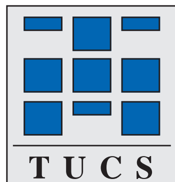
# Refining Concurrent Objects

**Martin Büchi**

Turku Centre for Computer Science,
Lemminkäisenkatu 14A, 20520 Turku, Finland
Martin.Buechi@abo.fi, http://www.abo.fi/~Martin.Buechi/

**Emil Sekerinski**

McMaster University,
1280 Main Street West, Hamilton, Ontario, Canada, L8S4K1
emil@mcmaster.ca, http://www.cas.mcmaster.ca/~emil/




**Turku Centre for Computer Science**
**TUCS Technical Report No 298**
**August 1999**
**ISBN 952-12-0509-1**
**ISSN 1239-1891**

**Abstract**

We study the notion of class refinement in a concurrent object-oriented setting. Classes, defining attributes and methods, serve as templates for creating objects. For expressing concurrency, actions are added to classes and methods with guards are considered. A class can be defined by inheriting from a given class. Class refinement is defined to support algorithmic refinement, data refinement, and atomicity refinement. Behavioral class refinement is defined in terms of trace refinement of action systems. A simulation-based proof rule for class refinement using a refinement relation is given. The special case of atomicity refinement by early returns is considered. The use of the class refinement rule is illustrated by examples.

**Keywords:** class refinement, concurrent objects, action-based concurrency, algorithmic/data/atomicity refinement, trace semantics, simulation relation

**TUCS Research Group**
Programming Methodology Research Group

# 1 Introduction

For the development of larger programs, a recommended practice is to separate a concise but precise specification of what the program should do from a possibly involved and detailed implementation. We view the specification as an abstract program $P$ and the implementation as a concrete program $Q$. The task of ensuring that the implementation is correct with respect to its specification is eased by introducing intermediate steps such that each step is a *refinement* of the previous, formally expressed as:

$$P = P_0 \sqsubseteq P_1 \sqsubseteq P_2 \sqsubseteq \ldots \sqsubseteq P_n = Q$$

In the development of sequential programs, *algorithmic refinement* steps replace abstract (or more abstract) statements by concrete (or more concrete) statements whereas *data refinement* steps replace abstract (or more abstract) data structures by concrete (or more concrete) data structures. In the development of concurrent programs, *atomicity refinement* steps replace sequential (or less concurrent) parts by concurrent (or more concurrent) ones.

These general principles are applied here to classes. For example, a file can be specified as an object of a class whose state is a sequence and a current position and whose read and write operations access the sequence at the current position. A typical implementation of this class would use pointers and blocks for storage and would process write operations in the background, hence changing the state space and introducing concurrency. Similarly, displaying (complex) graphical data on the screen may be specified by an operation that modifies the screen bitmap accordingly in one step. An implementation could process the update in small steps in the background, perhaps allowing other commands to be accepted, including those which may cause the update to be aborted. In both examples, the illusion to the user of the operations is maintained that the operations are executed *atomically*. In both examples, concurrency is introduced in the implementation for allowing a better utilization of resources, which is an aspect we are concerned about without formalizing it.

In this paper we propose a formal model for objects with (private) attributes and (public) methods, with self- and super-calls in methods, classes with inheritance, and action-based concurrency. Object have (private) actions which, as long as they are enabled, may execute and change the object's state while other parts of the program are in progress. Classes serve as templates for creating objects and inheritance is understood as a mechanism for modifying classes.

The notion of class refinement expresses that an object of the refining class behaves as an object of the refined class. Class refinement between two classes is defined in terms of the observable traces of objects of those classes. We give a simulation condition for establishing class refinement by using a relation between the attributes of those classes. As the main result, we prove that simulation by relation implies class refinement.

The proposed class refinement extends class refinement as defined for sequential objects [25, 24] by adding actions to classes. Class refinement has also been studied under the name behavioral subtyping in less formal settings guaranteeing only partial correctness by America [2] and by Liskov and Wing [22]. Different models for classes and objects have been proposed [1]. We extend the model of classes as self-referential structures with a delayed taking of the fixpoint of [27, 14].

The action system model for parallel, distributed, and reactive systems was proposed by Back and Kurki-Suonio [6, 7]. The same basic approach has later been used in other models for distributed computing, notably UNITY [12] and TLA [19].

Back and Sere [8] have added procedures to action systems. They, as well as Sere and Waldén [26] and Bonsangue et al [11], have also studied input/output refinement of action systems with methods, which is similar to our classes after self- and super-references have been resolved. Using trace refinement, we extend those results to reactive behavior and handle non-terminating systems.

The action system model has been extended with different notions of objects. Järvinen and Kurki-Suonio [16] used aggregation rather than inheritance and overriding, based their semantics on TLA, and concentrated on superposition refinement. Back et al [5] concentrated on the design of a language. Bonsangue et al [11] developed a less formal model with an action-system-per-object semantics.

Atomicity refinement has first been proposed by Lipton [21]. Back studied input/output behavior preserving atomicity refinement in action systems [3, 4]. Sere and Waldén [26] and Bonsangue et al [11] have extended this to procedures and methods, still refining only input/output behavior. Lamport and Schneider [20] and Cohen and Lamport [13] have studied atomicity refinement in TLA considering liveness properties beyond termination. de Bakker and de Vink [15] give an overview of atomicity refinement in process algebras and Petri nets. The idea of an early return, or release, statement has been proposed by Jones [17, 18] in a framework with explicit constructs for parallelism.

