition in the calculus. The rule (Emb) is weaker than the replacement rule in the original calculus; the replacement rule could be obtained by (Emb), (Equ), and a transitivity rule.

$$
\begin{array}{ll}
[R] & \overline{m} \\
 & \overline{\texttt{<}\ o_i : C_i \mid \overline{a_i : v_i}\ \texttt{>}} \\
 & \texttt{=>}\ \overline{\texttt{<}\ o_i : C_i' \mid \overline{a_i : w_i}\ \texttt{>}} \\
 & \quad \overline{m'}
\end{array}
$$

and a substitution $\sigma : \mathit{Vars} \to T(\Sigma, X)$,
where $D_i \leq C_i$ and

$$D_i' = \begin{cases} D_i, \text{ if } C_i = C_i' \\ C_i' \quad \text{else} \end{cases}$$

In the case of a conditional transition rule of the form:

$m_1' o_{i_1}' \ldots o_{i_n}' \to o_{j_1}' \ldots o_{j_m}' m_2' \ldots m_n'$ if $p_1 \wedge \ldots \wedge p_k$

(with equations or transitions $p_1, \ldots, p_k$) we require additionally that all $p_i[\sigma]$ are derivable. We need two more rules: (Emb) embeds the left-hand and the right-hand side of a transition into a configuration, containing objects and messages not changed by the transition and (Equ) makes the transition relation compatible with equations. Let c, d, c', d' and h be configurations and let $=_E$ denote equality modulo equations in the set E:

$$c\ h \to d\ h \text{ if } c \to d \tag{Emb}$$

$$c' \to d' \text{ if } c \to d \text{ and } c =_E c', d =_E d' \tag{Equ}$$

A structure (A, R) is an initial model of specification $\mathit{Sp} = (\Sigma, E, T)$, written (A,R) = I($\mathit{Sp}$), iff

- A is an order sorted Σ-algebra and A is the initial model of (Σ, E) [12].
- R is a relation, $R \subseteq A_{\mathrm{Cf}} \times A_{\mathrm{Cf}}$ such that $(c^A, d^A) \in R$ iff $\mathit{Sp} \vdash c \to d$ where the rules of the calculus are substitution rules, (Inst), (Emb) and (Equ).

Later in this paper we use labeled transitions systems. A transition is computed from one application of the rule (Inst) and applications of (Emb) and (Equ). The label is $m[\sigma]$, where m is the multiset of messages part of the left-hand side of the transition rule and σ the substitution applied by (Inst). Analogously we adapt definition of the relation $R \subseteq A_{\mathrm{Cf}} \times A_{\mathrm{Msg}} \times A_{\mathrm{Cf}}$ where $(c^A, m^A, d^A) \in R$ iff $\mathit{Sp} \vdash c \xrightarrow{m} d$. We also use the notation R_{m^A} for the subset of R with label m^A. (t^A is the representation of a ground term t in algebra A.)

2.2 The μ-calculus

The μ-calculus is used to reason about state transition systems at a property-oriented level. The language of μ-formulas consists of propositions, for reasoning about states, the modal connectives, quantifiers and fixpoint operators.

Our language of propositions for a specification is given by the grammar:

$$p ::= \texttt{tt} \mid \texttt{ff} \mid \neg p \mid \text{“}o\text{”} \mid \text{“}m\text{”}$$

where o, respectively m, is a term over a signature Σ representing an object respectively a message. The double quotes around an object or message represent the proposition "this object exists" or "this message exists", respectively. E.g., state C satisfies "`<B1:BdBuffer|in:1>`" if one of its elements is an object with object identifier `B1` belonging to class `BdBuffer` (which includes all subclasses of `BdBuffer`) whose value of attribute `in` is equal to `1`. Note that the use of negation is restricted to basic propositions.

Let p be a proposition. We define the formulas of the *modal μ-calculus* over a set of basic propositions of signature Σ as follows:

$$\begin{aligned}\phi ::= {} & p \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \\ & \mid \langle L\rangle\phi \mid [L]\phi \\ & \mid (\nu X : X \in \Phi : \phi) \mid (\mu X : X \in \Phi : \phi)\end{aligned}$$

L is a set of labels. $[L]\phi$ and $\langle L\rangle\phi$ are the labeled modal connectives. $[L]$ is called the box operator, $\langle L\rangle$ is called the diamond operator. Intuitively, $[L]\phi$ holds, if ϕ holds immediately after all transitions with labels in L. Dually, $\langle L\rangle\phi$ holds, if there is a transition with a label in L such that ϕ holds immediately afterwards. We use $\langle -\rangle$ and $[-]$ as abbreviations for modal connectives with the label set of all possible labels.

ν is the greatest fixpoint operator used, typically, for invariant (safety, "always") properties. μ is the least fixpoint operator used, typically, for variant (liveness, "sometime") properties.

We are interested in the truth of formulas in a structure (A, R) which is a model of a Maude specification $Sp = (\Sigma, E, T)$. Let $v : \mathrm{VAR} \to T(\Sigma)$ be a valuation and let $v^\star$ denote the canonical extension of v to an interpretation function for terms. Let $X := C$ be a valuation where C is assigned to X and $w + v$ a valuation such that $w + v(X) = w(X)$ if $X \in \mathrm{dom}(w)$ and $v(X)$ if $X \notin \mathrm{dom}(w)$. Let t^A denote the representation in algebra A of a ground term t. Let $FV(f)$ denote the free variables of formula f. We define $\mid \phi \mid_{(A,R),v}$ as the set of all states for which ϕ under the valuation v holds.

We define truth of formulas of the μ-calculus with respect to a state (or configuration) $C \in A_{\mathrm{Cf}}$ in Fig. 1.

