Concurrent Object-Oriented Design Specification in SPECTRUM

Martin Wirsing*, Friederike Nickl*, Ulrike Lechner**

*Institut für Informatik
Universität München
email: {wirsing, nickl}@informatik.uni-muenchen.de

**Fakultät für Mathematik und Informatik
Universität Passau
email: lechner@informatik.uni-passau.de

Abstract

In this paper an algebraic approach to formal object-oriented design specification is presented where the static and functional part of a software system is described by classical algebraic specification whereas the dynamic behavior is modeled by a transition relation. The approach is inspired by Astesiano's SMoLCS formalism and is based on Meseguer's rewriting logic; but it has two additional features, which are motivated by pragmatic object-oriented software development techniques: it supports the construction of subsystems, and the flow of messages can be controlled by use of a simple but powerful concurrent language. Liveness and safety properties of design specifications are formulated with the help of structured message expressions; methods for proving such properties are shortly discussed. As underlying specification language Broy's SPECTRUM is extended by features for concurrent object-oriented specification. An example of a component for interactive sorting is given and liveness and safety properties of the component are proven.

1. Introduction

As shown by numerous examples of abstract data types (see e.g. [WB 89]), many-sorted equational logic is well-suited for describing complex data structures and the functional aspects of a software system. But for modeling the concurrent, state-dependent behavior of systems pure equations seem to be less adequate. Therefore starting with [BW 82] during the last 10 years several proposals have been made for overcoming this problem, among them nondeterministic specifications, streams and the introduction of an additional transition or rewriting relation (for an overview see the excellent paper [AR 93]). Similarly there exist several approaches for describing design and requirements of object-oriented systems (for an overview see [EGS 93]).

In particular, Astesiano's SMoLCS approach ([AMRW 85], [Re 91]) and Meseguer's rewriting logic (cf. e.g. [M 92], [MM 95]) seem to be appropriate for describing both concurrency and object-orientation (at least for the design of object-oriented systems, for requirements some form of modal or temporal logic may be more adequate). In these approaches the static and functional part of a software system is described by classical algebraic specification whereas the dynamic behavior is modeled by an additional non-symmetric relation: a labelled transition relation in case of entity algebras ([Re 91]) and nondeterministic rewriting in case of rewriting logic. In this way dynamic and static aspects of software can be adequately integrated in contrast to pure algebraic or pure process-oriented approaches which have deficiencies in describing either dynamic or static behavior.
Our view on object-oriented design specification is based on these two approaches. Similar to rewriting logic we use a binary predicate for modeling state transitions; but similar to SMoLCS we axiomatize "one step transitions", not their reflexive, transitive and congruent closures as in Meseguer's approach. In our approach these closures are needed only for the semantics and possibly for some special cases. For classical algebraic specifications we use the language SPECTRUM [Broy et al. 93] - an algebraic specification language for requirement and design specifications. It contains explicit support for partial functions and higher order functions. SPECTRUM does not primarily aim at executable specifications; hence axioms are not restricted to equations or conditional equations but arbitrary typed well-formed formulae are allowed. Moreover, SPECTRUM offers a fine grain polymorphism by partitioning the sort universe in so-called sort classes (similar to the type concept in Haskell). The semantics of SPECTRUM is loose, i.e. a SPECTRUM spec-

ification is denoted by the class of all algebras with appropriate signature which fulfill the given axioms.

In this paper we present an object-oriented extension of SPECTRUM. The basic data structures are described by algebraic specifications in SPECTRUM; objects, messages and configurations ("states") are introduced by a particular syntax extending SPECTRUM. The interaction of objects via messages is described by nondeterministic rewrite rules. This extension is similar to and was inspired by Maude, Meseguer's extension of OBJ. However, there are two differences, which are motivated by pragmatic object-oriented software development techniques, in particular Jacobson's OOSE-method (see [J 92]):

The first difference concerns a facility for decribing subsystems. Maude has a global state, called configuration, which contains all currently existing objects and messages; it is not possible to divide this global state in particular subconfigurations (although Meseguer sketches such a possible extension in [M 92, section 4.6]). In our approach subconfigurations are an essential part for structuring the system and reasoning about subsystems.

Another drawback of nondeterministic rewriting as in Maude is the fact that it is difficult to control the order in which messages are executed, i.e. the order in which rewrite rules are applied. This is even worse since for an implementation in an object-oriented programming language messages correspond to and are converted to methods. We overcome this drawback by introducing further operators for composing messages. We present a simple abstract concurrent language containing sequential composition, interleaving and synchronous parallelism as an example for a message definition language. (As illustrated in Section 4 the axiomatisation of these language constructs is one of the reasons for introducing an abstract one step transition instead of full rewriting.) Structured message expressions are also used for formulating safety and liveness properties of design specifications. Proofs of liveness are performed with the help of induction on the form of message expressions whereas for safety properties we use invariant techniques.

Apart from the two extensions of Meseguers approach just described, there is another difference, which is of pure technical nature and stems from the fact that OBJ is a language with subsorting whereas SPECTRUM supports polymorphism and partial functions. Therefore in our approach class inheritance is modeled by partiality and (auxiliary) test predicates instead of the more elegant subsorting of Maude. On the other hand, the axioms for the rewriting predicates are directly specified in SPECTRUM (see section 2) whereas in

Maude concurrent rewriting is specified by a rewriting calculus extending OBJ.

For illustrating our approach a component for interactive sorting is specified in object-oriented SPECTRUM. We present two specifications: first, an abstract component where the sorting requirement is given by the underlying abstract data type and second, a distributed refinement which consists of a hierarchical net of simple sorting cells.

The paper is organized as follows: In section 2 the SPECTRUM specifications for multisets and nondeterministic rewriting are given. In section 3 the syntax for classes is introduced and a first abstract specification of the interactive sorting component is presented. In section 4 the concurrent language for messages is defined. In section 5 liveness and safety properties are studied and the correctness of the sorting component is proven. In section 6 subconfigurations are introduced, a refined version of interactive sorting is specified and its correctness is discussed. Moreover, the paper contains three appendices: Appendix 1 gives the SPECTRUM specifications of the basic data structures for sorting. Appendix 2 gives the formal definitions of the object-oriented extension of SPECTRUM and Appendix 3 contains the correctness proof of the refined version of interactive sorting.

2. Nondeterministic rewriting in SPECTRUM

In this section we present two examples for SPECTRUM specifications which should provide an intuitive understanding of the language; for a comprehensive introduction to SPECTRUM and the exact semantics see [Broy et al. 93].

To support associative commutative term rewriting we use a polymorphic specification of multisets. MSET0 is a loose specification that defines a polymorphic sort constructor "Mset": for each sort s the sort "Mset s" is an abstract sort denoting multisets with elements of sort s. Moreover, MSET0 introduces three standard operations for constructing multisets: the constant ε denoting the empty multiset and the function symbols "{.}" for singleton construction and $\cup$ for multiset union. The three axioms [neutral], [comm], [assoc] ensure that each element of Mset s behaves as a multiset where ε is the neutral element.

All function symbols used in this paper, in particular {.} and .$\cup$. are strict functions, i.e. if one of their arguments is undefined then the result is undefined as well. Moreover, as we will see later both functions are partial. The axiom

Mset α **generated by** ε, {.}, .$\cup$.;

requires that all interpretations of Mset s are term generated with the consequence that Mset s denotes finite multisets.