**Outline.** In Section 2 we review the fundamentals of statements and action systems. A point we like to make in the presentation is the (in-) dependence of various aspects. Section 3 introduces classes with attributes, methods, and actions as well as local object creation, inheritance, and self- and super-references in methods and actions. Section 4 defines class refinement, gives a condition for class simulation using a relation, and proves that class simulation implies class refinement. Class refinement is independent of how the classes are constructed by inheritance. Section 5 introduces dynamic object structures, which are necessary for allowing objects to run concurrently. In Section 6 we study early returns as a special case of atomicity refinement. Finally, Section 7 draws the conclusions.

## 2 Statements and Action Systems

We review the fundamentals of statement defined by predicate transformers following [10] and of action systems following [9].

**Predicate Transformers.** State predicates of type $P\Sigma$ are functions from elements of type $\Sigma$ to *Bool*. Relations of type $\Delta \leftrightarrow \Omega$ are functions from $\Delta$ to predicates over $\Omega$. Predicate transformers of type $\Delta \mapsto \Omega$ are functions from predicates over $\Omega$ (the postconditions) to predicates over $\Delta$ (the preconditions):

$$\begin{array}{lcl} P\Sigma & \widehat{=} & \Sigma \to Bool \\ \Delta \leftrightarrow \Omega & \widehat{=} & \Delta \to P\Omega \\ \Delta \mapsto \Omega & \widehat{=} & P\Omega \to P\Delta \end{array}$$

On predicates, conjunction $\wedge$, disjunction $\vee$, implication $\Rightarrow$, and negation $\neg$ are defined by the pointwise extension of the corresponding operations on *Bool*. The entailment ordering $\leq$ is defined by universal implication. The predicates *true* and *false* represent the universally true, respectively false predicates. On relations, we use union $\cup$, intersection $\cap$, relational composition $\circ$, and the relational image $R\,[p]$ of a predicate $p$, defined by $R\,[p]\ y \ \widehat{=}\ (\exists x \bullet R\ x\ y \wedge p\ x)$. The identity relation is denoted by *Id*. Statements are defined by predicate transformers because only their input/output behavior is of interest. Thus, for statement $S$ and predicate $q$ we have $S\ q = wp(S,q)$, where *wp* is in Dijkstra's notation the weakest precondition of statement $S$ to establish postcondition $q$. More precisely, we identify program statements with monotonic predicate transformers, i.e. predicate transformers $S$ for which $p \leq q \Rightarrow S\ p \leq S\ q$.

The sequential composition of predicate transformers $S$ and $T$ is defined by their functional composition:

$$(S\,;\,T)\ q \ \widehat{=}\ S\ (T\ q)$$

The identity on predicate transformers is denoted by *skip*. The guard $[p]$ skips if $p$ holds and established "miraculously" any postcondition if $p$ does not hold (by blocking execution). The (always blocking) guard $[false]$ is called *magic*. The assertion $\{p\}$ skips if $p$ holds and established no postcondition if $p$ does not hold (the system crashes). The (never holding) assertion $\{false\}$ is called *abort*:

$$\begin{array}{lclclcl} skip\ q & \widehat{=} & q & \quad & [p]\ q & \widehat{=} & p \Rightarrow q \\ magic\ q & \widehat{=} & true & & \{p\}\ q & \widehat{=} & p \wedge q \\ abort\ q & \widehat{=} & false & & & & \end{array}$$

The demonic (nondeterministic) choice $\sqcap$ establishes a postcondition only if both alternatives do. The angelic choice $\sqcup$ establishes a certain postcondition if already one alternative does. The relational updates $[R]$ and $\{R\}$ both update the state according to relation $R$. If several final states are possible, then $[R]$ chooses one

demonically and $\{R\}$ chooses one angelically. If $R$ is of type $\Delta \leftrightarrow \Omega$, then $[R]$ and $\{R\}$ are of type $\Delta \mapsto \Omega$:

$$\begin{array}{lclclcl} (S \sqcap T)\ q & \mathrel{\widehat{=}} & (S\ q) \wedge (T\ q) & \quad & [R]\ q\ \delta & \mathrel{\widehat{=}} & (\forall \omega \bullet R\ \delta\ \omega \Rightarrow q\ \omega) \\ (S \sqcup T)\ q & \mathrel{\widehat{=}} & (S\ q) \vee (T\ q) & \quad & \{R\}\ q\ \delta & \mathrel{\widehat{=}} & (\exists \omega \bullet R\ \delta\ \omega \wedge q\ \omega) \end{array}$$

All of the above constructs are monotonic. The universally and the positively conjunctive predicate transformers are two important subsets of the monotonic predicate transformers. Let $q_i$ for some index set $I$ be a set of predicates. If

$$S(\forall i \in I \bullet q_i) = (\forall i \in I \bullet S\ q_i)$$

holds for any index set $I$, then $S$ is universally conjunctive. If the condition holds for nonempty sets $I$ then $S$ is positively conjunctive. Any universally conjunctive predicate transformer is equal to $[R]$ for some relation $R$. Any positively conjunctive predicate transformer is equal to $\{p\}\ ;\ [R]$ for some predicate $p$ and some relation $R$.