We define for a μ-specification $\Phi = \{\phi_1, \ldots, \phi_n\}$ with μ-formulas over a algebraic specification (Σ, E) the class of models of Φ, $\mathrm{MOD}(\Phi)$, as the set of structures (A, R) where A is a model of (Σ, E) and $(A, R) \models \phi_1 \wedge \ldots \wedge \phi_n$.

3 Refinement relation

In the process of stepwise refinement an abstract (requirement) specification is transformed into a concrete specification, which might be a program [32]. We use the so-called loose approach to refinement: We consider the class of all models as the semantics of a specification. In the process of stepwise refinement, implementation details of data types and algorithms are added to the specification. This reduces the class of models (in several steps) to a singleton set – a program.

$(A,R),C,v \models$ "`<o:X|`$\overline{\texttt{a:v}}$`>`" iff $v^\star($`<o:X|`$\overline{\texttt{a:v}}, \overline{\texttt{b:w}}$`>`$) \in C$ for some $\overline{\texttt{w}}$
and $a \cup b$ are the attributes of class X

$(A,R),C,v \models$ "m" iff $v^\star(m) \in C$

$(A,R),C,v \models \neg\phi$ iff $(A,R),C,v \not\models \phi$

$(A,R),C,v \models \phi_1 \wedge \phi_2$ iff $(A,R),C,v \models \phi_1$ and $(A,R),C,v \models \phi_2$

$(A,R),C,v \models \phi_1 \vee \phi_2$ iff $(A,R),C,v \models \phi_1$ or $(A,R),C,v \models \phi_2$

$(A,R),C,v \models \langle L\rangle\phi$ iff for some $l \in L, (C,C') \in R_{v^\star(l)}$
and $(A,R),C',v \models \phi$

$(A,R),C,v \models [L]\phi$ iff $l \in L$ and $(C,C') \in R_{v^\star(l)}$
implies $(A,R),C',v \models \phi$

$(A,R),C,v \models (\mu X : X \in \Phi : \phi)$ iff

$$C \in (\cap C' : C' \subseteq \Phi^A, |\,\phi\,|_{(A,R),X:=C'+v} \subseteq C' : C')$$

$(A,R),C,v \models (\nu X : X \in \Phi : \phi)$ iff

$$C \in (\cup C' : C' \subseteq \Phi^A, |\,\phi\,|_{(A,R),X:=C'+v} \supseteq C' : C')$$

We define

$(A,R),C \models \phi$ iff for all $v : FV(\phi) \to A_s$ holds $(A,R),C,v \models \phi$

$(A,R) \quad \models \phi$ iff for all $C \in A_{\mathrm{Cf}}$ holds $(A,R),C \models \phi$

Fig. 1. Truth of μ-formulas

Definition 3.1 Let Σ be signature and (Σ, E) be an (algebraic) specification.
Let $\Phi = \{\phi_1, \ldots \phi_n\}$ and $\Phi' = \{\phi'_1, \ldots \phi'_{n'}\}$ be μ-specifications.
Φ is refined by Φ' written $\Phi \rightsquigarrow \Phi'$ iff $\phi'_1 \wedge \ldots \wedge \phi'_{n'} \implies \phi_1 \wedge \ldots \wedge \phi_n$.
Let $\Phi' = \{\phi'_1, \ldots \phi'_{n'}\}$ be a μ-specification over (Σ, E) and $Sp = (\Sigma, E, T)$ a Maude specification.
Φ' is refined by Sp, written $\Phi' \rightsquigarrow Sp$ iff $I(Sp) \models \phi'_1 \wedge \ldots \wedge \phi'_{n'}$

Let us briefly explain the relations between the specifications, particularly between specifications in different languages. The specifications have a basic signature andalso basic data types, specified by equations, in common. Thus the order-sorted algebra A and, in particular, the states, i.e., the terms of sort Configuration (abbreviated Cf) are the same. Different is the level of abstraction in the language for specifying the behavior of objects but common to these languages are the transition system: the semantics of a Maude specification is a transition system and the μ-properties are verified for the transition system.

4 The specifications

We give three specifications, Φ_A, Φ_M, Sp_C of a bounded buffer, each at a different level of abstraction, such that

$$\Phi_A \rightsquigarrow \Phi_M \rightsquigarrow Sp_C$$

4.1 The abstract level – Invariants and μ-calculus

Relevant at the most abstract level are –for us– only two issues:

- *What is a bounded buffer?*
- *How does a bounded buffer behave?*

The question "What is a bounded buffer?" can be answered by specifying a property each bounded buffer has to satisfy:

$$\mathit{State}(\texttt{B}) = \text{“}\texttt{<B:BdBuffer>}\text{”} \implies 0 \leq \texttt{length}(\texttt{B.cont}) \leq \texttt{B.max}$$

The number of elements stored in a buffer is `length(B.cont)`. It is not larger than `B.max`, the maximal number of elements of a buffer.

At this abstract level, we do not give the implementation of the state, but, instead, we require that a buffer has certain properties: it is able to determine the length of its contents and it stores only the maximum number of elements.

"What is a bounded buffer?" is answered by specifying properties of the state. This is usually not done in the μ-calculus. The specification of "How does a bounded buffer behave?" is answered by giving properties that do not involve the state but the interactions of objects via messages. For specifying this aspect we use the modal μ-calculus.

So, a general approach to answering "what is ... "? is to give basic properties and functions (`cont`, `max`, `length`) as well as invariants for objects and configurations (here we have only an invariant for one object, *State*). The answer to "how does ... behave?" gives possible actions (or messages) of an object together with their allowed or required (sequences of) actions.

