(Objects + Concurrency) & Reusability – A Proposal to Circumvent the Inheritance Anomaly

Ulrike Lechner[1], Christian Lengauer[1], Friederike Nickl[2] and Martin Wirsing[3]

[1] Fakultät für Mathematik und Informatik
Universität Passau, D–94030 Passau, Germany
email: {lechner,lengauer}@fmi.uni-passau.de
[2] sd&m GmbH & Co. KG
Thomas-Dehler-Str. 27, D–81737 München, Germany
email: Friederike.Nickl@sdm.de
[3] Institut für Informatik
Ludwig-Maximilians-Universität, D–80538 München, Germany
email: wirsing@informatik.uni-muenchen.de

Abstract. We enrich the *object-oriented + concurrent* specification language Maude with language constructs for *reuse* and gain a high degree of code reusability. We consider three reuse constructs: (1) Maude's inheritance relation, (2) an algebra of messages and (3) the construct of a subconfiguration. By employing these constructs for different kinds of reuse, we show for all examples of the seminal paper on the inheritance anomaly [11] how to circumvent the inheritance anomaly. Our running example is the bounded buffer.

Keywords: Concurrent rewriting, inheritance anomaly, Maude, parallelism, reusability.

1 Introduction

Inheritance and concurrency are two paradigms which are difficult to combine in a satisfactory way. In [11] the necessity of reprogramming a high proportion of reused code when using inheritance is called the *inheritance anomaly*. In [13] it is claimed that the main reason for the inheritance anomaly is the presence of *synchronization code*, the code which determines which method calls can be accepted in dependence of the state of the object. The classical example for the use and the necessity of synchronization code is the bounded buffer which allows its get method to be invoked only if it is not empty [5, 11, 13].

The language Maude claims to overcome the inheritance anomaly: in [13] Meseguer, the developer of Maude, demonstrates how to use Maude's inheritance mechanism and modules with various ways of reuse to specify concurrent systems in a structured and modular way.

We agree with [13] in that the inheritance anomaly is caused by the *presence of synchronization code*, not by the way the program is structured and implemented. But, in resolving the inheritance anomaly, we develop a new idea:

Maude's inheritance mechanism, which allows only to add new attributes and new possibilities of state changes, is not sufficient. We present constructs in a language with *object-oriented concepts* + *concurrency* and gain a high degree of *reusability* of code. For us, reusability means not necessarily only inheritance. We introduce two new concepts: a *subconfiguration* and an *algebra of messages*. They allow us to reuse code not only by enhancing the set of possible state transitions but also by restricting them and by composing actions from basic actions in an object-oriented way.

Throughout, we use the bounded buffer as our example since it has been investigated in several languages with different inheritance and reuse mechanisms and different kinds of synchronization code [1, 5, 11, 13, 15].

2 Our specification language

This section provides a brief introduction to our specification language, which is based on Maude as defined in [14]. We adopt some notational ideas from OOSpectrum [16] and add the concept of a subconfiguration and a message algebra to Maude.

2.1 Basic Concepts

Maude [14] has two parts: one which is functional and another which specifies states (so-called *configurations*) and state changes. The functional part is OBJ3 [6]; it forms the subset of the language used to specify the properties of data types in a purely algebraic way.

In the state-dependent part of Maude one writes object-oriented specifications consisting of an import list, a number of class declarations, message declarations, equations and rewrite rules. An object of a class is represented by a term –more precisely, by a tuple– comprising a unique object identifier (of sort `OId`), a class identifier and a set of attributes with their values; e.g., the term `< B:BdBuffer | cont:C, in:I, out:O, max:M >` represents an object of class `BdBuffer` with identifier `B` and attributes `cont`, `in`, `out`, and `max` with values `C`, `I`, `O` and `M`, respectively. A message is a term of sort `Msg` (in mixfix notation) that consists of the message's name, the identifiers of the objects the message is addressed to and, possibly, parameters; e.g., the term `(put E into B)` is a message addressed to the bounded buffer `B` with a parameter `E`. A configuration is a multiset of objects and messages. Multiset union is denoted by juxtaposition; the term `(put E into B) < B:BdBuffer | cont:C, in:I, out:O, max:M >` denotes the union of the buffer and the message of above.

State changes are specified by transition rules on configurations as defined by the rewriting calculus given below.