MSET0 = **{**
sort Mset α;
$\varepsilon : \rightarrow$ Mset α;
$\{.\} : \alpha \rightarrow$ Mset α;
$.\cup. :$ Mset $\alpha \times$ Mset $\alpha \rightarrow$ Mset α;
$\{.\}, .\cup.$ **strict**;
Mset α **generated by** $\varepsilon, \{.\}, .\cup.$;
axioms $\forall$s,s',s": Mset α **in**
[neutral] $\varepsilon \cup s = s$;
[comm] $s \cup s' = s' \cup s$;
[assoc] $s \cup (s' \cup s'') = (s \cup s') \cup s''$;
endaxioms; **}**

The specification REWRITE gives a polymorphic axiomatisation of nondeterministic rewriting. It introduces two polymorphic binary boolean function symbols: "==>" denotes a one-step reduction that describes the application of a rewrite rule; "==>*" is the reflexive and transitive closure of "==>". If its arguments are of type "Mset α" then "==>*" is compatible with multiset union (and subconfigurations, see Section 6). Since multiset union is a partial function (e.g. objects must have pairwise different names, cf. Section 3) the axiom [comp_$\cup$] holds only for defined multiset unions $s \cup s_1$, $s \cup s_2$.

REWRITE = **{ enriches** MSET0;
.==>. : $\alpha \times \alpha$ **to** Bool;
.==>*. : $\alpha \times \alpha$ **to** Bool;
.==>., .==>*. **strict, total**;
axioms $\forall x, x_1, x_2, x_3 : \alpha \ \forall y_1, y_2 : \beta \ \forall s, s_1, s_2 :$ Mset α **in**
[emb] $x_1 \Longrightarrow x_2 \quad \Rightarrow \quad x_1 \Longrightarrow^* x_2$;
[refl] $x \Longrightarrow^* x$;
[trans] $x_1 \Longrightarrow^* x_2 \wedge x_2 \Longrightarrow^* x_3 \Rightarrow x_1 \Longrightarrow^* x_3$;
[comp_$\cup$] $s_1 \Longrightarrow^* s_2 \wedge \delta(s \cup s_1) \wedge \delta(s \cup s_2)$
$\Rightarrow \quad (s \cup s_1) \Longrightarrow^* (s \cup s_2)$;
endaxioms; **}**

Please note the difference between the rewrite predicate symbols ("==>" and "==>*") and the logical connective "⇒". In order to avoid confusion between these symbols we often write
"t1 ==> t2 **if** cond" instead of "cond ⇒ (t1 ==> t2)".

3. Object-oriented SPECTRUM

The object-oriented extension of SPECTRUM consists of the introduction of a notation for the declaration of classes and subconfigurations.
An (*object*) *class* is declared by an identifier and three possibly empty lists of associated superclasses, attributes and messages. CId is the sort reserved for all class identifiers.
An *attribute* is declared by its name and its sort. Each instance is a pair (a : v) consisting of an identifier a for the name and a value v of appropriate sort. AId is the sort reserved for all attribute identifiers.
A *message* declaration has the form $m(s_1,...,s_p)$ where m is a function symbol with range Msg and parameters of sorts $s_1,...,s_p$. Here m also denotes the name of the message. Formally, a class C is introduced by a declaration of the following form (for the translation to pure SPECTRUM see Appendix 2.2):

> **class** C
> > **subclass of** $D_1,..., D_k$;
> > **attr** $a_1 : s_1,..., a_n: s_n$;
> > **msg** $m_1(s_{11},...,s_{1p_1}),...,m_r(s_{r1},...,s_{rp_r})$;

An *object* is represented by a term of the form

<o : C | atts>

where o is an object identifier of the reserved sort OId for object identifiers, C a class identifier and atts is a list of attributes. A term <o : C | atts> is defined only if atts contains all attributes of C including the attributes of its superclasses and if the attribute identifiers of atts are pairwise different. Thus here we make use of the partiality of multiset union.
A *class specification* is a SPECTRUM specification that contains a class declaration.
Computational progress is made by rewriting the state, the so-called *configuration.* A configuration is a multiset of objects and messages (such that all occurring object identifiers are pairwise different).

A *rewrite step* transforms a configuration into a subsequent configuration. Usually rewrite steps are specified via axioms of the form "cond $\Rightarrow$ (c_1 ==> c_2)" where "cond" is a formula, c_1, c_2 are terms of sort configuration and ==> is the one-step rewrite relation.

The creation of a new object of class C is a standard example for a rewrite step. By the declaration

new_C($atts_0$)

we introduce a message "new_C" which for any configuration c creates a new object <o' : C | $atts_0$> with a new object identifier o'. (For a formal specification see Appendix 2.2; the formal specification CONFIGURATION of the data structures for attributes, objects and configurations is given in Appendix 2.1.)

In the following we give an example for a class specification. A sorting component can receive messages of the form "receive(x)" and "request". If x is a natural number then it is inserted in the component, if it is a request then the component outputs the sorted list of its stored elements.

The class specification ABSTRACT_SORT defines a class Sort whose objects are sorting incoming values. The class has no superclasses and just one attribute "sortedlist" that contains a list of elements of a totally ordered sort N. An object o of this class can receive messages of the form "receive(x)" and "request". The value x is inserted in the "sorted list" l of o. If o receives a request message then o sends a message "send(l)" with the content l of "sortedlist". The crucial operation for the behavior of o is "insert"; it ensures that incoming elements are correctly inserted in l .

The sorts and operations relevant for sorting are specified in LIST_SORT which can be found in Appendix 1. In particular, LIST_SORT defines polymorphic lists where nil denotes the empty list, x+l adds the element x to the list l, l++l' concatenates two lists l and l'. Moreover, LIST_SORT introduces the following sorting functions: insert(x,l) inserts the element x in the sorted list l at the correct place, issorted(l) tests whether l is sorted and sort(l) yields a sorted permutation of l as result.

ABSTRACT_SORT = {
    enriches LIST_SORT + CONFIGURATION;
    sort N :: TO;
    class Sort
        attr sortedlist: List N;

msg receive(N),
request,
send (List N);

axioms ∀o:OId, n:N, l:List N **in**

[rcv] receive(n) <o : Sort | sortedlist : l> ==>
<o : Sort | sortedlist : insert(n, l)>;

[rqu] request <o : Sort | sortedlist : l> ==>
<o : Sort | sortedlist : nil> send(l);

endaxioms}

Note that for simplicity we have specified the messages “receive(n)”, “request” and “send(l)” without indicating the name of the object the message is addressed to.

By analyzing the axioms [rcv] and [rqu] one can see that an instance of class Sort sends on request the sorted list of all those elements it has received before. Such a sequential situation is difficult to specify since in our framework an object reacts to a single message not to a stream of messages. In order to overcome this problem we define in the next section a further extension.

4. Sequential and concurrent computation of messages

In a pure nondeterministic rewriting system it is difficult to control the (sequential) flow of messages. Such a control can be achieved by providing operators for composing messages.

In a first step we introduce message expressions of form $m_1;m_2$ for the sequential composition of two messages m_1 and m_2. The corresponding axiom [;] has to take into consideration that the one step rewriting relation ==> is applied to a subpart, (c_1 c_2) say, of the whole configuration c = (($m_1;m_2$) c_1 c_2 c’). In particular, m_1 can be fired with c_1 yielding a subconfiguration (d_1 h). By [emb] and [comp_∪] this leads to the ==>*-transition (m_1 c_1 c_2) ==>* (d_1 h c_2). Then m_2 is applied to the subconfiguration (h c_2) yielding a subconfiguration d_2 which by [emb] and [comp_∪] leads to the ==>*-transition (d_1 m_2 h c_2) ==>* (d_1 d_2). We interpret these two steps together as one step of the composite message ($m_1;m_2$) and get therefore the transition ($m_1;m_2$) c_1 c_2 ==> (d_1 d_2).

SEQUENTIAL_MSG = {

enriches CONFIGURATION

$.;. : Msg \times Msg \to Msg;$ -- sequential composition

.;. **strict, total**;

axioms $\forall m_1,m_2$:Msg $\forall c_1,c_2,d_1,d_2,h$:Configuration **in**

[;] $(m_1;m_2)\ c_1\ c_2$ **==>** $d_1\ d_2$

if $m_1\ c_1$ **==>** $d_1\ h \wedge m_2\ h\ c_2$ **==>** d_2 ;

endaxioms;}

Note that the use of ==> ensures that the messages m_1 and m_2 are really involved in the rewriting. If we would use ==>* instead of ==> we could rewrite e.g. the configuration $(m_1\ c_1)$ by a rule that matches a subconfiguration of c_1 which has nothing to do with the message m_1.