Other statements can be defined in terms of the above ones, for example the guarded statement $p \rightarrow S \mathrel{\widehat{=}} [p]\ ;\ S$ and the conditional:

$$\textbf{if}\ p\ \textbf{then}\ S\ \textbf{else}\ T\ \textbf{end} \mathrel{\widehat{=}} (p \rightarrow S) \sqcap (\neg p \rightarrow T)$$

The enabledness domain (guard) of a statement $S$ is denoted by $grd\ S$ and its termination domain by $trm\ S$:

$$grd\ S \mathrel{\widehat{=}} \neg S\ false \qquad trm\ S \mathrel{\widehat{=}} S\ true$$

For example, $grd\ (p \rightarrow S) = p \wedge grd\ S$ and $trm\ (\{p\}\ ;\ [R]) = p$.

**Refinement.** The reflexive and transitive refinement ordering $\sqsubseteq$ is defined by universal entailment:

$$S \sqsubseteq T \mathrel{\widehat{=}} \forall q \bullet S\ q \leq T\ q$$

The loop $\textbf{do}\ S\ \textbf{od}$ executes its body as long as it is enabled. This is defined by taking the least fixed point of the function $F = \lambda X \bullet S\ ;\ X \sqcap [\neg\ grd\ S]$. Sequential composition and nondeterministic choice are monotonic in both operands, so a least fixed point $\mu\ F$ exists and is unique:

$$\textbf{do}\ S\ \textbf{od} \mathrel{\widehat{=}} \mu\ X \bullet S\ ;\ X \sqcap [\neg\ grd\ S]$$

The loop $\textbf{while}\ p\ \textbf{do}\ B$ is defined as $\textbf{do}\ p \rightarrow B\ \textbf{od}$, provided that $B$ is always enabled, i.e. $grd\ B = true$.

Data refinement $S \sqsubseteq_R S'$ generalizes (plain) algorithmic refinement by relating the initial and final state spaces of $S : \Sigma \mapsto \Sigma$ and $S' : \Sigma' \mapsto \Sigma'$ with a relation $R : \Sigma \leftrightarrow \Sigma'$:

$$S \sqsubseteq_R S' \mathrel{\widehat{=}} S\ ;\ [R] \sqsubseteq [R]\ ;\ S'$$

Data refinement $S \sqsubseteq_R S'$ can be equivalently defined by $\{Q\}\ ;\ S \sqsubseteq S\ ;\ \{Q\}$. Algorithmic refinement is a special case of data refinement with the identity relation.

**Program Variables.** Typically the state space is made up of a number of program variables. Thus the state space is of the form $\Gamma_1 \times \ldots \times \Gamma_n$. States are tuples $(x_1, \ldots, x_n)$. The variable names serve for selecting components of the state. For example, if $x : \Gamma$ and $y : \Delta$ are the only program variables, then the assignment $x := e$ updates $x$ and leaves $y$ unchanged:

$$x := e \ \hat{=}\ [R] \quad \text{where} \quad R\ (x,y)\ (x',y') \equiv x' = e \wedge y' = y$$

The nondeterministic assignment $x :\in q$ assigns $x$ an arbitrary element of the set $q$:

$$x :\in q \ \hat{=}\ [R] \quad \text{where} \quad R\ (x,y)\ (x',y') \equiv x' \in q \wedge y' = y$$

The declaration of a local variable $y : \Delta$ with initialization predicate $y_0$ extends the state space and sets $y$ to any value for which $y_0\ y$ holds. A block construct allows us to temporarily extend the state space with local variables, execute the body of the block on the extended state space, and reduce the state space again:

$$\begin{array}{lcl} \mathbf{var}\ y \mid y_0 \bullet S & \hat{=} & \mathbf{enter}\ y \mid y_0\ ;\ S\ ;\ \mathbf{exit}\ y \\ \mathbf{enter}\ y \mid y_0 & \hat{=} & [R] \quad \text{where} \quad R\ x\ (x',y') \equiv x = x' \wedge y_0\ y' \\ \mathbf{exit}\ y & \hat{=} & [R] \quad \text{where} \quad R\ (x,y)\ x' \equiv x = x' \end{array}$$

Leaving out the initialization predicate as in $\mathbf{var}\ y \bullet S$ means initializing the variable arbitrarily, $\mathbf{var}\ y \mid \mathit{true} \bullet S$. Where necessary, we also explicitly indicate the type $\Delta$ of the new variable as in $\mathbf{var}\ y : \Delta$. Since $\Gamma \times (\Delta \times \Omega)$ is isomorphic to $(\Gamma \times \Delta) \times \Omega$, we can always find functions which transform an expression of one to the other type. Hence we simply write $\Gamma \times \Delta \times \Omega$. For example, if $\Gamma = \Gamma_1 \times \cdots \times \Gamma_n$ then $S$ above would have the type $\Gamma_1 \times \cdots \times \Gamma_n \times \Delta \mapsto \Gamma_1 \times \cdots \times \Gamma_n \times \Delta$.

**Product Statements.** For predicates $q_1 : P\Sigma_1$ and $q_2 : P\Sigma_2$ the product $q_1 \times q_2$ of type $P(\Sigma_1 \times \Sigma_2)$ is defined as $(q_1 \times q_2)\ (\sigma_1, \sigma_2)\ \hat{=}\ q_1\ \sigma_1 \wedge q_2\ \sigma_2$. For predicate transformers $S_1 : \Delta_1 \mapsto \Omega_1$ and $S_2 : \Delta_2 \mapsto \Omega_2$, their product $S_1 \times S_2$ is a predicate transformer of type $\Delta_1 \times \Delta_2 \mapsto \Omega_1 \times \Omega_2$ which corresponds to the simultaneous execution of $S_1$ and $S_2$:

$$(S_1 \times S_2)\ q\ \hat{=}\ \exists q_1, q_2 \mid q_1 \times q_2 \leq q \bullet S_1\ q_1 \times S_2\ q_2$$