As an example of a specification let us give the specification of buffers and bounded buffers and explain it subsequently. A similar Maude specification of bounded buffers can be found in [13]. The specification `CONFIGURATION` specifies the basic data types of objects, messages and configurations (for a formal definition see [12]). The specification `OIDLIST` (see App. A) specifies the sort `OIdList` of finite sequences of object identifiers together with a juxtaposition operation, where adding an element `E` to a list `C` on the left is written `E C` and adding it on the right is written `C E`.

Our syntax differs from Maude as follows. `BD_BUFFER = { ... }` corresponds to `omod Buffer is ... endo`. In Maude, each rewrite rule is preceded by `eq` or `rl`; we collect equations and transitions rules under the keywords `equations` and `transitions`. Our subclass declarations are part of a class declaration; in Maude class and subclass declarations are independent. Other than Maude, we collect variable declarations by universal quantification.

```
BD_BUFFER = {
enriches CONFIGURATION OIDLIST ;
classes Buffer attr cont: OIdList ;
        BdBuffer subclass of Buffer ;
          attr in: Nat, out: Nat, max: Nat ;
messages (new BdBuffer with _replyto _) : Nat OId -> Msg ;
   (to _ the new BdBuffer is _), (put _ into _),
   (get _ replyto _), (to _ answer to get is _) : OId OId -> Msg ;
transitions ∀ B,U,E:OId , C:OIdList, I,O:Nat in
[P](put E into B)
   < B:BdBuffer | cont:C, in:I, out:O, max:M >
   => < B:BdBuffer | cont:E C, in:I+1 >
            if (I - O < M) ;
[G](get B replyto U)
   < B:BdBuffer | cont:C E, in:I, out:O, max:M >
   => < B:BdBuffer | cont:C, out:O+1 >
      (to U answer to get is E) ;
[N](new BdBuffer with M replyto U)
   < P:Proto | class:BdBuffer, next:B >
   => < P:Proto | class:BdBuffer, next:inc(B) >
      < B:BdBuffer | cont:eps, in:0, out:0, max:M >
      (to U the new BdBuffer is B)
endtransitions }
```

In general, specifications have the following structure: Sp = {`enriches` R; `functions` F; `classes` C; `messages` M; `equations` E; `transitions` T}. The operator `enriches` imports specifications: each component of Sp consists of the union of the components of the imported specifications and of Sp. We may omit empty parts of the specifications as, e.g., functions and equations in the example above.

Class `Buffer` has only one attribute, `cont`, which is used to store object identifiers. Class `BdBuffer` inherits the attribute via the inheritance relation from class `Buffer`, as stated by the declaration `BdBuffer subclass of Buffer`. Subclasses inherit all attributes and all rewrite rules (equations and transition rules) from their superclasses.

A bounded buffer may react to two messages: `put` and `get`. `Put` stores an element in the buffer, `get` removes the first element being stored in the buffer and sends it to a "user". The transition rule with rule label `P` says that an object of class `BdBuffer` can react to a `put` message only if the actual number of objects being stored, `I-O`, is smaller than the upper bound `max`. Sending a `get` message not only triggers a state change of buffer `B` but also initiates an answer message to a "user" `U` which contains the result (an object identifier).

While rules `P` and `G` are standard, rule `N` is particular to Maude. To create a new buffer, a message `new` is sent to a proto-object (of class `Proto`) which is responsible for creating new objects. The message `new` triggers a new object to be created with default values for the attributes `in`, `out` and `cont`. The value of `max` is determined by a parameter of the message. The proto-object changes (increases) the value of the parameter `next` by `inc`, and we assume that this is done in such a way that the same object identifier is never created twice.

Generally speaking, transition rules specify *explicit, asynchronous communication* via message passing: if a message is part of a configuration, a state transition may happen and new (answer) messages waiting to be processed in subsequent state transitions may be created as part of the resulting configuration (in the specification given above only one new message is generated). The transition rules specify not only the behavior but also the equivalent of the synchronization code of other languages: the pattern given at the left-hand side involves not only the presence of objects and messages but also certain properties like the equality of object identifiers, values of attributes and parameters of the messages. (We could also specify more than one object at the left-hand side of a transition rule and specify a synchronous state transition of several objects.)

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 of signature Σ with variables from X. 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*.[4] 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 und a double arrow, $\overleftrightarrow{e}$, stands for a set whose elements are of the form e (with the exception that $\overleftrightarrow{m}$ is a multiset of messages).