If the predicate *State* holds in some state for a buffer `B`, then it holds in all subsequent states for `B`:

$$\phi_1(\texttt{B}) = \mathit{State}(\texttt{B}) \implies (\nu X :: \mathit{State}(\texttt{B}) \wedge [-]X)$$

The notation $[-]$ is an abbreviation for $[L]$, where L is the set of all possible actions. We do not care what happens with buffers that are in an inconsistent state. ϕ_1 ensures that a consistent buffer remains consistent.A bounded buffer accepts a `put` or a `get` and possibly both messages:

$$\begin{aligned}\phi_2(\texttt{B}) = \text{“}\texttt{<B:BdBuffer>}\text{”} &\implies \\ &(\nu X :: (\wedge \texttt{E} : \texttt{E} \in \texttt{OId} : \\ &\qquad (\langle \texttt{(to B: put E)} \rangle X \vee \langle \texttt{(to B: get)} \rangle X)))\end{aligned}$$

After an element has been put into the buffer, there is a sequence of `get` messages such that an `answer` carries the element. The result of a `get` message is an answer message that is part of the configuration waiting to be processed from –maybe– the object that sent the `get` message. At this point we apply already the asynchronous message passing mechanism of Maude to make refinement later feasible.

$$
\begin{aligned}
\phi_3(\mathtt{B}) = \text{“}\mathtt{<B:BdBuffer>}\text{”} \implies \\
&(\nu X :: (\wedge \mathtt{E} : \mathtt{E} \in \mathtt{OId} : \\
&\quad [\mathtt{(to\ B:\ put\ E)}] \\
&\quad (\mu Y :: \langle \mathtt{(to\ B:\ get)} \rangle \\
&\qquad (Y \vee \text{“}\mathtt{(answer\ to\ get\ is\ E)}\text{”})) \wedge X))
\end{aligned}
$$

After storing two elements in a buffer, the element stored first is retrieved before the one stored second (FIFO):

$$
\begin{aligned}
\phi_4(\mathtt{B}) = \text{“}\mathtt{<B:BdBuffer>}\text{”} \implies \\
&(\nu X :: (\wedge \mathtt{E1}, \mathtt{E2} : \mathtt{E1}, \mathtt{E2} \in \mathtt{OId} : \\
&\quad [\mathtt{(to\ B:\ put\ E1)}][\mathtt{(to\ B:\ put\ E2)}] \\
&\quad (\mu Y :: \langle \mathtt{(to\ B:\ get)} \rangle \\
&\qquad (Y \vee (\text{“}\mathtt{(answer\ to\ get\ is\ E1)}\text{”} \\
&\qquad\quad \wedge \langle \mathtt{(to\ B:\ get)} \rangle \\
&\qquad\quad\ \text{“}\mathtt{(answer\ to\ get\ is\ E2)}\text{”}))) \wedge X))
\end{aligned}
$$

The abstract specification of a bounded buffer is $\Phi_A = \{\phi_1, \phi_2, \phi_3, \phi_4\}$. This set of formulas is just one suggestion to specify the behavior of a bounded buffer. Naturally one could think about entirely different sets of formulas.

4.2 The intermediate level – Structured μ-calculus

At this level of abstraction, we make the decision about the implementation of the internal state of a buffer, namely that the internal state is represented by a proposition “`< B:BdBuffer|in:I,out:O,max:M,cont:L >`”. Implicitly, also the object model manifests itself in the structure of the formulas.

Five formulas determine the behavior of a class. Each formula corresponds to a certain view. We have two internal views which specify consistent states and the state changes induced by the object, we have a property stating that objects are persistent and views for two interfaces: the answer messages that are produced and the link between the incoming messages and the state changes.

Let us give the formula schemata and explain them when applied to the specification of the bounded buffer.

Definition 4.1 Let `C` be a classname and `atts` resp. `atts'` denote the attributes with their values of class `C`. Let $\mathit{SI}(\mathtt{<B:C|atts}_i\mathtt{>})$, $\phi_i(\mathtt{<B:C|atts}_i\mathtt{>})$ and $\psi_i(\mathtt{<B:C|atts}_i\mathtt{>})$ be propositions on the state of an object `B` of class `C`.

Let $\mathtt{a}_i$ be message and let $p \in P$ be all free variables in the formulas with their scope.

We define five formula schemata for a class `C` with n methods:

$$Persistence(B) = (\nu X : \ldots : (\wedge p : p \in P :$$

$$\text{``}\mathtt{<B:C>}\text{''} \implies [-](\text{``}\mathtt{<B:C>}\text{''} \wedge X)))$$

$$State(B) = (\nu X : \ldots : (\wedge p : p \in P :$$

$$SI(\text{``}\mathtt{<B:C|atts>}\text{''}) \implies [-](SI(\text{``}\mathtt{<B:C|atts'>}\text{''}) \wedge X)))$$

$$Synchronization(B) = (\nu X : \ldots : (\wedge p : p \in P : (\wedge i : 1 \le i \le n :$$

$$\text{``}\mathtt{<B:C|atts>}\text{''} \wedge SI(\mathtt{<B:C|atts>}) \wedge \psi_i(\mathtt{<B:C|atts>})$$

$$\implies \langle m_i \rangle X)))$$

$$StateChange(B) = (\nu X : \ldots : (\wedge p : p \in P : (\wedge i : 1 \le i \le n :$$

$$\text{``}\mathtt{<B:C|atts>}\text{''} \wedge SI(\mathtt{<B:C|atts>}) \wedge \psi_i(\mathtt{<B:C|atts>})$$

$$\implies [m_i](\phi_i(\mathtt{<B:C|atts}_i\mathtt{'>}) \wedge X))))$$

$$AnswerMessages(B) = (\nu X : \ldots : (\wedge p : p \in P : (\wedge i : 1 \le i \le n :$$

$$\text{``}\mathtt{<B:C|atts>}\text{''} \wedge SI(\mathtt{<B:C|atts>}) \wedge \psi_i(\mathtt{<B:C|atts>})$$

$$\implies [m_i](\text{``}\mathtt{a}_i\text{''} \wedge X))))$$

A rather basic safety-property of a bounded buffer is persistence: if a bounded buffer is part of a configuration it is also part of all successor states:

$$Persistence(B) =$$

$$(\nu X :: (\wedge \mathtt{I}, \mathtt{O}, \mathtt{M}, \mathtt{L} : \mathtt{I}, \mathtt{O}, \mathtt{M} \in \mathtt{Nat}, \mathtt{L} \in \mathtt{OIdList} :$$

$$\text{``}\mathtt{<B:BdBuffer>}\text{''} \implies [\mathtt{-}]\text{``}\mathtt{<B:BdBuffer>}\text{''} \wedge X))$$

