Constructs, Concepts and Criteria for Reuse in Concurrent Object-Oriented Languages

Ulrike Lechner

Institute for Media and Communications Management
University of St. Gallen
CH–9000 St. Gallen, Switzerland
email: Ulrike.Lechner@mcm.unisg.ch

Abstract. For reuse in concurrent object-oriented languages we present a set of *reuse constructs*. We give *criteria* for relations between classes that can be implemented by those reuse constructs, characterize the properties inherited via the constructs and explore that we have not only constructs but *concepts* for reuse.
We demonstrate the concepts and constructs with the object-oriented concurrent language Maude. We employ the μ-calculus to reason about these specifications and (bi)simulation relations parameterized with Galois connections to model reuse.
Keywords: Reuse, Object orientation, Concurrency, Rewriting, Maude, μ-calculus, Abstract Interpretation, Inheritance Anomaly.

1 Introduction

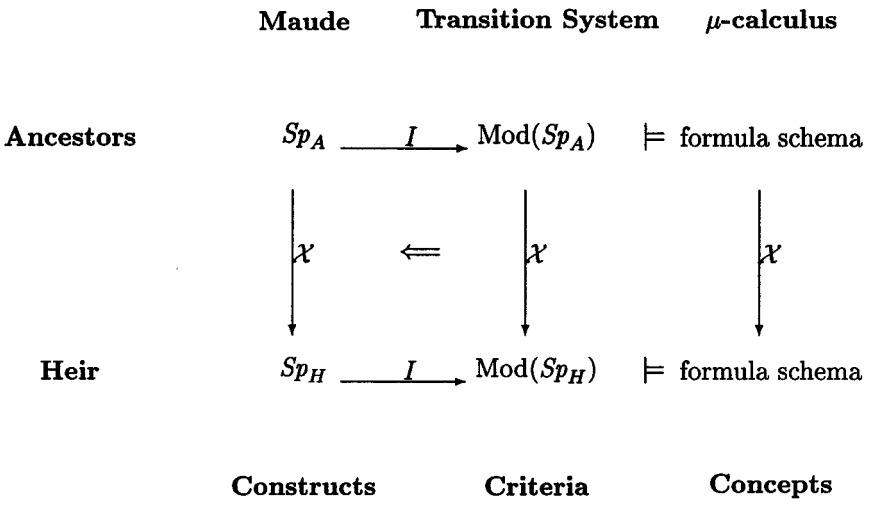
Reusability is considered to be one of the distinguishing advantages of object orientation. However, Matsuoka and Yonezawa demonstrated in their seminal paper on the inheritance anomaly that reuse in object-oriented concurrent languages is hardly feasible with inheritance alone [MY93].

To facilitate reuse in object-oriented concurrent languages more precisely in a particular language Maude we have developed a set of *reuse constructs* and we demonstrated that this set of reuse constructs is powerful enough to circumvent the inheritance anomaly [LLNW96].

However, reuse of code alone is not satisfactory, in particular, not at the level of a specification language, where *inheritance of properties* must have preference over mere *reuse of code*. We characterize the classes of properties that are inherited via our reuse constructs. We employ the modal μ-calculus [Koz83,Bra92] to reason about Maude specifications and we employ property preserving mappings, namely (bi-)simulation relations parameterized with Galois connections [LGS+95], for relations between classes and to characterize the properties that are inherited via the reuse relations.

We go one step further and give criteria for reuse relations, i.e., *criteria*, according to which classes can be implemented with a reuse relation. The scenario, we have in mind is object-oriented analysis and design of which establishing an appropriate class hierarchy is an essential part [WK96,BJR98]. In sequential

object-oriented languages, the information stored inside the classes together with the methods determine possible reuse relation [HP92]. The more concurrency a language admits, the more the behavior of classes becomes relevant in reuse and for relations between classes. We provide with our relations between classes of algebras criteria about which classes can be in a "reuse relation".



Let us explain this diagram. $\mathcal{X}$ is one of the three reuse relations. Note that we are able to have more than one ancestor. We have three different levels at which we explore reuse. The first level is our specification language, Maude with its reuse constructs. Maude specifications can be interpreted and the semantics of Maude specifications are classes of algebras. For those classes of algebras, we develop relations, which can be implemented as reuse constructs at the level of Maude. The second level, which we consider in this paper, is thus the semantic level of transition systems. The third level is the μ-calculus. Properties of objects can be phrased in the μ-calculus and we can prove whether those properties hold for a transition system [Lec97]. We characterize the classes of properties that are inherited via our reuse constructs.

This paper is organized as follows. We introduce in Sect. 2 our specification language Maude and in Sect. 3 our constructs for reuse. In Sect. 4, we give a brief introduction to the μ-calculus and the formula schemata. Sect. 5 contains the framework of property-preserving mappings. In Sect. 6, we explore the criteria for reuse and in Sect. 7 we characterize the properties that are inherited. Sect. 8 contains an example. We give a brief overview of related work in Sect. 9 and conclude our results in Sect. 10.

2 Maude

This section provides a brief introduction to our specification language, Maude [Mes96]. Note, that we employ the notation implemented in the CafeObj System [FN96]. E.g., [Mes96,Lec97] provide a more detailed introduction to Maude.

Maude [Mes96] has two parts: one which defines the basic data types using order-sorted equational specification and another which specifies states (so-called *configurations*) and state changes.

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 transition rules. An *object* of a class is represented by a term comprising an object identifier (of sort `ObjectId`), a class identifier and a set of attributes with their values; e.g., `< B : BdBuffer | cont = C, max = M >` represents an object of class `BdBuffer` with identifier `B` and attributes `cont` and `max` with values `C` and `M`, respectively. A *message* is a term of sort `Message` (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. A *configuration* (of sort ACZ-Configuration) is a multiset of objects and messages. Multiset union is denoted by juxtaposition. *State changes* are specified by *transition rules* (keyword `rl` or `crl`).