In the following we distinguish between *atomic message expressions* which do not contain any message composition operator (such as ";") and *composite messages*. We make the general assumption (*) that the specifications of rewrite steps in a class specification SP have the form

"cond $\Rightarrow$ (c_1 ==> c_2)" (*)

where c_1, c_2 are terms of sort configuration of form (m_1 ... m_k <o_1:C_1| $atts_1$> ... <o_n:C_n| $atts_n$>), m_1 ... m_k are *atomic* message expressions and "cond" is a formula.

Fact 1

Sequential composition satisfies the following associativity property. Let SP be a class specification satisfying the assumption (*). Let m_1, m_2, m_3, c, d be ground terms of sort Msg and Configuration:

SP + SEQUENTIAL_MSG |= $((m_1;m_2);m_3)$ c ==> d iff

SP + SEQUENTIAL_MSG |= $(m_1;(m_2;m_3))$ c ==> d.

Proof "$\Rightarrow$" Let SEQUENTIAL_MSG |= $((m_1;m_2);m_3)$ c ==> d .

Because of the special form of the rules of SP only axiom [;] can be used for proving $((m_1;m_2);m_3)$ c ==> d. Thus there exist c1,c2,c3 and d1,d2,d3 and h1, h2, h3 such that c = (c1 c2 c3), d = (d1 d2 d3) and

(m1;m2) c1 c2 ==> (d1 d2) (h2 h3) and m3 (h2 h3) c3 ==> d3

with m1 c1 ==> d1 h1 h3 and m2 h1 c2 ==> d2 h2.

Thus

m1 c1 ==> (d1 (h1 h3)) and (m2;m3) h1 h3 c2 c3 ==> d2 d3 implies
(m1;(m2;m3)) c1 c2 c3 ==> d1 d2 d3.
"⇐"Analogous to "⇒".
◊**Proof Fact 1**

Moreover, we introduce constructor symbols for interleaving parallel composition (||), synchronous parallel composition (|||), nondeterministic choice (or) and repetition (do).
Thus we distinguish between interleaving and true parallelism. Interleaving parallelism means that the messages are fired sequentially in an arbitrary order. Hence if m_1 or m_2 is atomic the behavior of (m_1|| m_2) can be defined using the behavior of (m_1;m_2) and (m_2;m_1) (see the axioms [||1], [||2]). For non-atomic messages either the first message ofm_1 (see axiom [;||]) or the first message of m_2 (see axiom [||;]) is performed firstSynchronous "true" parallelism requires that both messages are applied simultaneously (see axiom [|||]).

PARALLEL_MSG = { **enriches** SEQUENTIAL_MSG;
.||. , .|||. : Msg × Msg → Msg; -- parallel composition
or : Msg × Msg → Msg; -- non-deterministic choice
do : Msg → Msg; -- repetition
.||., .|||., or, do **strict**, **total**;
axioms ∀m,m_1,m_2: Msg ∀c,d: Configuration **in**
[||1] (m_1;m_2) c ==> d ⇒ (m_1|| m_2) c ==> d;
[||2] (m_2;m_1) c ==> d ⇒ (m_1|| m_2) c ==> d;
[;||] (m_1;(m_2||m_3)) c ==> d ⇒ ((m_1;m_2)|| m_3) c ==> d;
[||;] (m_2;(m_1||m_3)) c ==> d ⇒ (m_1||(m_2; m_3)) c ==> d;
[|||] m_1 c_1 c_2 ==> d_1 c_2 ∧ m_2 c_2 c_3 ==> c_2 d_2 ⇒
(m_1||| m_2) c_1 c_2 c_3 ==> d_1 c_2 d_2;
[or1] m_1 c ==> d ⇒ (m_1 or m_2) c ==> d;
[or2] m_2 c ==> d ⇒ (m_1 or m_2) c ==> d;
[or||] (m_1||m_3) c ==> d ∨ (m_2||m_3) c ==> d ⇒
((m_1 or m_2) || m_3) c ==> d ∧
(m_3|| (m_1 or m_2)) c ==> d
[do1] (do m) c = (m;do m) c;
[do2] (do m) ==> ε; **endaxioms; }**

The “do” operator satisfies the following property.

Fact 2
Let SP be a class specification satisfying assumption (*). Let m,c,d be ground terms of sort Msg and Configuration and (m;...;m) the n-fold sequential composition of m with $n \geq 1$.

If SP + PARALLEL_MSG |= (m;...;m) c ==> d then
SP + PARALLEL_MSG |= (do m) c ==> d.

Proof by natural induction.

"n=1”
By the assumption we have m c ==> d and by [do2] we have (do m) ==> ε . Thus by [;] we have (m;(do m)) c ==> d and hence by axiom [do1] it follows that (do m) c ==> d.

“n⇒n+1”
Let for the (n+1)-fold composition (m;...;m) c ==> d hold. Then by [;]
c = (c_1 c_2), d =(d_1 d_2), m c_1 ==> d_1 h and the n-fold composition
(m;...;m) h c_2 ==> d_2 for appropriate c_1, c_2, d_1, d_2, h.
By induction hypothesis we get (do m) h c_2 ==> d_2. Thus by [;] we have the desired result

(m; do m) c_1 c_2 ==> d_1 d_2. ◊**Proof Fact 2**

5. Liveness and safety properties

In this section we use message expressions to formulate and prove correctness properties of class specifications. As usual in systems with nondeterministic behavior we distinguish between liveness and safety properties. A liveness property ensures that a certain behavior is possible whereas a safety property is guaranteed to be satisfied by all (reachable) configurations.

In the case of class specifications we study the dynamic behavior of configurations of the form (m c) where m is a (composite) message expression. For liveness we prove that a certain desired configuration is reachable. If the composite message m contains parallel composition or non-deterministic choice, then, in general, there will be a number of different configurations which are reachable from the configuration (m c) in a one-step transition. In order to keep control over these different configurations we prove safety properties

which hold for all configurations which are reachable from (m c). The proof of a liveness property proceeds by induction on the form of the message expression m; for the proof of the safety properties we use the invariant technique explained below.
As an example we prove liveness and safety properties for ABSTRACT_-SORT.

5.1 Liveness properties of ABSTRACT_SORT

In this subsection we show the following liveness property of ABSTRACT_SORT: if an object of class Sort with empty store receives a sequence of natural numbers followed by a request then it reaches a configuration (by a one-step transition) that outputs these numbers as a sorted list. Here it does not matter whether the numbers arrive sequentially or interleaving in parallel. More formally, we will prove that the configuration

$$\langle o : Sort \mid sortedlist : nil\rangle \quad send(sort(n_1 ... n_k))$$

is reachable from the configurations

$$(receive(n_1);...;receive(n_k);request) \quad \langle o : Sort \mid sortedlist : nil\rangle$$

and

$$((receive(n_1) \| ... \| receive(n_k));request) \ \langle o : Sort \mid sortedlist : nil\rangle$$

by a one-step transition.

This property follows from the following slightly more general Lemma:

Lemma 1
ABSTRACT_SORT + PARALLEL_MESSAGE satisfies the following property:
$\forall n_1,...,n_k$:Nat.

$$issorted(l) \Rightarrow$$
$$(receive(n_1);...;receive(n_k);request) \ \langle o : Sort \mid sortedlist : l\rangle \ ==>$$
$$\langle o : Sort \mid sortedlist : nil\rangle \ send(sort(n_1 ... n_k{+}{+}l)).$$

Proof by induction on k:
k=0:
The configuration request <o : Sort | sortedlist : l> rewrites by [rqu] to <o : Sort | sortedlist : nil> send(l). Because of issorted(l) this is equal to <o : Sort | sortedlist : nil> send(sort(l)).
k⇒k+1:
By [rcv] we have

(1) receive(n_1) <o : Sort | sortedlist : l> ==> <o : Sort | sortedlist : insert(n_1,l)>.