[4] In contrast to [12] we do neither have a reflexivity nor a transitivity rule 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. If subconfigurations may occur we need also the rule (Sub), see Sect. 2.2.

A transition

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

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

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

and there is a substitution $\sigma : Vars \to T(\Sigma, X)$, where $D_i \leq C_i$ and

$$D'_i = \begin{cases} D_i, \text{ if } C_i = C'_i \\ C'_i \text{ else} \end{cases}$$

Let us explain this rule. A transition rule in T is instantiated such that all variables of the rule are substituted according to σ. The classes of the objects of the configuration to which the rule is applied are subclasses of the objects of the rule. Since an object of a subclass may have more attributes than the object of the superclass, we introduce $atts_i$ to match the additional attributes of the subclass. The values of those attributes are not changed in the transition. The values v_i are changed to w_i according to the rule. For simplicity we assume that no objects are created or deleted by the transition rule. We make two simplifying assumptions for the case that objects change their class: the class of the object at the right-hand side of the rule becomes the class of the (instance) object, and classes between which class changes are possible have the same attributes.

As a notational convention we may omit at the left- and right-hand side of a rule attributes whose values are not needed in the transition and, additionally, at the right-hand side attributes whose values are not changed.

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

$m'_1 o'_{i_1} \ldots o'_{i_n}$ `=>` $o'_{j_1} \ldots o'_{j_m} m'_2 \ldots m'_n$ 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:

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

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

2.2 Subconfigurations

With subconfigurations we can structure our configuration by permitting an object to contain configurations. This enables us to restrict the choice of classes to which the objects in a subconfiguration may belong. A subconfiguration, as defined below, must contain only objects belonging to a non-empty set of class names but it may contain arbitrary messages.

`Subconfiguration of {classnames}`$^+$

Since we specify explicitly which messages may pass from a configuration into a subconfiguration or vice versa, we impose no restriction on the messages in the subconfiguration construct.

An example of a class declaration using the subconfiguration construct is:

```
class HBuffer | conf: Subconfiguration of BdBuffer Flag .
```

Objects of class `HBuffer` have an attribute `conf` containing only objects belonging to class `BdBuffer` or class `Flag`.

In the presence of subconfigurations, we have to introduce one more rule, (Sub), to the rewriting calculus which specifies the application of transition rules to subconfigurations. Let $<\ o : C \mid a : S,\ atts\ >$ be an object containing a subconfiguration S stored under the attribute name a, and let *atts* denote all other attributes with their values of o apart from a:

$$<\ o : C \mid a : S,\ atts\ > \ \rightarrow\ <\ o : C \mid a : S',\ atts\ > \text{ if } S \rightarrow S' \qquad \text{(Sub)}$$

2.3 The message algebra

We introduce an algebra of messages which permits the formation of *composed messages*, much like process terms in process algebras, from basic messages as defined in Maude specifications. This provides us with a new way of reusing code: it is possible to specify messages which trigger more than one single computation step and to have some sort of control flow within these computation steps.

```
MSG_ALGEBRA = {
   enriches CONFIGURATION
       _+_, _;_, _;;_, _|_, _||_ : Msg Msg -> Msg;
   transitions ∀ m1,m2,n1,n2:Msg, c,d,c1,c2,d1,d2,h:Cf in
   [C] (m1 + m2) c => d
             if   m1 c => d   ∨   m2 c => d ;
   [S1](m1 ; m2) c1 c2 => d1 d2
             if   m1 c1 => d1 h   ∧   m2 c2 h => d2 ;
   [S2](m1 ;; m2) c1 c2 => d1 d2 (n1 ;; n2)
             if   m1 c1 => d1 n1 h   ∧   m2 c2 h => d2 n2 ;
   [P1](m1 | m2) c1 c2 => d1 d2
             if   m1 c1 => d1   ∧   m2 c2 => d2 ;
   [P2](m1 || m2) c1 c2 => d1 d2 (n1 || n2)
             if   m1 c1 => n1 d1   ∧   m2 c2 => n2 d2)
   endtransitions }
```

Specification `MSG_ALGEBRA` contains three message combinators: choice (`+`), sequential (`;` and `;;`) and parallel (`|` and `||`) composition. In sequential composition there can be a dependence between the left and the right message, in parallel composition there must be no dependence. We have two versions of sequential and parallel composition: one composes the answer messages following the structure of the input message (`;;` and $\|$) and one which does not pass this structure on to the next configuration (`;` and $|$). All combinators form atomic state transitions.