The formula *State* specifies an invariant for the internal state of an object. It has to hold for all objects of a class in all states, provided it holds once, and it ensures consistency of the internal state. We do not care for bounded buffers which are in an inconsistent state. At this level we make the design decision that the state of a bounded buffer is implemented by four attributes, namely `in`, `out`, `max`, and `cont`.

$$State(B) =$$

$$(\nu X :: (\wedge \mathtt{I}, \mathtt{O}, \mathtt{M}, \mathtt{L}, \mathtt{I'}, \mathtt{O'}, \mathtt{M'}, \mathtt{L'} :$$

$$\mathtt{I}, \mathtt{O}, \mathtt{M}, \mathtt{I'}, \mathtt{O'}, \mathtt{M'}, \in \mathtt{Nat}, \mathtt{L}, \mathtt{L'} \in \mathtt{OIdList} :$$

$$\text{``}\mathtt{<B:BdBuffer|in:I,out:O,max:M,cont:L>}\text{''}$$

$$\wedge \mathtt{length(L)} = \mathtt{I} - \mathtt{O} \wedge 0 \le \mathtt{I} - \mathtt{O} \le \mathtt{M}$$

$$\implies [\mathtt{-}]\ \text{``}\mathtt{<B:BdBuffer|in:I',out:O',max:M',cont:L'>}\text{''}$$

$$\wedge \mathtt{length(L')} = \mathtt{I'} - \mathtt{O'} \wedge 0 \le \mathtt{I'} - \mathtt{O'} \le \mathtt{M'} \wedge X))$$

We refer to the formula $\texttt{length(L)} = \texttt{I} - \texttt{O} \wedge 0 \leq \texttt{I} - \texttt{O} \leq \texttt{M}$ as *SI*, (state invariant) of a bounded buffer.

In the formula *Synchronization* we specify the so-called synchronization code. The synchronization code determines when an objects accepts which message. This synchronization code depends in our approach only on the state of an object.

$$
\begin{array}{l}
\mathit{Synchronization}(B) = \\
(\nu X :: (\wedge \texttt{I},\texttt{O},\texttt{M},\texttt{L},\texttt{E},\texttt{E'} : \texttt{I},\texttt{O},\texttt{M} \in \texttt{Nat}, \texttt{L} \in \texttt{OIdList}, \texttt{E},\texttt{E'} \in \texttt{OId} : \\
\qquad (\quad \text{``}\texttt{<B:BdBuffer|in:I,out:O,max:M,cont:L>}\text{''} \\
\qquad\quad \wedge\ \mathit{SI}(\texttt{<B:BdBuffer>}) \wedge \texttt{I} - \texttt{O} < \texttt{M} \\
\qquad\quad \Longrightarrow\ \langle\texttt{(to B: put E)}\rangle X)) \\
\qquad \wedge\, (\quad \text{``}\texttt{<B:BdBuffer|in:I,out:O,max:M,cont:L E'>}\text{''} \\
\qquad\quad \wedge\ \mathit{SI}(\texttt{<B:BdBuffer>}) \wedge \texttt{I} - \texttt{O} > 0 \\
\qquad\quad \Longrightarrow\ \langle\texttt{(to B: get)}\rangle X)))
\end{array}
$$

The two formulas *StateChange* and *Synchronization* specify the internal behavior of an object. From the synchronization code we obtain not only when a method may be invoked but also under which preconditions this method and the functions on data types must operate correctly on the state of the object. In the formula *StateChange*, we specify how methods change the state of an object. When a message is accepted it *always* changes the state of an object in the same way:

$$
\begin{array}{l}
\mathit{StateChange}(B) = \\
(\nu X :: (\wedge \texttt{I},\texttt{O},\texttt{M},\texttt{L},\texttt{E}, : \texttt{I},\texttt{O},\texttt{M} \in \texttt{Nat}, \texttt{L} \in \texttt{OIdList}, \texttt{E} \in \texttt{OId} : \\
\qquad (\quad \text{``}\texttt{<B:BdBuffer|in:I,out:O,max:M,cont:L>}\text{''} \\
\qquad\quad \wedge\ \mathit{SI}(\texttt{<B:BdBuffer>}) \wedge \texttt{I} - \texttt{O} < \texttt{M} \\
\qquad\quad \Longrightarrow\ [\texttt{(to B: put E)}] \\
\qquad\qquad (\text{``}\texttt{<B:BdBuffer|in:I+1,out:O,max:M,cont:E L >}\text{''} \wedge X) \\
\qquad \wedge(\quad \text{``}\texttt{<B:BdBuffer|in:I,out:O,max:M,cont:L E'>}\text{''} \\
\qquad\quad \wedge\ \mathit{SI}(\texttt{<B:BdBuffer>}) \wedge \texttt{I} - \texttt{O} > 0 \\
\qquad\quad \Longrightarrow\ [\texttt{(to B: get)}] \\
\qquad\qquad (\text{``}\texttt{<B:BdBuffer|in:I,out:O+1,max:M,cont:L>}\text{''} \wedge X)))
\end{array}
$$

After a `put` action, the value of attribute `in` is incremented and the element that is parameter to the message is added to the contents. After a `get` message, the value of `out` is incremented and the element which is parameter in the message added to the value of attribute `cont`.