By induction hypothesis we get

(2) (receive(n_2);...;receive(n_k);request) <o : Sort | sortedlist : insert(n_1,l)> ==>

<o : Sort | sortedlist : nil> send(sort(n_2...n_k++insert(n_1,l))). Obviously this is equal to

<o : Sort | sortedlist : nil> send(sort($n_1 n_2$...n_k++l)).

Hence by [;] we get from (1) and (2) the desired result:

(receive(n_1);...;receive(n_k);request) <o : Sort | sortedlist : l> ==>

<o : Sort | sortedlist : nil> send(sort($n_1 n_2$...n_k++l)).

◊**Proof Lemma 1**

As a Corollary we obtain the desired lifeness property:

Corollary 1

ABSTRACT_SORT + PARALLEL_MESSAGE satisfies the following properties:

1)$\forall n_1,...,n_k$:Nat.

(receive(n_1);...;receive(n_k);request) <o : Sort | sortedlist : nil> ==>

<o : Sort | sortedlist : nil> send(sort(n_1...n_k)).

2)$\forall n_1,...,n_k$:Nat.

((receive(n_1)||...||receive(n_k));request) <o : Sort | sortedlist : nil> ==>

<o : Sort | sortedlist : nil> send(sort(n_1...n_k)).

Proof : 1) Apply Lemma 1 with l=nil.

2)Let π be an arbitrary permutation of {1,...,k}. By 1) we have

(receive($n_{\pi(1)}$);...;receive($n_{\pi(k)}$);request) <o : Sort | sortedlist : nil> ==>

<o : Sort | sortedlist : nil> send(sort(n_1...n_k)).

Thus by the axioms [||1] and [||2] the required property 2) holds.

◊**Proof Corollary 1**

5.2 Proving safety properties

A well known technique for proving safety properties is the *invariant technique*: A predicate on configurations is defined which is shown to be invariant under all one-step transitions effected by (possibly) complex messages. In the following definition, we require certain properties of a predicate on configurations specified over a class specification SP, which guarantee that the predicate is invariant under all transitions effected by message expressions.

Here, as in Fact 1, we again make the general assumption (*) on the form of the axioms for rewrite steps of SP. By an instance of an axiom for a rewrite step we understand a formula obtained by applying an arbitrary ground term-substitution to the free variables in this axiom.

Definition 1

Let SP be a class specification as above and I : Configuration **to** Bool be a predicate which is well-defined over SP. Let SPI denote the extension of SP by the specification of I. Then I is called a *configuration invariant w.r.t. SP* iff the following holds:

(1) For every instance cond $\Rightarrow$ (m c ==> c') of an axiom for a rewrite step (where by assumption (*) cond is a formula, m is an atomic message and c a configuration term *not* containing a configuration variable) it holds

$$\text{If SP} \models \text{cond} \quad \text{then} \quad \text{SPI} \models I(c) \Rightarrow I(c')$$

(2) I is subset-closed, i.e.

$$\text{SPI} \models I(c) \wedge c = (c_1\ c_2) \Rightarrow I(c_1)$$

(3) I is compatible with multiset-union, i.e.

$$\text{SPI} \models I(c_1) \wedge I(c_2) \wedge \delta(c_1\ c_2) \Rightarrow I(c_1\ c_2)$$

◊**Definition 1**

From (2) and (3) it follows easily that for all configurations $c_1, c_2, ..., c_n$ with $\delta(c_1\ c_2 ...\ c_n)$ it holds

$$I(c_1\ c_2 ...\ c_n) = I(c_1) \wedge I(c_2) \wedge ... \wedge I(c_n).$$

Hence I is determined by its value on single objects and messages. Therefore no relations between different objects or messages can be formulated. More general invariants can be considered in connection with subconfigurations in Section 6.

The following Lemma states that configuration invariants are invariant under all transitions effected by message expressions:

Lemma 2

Let I be a configuration invariant w.r.t. SP. Then for every message expression m and every ground term c of sort Configuration built over the signature of SP it holds

$$\text{If SP + PARALLEL\_MESSAGE} \models (m\ c \ ==>\ c') \text{ then}$$
$$\text{SPI} \models (I(c) \Rightarrow I(c'))$$

Proof: The proof goes by induction on the structure of m. We assume that m c ==> c' can be proved and I(c) holds and prove that c' does not contain composite messages and I(c') holds.

(0) m is an atomic message. Then m c ==> c' cannot be deduced by the rules of PARALLEL_MESSAGE, since c does not contain composite messages (as it is a configuration over SP). Hence m c ==> c' is an instance of the conclusion of an axiom in SP and hence by condition (1) in Definition 1, I(c') holds. Since the axioms in SP do not contain composite messages, c' does not contain composite messages.

(1) m = m_1;m_2. Since SP satisfies assumption (*), (m_1;m_2) c ==> c' can only be proved using the axiom [;] or the equation [do1]. In the case of [;] there exists c_1, c_2, d_1, d_2, h such that

c = c_1 c_2, c' = d_1 d_2 and m_1 c_1 ==> d_1 h, m_2 h c_2 ==> d_2. From I(c1 c2) it follows by Def. 1, condition (2) that I(c_1) and I(c_2). Hence by induction hypothesis it follows that I(d_1 h) and d_1, h do not contain composite messages. Therefore I(d_1) and I(h) hold, and hence by condition (3) in Def. 1 also I(h c_2). Hence again by induction hypothesis we have I(d_2) and d_2 does not contain composite messages. With condition (3) in Def. 1 it now follows that I(c') = I(d_1 d_2) holds.

If (m_1;m_2) c ==> c' can be proved using [do1], then (m_1;m_2) = (m; do m) for some message m. Hence (m_1;m_2) c = (m; do m) c = (do m) c and the desired result follows by the induction hypothesis for (do m). This finishes the proof of **case (1).**

(2) m = do m_1. Here the proof procedes by induction on the number of the applications of the axiom [;] in a proof of(do m_1) c ==> c'. More precisely we prove by induction on n $\in \mathbb{N}$ that for all ground terms c of sort Configuration built only over the signature of SP, for all ground terms c' of sort Configuration and all proofs p of (do m_1) c ==> c' the following holds:

If the number of applications of the axiom [;] in p is n, then I(c) $\Rightarrow$ I(c') holds.

n = 0: Since the axioms of SP satisfy assumption (*), and since c does not contain composite message expressions, the proof p can only consist in an instance of axiom [do2]. Hence c' = ε and hence (I(c) $\Rightarrow$ I(c')) holds.

n > 0: Let us consider the "top-most" application of [;] in the proof p. Since c does not contain composite message expressions and by assumption (*) there exist m_1', m_2 such that (do m_1) = m_1'; m_2 is proved in p using the axiom

[do1], where $m_1' = m_1$ and $m_2 = (\text{do } m_1)$, and $(m_1'; m_2)\ c \implies c'$ is proved in p by application of [;] (modulo some applications of the multiset laws to c). Hence there exist c_1, c_2, d_1, d_2, h such that $c = c_1\ c_2$, $c' = d_1\ d_2$ and $m_1'\ c_1 \implies d_1\ h$, $m_2\ h\ c_2 \implies d_2$ is proved in p. Therefore $m_1\ c_1 \implies d_1\ h$ holds, since $m_1' = m_1$. Since the (structural) induction hypothesis applies to m_1 it follows that $I(c_1) \Rightarrow I(d_1\ h)$ holds. Moreover, since the (sub) proof of $m_2\ h\ c_2 \implies d_2$ in p involves less than n applications of [;] the (inner) induction hypothesis applies to this proof. Hence $I(h\ c_2) \Rightarrow I(d_2)$ holds. Hence as in the proof of the first case it follows that $I(c) \Rightarrow I(c')$. (Also the proof that c' does not contain composite messages can be led as in the first case). This finishes the proof of **case (2).** Since the **other cases** can be treated very similarly, we omit them here. ◊**Proof Lemma 2**