Generally, we have the freedom to specify any kind of sequential or parallel composition or choice in a message algebra. We have chosen the combinators given above only because we need them for the specifications in the next section.

3 Bounded Buffers and the inheritance anomaly

In this section, we extend the specification of the bounded buffer given in Sect. 2 with additional messages as originally suggested in [11] and partly also in [13]. With these extensions we demonstrate that Maude's inheritance relation, the message algebra and the concept of subconfiguration are powerful enough to overcome the inheritance anomaly. The first extension (by a message `last`) uses only Maude's inheritance mechanism, the second (by a message `get2`) the message algebra and for the third extension (by a message `gget`) we need the message algebra and subconfigurations. The class and module hierarchy we present in this section are depicted in Fig. 1.

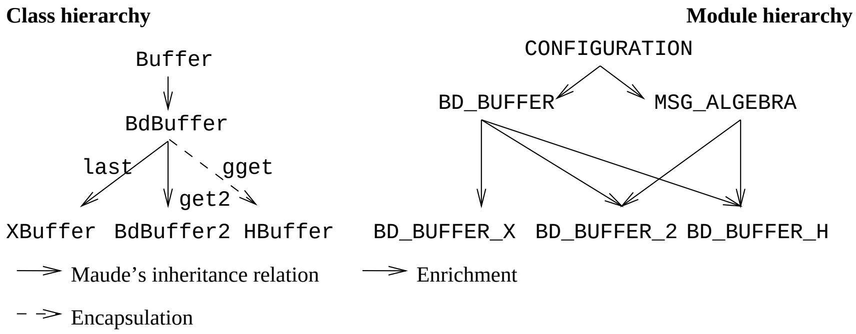

Fig. 1. Extensions of class `BdBuffer` and module `BD_BUFFER`

3.1 Message: `last`

One extension used in [11] to demonstrate how synchronization code requires the redefinition of existing code is the addition of a method `last` which returns the most recent element put into the buffer.

```
BD_BUFFER_X = {
    enriches BD_BUFFER ;
    class XBdBuffer subclass of BdBuffer ;
    messages (last _ replyto _),
             (to _ answer to last is _): OId OId -> Msg ;
transitions ∀ B,E,U:OId, C:OIdList, I,O:Nat in
[L] (last B replyto U)
    < B:XBdBuffer | cont:E C, out:O >
    => < B:XBdBuffer | cont:C, out:O+1 >
       (to U answer to last is E)
endtransitions }
```

Since `XBdBuffer` is a subclass of `BdBuffer`, an `XBdBuffer` inherits all attributes and all transition rules from `BdBuffer`. Thus an object of class `XBdBuffer` is capable of all state transitions an object of class `BdBuffer` is capable of in the same context. The transition rule `L` defines the behavior of a `XBdBuffer` when accepting a message `last`. When adding this new behavior to an existing specification we do not have to redefine or alter any piece of existing code and, thus, the inheritance anomaly does not apply.

3.2 Message: `get2`

Say we would like to extend the specification of class `BdBuffer` such that a buffer accepts an additional message, `(get2 B replyto U)`, which sends two elements of buffer `B` to an object `U`. The specification given in [13] is:

```
subclass BdBuffer2 < BdBuffer
rl (get2 B replyto U)
   < B:BdBuffer2 | cont:C E' E, in:I, out:O, max:M >
  => < B:BdBuffer2 | cont:C, out:O+2 >
     (to U answer to get2  is E' and E) .
```

This solution is not "suffering" from the inheritance anomaly but has one drawback: it does not reuse the specification of the message `get`. Our solution uses `MSG_ALGEBRA` to derive a composed message `get2` from the implementation of `get`:

```
BD_BUFFER_2 = {
    enriches BD_BUFFER MSG_ALGEBRA ;
    class BdBuffer2 subclass of BdBuffer ;
    messages (get2 _ replyto _) : OId OId -> Msg ;
       (to _ answer to get2 is _ and _) : OId OId OId -> Msg
equations ∀ B,U,E,E':OId, I,O,M:Nat in
[E1](get2 B replyto U)
    < B:BdBuffer2 >
    =  < B:BdBuffer2 >
       ((get B replyto U);;(get B replyto U)) ;
```

```
[E2]((to U answer to get is E);;(to U answer to get is E'))
    =  (to U answer to get2 is E' and E)
endequations }
```