As an example of a specification let us give the specification of bounded buffers and explain it subsequently. The specification `EXT-ACZ-CONFIGURATION` specifies the basic data types of objects, messages and configurations (for a formal definition see [Mes96,Lec97]). The empty state, i.e., the element of sort `ACZ-CONFIGURATION` is denoted by `acz-empty`. `LIST` specifies the sort `List` of finite sequences together with a juxtaposition operation where adding an element `E` to a list `C` on the left is written `E C` and a list consisting of a list and a single element is written `C E`. `NAT` contains the specification of natural numbers (`Nat`) and the sort `NzNat` for natural numbers strictly greater than `zero`.

```
module BD-BUFFER {
  import {
    protecting (NAT)
    protecting (LIST)
    protecting (EXT-ACZ-CONFIGURATION) }

  signature {
    class BdBuffer {
      max  : NzNat
      cont : List }

     op get _ replyto _  : ObjectId ObjectId -> Message
     op to _ answer is _ : ObjectId Elem -> Message
     op put _ into _     : Elem ObjectId -> Message }

  axioms {
```

```
  vars B R  : ObjectId
  var  E    : Elem
  var  C    : List
  var  M    : NzNat
  var  ATTS : Attributes

crl [P]: (put E into B)
         < B : BdBuffer | cont = C,   max = M, ATTS >
      => < B : BdBuffer | cont = E C, max = M, ATTS >
         if length(C) < M .

rl  [G]: (get B replyto R)
         < B : BdBuffer | cont = C E, max = M, ATTS >
       => < B : BdBuffer | cont = C  , max = M, ATTS >
         (to R answer is E) . } }
```

The class `BdBuffer` has two attributes, `max` is the capacity of a bounded buffer and `cont` stores the buffered elements. The variable `ATTS` collects—according to the syntax supported by the CafeObj system [FN96]—attributes not mentioned in a rule or additional attributes particular to heirs of `BdBuffer`.

A bounded buffer may react to two messages: `put` and `get`. `Put` stores an element in the buffer, `get` removes the oldest element being stored in the buffer and sends it to a "receiver". 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, `length(C)` is smaller than the capacity `max`. Sending a `get` message triggers not only a state change of buffer `B` but also initiates an answer message to `R` which contains the result (an element). Note, that a `get` is only accepted if the buffer is not empty, i.e., if attribute `cont` contains a structure `C E` indicating that there is at least one element part of the list.

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). We could also have more than one object at the left-hand side of a transition rule and specify thereby a synchronous state transition of several objects [Lec97].

The matching itself is done by a Rewriting Calculus. Examples for rewriting calculi can be found in [Mes92,Mes96,Lec97,LLNW96]. Note, that we consider in contrast to [Mes96] labeled transition systems, whose labels are the messages triggering a state transition.

A specification comprises thus a signature, a set of equations and a set of transition rules. Later, we use the notation $Sp = (\Sigma, E, T)$ for Maude specifications. The signature itself consists of a set of sorts, a subsort relation and a set of operators and is written as $\Sigma = (S, \leq, OP)$.

3 Reuse Constructs for Maude

We have developed a set of three reuse constructs for Maude: (1) Maude's inheritance relation, (2) subconfiguration and (3) message algebra. We explain the constructs briefly and give a typical example for each of them. Sect. 8 contains the specification code.

According to *Maude's inheritance relation* [Mes96], an heir inherits all attributes, all equations and all transition rules from all its ancestors. Thus, an heir reacts at least in all situations in which one of its ancestors was able to react to a message. A typical example for the use of inheritance is a bounded buffer that reacts to more messages than `BdBuffer`.

The construct of *subconfiguration* is dual to inheritance [LLNW96]. It allows to restrict the ability of a class encapsulated in a subconfiguration to react to messages. A typical example is a bounded buffer which is implemented by reusing an unbounded buffer. The unbounded buffer, providing the facilities to store elements is encapsulated inside a bounded buffer that restricts the messages that come into contact with the unbounded buffer [Lec97].

The concept of *message algebra* is particular to Maude [LLNW96]. We specify message combinators and their semantics that allows us to construct composed messages from atomic messages. A typical example is a `get2` message implemented as a sequential composition of two `get` messages. The semantics of the sequential composition, provides (1) non-interference and (2) that a `get2` message is accepted if and only if both its `get` messages can be accepted in sequence. Moreover, we ensure by the semantics of the message combinator that the `answer` messages are arranged properly such that they can be transformed into an answer containing two elements in proper order. Note, that the message combinators and, their semantics is subject to a Maude specification and thus, this concept gives us a large amount of freedom and expressivity.

Note, that we employ equations and not only transition rules alone. Thus, transformations of the state, necessary for implementing a `get2` message by a sequential composition of two `get` messages, or by modeling the migration of messages into and out from subconfigurations does not involve additional "administrative" transitions.

4 The μ-calculus

The μ-calculus is used to reason about state transition systems at a property-oriented level [Koz83,Bra92]. The language of μ-formulas, denoted by $\mathcal{L}_\mu$, is constructed from atomic propositions, conjunction and disjunction, modal connectives and fixpoint operators according to the following grammar. Let the set T be non-empty (but possibly infinite).

$$
\begin{aligned}
p &::= \mathtt{tt} \mid \mathtt{ff} \mid \neg p \mid \text{“}o\text{”} \mid \text{“}m\text{”} \\
\phi &::= p \mid (\wedge i : i \in T : \phi_i) \mid (\vee i : i \in T : \phi_i) \mid (\exists x \in T : \phi) \mid (\forall x \in T : \phi) \\
&\quad \mid \langle L \rangle \phi \mid [L]\phi \mid (\nu X.\phi) \mid (\mu X.\phi)
\end{aligned}
$$

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

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

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

We are interested in the truth of formulas in a structure (A, R) which is a model of a Maude specification. Let us introduce some notation. Let v be a valuation and I be an interpretation function which indicates in which structure formulas are interpreted. $\mid \phi \mid_{(A,R),I} v$ denotes all elements of A, for which ϕ holds under valuation v and under an interpretation $I : \mathcal{L}_\mu \to (A, R)$.