In the case of the class specification ABSTRACT_SORT it can be shown easily, that the predicate I_sort is a configuration invariant, where I_sort is specified by

$$
\begin{aligned}
&\text{I\_sort}(c) = (\forall\ l\text{:List Nat}.\\
&\qquad (\exists\ c', o.\ \ c = c' < o : \text{Sort} \mid \text{sortedlist} : l > \vee\ c = c'\ \text{send}(l)\ )\\
&\qquad\qquad \Rightarrow \text{issorted}(l)\ )
\end{aligned}
$$

Hence from Lemma 2 the following safety property follows for ABSTRACT_SORT:

Corollary 2

For every message expression m built over the messages in ABSTRACT_-SORT and all ground terms $l, l_1,...,l_n$ of sort List Nat ((where n ranges over arbitrary n>0) the following holds

If ABSTRACT_SORT + PARALLEL_MESSAGE |=
m <o : Sort | sortedlist : nil>==> <o : Sort | sortedlist :l>$\text{send}(l_1)...\text{send}(l_n)$
then ABSTRACT_SORT |= $\text{issorted}(l) \wedge \text{issorted}(l_1) \wedge ...\ \text{issorted}(l_n)$.

◊ **Corollary 2**

Hence, in particular, only sorted lists are output. Another safety property, that has to be considered for a sorting object, is, that if m<o : Sort | sortedlist : nil>

rewrites to a configuration c, then the multiset of incoming elements in m equals the (multiset) union of the elements currently stored in objects in c and the elements already sent out. This safety property can be proven, if we restrict ourselves to such message expressions m which do not contain the do-operator and the non-deterministic choice (since it is not possible to decide what are the incoming elements in (m_1 or m_2) and in (do m)). The details of this proof are omitted here.

6. Subconfigurations and interactive sorting with sort cells

Configurations can contain all defined objects and messages. Often it is useful to restrict the set of admissible classes. This can be achieved with the notion of a subconfiguration. A *subconfiguration* is a specific kind of configuration which admits only objects and messages from a well-specified set of classes. We write

configuration SubConf **of** $D_1,..., D_k$;

for declaring a sort SubConf whose elements are configurations consisting of objects and messages of the classes $D_1,..., D_k$. For a formal definition of the translation to SPECTRUM see Appendix 2.3.

As an example we give another specification of interactive sorting. For implementing the sorting algorithm we introduce a class of objects, called SortCell, with two attributes: “content” stores a single element whereas “conf” has a subconfiguration as value. If a sorting cell receives an element it compares it with the element in its store. Then it stores the smaller number and sends the larger one to its subconfiguration. If it receives a request then it outputs its store and passes the request message to its subconfiguration.

SORT_CELL = **{**
  enriches LIST_SORT;
  class SortCell
    attr content : N ∪ {∞},
      conf : SortConf;
    msg receive(N),
      request,
      send(List N);
  new_SortCell(content : ∞, conf : ε);
  configuration SortConf **of** SortCell;

```
axioms ∀ o, o1: OId ∀c, c1: SortConf ∀m, n:N  in
[∞]        m ≤ ∞;
[rcv ∞]    receive(n) <o : SortCell | content : ∞, conf : c> ==>
                  <o : SortCell | content : n, conf : c1>
     if (conf: new_SortCell)   ==> (conf: c1);
[rcv m]    receive(n) <o : SortCell | content : m, conf : c> ==>
                  <o : SortCell | content : min(n,m), conf : c1>
     if receive(max(n,m)) c ==> c1;
[rqu ∞]    request <o : SortCell | content : ∞, conf : ε> ==>
                  <o : SortCell | content :  ∞, conf : ε> send(nil) ;
[rqu m]    request <o : SortCell | content : m,   conf : <o1 : SortCell>> ==>
                  <o : SortCell | content :  ∞, conf : ε> send(m+l)
     if request <o1 :  SortCell>==> <o1 : SortCell |  content :∞, conf : c> send(l).
endaxioms;}
```

Note that the condition in the axiom [rcv∞] could easily be eliminated by replacing c1 for instance by the configuration <o: SortCell | content :∞, conf : ε>. The above specification is more abstract, in that it does not specify the choice of a particular object identifier for the new sort cell (compare the specification of new_C in appendix 2.3).

To show the relationship with ABSTRACT_SORT we extend SORT_CELL by two auxiliary functions: the function “min” computes the minimal element of a list; “elemlist” computes the list of elements stored in a sort cell. It has the functionality

```
elemlist : SortConf ⟶ List N
```

and is partially axiomatised by the following equations:

```
elemlist(<o : SortCell | content : ∞, conf : ε>) = nil
elemlist(<o : SortCell | content : n, conf : c>) = n+elemlist(c).
```

Thus the specification SORT_CELL1 has the following form:

```
SORT_CELL1 = {
     enriches SORT_CELL;
     min : List N⟶ N;
     elemlist : SortConf ⟶ List N;
```

axioms ∀o: OId ∀c: SortConf ∀n: N ∀l: List N **in**

min(ε) = ∞;

min(n+l) = min(n, min(l));

elemlist(ε) = nil;

elemlist(<o : SortCell | content : ∞, conf : ε>) = nil;

elemlist(<o : SortCell | content : n, conf : c>) = n+elemlist(c);

endaxioms;}

Each instance <o : Sort | sortedlist : nil> is represented by a term of the form <o : SortCell | content : ∞, conf : ε>.

If x+l is sorted then each instance <o : Sort | sortedlist : x+l> is represented by a term of the form <o : SortCell | content : x, conf : <o' : SortCell | ATTS>> with elemlist (<o' : SortCell | ATTS>) = l .

Then we can show three liveness properties of sort cells that explain the dynamic behavior: if a sort cell sc with wellformed "element list" receives a request then it outputs its element list; moreover, if its element list is sorted and sc receives an element then the resulting element list will be sorted again. This implies that if sc receives a sequence of elements followed by a request then it returns a a sorted permutation of the received elements and its element list.

Lemma 3

SORT_CELL1 satisfies the following properties for all ground terms l such that SORT_CELL1 |= l = elemlist(<o : SortCell | ATTS>):

0) ATTS is an element of the set A which can be inductively defined as follows:

1) (content : ∞, conf : ε) ∈ A,

2) (content : n, conf : <o1 : SortCell | ATTS1>) where ATTS1∈ A, n∈ N.

1) request <o : SortCell | ATTS> ==> <o : SortCell | content : ∞, conf : ε> send(l);

2) ∀n:N. issorted(l) ⇒

receive(n) <o : SortCell | ATTS> ==> <o : SortCell | ATTS1>

such that SORT_CELL1 |– elemlist(<o : SortCell | ATTS1>) = insert(n, l) ;

3)∀n_1,...,n_k:N. issorted(l) ⇒

(receive(n_1);...;receive(n_k);request) <o : SortCell | ATTS> ==>

<o : SortCell | content : ∞, conf : ε> send(sort(n_1...n_k++l)).

The **Proof** is given in Appendix 3.

Analyzing the relationship of ABSTRACT_SORT and SORTCELL1 we see that SORTCELL1 is not a refinement of ABSTRACT_SORT since property (2) above is constrained by issorted(l) whereas the corresponding axiom [rcv] of ABSTRACT_SORT holds for all lists. In fact, if the list l is not sorted then SORTCELL1 and ABSTRACT_SORT behave differently: let l = 1+4+3+nil; then rewriting the configuration "receive(2) <o : Sort | sortedlist : l>" in ABSTRACT_SORT yields "<o : Sort | sortedlist : 1+2*4*+*3*+nil>" whereas the corresponding "elemlist" in SORTCELL1 is 1+2*3*+*4*+nil (since by applying [rcv m] recursively the elements 3 and 4 are permuted).
But SORTCELL has the same sorting property as ABSTRACT_SORT:

Corollary 3
SORT_CELL satisfies the following properties:

1) $\forall n_1,...,n_k$:Nat.