Messages not only change the internal state of the objects, they also trigger answer messages to be created as part of the global state:

$$
\begin{array}{l}
\mathit{AnswerMessages}(B) = \\
(\nu X :: (\wedge \texttt{I,O,M,E,E',L} : \texttt{I,O,M} \in \texttt{Nat}, \texttt{E,E'} \in \texttt{OId}, \texttt{L} \in \texttt{OIdList} : \\
\qquad (\quad \text{“}\texttt{<B:BdBuffer|in:I,out:O,max:M,cont:L>}\text{”} \\
\qquad\ \ \wedge\ \mathit{SI}(\texttt{<B:BdBuffer>}) \wedge \texttt{I} - \texttt{O} < \texttt{M} \\
\qquad \Longrightarrow\ [\texttt{(to B: put E)}]X) \\
\qquad \wedge(\quad \text{“}\texttt{<B:BdBuffer|in:I,out:O,max:M,cont:L E'>}\text{”} \\
\qquad\quad \wedge\ \mathit{SI}(\texttt{<B:BdBuffer>}) \wedge \texttt{I} - \texttt{O} > 0 \\
\qquad \Longrightarrow\ [\texttt{(to B: get)}] \\
\qquad\qquad (\text{“}\texttt{(to U: answer to get is E)}\text{”} \wedge X)))
\end{array}
$$

After the message `(to P: get)`, message “`(answer to get is E)`” is part of the global state of the system. Again, we specify that after a method is invoked an answer message is *always* part of the (global) state.

The specification at the intermediate level is given by

$$
\begin{array}{l}
\Phi_M = \{\, \mathit{State}(B), \mathit{Persistence}(B), \\
\qquad\qquad \mathit{StateChange}(B), \mathit{AnswerMessages}(B), \mathit{Synchronization}(B)\}
\end{array}
$$

After giving those five formulas as a specification of a class, we would like to give a brief motivation why we use both diamond and box operators for modeling the different aspects of an object. The use of the diamond operator is quite easy to motivate: we are interested in which state transitions are possible for an object, which transitions an object may perform. This is the kind of property expressible by the diamond operator.

The use of the box operator needs more motivation and we give two reasons for preferring the box to the diamond operator for specifying the internal properties and behavior of objects. The first motivation is that, typically, even in object-oriented concurrent languages, an object is sequential and deterministic. Thus a property $\langle \texttt{get} \rangle \langle \texttt{answer to get is E} \rangle \phi$ would not reflect the situation that after a `get` action there is always an `answer` message possible for the overall system.

The second motivation for the use of the box operators lies in the properties of the overall system we are interested in. One very important property is absence of deadlocks specified by

$$\mathit{Deadlockfree} = (\nu X : \ldots : \langle - \rangle \texttt{tt} \wedge [-]X)$$

Let us explain this formula: in every state, a transition (with arbitrary label) is possible and after every transition the property *Deadlockfree* is satisfied.

Our schemata and formulas specify the behavior of a single class but they do not specify the behavior of the global system. From the global point of view a message must be part of the state to make the local transition of an object possible, provided the precondition specified in *Synchronization* holds as well. Assume we have in our specification an object that consumes always all `answer` messages, i.e., if a message is part of the global state then there is always a transition with this label possible.

If we use only diamond properties to specify the behavior of an object we would obtain the property

$$\langle \texttt{(to B get)} \rangle \texttt{tt} \wedge \langle \texttt{(to B get)} \rangle \langle \texttt{(answer to get is E)} \rangle \texttt{tt}$$

With the box operator we obtain

$$\langle \texttt{(to B get)} \rangle \texttt{tt} \wedge [\texttt{(to B get)}] \langle \texttt{(answer to get is E)} \rangle \texttt{tt}$$

This formula, which uses the box operator, is stronger and models the absence of a deadlock in a situation when a user waits for the bounded buffer.

Deadlocks are typically caused by the composition of a system as a (large) collection of objects belonging to different classes and deadlocks inside objects are not really an issue in specification. The box property specifying the internal behavior gives us a property which is important when composing a large system.

4.3 The concrete level – Maude

Maude's transition rules and rewriting calculus provide only the possibility to express sometime-properties on single actions, but not always-properties and not properties on sequences of actions. Each transition rule specifies the reaction (or one way to react to a message) of an object to a message. Thus we have to refine the intermediate specification which focuses on certain aspects of the behavior of a class to a specification which focuses on the local and global reaction to a message.

There is one more, severe difference between the concrete level of Maude and the object-oriented μ-specification: The rules of Maude describe the global transition system, and the formulas in structured μ-calculus properties of a single class.

Lemma 4.2 $Sp_M \rightsquigarrow$*BDBUFFER*.

Proof. $Sp_M \rightsquigarrow \texttt{BDBUFFER}$ iff

$$\begin{aligned} \texttt{I(BDBUFFER)} \models\ & Persistence(B) \wedge State(B) \\ \wedge\ & Synchronization(B) \\ \wedge\ & StateChange(B) \wedge AnswerMessages(B) \end{aligned}$$

We proceed as follows:

(i) We strengthen the formulas *Synchronization*, *StateChange*, *AnswerMessages* such that each precondition requires additionally the presence of the appropriate message in the global state. (The formulas are not given explicitly and their names are primed.)

(ii) We show that we may omit the fixpoint operator in the formulas in our particular setting.

(iii) We show that a formula containing a diamond operator $\langle m \rangle$ implies a formula containing a box operator $[m]$, provided there is always just one successor state reachable via a transition with label m.

(iv) We compose the strengthened formulas to a formula *Rule* such that *Rule* implies all three strengthened formulas.

(v) We prove that $I(\texttt{BDBUFFER}) \models \mathit{Rule}$.