We introduce a set of formula schemata describing the behavior of classes.

Definition 1 (Formula schemata). Let C be a class and atts resp. atts' denote the attributes with their values of class C. Let SI(< B : C | $\mathtt{atts}_i$ >), ϕ_i(< B : C | $\mathtt{atts}_i$ >) and ψ_i(< B : C | $\mathtt{atts}_i$ >) be propositions on the state of an object B of class C. SI is the state invariant of class C. Let $\mathtt{a}_i$ be a message and let p be variables in the formulas with the range P. We define five formula schemata by closed μ-formulas for a class C with n methods:

$$
\begin{aligned}
&\mathit{Persistence}(B) = (\nu X.(\forall p{\in}P : \\
&\quad \text{"< B : C >"} \Rightarrow [-](\text{"< B : C >"} \wedge X))) \\
&\mathit{State}(B) = (\nu X.(\forall p{\in}P : \\
&\quad SI(\text{"< B : C | atts >"}) \Rightarrow [-](SI(\text{"< B : C | atts' >"}) \wedge X))) \\
&\mathit{Synchronization}(B) = (\wedge i : 1 \leq i \leq n : (\forall p{\in}P : \\
&\quad \text{"}m_i\text{"} \wedge \text{"< B : C | atts >"} \wedge \psi_i(\text{< B : C | atts >}) \wedge \langle m_i\rangle \mathtt{tt})) \\
&\mathit{StateChange}(B) = (\nu X.(\forall p{\in}P : (\wedge i : 1 \leq i \leq n : \\
&\quad \text{"< B : C | atts >"} \wedge SI(\text{< B : C | atts >}) \wedge \psi_i(\text{< B : C | atts >}) \\
&\qquad \Rightarrow [m_i](\phi_i(\text{< B : C | } \mathtt{atts}_i\text{' >}) \wedge X)))) \\
&\mathit{AnswerMessages}(B) = (\nu X.(\forall p{\in}P : (\wedge i : 1 \leq i \leq n : \\
&\quad \text{"< B : C | atts >"} \wedge SI(\text{< B : C | atts >}) \wedge \psi_i(\text{< B : C | atts >}) \\
&\qquad \Rightarrow [m_i](\text{"}\mathtt{a}_i\text{"} \wedge X))))
\end{aligned}
$$

Each of the formula schemata reflects one particular notion of the object model of Maude. *Persistence* describes that objects do not disappear, *State* that a state invariant holds, *Synchronization* under which circumstances an object reacts to a message, *AnswerMessages* gives the messages created as a result of a state transition of an object and *StateChange* describes the changes in the state of an object.

5 Property Preserving Mappings

The property-preserving mappings we employ to relate transition systems comprise (1) Galois connections as a relation between (sets of) states and (2) (bi)simulation relations parameterized with Galois connections as a relation between transition systems, whose states are in Galois connection. We rely on [LGS+95] for notation and formal framework.

Let us introduce some abbreviations and notation: $\overline{X}$ is the complement of X in the domain of X. Id^Q is the identity function on a set Q. The dual of a function α is $\widetilde{\alpha}$, defined by $\widetilde{\alpha}(X) =_{\text{def}} \overline{\alpha(\overline{X})}$. Let Q be a set of states, $X \subseteq Q$, L a set of labels and R a relation; the set of predecessors in a labeled transition relation R by transitions with a label in the label set L is represented by $pre(R)(L)(X)$, the set of successors respectively by *post*. Let S_1, S_2 be two sets of configurations: $S_1 \uplus S_2 =_{\text{def}} \{C_1\ C_2 \mid C_1 \in S_1, C_2 \in S_2\}$. (Remember that the multiset union of configurations is written $C_1\ C_2$.)

A Galois connection is a relation between sets, which is determined by two functions α and γ. As the names of the two functions suggest, we refer to them as the abstraction and concretion function, respectively.

Definition 2 (Galois connection). Let Q_1 and Q_2 be two sets. A *Galois connection* (α, γ), from $\wp(Q_1)$ to $\wp(Q_2)$ is a pair of continuous functions $\alpha : \wp(Q_1) \to \wp(Q_2)$, $\gamma : \wp(Q_2) \to \wp(Q_1)$ such that $Id^{Q_1} \subseteq \gamma \circ \alpha$ and $\alpha \circ \gamma \subseteq Id^{Q_2}$.

Note, that α distributes over union of sets, i.e., $\alpha(S_1 \cup S_2) = \alpha(S_1) \cup \alpha(S_2)$.

Galois connections provide the formal framework for relating sets of states. Let us now define a simulation relation between transition systems whose states are in a Galois connection.

Definition 3 ($\sqsubseteq_{(\alpha,\gamma)}$ and $\simeq_{(\alpha,\gamma)}$). Let $S_1 = (Q_1, R_1)$ and $S_2 = (Q_2, R_2)$ be two transition systems, L_1 the set of labels of S_1 and (α, γ) a Galois connection from $\wp(Q_1)$ to $\wp(Q_2)$. S_2 is an (α, γ)-*simulation* of S_1, written $S_1 \sqsubseteq_{(\alpha,\gamma)} S_2$, if and only if, for any $L \subseteq L_1$, $\alpha \circ pre(R_1)(L) \circ \gamma \subseteq pre(R_2)(\alpha(L))$.