We allow the "transformation" of a `get2` message to two `get` messages in sequence only if `get2` is addressed to a buffer of class `BdBuffer2` and, thus, we do not extend the set of messages a `BdBuffer` object accepts. The rewriting calculus and the algebra of messages allow us to process a `get2` message in one step, since we have an equational "transformation" of `get2` to `get` messages which does not require a computation step and since the algebra of messages allows us to build an atomic state transition `get2`.

Assume that we would like to implement `get2` in a more conventional language, with methods encapsulated in objects and guards for the methods such as in [1, 5, 7, 11]. This implementation of `get2` would apply `get` twice and its guard would have to make sure that `get2` is only invoked if the buffer contains (at least) two elements. The synchronization code of `get2` would have to be either derived from the synchronization code of the `get` method invoked twice by `get2` or written "by hand". Both might be hard, although in our example both is rather trivial. In our approach the message algebra ensures that a `get2` method may only be invoked if both invocations of `get` can be executed in sequence. This replaces the synchronization code of the message `get2` and facilitates the reuse of methods.

3.3 Message: `gget`

Our last modification is to make bounded buffer history-sensitive: a new message, `gget`, is only accepted if the latest message was a `put` message. Adding a message with a history-sensitive behavior involves a change of the behavior of all messages: `put` has to set a flag, all other messages have to reset it.

```
BD_BUFFER_H = {
enriches BD_BUFFER MSG_ALGEBRA ;
classes
    Flag atts buffer : OId ;
    FSet subclass of Flag ;
    FUnset subclass of Flag ;
    HBuffer atts conf: Subconfiguration of BdBuffer Flag
messages (set _), (unset _), (reset _) : OId → Msg ;
    (gget _ replyto _) : OId OId -> Msg
equations ∀ B,U,F:OId, C:Subconfiguration of BdBuffer Flag in:
[E1](gget B replyto U)
    < B:HBuffer | conf:<B:BdBuffer> <F:Flag|buffer:B> C >
    = < B:HBuffer | conf:((get B replyto U)|(reset F))
                         <B:BdBuffer> <F:Flag> C > ;
[E2]< B:HBuffer | conf:(to U answer to get is E) C >
    = (to U answer to get is E) < B:HBuffer | conf:C > ;
```

```
[E3](put E into B)
    < B:HBuffer | conf:<B:BdBuffer> <F:Flag|buffer:B> C >
    = < B:HBuffer | conf:((put E into B)|(set F))
                         <B:BdBuffer> <F:Flag> C > ;
[E4](get B replyto U)
    < B:HBuffer | conf:<B:BdBuffer> <F:Flag|buffer:B> C >
    = < B:HBuffer | conf:((get B replyto U)|(unset F))
                         <B:BdBuffer> <F:Flag> C >
endequations
transitions ∀ F:OId in
[S] (set F) <F:Flag> => <F:FSet> ;
[U] (unset F) <F:Flag> => <F:FUnset> ;
[R] (reset F) <F:FSet> => <F:FUnset>
endtransitions }
```

A history-sensitive buffer `HBuffer` encapsulates, in a subconfiguration, a buffer and a flag which indicates whether the last message accepted was a `put`.

We model the state of the flag, according to the states-as-classes approach [10], as classes and not, as usual, as attributes. In doing so we are able to refine the state of class `Flag` and, thus, its ability to process messages by introducing more subclasses. A flag accepts three messages and, while the state (and class) is irrelevant for a `set` and `unset` message to be accepted, a `reset` message is only accepted if the actual state of the flag is `FSet`.

The messages addressed to an object `HBuffer` are transformed by equations to a composed message inside the subconfiguration. This composed message consists of a message addressed to the `BdBuffer` and one message addressed to the `Flag`. The combinator | ensures that the message responsible for the manipulation of the buffer and the message triggering the state change of the flag are processed in parallel. In equation `E1` a message `gget` can be transformed into a parallel combination of a `get` and a `reset` message at any time. The state of the flag is only relevant for processing of the composed message consisting of a `reset` and a `get` message.