(vi) We prove that $I(\texttt{BDBUFFER}) \models \mathit{Persistence}(B) \wedge \mathit{State}(B)$. (The proof is not given)

Step i: Is not given here.
Step ii: Let (A, R) be $I(Sp)$ of some specification Sp and ϕ a μ-formula. Then $(A, R) \models \phi$ iff for all $C \in A_{\mathrm{Cf}}$ holds $(A, R), C \models \phi$

$$\text{We prove:} \qquad (A,R), C \models \mathit{pre} \implies \langle L \rangle \mathit{post} \text{ for all } C \in A_{\mathrm{Cf}}$$

$$\text{implies } (A,R) \models (\nu X :: \mathit{pre} \implies \langle L \rangle(\mathit{post} \wedge X))$$

$(A,R) \models (\nu X :: \mathit{pre} \implies \langle L \rangle(\mathit{post} \wedge X))$
$\Leftrightarrow \quad \{$ Definition of $\models$ $\}$
$(A,R), C \models (\nu X :: \mathit{pre} \implies \langle L \rangle(\mathit{post} \wedge X))$
for all $C \in A_{\mathrm{Cf}}$
$\Leftrightarrow \quad \{$ Definition of $\models$ $\}$
$C \in (\cup C' : C' \subseteq A_{\mathrm{Cf}}, \mid \mathit{pre} \implies \langle L \rangle(\mathit{post} \wedge X) \mid_{(A,R), X := C'} \supseteq C' : C')$
for all $C \in A_{\mathrm{Cf}}$
$\Leftrightarrow \quad \{ \ S \subseteq S'$ and $C \in S$ for all $C \in S' \Leftrightarrow S = S' \ \}$
$A_{\mathrm{Cf}} = (\cup C' : C' \subseteq A_{\mathrm{Cf}}, \mid \mathit{pre} \implies \langle L \rangle(\mathit{post} \wedge X) \mid_{(A,R), X := C'} \supseteq C' : C')$
$\Leftarrow \quad \{ \ C' = A_{\mathrm{Cf}} \ \}$
$A_{\mathrm{Cf}} = \mid \mathit{pre} \implies \langle L \rangle(\mathit{post} \wedge X) \mid_{(A,R), X := A_{\mathrm{Cf}}}$
$\Leftrightarrow \quad \{ \ \}$
$A_{\mathrm{Cf}} = \mid \mathit{pre} \implies \langle L \rangle \mathit{post} \mid_{(A,R), X := A_{\mathrm{Cf}}}$

Thus, we may omit the fixpoint operator in the three formulas *Synchronization*, *AnswerMessages* and *StateChange*.
Step iii: In our specification `BDBUFFER` each method is implemented in only one rule, and thus $(A, R), C \models \langle l \rangle \phi \implies (A, R), C \models [l]\phi$ if l is a singleton set of labels.
Step iv: We define a new formula schema :

$$\begin{aligned} \mathit{Rule} = (\wedge\ B, p : B \in \texttt{OId}, p \in P : (\wedge i : 1 \leq i \leq n :& \\ \text{“}\texttt{<B:C|atts>}\text{”} \wedge \mathit{SI}(\texttt{<B:C|atts>}) \wedge \psi_i(\texttt{<B:C|atts>})& \\ \implies \langle m_i \rangle(\phi_i(\texttt{<B:C|atts}_i\texttt{'>}) \wedge \text{“}\texttt{a}_i\text{”})))& \end{aligned}$$

such that

$$\begin{aligned} \mathit{Rule} \implies\ & \mathit{Synchronization}'(B) \wedge \mathit{StateChange}'(B) \\ & \wedge \mathit{AnswerMessage}'(B) \end{aligned}$$

Step v:
To prove: for all $C \in A_{\mathrm{Cf}}$ holds
$(A, R), C \models$

$(\wedge$I, O, M, L, E, E' : I, O, M $\in$ Nat, L $\in$ OIdList, E, E' $\in$ OId :

$(($"(to B: put E)" $\wedge$ "<B:BdBuffer|in:I,out:O,max:M,cont:L>"

$\wedge SI($<B:BdBuffer>$) \wedge$ I $-$ O $<$ M$)$

$\implies \langle$(to B: put E)$\rangle$

"<B:BdBuffer|in:I+1,out:O,max:M,cont:E L>"$)$

$\wedge$ $(($"(to B: get)" $\wedge$ "<B:BdBuffer|in:I,out:O,max:M,cont:L E'>"

$\wedge SI($<B:BdBuffer>$) \wedge$ I $-$ O $>$ 0$)$

$\implies \langle$(to B: get)$\rangle($"(answer to get is E')"

$\wedge$"<B:BdBuffer|in:I,out:O+1,max:M,cont:L >"$))$

Proof by structural induction on the size of configurations.

Case 1 : $C =$ eps (the empty configuration)

$(A, R), C \models (\wedge \ldots : \ldots :$ ff $\implies \langle$(to B: put E)$\rangle \ldots$

$\wedge$ff $\implies \langle$(to B: get)$\rangle \ldots)$

Case 2 : $C \neq$ eps, no state transition possible.
No state transition possible if there exists no σ, such that σ applied to the left-hand side of rule [get] or [put] is part of C.
And, thus, for all valuations v

$(A, R), C, v \not\models$ "<B:BdBuffer|in:I,out:O,max:M,cont:L>"

$\wedge SI($<B:BdBuffer$) \wedge ($"(to B get)" $\vee$ "(to B put E)"$)$

and, thus, $(A, R), C \models \mathit{Rule}$

Case 3.1 : $C =$ <b:BdBuffer|in:i,out:o,max:m,cont:l> (to b put e) such that the transition rule with label [put] is applicable (small letters denote values)
Then $\sigma = [\mathtt{B} \to \mathtt{b}, \mathtt{I} \to \mathtt{i}, \mathtt{O} \to \mathtt{o}, \mathtt{M} \to \mathtt{m}, \mathtt{L} \to \mathtt{l}, \mathtt{e} \to \mathtt{E}]$ is the only substitution such that rule (Inst) of the calculus is applicable.