S_1 and S_2 are (α, γ)-*bisimilar*, written $S_1 \simeq_{(\alpha,\gamma)} S_2$, if and only if, S_1 (α, γ)-simulates S_2 and S_2 $(\widetilde{\gamma}, \widetilde{\alpha})$-simulates S_1, i.e., $S_1 \sqsubseteq_{(\alpha,\gamma)} S_2$ and $S_2 \sqsubseteq_{(\widetilde{\gamma},\widetilde{\alpha})} S_1$.

Note, that a Σ-homomorphism $f : A_1 \to A_2$, more precisely, its extension to sets, which we also denote by f, is an abstraction function and induces a simulation relation $A_1 \sqsubseteq_{(f,f^{-1})} A_2$ [LGS+95,Lec97]. Note that a (simulation) relation $\rho \subseteq A_1 \times A_2$ induces a simulation relation $A_1 \sqsubseteq_{(post(\rho),\widetilde{pre(\rho)})} A_2$[LGS+95,Lec97].

Preservation of a formula by a function α means that, if a formula holds for a set of states, then it holds for the image of this set under α as well. Let (A_1, R_1) and (A_2, R_2) be two transition systems, $\phi \in \mathcal{L}_\mu$ a formula, and $I : \mathcal{L}_\mu \to \wp(A_1)$ an interpretation function. f *preserves* ϕ for I iff for $q \in Q_1$, $q \in |\phi|_{(A_1,R_1),I}$ v implies $f(q) \subseteq |\phi|_{(A_2,R_2),f \circ I}$ $f(v)$. A function f is *consistent* with an interpretation function I if, for all formulas ϕ, $f(\overline{I(\phi)}) \cap f(I(\phi)) = \emptyset$.

Theorem 4 (Preservation of properties). *Let (A_1, R_1) and (A_2, R_2) be two transition systems. Let $I_1 : \mathcal{L}_\mu \to A_1$ and $I_2 : \mathcal{L}_\mu \to A_2$ be two interpretation functions.*

1. *If $(A_1, R_1) \sqsubseteq_{(\alpha,\gamma)} (A_2, R_2)$ then α preserves $[\,]$-free, positive formulas, and if α is consistent with I_1, then α preserves $[\,]$-free formulas.*
2. *If $(A_1, R_1) \sqsubseteq_{(\alpha,\gamma)} (A_2, R_2)$ then $\widetilde{\gamma}$ preserves $\langle\,\rangle$-free positive formulas for I_2 and, if $\widetilde{\gamma}$ is consistent with I_2, then $\widetilde{\gamma}$ preserves $\langle\,\rangle$-free formulas.*
3. *If $(A_1, R_1) \simeq_{(\alpha,\gamma)} (A_2, R_2)$ then α preserves positive formulas for I_1 and, if α is consistent with I_1, then α preserves all formulas for I_1.*

Proof. Proof by induction on the size of formulas. See [LGS+95] or [Lec97].

6 Criteria for Reuse

Let us sketch briefly our design scenario and the role of our results for the object-oriented specification of distributed systems. In object-oriented design, the class hierarchy has to be established with the reuse relations between the different classes. We provide via our relations information about criteria which classes are similar so that they can be implemented via a reuse relation. The formal basis for "similarity" is the property preserving relation introduced in Sect. 5.

The design of the class hierarchy is the first phase: only in the second phase a system is modeled as a collection of objects. Thus, the properties (and possibly proofs) whose inheritability one is interested in are properties of single classes. We are interested in the inheritability of the instances of the formula schemata of Def. 1.

In the following, we define relations between classes of algebras, which can be implemented by reuse relations. The relations consist of two parts: (1) a relation between the algebras and (2) a relation between transition systems. Common to the three criteria for reuse is also the function fl (an abbreviation for filter), which abstracts from the structures of the heir and relates ancestor states (terms of sort `ACZ-Configuration`) to heir states. Note that we consider only specifications with coherent order-sorted signatures [HN96].

Definition 5 (Common basis for reuse criteria). We consider an "ancestor" specification $Sp_A = (\Sigma_A, E_A, T_A)$ with $\Sigma_A = (S_A, \leq_A, OP_A)$ and a "heir" specification $Sp_H = (\Sigma_H, E_H, T_H)$ with $\Sigma_H = (S_H, \leq_H, OP_H)$. Σ_A and Σ_H are coherent order-sorted signatures. Let $\sigma : \Sigma_A \to \Sigma_H$ be the canonical injection.

Let $\mathrm{fl} \subseteq \Sigma_H \times \Sigma_A$ be given by

$$
\begin{array}{lll}
\mathrm{fl}(D_1 D_2) & = \mathrm{fl}(D_1) \uplus \mathrm{fl}(D_2) & \\
\mathrm{fl}(\texttt{acz-empty}) & = \texttt{acz-empty} & \\
\mathrm{fl}(\texttt{<}O : C \,|\, \{a = v\}\texttt{>}) & = \{\texttt{<}O : C \,|\, \{a = w\}\texttt{>} | w \in \mathrm{fl}(v)\} & \text{for } C \in \Sigma_A, C \not\leq_H C_H \\
\mathrm{fl}(\texttt{<}O : C \,|\, \{a = v\}\texttt{>}) & = \texttt{acz-empty} & \text{for } C \in \Sigma_H \setminus \Sigma_A, C \not\leq_H C_H \\
\mathrm{fl}(m(p_1 \ldots p_n)) & = m(\mathrm{fl}(p_1) \ldots \mathrm{fl}(p_n)) & \text{for } m \in \Sigma_A \\
\mathrm{fl}(m(p_1 \ldots p_n)) & = \texttt{acz-empty} & \text{for } m \in \Sigma_H \setminus \Sigma_A \\
\mathrm{fl}(v) & = v & \text{for } v : s, s \not\leq \texttt{Cf}
\end{array}
$$

Let us motivate the common basis for the reuse relations. Common to the reuse relations is that we require that the heir specification has at least the sorts and function symbols of the ancestor. We ensure this by the existence of a canonical embedding $\sigma : \Sigma_A \to \Sigma_H$ and we require that the reduct of a Sp_H-algebra is a Sp_A-algebra, i.e., $H|_\sigma = A$ for some $(A, R) \in Mod(Sp_H)$.