The two messages `put` and `get` migrate –like the message `gget`– into a subconfiguration and are transformed into a `put`, respectively a `get` message, and a message addressed to the flag. A `put` message is transformed into a composed message, consisting of a `put` and a `set` message, a `get` into a `get` and an `unset` message. The migration of `answer` messages from a subconfiguration into a configuration is also modeled by an equation.

The variable `C` of type subconfiguration in the equation matches messages which have already passed from the overall configuration into the subconfiguration. The use of configuration variables enhances also the reusability of the specification `BD_BUFFER_H`: if in a reusing specification more than two objects are contained in an `HBuffer` then the variable `C` can match these objects in the application of the transition rules by the rewriting calculus.

3.4 Analysis

In [11] several kinds of synchronization code are investigated. The inheritance anomaly occurs because no kind of synchronization code supports all types of modification when reusing a class declaration which contains synchronization code. Maude's equivalent to "synchronization code" is the pattern to be matched at the left-hand side of a transition rule. It consists of one or more messages, their parameters and the internal state of the object(s). Thus, each transition rule specifies a "synchronization constraint", more precisely an "enabled set", i.e., a condition under which a message may be accepted, individually for each message. Since each rule specifies such a pattern, adding new "enabled conditions" by adding new rules is the kind of modification of existing code which can be expressed straight-forwardly by Maude's inheritance relation (like `get2` and `last`).

For other kinds of modification of the behavior of reused code, we provide different mechanisms. With encapsulation in subconfigurations we are able to restrict the ability of objects to react to messages. The algebra of messages supports the specification of complex systems with a large number of objects and a complex control flow in the reuse of specifications with "simple" transition rules.

The reason why we are able to reuse code in many ways is that we provide different kinds of synchronization or control code: for each particular type of modification there is one particular construct for reuse.

One of the advantages of using equations to model the migration of messages into and out of subconfigurations is that we do not add additional state transitions or actions to a rewrite system. The migration would be some special kind of state transition and could be modeled by an internal action (like τ in CCS), but this would cause the same problems in compositionality and verification as it does in CCS.

4 Another buffer

In the previous sections we have given methods and language constructs for the reuse of specifications. Each method covers a particular situation in reuse. But it remains to demonstrate that these three techniques of reuse fit and can be used together.

We specify a bounded buffer, `PBuffer`, which is capable of processing all the three messages `last`, `get2` and `gget`.

```
BD_BUFFER_P = {
enriches BD_BUFFER_X BD_BUFFER_H BD_BUFFER2 ;
classes PBufferI subclass of XBdBuffer BdBuffer2 ;
        PBuffer subclass of HBuffer ;
messages (new PBuffer with _ replyto _): Nat OId -> Msg;
    (to _ the new PBuffer is _): OId OId -> Msg
equations ∀ B,E,U: OId, C:Cf in
```

```
[E1](last B replyto U)
    < B:PBuffer | conf:C >
    =  < B:PBuffer | conf:C ((last B replyto U)|(reset B)) > ;
[E2]< B:PBuffer | conf:C (to U answer to last is E) >
    =  < B:PBuffer | conf:C >
       (to U answer to last is E) ;
[E3](get2 B replyto U)
    < B:PBuffer | conf:C >
    =  < B:PBuffer | conf:C ((get2 B replyto U)|(reset B))) > ;
[E4]< B:PBuffer | conf:C (to U answer to get2 is E' and E) >
    =  < B:PBuffer | conf:C >
       (to U answer to get2 is E' and E) >
endequations
transitions ∀ P,B,U,F:OId, M:Nat in
[N] (new PBuffer with M replyto U)
    < P:Proto | class:PBuffer, next:(B,F) >
    => < P:Proto | class:PBuffer, next:(inc(B),inc(F)) >
       < B:PBuffer | conf:<B:PBufferI|cont:eps,in:0,out:0,max:M>
                          <F:FUnset|buffer:B> >
       (to U the new PBuffer is B)
endtransitions }
```

We have two subclass relations which inherit behavior from ancestor classes. The subclass definition of `PBufferI` ensures that `put`, `get` and `last` can be processed by `PBufferI`. Furthermore, it ensures that a `get2` method can actually be converted to a sequence of two `get` messages. The subclass definition of `PBuffer` ensures that all messages which may migrate into an `HBuffer`, namely `put`, `get` and `gget`, may migrate into a `PBuffer` as well. Equations `E1` to `E4` specify the migration of the messages `last` and `get2` and the answer message into and out of the subconfiguration. Rule `N` ensures that the buffer contained in the subconfiguration of `PBuffer` is of class `PBufferI`.