Two statements $S$ and $T$ over the same state space are *independent* if they operate on different components of the state space (disjoint variables). This is expressed by stating that there must exist $S', T'$ such that $S = S' \times \mathit{skip}$ and $T = \mathit{skip} \times T'$. If $R$ is a relation we say that $R$ is independent of $S$ if $[R]$ and $S$ are independent, or equivalently $\{R\}$ and $S$ are independent. If $R$ and $Q$ are independent of $S$ we have following subcommutativity properties:

$$S\ ;\ [R] \sqsubseteq [R]\ ;\ S \qquad \{Q\}\ ;\ S \sqsubseteq S\ ;\ \{Q\}$$

For simplicity and readability, we usually omit the natural extensions of predicates by *true* and of statements by *skip* when operating on an extended state space.

**Procedures.** Declaration of a procedure $p$ with value parameters $v : \Delta$, result parameters $r : \Omega$, and body $S$, written

$$\textbf{procedure}\ p(\ \textbf{val}\ v : \Delta,\ \textbf{res}\ r : \Omega)\ \textbf{is}\ S$$

defines $p$ to stand for $S$ of type $\Gamma \times \Delta \times \Omega \mapsto \Gamma \times \Delta \times \Omega$, if $\Gamma$ is the type of the global variables. A procedure call $p(e,x)$ extends the state space by the value and result parameters, sets the value parameters to $e$, executes the procedure body, sets the result parameter $x$, and removes the parameters:

$$p(e,x)\ \widehat{=}\ \ \textbf{var}\ v,r \bullet v := e\ ;\ p\ ;\ x := r$$

Now suppose that $p$ is a recursive procedure, which is expressed by assuming that $S$ is of the form $s\ p$ for some $s$. That is, $S$ has a free occurrence of $p$. The meaning of $p$ is then given by taking the least fixed point of the function $s$, i.e. the least solution of $\lambda X \bullet X = s\ X$. Statements form a complete lattice with the refinement ordering. Furthermore, we assume that $s$ is defined with $p$ occurring in monotonic positions only. These two conditions guarantee that the least fixed point $\mu\ s$ of $s$ exists and is unique. Hence we can define $p\ \widehat{=}\ \mu\ s$.

A set of mutually recursive procedures is defined by taking the fixpoint of statement tuples. For tuples $(s_1,\dots,s_n)$ and $(s'_1,\dots,s'_n)$, where $s_i$ and $s'_i$ are statements of the same type, the refinement ordering is defined elementwise:

$$(s_1,\dots,s_n) \sqsubseteq (s'_1,\dots,s'_n)\ \widehat{=}\ (s_1 \sqsubseteq s'_1) \wedge \dots \wedge (s_n \sqsubseteq s'_n)$$

Statement tuples also form a complete lattice with the refinement ordering. Let $p$ stand for the tuple $(p_1,\dots,p_n)$, assume that $S_1 = s_1\ p,\dots,S_n = s_n\ p$, and let $s$ stand for $\lambda p \bullet (s_1\ p,\dots,s_n\ p)$. The set of procedure declarations

$$\textbf{procedure}\ p_1\ \textbf{is}\ S_1,\dots,\ \textbf{procedure}\ p_n\ \textbf{is}\ S_n$$

defines $p$ to be the least fixpoint of $s$, i.e. $p\ \widehat{=}\ \mu\, s$. Assuming again that all $p_i$ occur only in monotonic positions in all $s_j$, a least fixed point exists and is unique.

**Action Systems.** Statements modeled as predicate transformers can express only atomic computations. In concurrent programs, components of the program interact during the computation. For reactive systems, the possible sequences of observable states rather than the input/output behavior are of interest. Such components can be modeled by action systems. Action systems have local variables and a body that is repeatedly executed as long as it is enabled. Action systems can represent terminating, non-terminating, and aborting computations. Formally an action system is a pair $AS = (a_0, A)$ where $a_0 : \mathcal{P}\Sigma$ is the initializing predicate of the local state. Upon initialization, arbitrary values satisfying $a_0$ are chosen for the local variables. The global state space $\Gamma$ is declared and initialized outside. Action $A : \Gamma \times \Sigma \mapsto \Gamma \times \Sigma$ is a positively conjunctive statement, which acts on the local state of type $\Sigma$ and global state of type $\Gamma$. Because $A$ is positively conjunctive, it can be written as $\{p\}\ ;\ [R]$.

The next relation of $A$ relates an initial state $(u,v)$ in both the enabledness domain and termination domain to all possible next states $(u',v')$:

$$nxt\,A\ (u,v)\ (u',v')\ \widehat{=}\ p\ (u,v) \wedge R\ (u,v)\ (u',v')$$

A behavior of $A$ is a sequence of pairs

$$s = \langle (u_0,v_0),(u_1,v_1),\ldots \rangle$$

where $v_0$ is the initial value of the local state, such that $a_0\ v_0$, and all consecutive elements of the sequence are in the next relation:

$$nxt\,A\ (u_i,v_i)\ (u_{i+1},v_{i+1})$$

The set $beh\,AS$ is the set of all behaviors. A behavior is terminating if it is finite and for the last element $(u_n,v_n)$ the action $A$ is not enabled, $\neg\, grd\,A(u_n,v_n)$. A behavior is aborting if it is finite and for the last element $(u_n,v_n)$ the action aborts, i.e. $(u_n,v_n)$ is not in the termination domain, $\neg\, trm\,A(u_n,v_n)$. A behavior is non-terminating if it is not of finite length. The set $beh\,A$ can be thought of as the (disjoint) union of terminating, aborting, and non-terminating behaviors of $AS$.