We apply fl to abstract from the new classes and relations and to relate ancestor and heir configurations of the transition systems.

In Maude, an heir inherits from its ancestor the implementation of the state and the ability to react to messages. Thus, in order to establish inheritance one needs a relation in which the heir acts and reacts if the ancestors act and react. This is captured by a simulation relation.

Definition 6 (Inheritance Criterion). Let Def. 5 be included. Let C_{A_i} for $1 \leq i \leq n$ be classes in Σ_A and C_H a class in Σ_H. Sp_H is an *heir of* Sp_A via $C_H \leq_H C_{A_1}, \ldots, C_{A_n}$ if

$$(\forall (H,S) \in Mod(Sp_H) : (\exists (A,R) \in Mod(Sp_A) : H|_\sigma = A \wedge (A,R) \sqsubseteq_{(pre(\mathrm{fl}),\widetilde{post}(\mathrm{fl}))} (H,S)))$$

where for $C_{H'} \leq_H C_H$

$$\mathrm{fl}(< O : C_{H'} \mid atts >) = \{< O : C_{A_i} \mid atts_{A_i} > \mid (\forall a = w \in atts, a \text{ attribute of } C_{A_i}, v \in \mathrm{fl}(w) : a = v \in atts_{A_i})\}$$

Let us explain and motivate this inheritance relation. We relate in the simulation relation modeling inheritance those states whose parts belonging to the ancestor specification are equal. Function fl provides this abstraction for the heir configurations and induces a simulation relation on states.

The abstraction filters the "new observations", which are particular to the heir specification, while the reduct excludes "new elements". This difference in the treatment of the inheritance relation reflects the difference in the construction in algebras and observation in transition systems. fl links the two concepts by abstracting in a way such that behaviorally equal configurations, which are constructed differently, are related in the inheritance relation. Maude's object model is the reason why we cannot abstract from the values and consider only the sorts, since the values of the attributes determine whether and how an object reacts to a message. Thus, we cannot extend the domain of basic values.

Our second construct and concept for reuse is subconfiguration. Subconfiguration are a means to restrict the ability of the reused classes to act and react. Thus, subconfiguration and accordingly the simulation relation and the criterion are dual to inheritance.

Definition 7 (Subconfiguration Criterion). Let Def. 5 be included. Let Σ_A comprise the classes C_{A_i} for $1 \leq i \leq n$ and let C_H with attribute a be a class in Σ_H. Sp_H is an *heir by subconfiguration via* $(C_H$ `Subconfiguration of` $C_{A_1} \ldots C_{A_n})$ of Sp_A if

$$(\forall (H,S) \in Mod(Sp_H) : (\exists (A,R) \in Mod(Sp_A) : H|_\sigma = A \wedge (H,S) \sqsubseteq_{(post(\mathrm{fl}),\widetilde{pre}(\mathrm{fl}))} (A,R)))$$

where for $C_{H'} \leq_H C_H$
$\mathrm{fl}(<O : C_{H'} \mid atts>) = < O : C_{A_i} \mid atts_{A_i} > \mid$
$(\forall a = w \in atts, a \text{ attribute of } C_{A_i}, v \in \mathrm{fl}(w) : a = v \in atts_{A_i})\}$

The criterion for reusability via subconfiguration is that an object of class C_H can be replaced by a number of objects of class $C_{A_1} \ldots C_{A_n}$ and that this increases the number of possible transitions. Hereby, the values of the respective attributes of the ancestor(s) and the heir are identical. When two specifications for which the criterion holds are implemented by reuse we replace a "normal" value of an attribute (of class C_H) by an object of the reused class (C_{A_i}).

The third reuse construct and concept is the message algebra with which new message combinators together with their semantics, i.e., the way composed messages are being processed, are specified. A message combinator such as, e.g., sequential composition, does not affect the state changes triggered by these single messages. Maude provides us with the flexibility to combine less benign message combinators that allow us to manipulate the states of the objects in a way which cannot be achieved by processing (uncomposed) messages with the rules of the rewriting calculus. Such message combinators alter the properties of the objects involved in an arbitrary way. We are not interested in such a kind of reuse, which we consider to be dangerous, and we restrict the reuse relation "via message combinator" to message combinators which compose messages and transitions only.

Definition 8 (Message Algebra Criterion). Let Def. 5 be included. Let op_i be message combinators for $1 \leq i \leq n$ such that op_i : `Message`^{i}`->` `Message` $\in \Sigma_H$.

Sp_H inherits via m_H combined from $m_0 op_1 \ldots op_n m_n$ from Sp_A if

$$\begin{aligned}(\forall (H,S) \in Mod(Sp_H) : (\exists (A,R) \in Mod(Sp_A), \mathrm{f} \subseteq H \times A : \\ H|_\sigma = A \\ \wedge\ (H,S) \simeq_{(post(\mathrm{f}),\widetilde{pre}(\mathrm{f}))} (A,R) \\ \wedge\ (H,S) \sqsubseteq_{(post(\mathrm{fl}),\widetilde{pre}(\mathrm{fl}))} (A,R)))\end{aligned}$$

where $\mathrm{fl}(D) \supseteq \mathrm{f}(D)$ and
$\mathrm{fl}(m_H(p)) = \mathrm{fl}(m_1(p_1)) \ \ldots \ \mathrm{fl}(m_n(p_n))$ for $p_i \subseteq p$.