Note that again no changes of the reused specification are necessary. In fact, this specification demonstrates that our three concepts of reuse can be used together.

5 Buffers with synchronous communication

The transition rules of Maude offer a very powerful communication mechanism which we have not used up to now in our model of bounded buffers: a rewrite rule can employ more than one object and one message at its left-hand and right-hand side. Such a rule specifies joint atomic state transitions by all the objects involved. As an example we give a rule specifying a synchronous get message.

A message `(sget by B and U)` triggers a joint state transition of a buffer and a user. In it, the element which is retrieved from the buffer is stored in the attribute `elem` of the user; `sget` replaces the `answer` message used in the previous sections. We call this a synchronous implicit communication.

```
[SG] (sget by B and U)
     < U:User | buffer:B, elem:X >
     < B:BdBuffer | cont:C E, in:I, out:O, max:M >
     => < U:User | buffer:B, elem:E >
        < B:BdBuffer | cont:C, out:O+1 > ;
```

But is this kind of rule really appropriate for reuse? Of course, this depends on the kind of reuse, but a rule like this cannot be inherited easily when encapsulating one of the participants, say, the buffer in a subconfiguration. The rule relies very much on the structure of the configuration –namely, that both user and buffer are part of the same configuration– and on the shape of the objects.

We propose a specification, which uses the message algebra to specify a joint atomic state transition:

```
[H]  (h(U))
     (to U answer to get is E)
     < U:User | buffer:B, elem:X >
     => < U:User | buffer:B, elem:E > ;

[SE] (sget by B and U) = ((get B replyto U);h(U)) ;
```

Transition rule `H` specifies that a user requires two messages, an `answer` message and a help message `h`, to store an element retrieved from the buffer. Equation `SE` specifies that a synchronous `get` is a sequential composition of a `get` and an `h` message. The sequential composition ensures that the `h` message is always processed after the `get` message and, since the `h` and the `answer` message can only be processed in one joint synchronous transition, the `answer` message is also processed.

One can imagine that the other messages of the various buffers can be specified in a synchronous version using this technique. Moreover this specification of a synchronous communication can also be inherited to heirs which are encapsulated in a subconfiguration.

6 Related work

The SMoLCS approach [3, 4] combines algebraic specifications of data types and labeled state transition relations for the specification of behavior. Particular to this approach is a hierarchy of layers of specification with algebraic data types at the lowest layer and communication, parallel composition, abstraction and monitoring at successive higher layers. Algebras of actions allow to define the semantics of composed actions, modularized in the various layers.

The concept of subconfiguration also exists in several other object-oriented languages as, e.g., in Actor languages [1, 2] and Troll [7]. Particular to our approach is that we model the migration of messages into and out of subconfigurations by equations, not by transitions. This keeps the number of transitions reasonably small.

7 Conclusions

Maude's inheritance relation together with the concept of subconfiguration and an algebra of messages make it possible to reuse specifications. Our work on a verification technique for Maude specifications [9] demonstrates also that asynchronous message passing is better than synchronous message passing with respect to the inheritability of properties of specifications, a necessity for modular verification. The use of equations in modeling the migration of messages into and out of configurations keeps the number of transitions small. This helps to make the verification of properties of the behavior of single objects or configurations feasible. The use of equations contributes to a very abstract level of specification, where the structure of the state is of less importance, the focus of the specification lies on synchronization and communication of the objects, and one has few but powerful messages.

The degree of reusability of our specifications is far higher than one would expect for such a simple language in the presence of the inheritance anomaly [11]. This suggests that the basic design concepts of Maude, especially the object model, the communication mechanism and the transition rules for the specification of the behavior, are more appropriate for a structured design of specifications than the design concepts of more conventional languages with synchronous communication, explicit synchronization code and code of methods encapsulated in objects.

Together with our work on verification techniques for specifications in Maude [16], our picture of a sensible object-oriented specification language for the design of complex, concurrent systems is becoming more and more precise: Maude's object model, Maude's communication mechanism, a message algebra, subconfigurations and, maybe, also a module concept for programming in the large. In this paper we use Maude and made some small enhancements of its syntax. But, of course, our reuse mechanisms could be part of any object-oriented specification or programming language independent of the object model.