(receive(n_1);...;receive(n_k);request) <o : SortCell | content :∞, conf : ε> ==>
<o : SortCell | content : ∞, conf : ε> send(sort($n_1...n_k$)).

2) $\forall n_1,...,n_k$:Nat.

((receive(n_1)||...||receive(n_k));request) <o : SortCell | content :∞, conf : ε> ==>
<o : SortCell | content : ∞, conf : ε> send(sort($n_1...n_k$)).

Proof. 1) Apply Lemma 3.3 with l=nil. Since SORT_CELL1 is a persistent extension of SORT_CELL and the property does not use the auxiliary function elemlist the property also holds in SORTCELL.

2) follows from 1) as in Corollary 2. ◊**Proof Corollary 3**

7. Concluding Remarks

The extension of Meseguer's rewriting logic by the introduction of subconfigurations and complex message expressions as shown in this paper, gives a new basis for a formal foundation of current object-oriented software development methods. It has already been used in a course given by the first author [HW 94] on methods of software development. In this course Jacobson's method OOSE [J 92] for object-oriented software development was presented. The techniques of this paper have been used to formalize informal diagrams such as interaction diagrams and state transition diagrams. Therefore,

different informal diagrammatic representations can be integrated into one semantic framework.
Clearly, the results of this paper are only a first step in the formalization of object-oriented development methods. We are currently working on extending the message language by guards and recursive method declarations and on including timing constraints in our framework. Also the techniques for verification and refinement deserve further study (cf. also [LLW 95]).

Acknowledgment This work has been partially sponsored by the DFG project Osidris, contract Wi 841/3-1, and the ESPRIT working group COMPASS.

References

[Note: LNSC n = Springer Lecture Notes in Computer Science, Volume n]

[AR 93] E. Astesiano, G. Reggio: Algebraic specification of concurrency. In: [BC 93], 1-39.

[AMRW 85] E. Astesiano, G. F. Mascari, G. Reggio, M. Wirsing: On the parameterized algebraic specification of concurrent systems. In: H. Ehrig, C. Floyd, M. Nivat, J. Thatcher (eds.): Mathematical Foundations of Software Development, TAPSOFT. LNCS 185. 1985, 342-358.

[BC 93] M. Bidoit, C. Choppy (eds.): Recent Trends in Data Type Specification, LNCS 655, 1993.

[Broy et al. 93] M. Broy, C. Facchi, R. Grosu, R. Hettler, H. Hußmann, D. Nazareth, F. Regensburger, O. Slotosch, K. Stolen: The requirement and design specification language SPECTRUM: An informal introduction, version 1.0, part I. TU München, Tech. Rep. TUM-I9311, 1993.

[BW 82] M. Broy, M. Wirsing: On the algebraic specification of finitary infinite communicating sequential processes. In: D. Bjørner (ed.): TC-2 Working Conference on Formal Description of Programming Concepts II, Garmisch-Partenkirchen, 1982. - North-Holland. 1982, 171-197.

[EGS 93] H. Ehrig, M. Gogolla, A. Sernadas: Objects and their specification. In: [BC 93], 40-65.

[HW 94] R. Haggenmüller, M. Wirsing: Methoden der Software-Entwicklung. Vorlesungsausarbeitung, Institut für Informatik, Universität München, 1994.

[J 92] I. Jacobson: Object-Oriented Software Engineering. Wokingham: Addison-Wesley, 1992.

[LLW 95] U. Lechner, C. Lengauer, M.Wirsing: An object-oriented airport. To appear in: Proc. 10th ADT Workshop, San Miniato, May 1994, LNCS.
[M 92] J. Meseguer: A logical theory of concurrent objects and its realization in the Maude language. Technical Rep. SRI-CSL-9208, SRI International, July 1992.
[MM 95] N. Marti-Oliet, J. Meseguer: From abstract data types to logical frameworks. To appear in: Proc. 10th ADT Workshop, San Miniato, May 1994, LNCS.
[Re 91] G. Reggio: Entities: an institution for dynamic systems. In H. Ehrig, K.P. Jantke, F. Orejas and H. Reichel, editors, Recent Trends in Data Type Specification, LNCS 534, pp. 244-265, Springer, 1991.
[WB 89] M. Wirsing, J.A. Bergstra (eds.): Algebraic methods: theory, tools and applications. LNCS 394, 1989.

Appendix 1 Basic SPECTRUM specifications for sorting

The following specification PARTIAL_ORDER defines a sort class PO where each sort in this class comes along with a boolean function ≤ that satisfies the laws of a partial order relation; PO itself is a subclass of the class EQ that provides a boolean equality symbol == .

PARTIAL_ORDER =
    {enriches EQUALITY;
    class PO **subclass of** EQ;
    ≤: α :: PO $\Rightarrow \alpha \times \alpha \rightarrow$ Bool;
    axioms α :: PO $\Rightarrow \forall$x,y,z:α **in**
    [refl] $x \leq x$;
    [trans] $x \leq y \wedge y \leq z \Rightarrow x \leq z$;
    [ant] $x \leq y \wedge y \leq x \Rightarrow x == y$;
    endaxioms;**}**

The specification ORDERING of total orders is obtained from PARTIAL_ORDER by imposing the additional law of a total ordering on ≤ :

ORDERING =
    {enriches PARTIAL_ORDER;
    class TO **subclass of** PO;
    axioms α :: TO $\Rightarrow \forall$x,y,z:α **in**
    [total] $x \leq y \vee y \leq x$;
    endaxioms;**}**

The specification ORDERING1 provides polymorphic functions for computing the minimum and maximum of two values.

ORDERING1 =
    {enriches ORDERING;
    min,max: α :: TO $\Rightarrow \alpha \times \alpha \rightarrow \alpha$;
    axioms α :: TO $\Rightarrow \forall$x,y:α **in**
        min(x,y) = **if** $x \leq y$ **then** x **else** y **fi**;
        max(x,y) = **if** $x \leq y$ **then** y **else** x **fi**;
    endaxioms;**}**

The specification LIST_TO defines polymorphic lists where nil denotes the empty list, x+l adds the element x to the list l, l++l' concatenates two lists l and l'. The selector functions for the constructor "+" are first and rest.

```
LIST_TO =
    { enriches ORDERING1;
    data α :: TO ⇒List α = nil | .+.(!first:α, !rest: List α);
    List ::(TO)EQ;
    .++.: α :: TO ⇒ List α × List α → List α;
    .++. strict total;
    axioms α :: TO ⇒ ∀l,l':List α ∀x:α in
        nil ++l = l;
        (x +l') ++l = x +(l'++l);
    endaxioms;}
```

The specification LIST_SORT is a requirement specification for sorting functions: insert(x,l) inserts the element x in the sorted list l at the correct place, issorted(l) tests whether l is sorted and sort(l) yields a sorted permutation of l as result; .©. is an auxiliary filter function.

```
LIST_SORT =
    { enriches LIST_TO ;
    .©., insert : α :: TO ⇒ α × List α → List α;
    sort: α :: TO ⇒ List α → List α;
    issorted: α :: TO ⇒ List α → Bool;
    axioms α :: TO ⇒ ∀x,y:α, l,l1,l2:List α in
        x©nil = nil;
        x©(y+l) = if x==y then x+(x©l) else x©l fi;
        insert(x,nil) = x+nil;
        insert(x,y+l) = if x≤y then x+(y+l) else y+insert(x,l) fi;
        x©sort(l ) = x©l;
        sort l = l1 ++(x+y+l2) ⇒ x≤y;
        issorted l = (l == sort l);
    endaxioms;}
```

Appendix 2: The specification of CONFIGURATION

In the following we give exact definitions of the specifications of attributes, objects and configurations and the extensions of the SPECTRUM syntax for declaring classes and subconfigurations.