Let $D = \sigma($<B:BdBuffer|in:I,out:O+1,max:M,cont:L>$)$

$\sigma(($answer to get is E$)))$

Thus, $(C, D) \in R$ and

$(A, R), D \models$ "(answer to get is e)"

$\wedge$ "<b:BdBuffer|in:i,out:o+1,max:m,cont:l>"

and thus $(A, R), C \models \mathit{Rule}$

Case 3.2 : $C =$ <b:BdBuffer|in:i,out:o,max:m,cont:l> (to b get)
Analogously to Case 3.1.

Case 4 : $C = C_1\ C_2$
For all propositional variables ϕ and for $\langle L\rangle\phi$ holds:
$\phi(C_i) \implies \phi(C)$ and $(A, R), C_i \models \langle L\rangle\phi \implies (A, R), C \models \langle L\rangle\phi$
And by application of rule (Emb): $C_i \xrightarrow{m} D_i \implies C \xrightarrow{m} D_i\ C_j (i \neq j)$.

Thus we consider only a transition which is generated by an application

of (Inst) and (Emb) which not applicable by (Inst) to C_1 or C_2, This proof follows cases 3.1 or 3.2.
Step vi: Proof by induction on the size of configurations. □

5 Related Work

The main issues in our work are (1) the specification of objects (2) the expressiveness of specification languages for concurrent systems and (3) refinement of specifications.

In the specification of objects and, in particular, of the bounded buffer we combine, in all three levels of specification, algebraic specification [32] and specification languages for concurrent systems. The SMoLCS approach [2,3] combines also algebraic specification and specification of transition systems. The SMoLCS approach is designed for modular specification of the semantics of programming languages. Particular to the approach is the way the transition systems that model the behavior of a program are specified. Since algebras and transition systems are also the semantic foundations of our specifications, the SMoLCS approach could be used to specify a semantics for our μ and Maude specifications.

When comparing our object model and our two specification languages, Maude and the μ-calculus, to other object-oriented concurrent approaches, we notice that most other approaches like Troll [13], $\pi o\beta\lambda$ [18] use synchronous communication. In this respect, our μ and Maude specifications are related more closely to actor languages [1,11]. Troll and $\pi o\beta\lambda$ specify the implementation of methods inside objects and focus thus in specification on the intra-object view with properties of classes, while our approach abstracts from the implementation of methods and provides together with [20] an intra- and inter-object view.

The use of greatest fixpoints is inspired by the coalgebraic specifications of classes in [16,17,31]. While an algebraic specification specifies the properties of a class [6] the coalgebraic specification gives the observable behavior and properties. Thus the use of greatest fixpoint and the coalgebraic specification style reflect the principle of encapsulation, the basic concept of object orientation, more than the algebraic approach.

Refinement relations have been studied in various versions for algebraic specifications [4,15,32]. The initial and the loose approach to refinement are compared in [30]. The initial approach to refinement of Maude specifications is used in [24,33]. Refinement of object-oriented languages is studied also for Troll [8,9] and $\pi o\beta\lambda$[18]. These refinement approaches are mainly concerned with action refinement and refinement of communication between objects. Their correctness criterium of the refinement is the state of the global system, but not, like in our approach the behavior of a class. While all these approaches as well as our approach remain in the formal world of specification is in [33] an abstract informal specification in the object-oriented analysis method of Jacobsen refined to a Maude specification and from the Maude specification to a Java program.

6 Conclusions

Both Maude and the μ-calculus are specification languages that can be used for the specification of concurrent systems. While Maude is designed to make it executable, the advantage of the μ-calculus is its expressiveness. But, when we combine these two specification mechanisms, as in the intermediate level of our specification, we get a very expressive language, appropriate for specification. At this level, it should be relatively easy to find operators for various property-preserving kinds of reuse as, e.g., inheritance and, thus, *modal-μ-Maude* could be a specification language on its own.

Acknowledgement

We would like to thank the anonymous referees for their helpful comments. Many fruitful discussions with Christian Lengauer and Martin Wirsing helped to improve this work.

References

[1] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* Artificial Intelligence. MIT Press, 1986.

[2] E. Astesiano, G.F. Mascari, G. Reggio, and M. Wirsing. On the parameterized algebraic specification of concurrent systems. In H. Ehrig, C. Floyd, M. Nivat, and M. Thatcher, editors, *Mathematical Foundations of Software Development*, Lecture Notes in Computer Science 185, pages 342–358. Springer-Verlag, 1985.

[3] E. Astesiano and M. Wirsing. Bisimulation in algebraic specifications. In H. Ait-Kaci and M. Nivat, editors, *Resolution of Equations in Algebraic Structures*, Algebraic Techniques, Vol. 1, pages 1–32. Academic Press, London, 1989.

[4] M. Bidoit, R. Hennicker, and M. Wirsing. Behavioural and abstractor specifications. *Science of Computer Programming*, 2-3(25):146–186, 1995.

[5] J.C. Bradfield. *Verifying Temporal Properties of Systems.* Birkhäuser, 1992.

[6] R. Breu. *Algebraic Specification Techniques in Object-Oriented Programming Environments.* Lecture Notes in Computer Science 562. Springer-Verlag, 1991.

[7] G. Bruns. A practical technique for process abstraction. In E. Best, editor, *4th Int. Conf. on Concurrency Theory (CONCUR'93)*, Lecture Notes in Computer Science 715, pages 37–49. Springer-Verlag, 1993.

[8] G. Denker. A semantic characterisation of correct refinement of object specifications based on database schedules. In B. Freitag, C.B. Jones, C. Lengauer, and H.-J. Schek, editors, *Object Orientation with Parallelism and Persistence.* Kluwer, 1996. To appear.

[9] G. Denker and H.-D. Ehrich. Action Reification In Object Oriented Specifications. In *Proc. ISCORE Workshop Amsterdam, Sep. 94.* World Scientific, 1995.

[10] E.W. Dijkstra and C.S. Scholten. *Predicate Calculus and Program Semantics.* Texts and Monographs in Computer Science. Springer-Verlag, 1990.