Action systems are typically composed of a set of actions $A_1,\ldots,A_n$ operating on different parts of the state space. In the interleaving model, parallelism of two actions is modeled by taking them in arbitrary, demonically chosen order. Hence the meaning of such an action system is given by taking the nondeterministic choice between all actions, $AS = (a_0,A)$ where $A = A_1 \sqcap \ldots \sqcap A_n$. We use the following syntax for an action system with local variables named $a$:

$$AS = \mathbf{var}\ a \mid a_0 \bullet \mathbf{do}\ A_1\ [\!]\ \ldots\ [\!]\ A_n\ \mathbf{od}$$

**Parallel Composition.** The parallel composition of action systems $AS = (a_0,A)$ and $BS = (b_0,B)$ with the same global state space merges the local state spaces (possibly renaming variables to make them mutually distinct) and combines the actions by nondeterministic choice:

$$AS \parallel BS\ \widehat{=}\ (a_0 \wedge b_0, A \sqcap B)$$

This models an arbitrary interleaving of the action of $AS$ and $BS$ without any assumption of fairness. As $grd\ (A \sqcap B) = grd\,A \vee grd\,B$, the combined system terminates only if both $A$ and $B$ are not enabled. As $trm\ (A \sqcap B) = trm\,A \wedge trm\,B$, the combined action system aborts if either $A$ or $B$ aborts. Parallel composition is commutative and associative, up to the order of state components.

We have omitted the explicit state space reordering and the natural extensions by $skip$ for $A$ and $B$ to operate on the global state space and their respective local state space in $A \sqcap B$.

Given an action system $AS$, we can make part of its global state space local by $\mathbf{var}\ b \mid b_0 \bullet AS$ (typically for hiding common variables of two action systems composed in parallel). If $a$ and $b$ are disjoint then:

$$\mathbf{var}\ b \mid b_0 \bullet \mathbf{var}\ a \mid a_0 \bullet \mathbf{do}\ A\ \mathbf{od}\ \widehat{=}\ \mathbf{var}\ a,b \mid a_0 \wedge b_0 \bullet \mathbf{do}\ A\ \mathbf{od}$$

**Trace Refinement.** Behaviors contain a local state component, which is not observable from outside. Furthermore, behaviors may contain stuttering steps which are not observable from the outside either. A state $(u_{i+1}, v_{i+1})$ is a stuttering state if $u_i = u_{i+1}$. Traces on the other hand capture only the observable part of behaviors. For a behavior $s$, its trace $tr\ s$ is obtained by

1. removing all finite stuttering states from $s$, and
2. removing the local state component from all states in $s$.

Behavior $s$ approximates behavior $t$, written $s \preceq t$, if either

- $s$ is aborting and $tr\ s$ is a prefix of $tr\ t$, or
- neither $s$ nor $t$ are aborting and $tr\ s = tr\ t$.

Trace refinement between action systems $AS$ and $AS'$ with same global state space holds if all behaviors of $AS'$ have an approximating behavior of $AS$:

$$AS \overset{\text{tr}}{\sqsubseteq} AS' \ \widehat{=}\ \forall t \in beh\, AS' \bullet \exists s \in beh\, AS \bullet s \preceq t$$

Since only finite stuttering is removed, an infinite behavior gives rise to an infinite trace and an a finite behavior gives rise to a finite trace. Both "concrete stuttering" in $AS'$ as well as "abstract stuttering" in $AS$ is allowed.

**Simulation.** Trace refinement can be shown to hold by simulation. Here we consider forward simulation between $AS = (a_0, A)$ and $AS' = (a'_0, A')$ with the same global state space using a relation $R$. An action $A_\natural$ is a stuttering action if it always terminates and it leaves the global state unchanged:

$$trm\, A_\natural = true \qquad \text{and} \qquad nxt\, A_\natural\ (u,v)\ (u',v') \Rightarrow u = u'$$

Let $S^n$ be the $n$-fold sequential composition of statement $S$, defined by $S^0 = skip$ and $S^{n+1} = S\, ;\, S^n$. Let $S^*$ stand for the nondeterministic choice between all $n$-fold sequential compositions of $S$, defined by $S^* = (\sqcap\ i \in Nat \bullet S^n)$. Define $A_0 =$ **enter** $a \mid a_0$ and $A'_0 =$ **enter** $a' \mid a'_0$. Action system $AS$ is forward simulated by $AS'$ using $R$, written $AS \sqsubseteq_R AS'$, if there are decompositions $A = A_\sharp \sqcap A_\natural$ and $A' = A'_\sharp \sqcap A'_\natural$ such that $A_\natural$ and $A'_\natural$ are stuttering actions and

| | | |
|---|---|---|
| (a) | Initialization: | $A_0\, ;\, A_\natural^*\, ;\, [R] \sqsubseteq A'_0\, ;\, A_\natural'^*$ |
| (b) | Actions: | $A_\sharp\, ;\, A_\natural^*\, ;\, [R] \sqsubseteq [R]\, ;\, A'_\sharp\, ;\, A_\natural'^*$ |
| (c) | Exit Condition: | $R[trm\, A \wedge grd\, A] \leq grd\, A'$ |
| (d) | Internal Convergence: | $R[trm\, A \wedge trm\ (\mathbf{do}\, A_\natural\ \mathbf{od})] \leq trm\ (\mathbf{do}\, A'_\natural\ \mathbf{od})$ |