Let us motivate this relation. We have two different relations, a simulation and a bisimulation relation, to model the inheritance relation via message algebras. Function fl relates states of the ancestor and the heir specification, provided they consist of the same objects and the same messages, regardless of whether they are composed in the reusing specification.

The simulation relation abstracts from the message combinators and relates all states with the same objects and the same messages, regardless of whether they are part of a composed message or whether they are "simply" part of the configuration.

The bisimulation relation relates—like the simulation relation—states, which consist of the same objects and messages. But, in contrast to a simulation relation, it also takes into account that the composed messages are accepted, provided the state of the reused specification accepts the uncomposed messages.

Naturally, for this relation there is no purely syntactical criterion like for the simulation relation and thus, we only know that the bisimulation relation is a subset of the simulation relation.

7 Inheritance of Properties

Up to now we have considered reuse at the syntactical level with constructs and at the semantical level with criteria. Constructs and criteria, Maude and transition systems have all a quite operational "flavor". Now, we reason about these reuse relations and the objects and classes at the more property-oriented level of the μ-calculus and characterize the properties preserved by simulation and bisimulation relations modeling the reuse relations.

Proposition 9 (Inheritance of Properties). *Let Sp_A and Sp_H be two specifications, such that Sp_H inherits via $\mathcal{X}$ from Sp_A where $\mathcal{X}$ is one of the three reuse relations and $\rho_{(\mathcal{X},\mathcal{X}')}$ the criterion. Choose $(A,R) \in Mod(Sp_A)$ and $(H,S) \in Mod(Sp_A)$ such that $(A,R)\rho_{(\mathcal{X},\mathcal{X}')}(H,S)$.*

A property ϕ is called inheritable via $\mathcal{X}$ if $C \in |\phi|_{(A,R),I_A} \Rightarrow D \in |\phi|_{(A,R),I_H}$ for all $C \in A$, $D \in H$, $C\rho_{(\mathcal{X},\mathcal{X}')}D$. Then, the properties inheritable via the reuse relations are marked by $\surd$:

	Persistence	*State*	*Synchronization*	*Answer-Messages*	*State-Change*
Inheritance			$\surd$		
Subconfiguration	$\surd$	$\surd$		$\surd$	$\surd$
MessageAlgebra ($\sqsubseteq$)	$\surd$	$\surd$		$\surd$	$\surd$
MessageAlgebra ($\simeq$)	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$

Proof. (Sketch) *Persistence*, *State*, *AnswerMessages* and *StateChange* are $\langle\ \rangle$-free formulas, which are preserved by simulation relations, more precisely by the consistent dual of a concretion function. Inheritance and the simulation relation modeling reuse via Message Algebra are modeled by a simulation relation employing a Galois connection with a consistent concretion function.

Synchronization is a $[\ ]$-free, positive formula, which is preserved by simulation and bisimulation relations, as defined by Inheritance and Message Algebra.

The proof can be found in [Lec97].

Naturally, one cannot expect to inherit all properties, and, probably all proofs when reusing code. The correspondence between the operational paradigm with Maude and the transition system on the one hand and the property-oriented paradigm with μ-calculus on the other hand shows that we have not only constructs for a language but concepts which work at different levels of abstraction.

We are interested in properties for single classes, more precisely in the instances of the formula schemata. The reason for this is that those properties are the ones that are of interest in establishing the class hierarchy. More complex properties, e.g., involving many objects of different classes are more of interest, when a system is composed from different objects in a later phase in the design.

8 An Example: Buffer, BdBuffer and BdBuffer2

Let us sketch the scenario of our example first. Assume we would like to have three buffers with different properties (0) an unbounded buffer, (1) a bounded buffer like `BdBuffer` of Sect. 2 (2) a bounded buffer `BdBuffer2` that accepts `put`, `get` as well as `get2`, a message that triggers a retrieval of two elements from the bounded buffer.

Assume that we have finished the phase in the design where we have identified the classes, the objects and the messages and assume we have given the system as a Maude specification, which contains now three different, not related descriptions of the three buffers. Assume furthermore, that we would like to start with the specification of the unbounded buffer (maybe because it is implemented in the standard library) and implement the other two buffers by reusing this specification.

The specification `BUFFER` containing a class `Buffer` is the starting point.

```
module BUFFER {
  import {
    protecting (LIST)
    protecting (EXT-ACZ-CONFIGURATION) }

  signature {
    class Buffer { cont : List }

    op get _ replyto _   : ObjectId ObjectId -> Message
    op to _ answer is _  : ObjectId Elem -> Message
    op put _ into _      : Elem ObjectId -> Message }

  axioms {
    vars B R   : ObjectId
    var  E     : Elem
    var  C     : List
    var  ATTS  : Attributes

  rl [P]: (put E into B)
          < B : Buffer | cont = C  , ATTS >
       => < B : Buffer | cont = E C, ATTS > .

  rl [G]: (get B replyto R)
          < B : Buffer | cont = C E, ATTS >
       => < B : Buffer | cont = C,   ATTS >
          (to R answer is E) . } }
```

First, we implement `BdBuffer` (as it is given in Sect. 2) by reusing `Buffer`. Since the bounded buffer is more restricted to acting and to reacting than the buffer, we employ subconfiguration for reuse. We establish the relation by:

```
fl(< B : BdBuffer | max = M, cont = C, ATTS >) =
                          < B : Buffer | cont = C, ATTS >
```

We have to check that BDBUFFER simulates BUFFER. Thus, class BdBuffer can be implemented by reusing Buffer, more precisely, the value of attribute cont of BdBuffer can be replaced by an object of class Buffer. The reuse relation is BdBuffer cont : Subconfiguration of Buffer.