Appendix 2.1 Specifications of objects and configurations

The object-oriented extension of SPECTRUM is based on specifications of the data structures for objects and configurations. Every object is denoted by a term of the form

$<o : C \mid a_1 : v_1, ..., a_n : v_n>$ (cf. the notational convention at the end of this section) where o is an object identifier, C a class identifier and the pairs $a_i : v_i$ denote attributes consisting of an attribute identifier a_i and a value v_i.

In the specification ID, the only function symbols required for identifiers are an equality test and a partial order relation: the sorts AId for attribute identifiers and OId for object identifiers just need an equality test that is provided by the predefined SPECTRUM sort class EQ.
The sort CId for class identifiers also needs a partial order relation for checking the subclass relationship.
In order to be able to generate new object identifiers, we consider an injective dot-operator on object identifiers, which allows to suffix an object identifier by a natural number.

ID = { **enriches** PARTIAL_ORDER;
  sorts AId,CId,OId;
  AId::EQ;
  CId::PO;
  OId::EQ;
  . dot . : OId $\times$ Nat $\rightarrow$ OId;
  . dot . **strict total**;
  axioms $\forall$o,o':OId $\forall$ n, n': Nat **in**
    o dot n = o' dot n' $\Rightarrow$ o = o' $\wedge$ n = n'**endaxioms**;}

The specification ATTRIBUTE introduces the sort Attribute, the polymorphic constructor function (.,.) for attributes, a selector function for attribute names and an equality test for attributes which only compares the identifiers and the sorts of the attributes. Moreover, it ensures that all elements of a multiset of attributes have different names (see axiom [unique_AId]; this property is needed in Appendix 2.2 for the definedness of the objects of a class C).

ATTRIBUTE = **{**
 enriches ID + MSETO;
 sort Attribute;
 (. : .) : AId $\times$ β $\rightarrow$ Attribute;
 aid: Attribute $\rightarrow$ AId;
 (. : .), aid **strict**;
 Attribute **generated by** (. : .);
 axioms $\forall$a,b:AId $\forall$ v:β $\forall$z:γ **in**
   aid(a : v) = a;
[unique_AId] $\delta(\{(a{:}v)\} \cup \{(b : z)\} \cup atts) \Rightarrow (a == b) = false$;
 endaxioms;**}**

The specification OBJECT0 introduces the sorts Object and Msg (for messages), the constructor function $< .{:}. \mid . >$ for objects and selector functions for object and class identifiers.

OBJECT0 = **{**
 enriches ATTRIBUTE;
 sort Object, Msg;
 $< .{:}. \mid . >$: OId $\times$ CId $\times$ Mset Attribute $\rightarrow$ Object;
 oid: Object $\rightarrow$ OId;
 cid: Object $\rightarrow$ CId;
 $< .{:}. \mid . >$, oid, cid **strict**;
 Object **generated by** $< .{:}. \mid . >$;
 axioms $\forall$o:OId $\forall$C:CId $\forall$atts: Mset Attribute **in**
  $\delta(<o : C \mid atts>) \Rightarrow oid(<o : C \mid atts>) = o$;
  $\delta(<o : C \mid atts>) \Rightarrow cid(<o : C \mid atts>) = C$;
 endaxioms;**}**

A configuration is a multiset of objects and messages. Since subsorting is not available in SPECTRUM we define an auxiliary sort ConfElem as sum of Object and Msg with explicit coercion functions inobject and inmsg. The axiom [unique_OId] ensures that the names of the objects in a configuration are pairwise different. The function oid collects all object identifiers of a configuration.

CONFIGURATION = **{**

enriches OBJECT0 + REWRITE;

sort ConfElem;

sortsyn Configuration = Mset ConfElem;

inobject: Object $\rightarrow$ ConfElem;

inmsg: Msg $\rightarrow$ ConfElem;

oid: Configuration $\rightarrow$ Mset OId;

inobject, inmsg, oid **strict, total**;

axioms $\forall$O1,O2:Object $\forall$c,c_1,c_2:Configuration $\forall$msg:Msg $\forall$atts: Mset Attribute $\forall$C:Cid $\forall$o:Oid **in**

[unique_OId] $\delta(\text{inobject}(O1) \cup \text{inobject}(O2) \cup c) \Rightarrow (\text{oid}(O1) == \text{oid}(O2)) = \text{false}$;

$\text{oid}(\varepsilon) = \varepsilon$;

$\text{oid}(\text{inmsg}(msg)) = \varepsilon$;

$\delta(\langle o : C \mid atts \rangle) \Rightarrow \text{oid}(\text{inobject}(\langle o : C \mid atts \rangle)) = \{o\}$;

$\text{oid}(c_1 \cup c_2) = \text{oid}(c_1) \cup \text{oid}(c_2)$;

endaxioms;**}**

Notational convention

In order to improve readability we allow objects to be written as

$\langle o : C \mid a_1 : v_1, ..., a_n : v_n \rangle$ or $\langle o : C \mid a_1 : v_1, ..., a_n : v_n, atts \rangle$

and translate them as

$\langle o : C \mid \{(a_1 : v_1)\} \cup ... \cup \{ (a_n : v_n)\} \rangle$ and

$\langle o : C \mid \{(a_1 : v_1)\} \cup ... \cup \{ (a_n : v_n)\} \cup atts \rangle$.

Similarly, we allow configurations to be written as

$x_1\ x_2\ ...\ x_n\ c$

and translate them as

$\{in_1(x_1)\} \cup \{in_2(x_2)\} \cup ... \cup \{in_n(x_n)\} \cup c$

with $in_1, ..., in_n \in \{\text{inobject}, \text{inmsg}\}$ depending on the type of x_i .

Appendix 2.2: Formal definition of object classes

A class C is given by its name, its attributes and its messages. In SPECTRUM the name of a class is represented by a class identifier denoted also by C, messages are created by function symbols with range Msg and there exist test operations isobj_C and ismsg_C for determining the correct classes of a multiset of attributes and of a message. A term <o : C | atts> is defined only if the attribute identifiers of atts are pairwise different (cf. the specification ATTRIBUTE) and if atts contains all attributes of C including the attributes of its superclasses (i.e. isobj_C(atts) = true). The expression

class C
 subclass of $D_1, ..., D_k$
 attr $a_1: s_1, ..., a_n: s_n$;
 msg $m_1(s_{11},...,s_{1l_1}),...,m_r(s_{r1},...,s_{rl_r})$;

is an abbreviation for the following SPECTRUM text:

C: $\rightarrow$ CId;
$m_1 : s_{11} \times ... \times s_{1l_1} \rightarrow$ Msg; ...; $m_r : s_{r1} \times ... \times s_{rl_r} \rightarrow$ Msg;
isobj_C: Mset Attribute $\rightarrow$ Bool;
ismsg : CId $\times$ Msg $\rightarrow$ Bool;
$m_1,..., m_r$, ismsg, isobj_C **strict total;**
axioms $\forall\ v_1{:}s_1,...,v_n{:}s_n\ \forall o$: OId $\forall$atts: Mset Attribute
$\forall\ x_{11}{:}s_{11},...,x_{rl_r}{:}s_{rl_r}\ \forall D$: CId $\forall$msg,msg_1,msg_2: Msg **in**
 $C \le D \Leftrightarrow (C == D \vee D_1 \le D \vee ... \vee D_k \le D)$;

 isobj_C(atts) =
 $\exists\ v_1{:}s_1,...,v_n{:}s_n\ \exists$ atts' : Mset Attribute.
  atts = $\{(a_1 : v_1)\} \cup ... \cup \{ (a_n : v_n)\} \cup$atts' $\wedge$
  isobj_D_1(atts') $\wedge ... \wedge$ isobj_D_k(atts'));