**Theorem 1** *Let $AS$ and $AS'$ be action systems and $R$ a relation. Then:*

$$AS \sqsubseteq_R AS' \Rightarrow AS \overset{\text{tr}}{\sqsubseteq} AS'$$

In general, action system refinement is not compositional in the sense that refining one action system would lead to a refinement in an environment with other action systems running in parallel. However, we get compositionality under the additional constraint of *non-interference*. Let $BS = (b_0, B)$ be an action system and let $R$ be refinement relation for $AS$. Action $B$ does not interfere with $R$ if

$$trm\ B \wedge r \leq B\ r$$

where $r\ u\ b = R\ (u,a)\ (u,a')$. In other words, $r$ is an invariant of $B$.

**Theorem 2** *Let AS, AS′, and BS be action systems, let R be a relation, and assume $BS = (b_0, B)$. If B does not interfere with R then:*

$$AS \sqsubseteq_R AS' \Rightarrow AS \parallel BS \overset{\text{tr}}{\sqsubseteq} AS' \parallel BS$$

# 3 Objects and Classes

Conventionally, a class is a template that defines a set of attributes and methods. Methods of a class may contain self-references to the method itself and other methods of the class. Instantiating a class creates a new object with initialized attributes and method bodies as defined by the class. A subclass inherits attributes and methods from its superclass, possibly adding new attributes and methods and overwriting inherited methods. Methods in a subclass may contain super-references to methods in the superclass. Formally, classes are modeled as self-referential recursive structures, where self-references are not resolved at the time the class is declared, but resolving is delayed until objects are created [27]. These principles are extended here: classes define additionally a set of actions, which are inherited and can be overwritten and extended in subclasses. Self-references are then possible between both methods and actions, and are resolved only at the time when objects are created. Also, both methods and actions may contain super-references to methods and actions in a superclass. This implies that for this purpose actions are given names.

Let $\Sigma$ the type of the attributes of some class $C$ and let $\alpha$ be a type variable to be instantiated by the type of the global variables and possibly by the type of further attributes of subclasses. Typically, there are a number of attributes and global variables, so elements of the types will be tuples and the variable and attribute names are used for accessing the corresponding components. The set of methods and actions of a class are also modeled by a tuple with the method and action name accessing the corresponding component. For the types of methods $m_i$ and actions $a_j$ of $C$ we define

$$\begin{aligned} CM_i &= \alpha \times \Sigma \times \Delta_i \times \Omega_i \mapsto \alpha \times \Sigma \times \Delta_i \times \Omega_i \\ CA &= \alpha \times \Sigma \mapsto \alpha \times \Sigma \end{aligned}$$

where $\Delta_i$ and $\Omega_i$ are the types of the value and result parameters of method $m_i$, respectively. Within a class, methods $m_i$ and actions $a_j$ of that class can be referred to

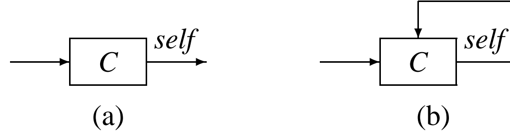



Figure 1: Illustration of (a) class $C$ and of (b) taking the fixpoint of $C$. The incoming arrow represents calls to $C$, the outgoing arrow stands for self-calls of $C$.

by $self.m_i$ and $self.a_j$, respectively. This is formalized by having $self.m_i$ and $self.a_j$ as parameters of all methods and actions, allowing all methods and actions to be referred to by all methods and actions, a generalization whose usefulness becomes clearer when considering inheritance. Let $self$ stand for the tuple of method and action names prefixed by $self$:

$$self = (self.m_1, \ldots, self.m_m, self.a_1, \ldots, self.a_a)$$

The collection of all methods and actions of a class can then be expressed as a tuple $cs$ parameterized with $self$,

$$cs = \lambda self \bullet (cm_1, \ldots, cm_m, ca_1, \ldots, ca_a)$$

where $cm_i : CM_i$, $ca_j : CA$, $self.m_i : CM_i$, and $self.a_j : CA$. A class does also specify possible initial values $c_0 : \mathcal{P}\Sigma$ of its attributes $c$. Hence a class $C$ takes the form of a tuple:

$$C = (c_0, cs)$$

Note that $self$ is here used to refer to methods and actions, but not to reference attributes (fields) of an object. Figure 1(a) illustrates the definition of a class. For defining class $C$ with attributes, methods, and actions as above we use the syntax:

**class** $C$
  **attr** $c \mid c_0$,
  **meth** $m_1($ **val** $v_1$, **res** $r_1)$ **is** $cm_1$,
  $\ldots$,
  **meth** $m_m($ **val** $v_m$, **res** $r_m)$ **is** $cm_m$,
  **action** $a_1$ **is** $ca_1$,
  $\ldots$,
  **action** $a_a$ **is** $ca_a$
**end**

Objects have all self-calls resolved with methods of the object itself. Self-calls may be mutually recursive, like mutually recursive procedures. Modeling this formally amounts to taking the least fixed point of the function $cs$ (Figure 1(b)). Methods and actions of objects of class $C$, denoted by $C.m_i$ and $C.a_i$, respectively, are

defined by taking the fixpoint of the tuple of all methods and actions and then selecting the corresponding method and action, respectively:

$$C.m_i \mathrel{\widehat{=}} (\mu\, cs).m_i \qquad C.a_i \mathrel{\widehat{=}} (\mu\, cs).a_i$$

Declaring a variable $x$ to be of class $C$ means declaring it to be of type $\Sigma$ and initializing it with $c_0$:

$$\mathbf{var}\, x : C \bullet S \mathrel{\widehat{=}} \mathbf{var}\, x \mid c_0 \bullet S$$

Such a variable corresponds to a local, stack allocated object in programming languages. Since actions cannot access variables which are local to some statements, concurrency cannot be expressed this way. For this purpose dynamic object structures are introduced later.