Let us deal with the second task, namely to implement a class BdBuffer2, which accepts put, get and get2. First we apply inheritance to let BdBuffer2 inherit put and get from BdBuffer. Hereby, fl is given by

```
fl(< B : BdBuffer2 | ATTS >) = < B : BdBuffer | ATTS >
```

Since we do not give the original specification of BdBuffer2 here, we have to assume that BdBuffer2 and BdBuffer are in the appropriate relation.

The second step is to implement get2 by subconfiguration as a sequential composition of two get messages. Assume we have a message algebra, called MSG-ALGEBRA, containing the following fragments of a specification, describing the message combinator for sequential composition ;; and its semantics in rule Seq:

```
op _ ;; _ : Message Message -> Message
vars m1 m2 n1 n2 : Message
vars c1 c2 d1 d2 h : ACZ-Configuration

crl [Seq]  (m1;;m2) c1 c2 => d1 d2 (n1;;n2)
             if (m1 c1 ==> d1 h n1) and (m2 c2 h ==> d2 n2) .
```

Then fl relates all configurations containing a get2 to configurations containing two get messages. Again, one has to check whether the appropriate simulation relations can be established. Critical here is that get2 does not provide the possibility to reach states, that are not reachable by two get messages.

Finally, we give the specification of the three buffers with their reuse relations.

```
module ALL-BUFFERS {
  import {
    protecting (BUFFER)
    protecting (MSG-ALGEBRA) }

  signature {
    class BdBuffer {                  -- A new bounded buffer
      max  : NzNat                    -- Capacity
      cont : ACZ-Configuration }      -- Encapsulating a buffer

    class BdBuffer2 [BdBuffer] { }    -- BdBuffer2 inherits
                                      -- from BdBuffer
```

```
  op get2 _ replyto _        : ObjectId ObjectId  -> Message
  op to _ answer is _ and _  : ObjectId Elem Elem -> Message }

axioms {
  vars B R      : ObjectId
  vars E E1 E2  : Elem
  var  C        : List
  var  M        : NzNat
  vars ATTS A   : Attributes

ceq [P]: (put E into B)
         < B : BdBuffer |
               ( cont = < B : Buffer | cont = C, A > ), ATTS >
      =  < B : BdBuffer | ( max = M ),
               ( cont = < B : Buffer | cont = C, A >
                        (put E into B) ),  ATTS >
         if length(C) < M .

eq [G]: (get B replyto R)
        < B : BdBuffer | ( max = M ),
            ( cont = < B : Buffer | cont = C E, A > ), ATTS >
      = < B : BdBuffer | ( max = M ),
            ( cont = < B : Buffer | cont = C E, A >
                     (get B replyto R) ), ATTS > .

eq [A]: < B : BdBuffer | ( max = M ),
            ( cont = < B : Buffer | A >
                     (to R answer is E) ), ATTS >
      = < B : BdBuffer | ( max = M ),
            ( cont = < B : Buffer | A > ), ATTS >
        (to R answer is E) .

eq [E2]: (get2 B replyto R)
         < B : BdBuffer2 | ATTS >
       = ((get B replyto R);;(get B replyto R))
         < B : BdBuffer2 | ATTS > .

eq [A2]: (to R answer is E1 and E2)
       = (to R answer is E);;(to R answer is E) . } }
```

Let us discuss this specification. In this example, the specification is not shorter than the original specification, containing the three entirely different specification with 7 rules (1 rule for each of the three buffers to implement get, 1 for each buffer for put, and 1 rule to implement get2). However, one can imagine to applying schemata, in particular, for rules describing the migration into and out of subconfigurations. This would make our reuse concepts more

effective in terms of length of code. However, establishing the class hierarchy at this abstract level of Maude is much easier than at the concrete level of a programming language, and it is more concise than it would be based on a semi-formal design notation only.

9 Related Work

Object-oriented concurrent language deal differently with the inheritance anomaly [MY93]. Languages as, e.g., $\pi o \beta \lambda$ [Jon93], do not provide inheritance at all. Other languages separate the methods from the synchronization code that decides which methods are accepted [DH97] and/or provide sophisticated constructs to reuse the synchronization code [Frø92].

The formal framework of property-preserving simulation relations [MPW93] stems from abstract interpretation [CC78,LGS+95,Bru93,SMC96]. Relations between classes that are based on the behavior respectively (behavioral) subtyping are [Ame90,HP92,PS94,Vas94]. Roles and views [AB91,ABGO93] could be expressed within our framework as well. Bisimulation relations are employed in [Jac96] as the abstraction from the constructive, algebraic, intra-object to the behavioral, coalgebraic view.

We restrict ourselves to the world of formal specifications and start with the criteria at a point in the design process where objects and classes are already specified in Maude. [WK96] integrates Maude and semi-formal object-oriented design notations.

10 Concluding Remarks

We have established a link from reuse at the syntactic level of Maude and the reuse constructs, to reuse at the semantic level and reuse at the property-oriented level. We distinguish three kinds of reuse: (1) via inheritance, (2) via subconfiguration and (3) via message algebra.

In [LLNW96], we have already explored the power of these reuse constructs. Together they are *powerful* enough to circumvent the inheritance anomaly. The upshot of our work is that the are also *safe* kinds of reuse, since we can reflect the syntactic reuse at the semantic level by an operation on the classes of algebras, which are the model of our specifications. This suggests that we do not only have constructs but concepts that work independent from the language and from the level of abstraction. Thus, our means of reuse are *adequate* both for the property-oriented level of a specification language, when one would like to achieve presumably not reuse of specification text but reuse of properties and for the concrete level of a programming language with a class hierarchy reflecting ideas and concepts and not mere reuse of code.

Acknowledgments

We are indebted to Christian Lengauer and Martin Wirsing for their support and for many fruitful discussions.