[11] S. Frølund. Inheritance of synchronisation constraints in concurrent object-oriented programming languages. In O. Lehrmann Madsen, editor, *European Conf. on Object-Oriented Programming (ECOOP'92)*, Lecture Notes in Computer Science 615, pages 185–196. Springer-Verlag, 1992.

[12] J.A. Goguen and J. Meseguer. Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theoretical Computer Science*, 105:217–273, 1992.

[13] T. Hartmann, G. Saake, R. Jungclaus, P. Hartel, and J. Kusch. Revised version of the modelling language TROLL (Version 2.0). Technical Report Informatik-Bericht 94–03, TU Braunschweig, 1994.

[14] A.E. Haxthausen and F. Nickl. Pushouts of order-sorted algebraic specifications. In M. Wirsing and M. Nivat, editors, *Algebraic Methodology and Software Technology (AMAST 96)*, Lecture Notes in Computer Science 1101. Springer-Verlag, 1996.

[15] M. Hofmann and D. Sannella. On behavioral abstraction and behavioral satisfaction in higher-order logic. In *Proc. Theory and Practice of Software Development (TAPSOFT'95)*, Lecture Notes in Computer Science. Springer-Verlag, 1995. To appear.

[16] B. Jacobs. Inheritance and cofree constructions. In P. Cointe, editor, *European Conference on object-oriented programming (ECOOP)*, Lecture Notes in Computer Science 1098, pages 210–231. Springer-Verlag, 1996.

[17] B. Jacobs. Objects and classes, co-algebraically. In B. Freitag, C.B. Jones, C. Lengauer, and H.-J. Schek, editors, *Object Orientation with Parallelism and Persistence*. Kluwer, 1996. To appear.

[18] C.B. Jones. Constraining Interference in an Object-Based Design Method. In M.-C. Gaudel and J.-P. Jouannaud, editors, *Theory and Practice of Software Development (TAPSOFT'93)*, Lecture Notes in Computer Science 668, pages 136–150. Springer-Verlag, 1993.

[19] D. Kozen. Results on the propositional mu-calculus. *Theoretical Computer Science*, 27:333–354, December 1983.

[20] U. Lechner and C. Lengauer. Modal–μ–Maude — properties and specification of concurrent objects. In B. Freitag, C.B. Jones, C. Lengauer, and H.-J. Schek, editors, *Object Orientation with Parallelism and Persistence*. Kluwer, 1996. To appear.

[21] U. Lechner, C. Lengauer, F. Nickl, and M. Wirsing. (Objects + Concurrency) & Reusability – A Proposal to Circumvent the Inheritance Anomaly. In *ECOOP'96*, Lecture Notes in Computer Science 1098, pages 232–248. Springer-Verlag, 1996.

[22] U. Lechner, C. Lengauer, and M. Wirsing. An Object-Oriented Airport: Specification and Refinement in Maude. In E. Astesiano, G. Reggio, and

A. Tarlecki, editors, *10th Workshop on Specification of Abstract Data Types Joint with the 5th COMPASS Workshop, Selected papers*, Lecture Notes in Computer Science 906, pages 351–367. Springer-Verlag, 1995.

[23] C. Loiseaux, A. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. Property preserving abstraction for the verification of concurrent systems. *Formal Methods in System Design*, 6(1):11–45, 1995.

[24] J. Meseguer. A logical theory of concurrent objects and its realization in the Maude language. Technical Report SRI-CSL-92-08, SRI International, July 1992.

[25] 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 314–390. MIT Press, 1993.

[26] J. Meseguer. Solving the inheritance anomaly in concurrent object-oriented programming. In O. Nierstrasz, editor, *ECOOP '93 – Object-Oriented Programming*, Lecture Notes in Computer Science 707, pages 220–246. Springer-Verlag, 1993.

[27] J. Meseguer. Rewriting logic as a semantic framework for concurrency: a progress report. In *Concur 96*, Lecture Notes in Computer Science. Springer-Verlag, 1996. To appear.

[28] J. Meseguer and T. Winkler. Parallel programming in Maude. In J.-P. Banâtre and D. Le Métayer, editors, *Research Directions in High-Level Parallel Programming Languages*, Lecture Notes in Computer Science 574, pages 253–293. Springer-Verlag, 1992.

[29] R. Milner, J. Parrow, and D. Walker. Modal logics for mobile processes. *Theoretical Computer Science*, 25:267–310, 1993.

[30] F. Orejas, M. Navarro, and A. Sanchez. Implementation and behavioural equivalence: A survey. In M. Bidoit and C. Choppy, editors, *Recent Trends in Data Type Specifications*, Lecture Notes in Computer Science 655, pages 93–125. Springer-Verlag, 1993.

[31] H. Reichel. An approach to object semantics based on terminal co-algebras. *Math. Struct. Comp. Sci.*, 5:129–152, 1995.

[32] M. Wirsing. Algebraic specification. In J.V. Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 675–788. Elsevier (North-Holland), 1990.

[33] M. Wirsing and A. Knapp. A formal approach to object-oriented software engineering. *ENTCS*, 5, 1996. To appear.

A Specification of OIDLIST

```
obj OIDLIST is
  protecting OID .
  sort OIdList .

  op eps       : -> OIdList  .
  op _ _       : OId OIdList -> OIdList . (* left append *)
  op _ _       : OIdList OId -> OIdList . (* right append *)
  op length    : OIdList -> Nat .

  var E E1 E2 : OId .
  var L : OIdList .

  eq E2 eps = eps E2 .
  eq E1 (L E2)  = (E1 L) E2 .

  eq length (eps) = 0 .
  eq length (E L) = succ(length(L)).
endfm
```

Note that we do not use subsorting to implement lists. The reason for this is that in the presence of subsorting, the union operation, in general, does not preserve the coherence of signatures [14,21].