A method call $x.m_i$ of object $x$ of class $C$ corresponds to a procedure call with $x$ as a value-result parameter.

$$x.m_i \mathrel{\widehat{=}} \mathbf{var}\, c \bullet c := x\, ;\, C.m_i\, ;\, x := c$$

The name of the formal parameter is that of the attributes, namely $c$. Therefore, $c$ is used to access local data in the body of $C.m_i$. This corresponds to *this* in some programming languages.

Additional value and result parameters are treated like for procedure calls. For convenience, we also use the same notation for selecting an action of an object:

$$x.a_i \mathrel{\widehat{=}} C.a_i$$

Inheritance is expressed by the application of a modifier to a base class: If $D$ inherits from $C$, then $D$ is equivalent to $L$ **mod** $C$, where modifier $L$ corresponds to the extending part of the definition of $D$. This model of single inheritance is equivalent to dynamic method lookups along the inheritance structure as implemented in object-oriented languages [14]. We call $C$ the superclass of $D$ and $D$ a subclass of $C$.

Let $C$ be as above. A modifier $L$ specifies additional attributes, say $l$ of type $\Lambda$. We consider only modifiers which redefine all methods of the base class. If a method should remain unchanged, this is expressed by making a supercall to the same method of the base class. A modifier also redefines all actions of the base class and possibly adds new actions. For the types of methods $m_i$ and actions $a_j$ of $L$ we define

$$\begin{aligned} LM_i &= \beta \times \Lambda \times \Sigma \times \Delta_i \times \Omega_i \mapsto \beta \times \Lambda \times \Sigma \times \Delta_i \times \Omega_i \\ LA &= \beta \times \Lambda \times \Sigma \mapsto \beta \times \Lambda \times \Sigma \end{aligned}$$

where $\beta$ is the type variable for further attributes in subclasses. Thus, we instantiate $\alpha$ of $CM_i$ and $CA$ by $\beta \times \Lambda$. The types of the value and result parameters of method $m_i$ are, exactly as in $C$, $\Delta_i$ and $\Omega_i$. Within $L$, methods $m_i$ and actions $a_j$ of that class can be referred to by $self.m_h$ and $self.a_k$, and those of the superclass $C$ by $super.m_i$

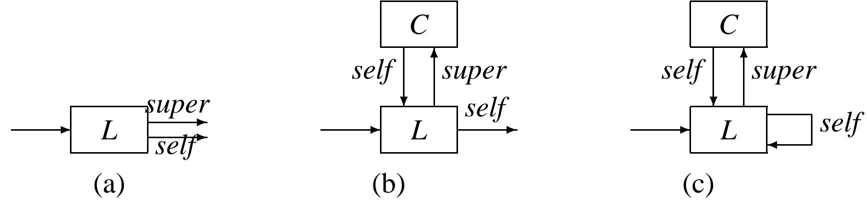



Figure 2: Illustration of (a) modifier $L$, of (b) $L$ **mod** $C$, and of (c) taking the fix-point of $L$ **mod** $C$

and $super.a_j$, respectively. This is formalized by having $self.m_h, self.a_k, super.m_i$, and $super.a_j$ as parameters of all methods and actions:

$$\begin{array}{lcl} self & = & (self.m_1, \ldots, self.m_m, self.a_1, \ldots, self.a_b) \\ super & = & (super.m_1, \ldots, super.m_m, super.a_1, \ldots, super.a_a) \end{array}$$

The collection of all methods and actions of modifier $L$ can then be expressed as a tuple $ls$ parameterized with both $self$ and $super$,

$$ls = \lambda self \bullet \lambda super \bullet (lm_1, \ldots, lm_m, la_1, \ldots, la_b)$$

where $lm_k : LM_k$, $la_h : LA$, $self.m_h : LM_h$, $self.a_k : LA$, $super.m_i : CM_i$, and $super.a_j : CA$. A modifier also specifies initial values $l_0 : \Lambda$ of the new attributes $l$. Hence a modifier $L$ takes the form of a tuple:

$$L = (l_0, ls)$$

For defining modifier $L$ with attributes, methods, and actions as above we use the following syntax, where unmentioned methods $m_i$ and actions $a_j$ are defined as $super.m_i$ and $super.a_j$, respectively:

**modifier** $L$
  **attr** $l \mid l_0$,
  **meth** $m_1($**val** $v_1$, **res** $r_1)$ **is** $lm_1$,
  $\ldots$,
  **meth** $m_m($**val** $v_1$, **res** $r_m)$ **is** $lm_m$,
  **action** $a_1$ **is** $la_1$,
  $\ldots$,
  **action** $a_b$ **is** $la_b$
**end**

The modification of $C$ by $L$ redirects self-calls in $C$ to $L$, redirects super-calls in $L$ to $C$, and leaves the self-calls in $L$ unresolved for possible further modification (see also Figure 2(b)):

$$L \textbf{ mod } C \;\widehat{=}\; (l_0 \wedge c_0, \lambda self \bullet ls\ self(cs\ \overline{self}))$$