We would like to thank the anonymous referees for their helpful comments and detailed suggestions.

Funding was granted by the Grundlagenforschungsfonds of the University of St. Gallen.

Part of this work war carried out while Ulrike Lechner was a member of the Lehrstuhl für Programmierung, Fakultät für Mathematik und Informatik, Universität Passau. During that time, funding was granted by the Deutsche Forschungsgemeinschaft (Project OSIDRIS) and travel support by the ARC and the DFG.

References

[AB91] S. Abiteboul and A. Bonner. Objects and views. In *Proc. ACM SIGMOD Conference on the Management of Data*, pages 238–247. ACM, 1991.

[ABGO93] A. Albano, R. Bergamini, G. Ghelli, and R. Orsini. An object data model with roles. In *Proc. 19th International Conference on Very Large Databases (VLDB'93)*, pages 39–51, Dublin, Ireland, 1993.

[Ame90] P. America. Designing an object-oriented programming language with behavioural subtyping. In J.W. de Bakker, W.P. de Roever, and G. Rozenberg, editors, *Proc. REX/FOOLS Workshop*, Lecture Notes in Computer Science 489, pages 60–90. Springer-Verlag, 1990.

[BJR98] G. Booch, I. Jacobson, and J. Rumbaugh. *Unified Modeling Language User Guide.* Addison-Wesley, 1998.

[Bra92] J.C. Bradfield. *Verifying Temporal Properties of Systems.* Birkhäuser, 1992.

[Bru93] G. Bruns. A practical technique for process abstraction. In E. Best, editor, *4th Int. Conf. on Concurrency Theory (CONCUR'93)*, Lecture Notes in Computer Science 715, pages 37–49. Springer-Verlag, 1993.

[CC78] P. Cousot and R. Cousot. Static determination of dynamic properties of recursive procedures. In E.J. Neuhold, editor, *Proc. 2nd IFIP TC-2 Working Conf. on Formal Description of Programming Concepts*, pages 237–277. North-Holland, August 1978.

[DH97] G. Denker and P. Hartel. TROLL – An Object Oriented Formal Method for Distributed Information System Design: Syntax and Pragmatics (Version 3.0). Technical Report Informatik-Bericht 97–03, TU Braunschweig, 1997.

[FN96] K. Futatsugi and A. Nakagawa. An overview of Cafe project. In *First CafeOBJ workshop, Yokohama, Japan*, 1996. Available at: `http://ldl-www.jaist.ac.jp:8080/cafeobj/abstracts/ocp.html`.

[Frø92] 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.

[HN96] A.E. Haxthausen and F. Nickl. Pushouts of order-sorted algebraic specifications. In M. Wirsing and M. Nivat, editors, *Algebraic Methodology and Software Technology (AMAST 96)*, Lecture Notes in Computer Science 1101, pages 132–147. Springer-Verlag, 1996.

[HP92] M. Hofmann and B.C. Pierce. An abstract view of objects and subtyping. Technical Report ECS-LFCS-92-226, August, 1992.

[Jac96] B. Jacobs. Inheritance and cofree constructions. In P. Cointe, editor, *European Conf. on Object-Oriented Programming (ECOOP'96)*, Lecture Notes in Computer Science 1098, pages 210–231. Springer-Verlag, 1996.

[Jon93] C.B. Jones. Reasoning about Interference in an Object-Based Design Method. In J.C.P. Woodcock and P.G. Larsen, editors, *Industrial-Strength Formal Methods (FME'93)*, Lecture Notes in Computer Science 670, pages 1–18. Springer-Verlag, 1993.

[Koz83] D. Kozen. Results on the propositional mu-calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[Lec97] U. Lechner. *Object-Oriented Specification of Distributed Systems.* PhD thesis, University of Passau, 1997. Technical Report: MIP-9717. Available at: `www.mcm.unisg.ch/~ulechner` or `www.fmi.uni-passau.de/~lechner`.

[LGS+95] C. Loiseaux, A. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. Property preserving abstraction for the verification of concurrent systems. *Formal Methods in System Design*, 6(1):11–45, 1995.

[LLNW96] U. Lechner, C. Lengauer, F. Nickl, and M. Wirsing. (Objects + Concurrency) & Reusability – A Proposal to Circumvent the Inheritance Anomaly. In *European Conf. on Object-Oriented Programming (ECOOP'96)*, Lecture Notes in Computer Science 1098, pages 232–248. Springer-Verlag, 1996.

[Mes92] J. Meseguer. Conditional rewriting as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[Mes96] J. Meseguer. Rewriting logic as a semantic framework for concurrency: a progress report. In U. Montanari and V. Sassone, editors, *7th Int. Conf. on Concurrency Theory (CONCUR'96)*, Lecture Notes in Computer Science 1119, pages 331–372. Springer-Verlag, 1996.

[MPW93] R. Milner, J. Parrow, and D. Walker. Modal logics for mobile processes. *Theoretical Computer Science*, 25:267–310, 1993.

[MY93] 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.

[PS94] J. Palsberg and M.I. Schwartzbach. *Object-Oriented Type Systems.* Wiley, 1994.

[SMC96] B. Steffen, T. Margaria, and A. Claßen. Heterogeneous analysis and verification for distributed systems. *SOFTWARE: Concepts and Tools*, 17:13–25, 1996.

[Vas94] V.T. Vasconcelos. Typed concurrent objects. In M. Tokoro and R. Pareschi, editors, *European Conf. on Object Oriented Programming (ECOOP'94)*, Lecture Notes in Computer Science 821, pages 100–117. Springer-Verlag, 1994.

[WK96] M. Wirsing and A. Knapp. A formal approach to object-oriented software engineering. *Electronic Notes in Theoretical Computer Science*, 4:321–359, 1996. Proc. First International Workshop on Rewriting Logic and its Applications.