 δ(<o : C | atts>) $\Leftrightarrow$ isobj_C(atts);

 ismsg(D, $m_i(x_{i1},...,x_{il_i})$) = D$\le$C for i=1,...,r;

 ismsg(D, msg_1op msg_2) = (ismsg(D, msg_1) $\wedge$ ismsg(D, msg_2))
  for op $\in$ {;, ||, |||, or};

 ismsg(D, do msg) = ismsg(D, msg);

endaxioms;

Object creation is performed by means of a declaration of the form

new_C(atts);

where atts gives the initial value of the new object. As in [M 92] we assume the existence of a class ProtoObject with a single attribute “counter” whose value is a natural number. We assume that in the initial configuration we only have a collection of different objects of class ProtoObject. The expression new_C(atts) is an abbreviation for the introduction of a message

new_C : Mset Attribute $\rightarrow$ Msg;

with the axiom

axioms $\forall$o:OId $\forall$atts: Mset Attribute $\forall$ n: Nat **in**

δ(<o.n :C | atts>) $\Rightarrow$

new_C(atts) <o:ProtoObject | counter: n> ==>

<o:ProtoObject | counter: n+1> <(o dot n) :C | atts>;

endaxioms;

Appendix 2.3: Formal definition of subconfigurations

A subconfiguration is a specific kind of configuration which does not occur at the topmost level but at a lower level. In contrast to the sort Configuration which admits all defined messages and objects, a subconfiguration is restricted to a set of acceptable classes. The rewrite relation ==>* is compatible with subconfigurations.

The expression

configuration SubConf **of** D_1, ..., D_k;

is an abbreviation of

sort SubConf ;

subconf : Configuration $\rightarrow$ SubConf;

is_SubConf : Configuration $\rightarrow$ Bool;

subconf, is_SubConf **strict**;

axioms $\forall$msg:Msg $\forall$o: OId $\forall$atts, $atts_1$, $atts_2$: Mset Attribute
$\forall$x: CId $\forall$c,c1,c2: Configuration $\forall$a: AId **in**

is_SubConf(ε) =true;

is_SubConf({msg}) = (ismsg_D_1(msg) $\vee$... $\vee$ ismsg_D_k(msg));

is_SubConf({<o : x | atts>}) = (isobj_D_1(atts) $\vee$... $\vee$ isobj_D_k(atts));

$\delta(c1 \cup c2) \Rightarrow$
is_SubConf(c1∪c2) = (is_SubConf(c1)∧ is_SubConf(c2));

δ(subconf(c)) ⇔ is_SubConf(c);

c_1 ==>* c_2 ⇒ (a : subconf(c_1)) ==>* (a : subconf(c_2)) ;

$atts_1$ ==>* $atts_2$ ∧ δ(<o : x | $atts_1$>) ⇒
<o : x | $atts_1$> ==>* <o : x | $atts_2$>;

endaxioms;

Object creation for subconfigurations could be handled similarly as before; we prefer however a more abstract description which avoids the explicit introduction of a class ProtoObject:

axioms∀a: AId ∀c:Configuration ∀atts: Mset Attribute ∃o:OId **in**
δ(<o :C | atts>) ⇒
¬(o ∈ oid(c)) ∧
(a : subconf(new_C(atts) c)) ==> (a : subconf(<o :C | atts> c));
endaxioms;

Notational convention
In order to improve readability we write in examples
<o : C | a : c, atts>
instead of
<o : C | a : subconf(c), atts> .

Appendix 3: Proof of Lemma 3

Proof of Lemma 3.0)
The least fixpoint of elemlist is defined only for configurations of the form ε or <o : SortCell | content : ∞, conf : ε> or <o : SortCell | content : n, conf : c> where c is a configuration of the required form. Hence if elemlist(<o : SortCell | ATTS >) is defined in all models, ATTS has the required form. ◊**Proof of Lemma 3.0)**

Proof of Lemma 3.1) by case analysis and structural induction on the form of ATTS:

1i) ATTS = content : ∞, conf : ε

request <o : SortCell | content : ∞, conf : ε> ==> (by [rqu ∞])

<o : SortCell | content : ∞, conf : ε> send(nil)

with nil =elemlist(<o : SortCell | content : ∞, conf : ε>).

1ii) ATTS = content : n**, conf :** <o1 : SortCell | ATTS1>

Then

(*) elemlist(<o : SortCell | ATTS) = n+elemlist(<o1 : SortCell | ATTS1>).

We consider the configuration

request <o : SortCell | content : n, conf : <o1 : SortCell | ATTS1>>.

By induction hypothesis the configuration

request <o1 : SortCell | ATTS1>

rewrites (==>) to

<o1 : SortCell | content : ∞, conf : ε> send(l)

with l = elemlist(<o1 : SortCell | ATTS1>)

Hence by [rqu no] we get

<o : SortCell | content : ∞, conf : ε> send(n+l)

which by (*) is equal to

<o : SortCell | content :∞, conf : ε> send(elemlist(<o : SortCell | ATTS)).

◊**Proof of Lemma 3.1)**

Proof of Lemma 3.2) by structural induction on the form of ATTS:

2i) ATTS = content : ∞, conf : ε

Then

SORT_CELL |– l = elemlist(<o : SortCell | content : ∞, conf : ε>) = nil .

By [rcv ∞] and [new]

receive(n) <o : SortCell | content : ∞, conf : ε> ==> sc

where

sc = <o : SortCell | content : n, conf : <o' : SortCell | content :∞, conf :ε >> .

and elemlist(sc) = n+nil =insert(n,l) .

2ii) ATTS = content : m**, conf :** <o1 : SortCell | ATTS2> where ATTS2 is an element of A. Since

SORT_CELL1 |–

l = elemlist(<o : SortCell | content m, conf : <o1 : SortCell | ATTS2>>) and issorted l = true we have for l1 = elemlist(<o1 : SortCell | ATTS2>)

(***)

elemlist(<o : SortCell |content : m, conf : <o1 : SortCell | ATTS2>>) =m+l1, issorted l1 and m ≤ min(l1) .

Thus

(**) min(n,m) ≤m ≤ min(l1) .

Then by [rcv m]

receive(n) <o : SortCell | content : m, conf : <o1 : SortCell | ATTS2>>
==> <o : SortCell | content: min(n,m), conf: c1>

if receive(max(n,m)) <o1 : SortCell | ATTS2>> ==> c1 .

By induction hypothesis

receive(max(n,m)) <o1 : SortCell | ATTS2>> ==>
<o1 : SortCell | ATTS1>

such that

SORT_CELL1 |–
elemlist(<o1 : SortCell | ATTS1>) = insert(max(n,m), l1) .

Because of (**) min(n,m) ≤ min(l1) we can prove for SORTCELL1

issorted(min(n,m)+insert(max(n,m), l1)) and
elemlist(<o : SortCell | content : min(n,m), conf : <o1 : SortCell |ATTS1>>)
= min(n,m)+ insert(max(n,m), l1) .

Thus it suffices to choose c1 = <o1 : SortCell | ATTS1>.

◊**Proof of Lemma 3.2)**

Proof of Lemma 3.3) by natural induction:

k=0: Follows by Lemma 3.1) using the axiom

issorted l = (l ==sort l) in LIST_SORT.

k ⇒ k+1

By Lemma 3.2

receive(n_1) <o : SortCell | ATTS> ==> <o : SortCell | ATTS1>

such that SORTCELL1 |– elemlist(<o : SortCell | ATTS1>) = insert(n_1, l).

By induction hypothesis

(receive(n_2);...;receive(n_k);request) <o : SortCell | ATTS1> ==>
<o : SortCell | content : ∞, conf : ε> send(sort(n_2...n_k++ insert(n_1, l)))

Thus by [;]

(receive(n_1);receive(n_2);...;receive(n_k);request) <o : SortCell | ATTS>
==> <o : SortCell | content : ∞, conf : ε> send(sort(n_1...n_k++l)).

◊**Proof of Lemma 3.3)**

◊**Proof of Lemma 3**