8 Acknowledgement

We wish to thank the anonymous referees for their valuable comments.

This work is part of the DFG project OSIDRIS and received travel support from the ARC program of the DAAD. The work of F. Nickl was carried out while she was a member of the Institut für Informatik, Ludwig-Maximilians-Universität München.

References

1. G. Agha, S. Frølund, W.Y. Kim, R. Panwar, A. Patterson, and D. Sturman. Abstraction and modularity mechanisms for concurrent computing. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 3–21. MIT Press, 1993.

2. M. Aksit, K. Wakita, J. Bosch, L. Bergmans, and A. Yonezawa. Abstracting object interactions using composition filters. In R. Guerraoui, I. Nierstrasz, and M. Riveill, editors, *Object-based Distributed Programming*, Lecture Notes in Computer Science 791, pages 152–183. Springer-Verlag, 1993.
3. 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.
4. 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.
5. 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.
6. J.A. Goguen, C. Kirchner, H. Kirchner, A. Megrélis, J. Meseguer, and T. Winkler. An introduction to OBJ3. In J.-P. Jouannaud and S. Kaplan, editors, *Proc. First Int. Workshop on Conditional Term Rewriting Systems*, Lecture Notes in Computer Science 308, pages 258–263. Springer-Verlag, 1988.
7. 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.
8. A.E. Haxthausen and F. Nickl. Pushouts of order-sorted algebraic specifications. In *AMAST'96*, Lecture Notes in Computer Science. Springer-Verlag, 1996. To appear.
9. 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.
10. 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.
11. S. Matsuoka and A. Yonezawa. Analysis of inheritance anomaly in concurrent object-oriented languages. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 107–150. MIT Press, 1993.
12. J. Meseguer. A logical theory of concurrenty 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.
13. 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.
14. 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.

15. O. Nierstrasz and M. Papathomas. Towards a type theory for active objects. *ACM OOPS Messenger*, 2(2):89–93, April 1991. (*Proceedings OOPSLA/ECOOP 90 Workshop on Object-Based Concurrent Systems*).
16. M. Wirsing, F. Nickl, and U. Lechner. Concurrent Object-Oriented Design Specification in Spectrum. In P.-Y. Schobbens, editor, *MeDiCis'94: Methodology for the Development of Computer System Specifications, Working Notes of a Workshop held in the Chateau de Namur 1994*, pages 163–179, 1994. Full version: Concurrent Object-Oriented Design Specification in SPECTRUM, Technical Report 9418, Institut für Informatik, Ludwig-Maximilians-Universität München, December 1994.

A Specification of `OIDLIST`

```
OIDLIST = {
   enriches OID ;
   sort OIdList ;
   functions
      eps : → OIdList ;
      _ _ : OId OIdList → OIdList ;    (* left append *)
      _ _ : OIdList OId → OIdList      (* right append *)
   equations ∀ E1,E2:OId, L:OIdList in
      E2 eps = eps E2 ;
      E1 (L E2) = (E1 L) E2
   endequations }
```

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

Coherence is a central property of order sorted signatures: in a coherent signature two connected subsorts have a common supersort (locally upward filteredness) and there is always a unique least sort for each term (regularity) [8]. This problem is illustrated by the following example: Assume we have signatures

$\Sigma_1 =$ `Nat, NatList; subsorts Nat < NatList;`

$\Sigma_2 =$ `sorts Nat, Int; subsorts Nat < Int;`

sharing the common subsignature

$\Sigma_0 =$ `sorts Nat;`

Then the union of Σ_1 and Σ_2 (more precisely, the pushout of the inclusions $\sigma_1 : \Sigma_0 \hookrightarrow \Sigma_1$ and $\sigma_2 : \Sigma_0 \hookrightarrow \Sigma_2$) is

$\Sigma =$ `sorts Nat, Int, NatList; subsorts Nat < Int, Nat < NatList;`

Σ is not locally upwards filtered. Hence, in the above specification `OIDLIST`, problems w.r.t. the coherence of its signature arise if `OId` is declared as a subsort of `OIdList` and `OId` is also declared as a subsort in the specification `OID`. Since we use overloading, we avoid this problem.

This article was processed using the LaTeX macro package with LLNCS style