Here $\overline{self} = (self.m_1, \ldots, self.m_m, self.a_1, \ldots, self.a_a)$ is identical as *self* in the definition of *cs*. For modifying some class $C$ with $L$ as above and calling the result $D$ we can also use a more conventional syntax:

$$\begin{array}{l} \textbf{class } D \\ \textbf{inherit } C \\ \quad \textbf{attr } l \mid l_0, \\ \quad \textbf{meth } m_1(\,\textbf{val } v_1, \textbf{res } r_1) \textbf{ is } lm_1, \\ \quad \ldots, \\ \quad \textbf{meth } m_m(\,\textbf{val } v_1, \textbf{res } r_m) \textbf{ is } lm_m, \\ \quad \textbf{action } a_1 \textbf{ is } la_1, \\ \quad \ldots, \\ \quad \textbf{action } a_b \textbf{ is } la_a \\ \textbf{end} \end{array}$$

# 4 Class Refinement

In this section we define class refinement in terms of trace refinement. Also, a simulation condition between classes with a relation is defined and proved to imply class refinement. The reasoning is done with a single object of a class running in isolation; dynamic object creation is considered later.

For an object $x$ of class $C$, let $\mathcal{A}[x]$ be the action system with all its actions. Thus $\mathcal{A}[x]$ specifies how $x$ behaves between external method calls to $x$:

$$\mathcal{A}[x] = \textbf{do}\, x.a_1 \;[\!]\; \ldots \;[\!]\; x.a_a \,\textbf{od}$$

Let $\mathcal{O}[x]$ be an arbitrary action system observing object $x$ only through method calls. Let $S_0, \ldots, S_m$ be universally conjunctive statements that are independent of the global state, i.e. they access only local variables $h$:

$$\mathcal{O}[x] = \textbf{var}\, h \mid h_0 \bullet \textbf{do}\, S_0 \,;\, abort \;[\!]\; S_1 \,;\, x.m_1 \;[\!]\; \ldots \;[\!]\; S_m \,;\, x.m_m \,\textbf{od}$$

Let $\mathcal{K}[C]$ be an arbitrary program operating on an object $x$ of class $C$ such that $\mathcal{K}$ is the full context of $x$, in the sense that no other program accesses $x$. Any such program can be described by an interleaving of method calls to $x$ and of actions of $x$:

$$\mathcal{K}[C] = \textbf{var}\, x : C \bullet \mathcal{O}[x] \parallel \mathcal{A}[x]$$

Class $D$ is a refinement of class $C$, written $C \overset{\text{cl}}{\sqsubseteq} D$, if using an object of class $D$ instead of $C$ in all possible programs yields a trace refinement of the original program:

$$C \overset{\text{cl}}{\sqsubseteq} D \;\widehat{=}\; \forall \mathcal{K} \bullet \mathcal{K}[C] \overset{\text{tr}}{\sqsubseteq} \mathcal{K}[D]$$

Note that class refinement of two classes is independent of how the classes are constructed using inheritance. Also, self-calls within the classes are resolved first.

This means that this definition applies independently whether inheritance and self-calls are considered.

For proving refinement between classes $C = (c_0, cs)$ and $D = (d_0, ds)$ we use a simulation with a refinement relation $R$. Define $C_0 = \textbf{enter}\ c \mid c_0$, $D_0 = \textbf{enter}\ d \mid d_0$, and:

$$CX = C.a_1 \sqcap \ldots \sqcap C.a_a \qquad \text{and} \qquad DX = D.a_1 \sqcap \ldots \sqcap D.a_b$$

Class $C$ is simulated by $D$ using $R$, written $C \sqsubseteq_R D$, if there is a decomposition $CX = CX_\sharp \sqcap CX_\natural$ and $DX = DX_\sharp \sqcap DX_\natural$ such that $CX_\natural$ and $DX_\natural$ are stuttering actions and:

| | | |
|---|---|---|
| (a) | Initialization: | $C_0\ ;\ CX_\natural^*\ ;\ [R] \sqsubseteq D_0\ ;\ DX_\natural^*$ |
| (b) | Methods: | $C.m_i\ ;\ CX_\natural^*\ ;\ [R] \sqsubseteq [R]\ ;\ D.m_i\ ;\ DX_\natural^*$ for all $m_i$ in $m_1, \ldots, m_m$ |
| (c) | Actions: | $CX_\sharp\ ;\ CX_\natural^*\ ;\ [R] \sqsubseteq [R]\ ;\ DX_\sharp\ ;\ DX_\natural^*$ |
| (d) | Method Guards: | $R[\mathit{trm}\ C.m_i \wedge \mathit{trm}\ CX \wedge \mathit{grd}\ C.m_i] \leq \mathit{grd}\ D.m_i \vee \mathit{grd}\ DX$ for all $m_i$ in $m_1, \ldots, m_m$ |
| (e) | Exit Condition: | $R[\mathit{trm}\ CX \wedge \mathit{grd}\ CX] \leq \mathit{grd}\ DX$ |
| (f) | Internal Convergence: | $R[\mathit{trm}\ CX \wedge \mathit{trm}\ (\textbf{do}\ CX_\natural\ \textbf{od})] \leq \mathit{trm}\ (\textbf{do}\ DX_\natural\ \textbf{od})$ |

**Theorem 3** *Let C and D be classes and R a relation. Then:*

$$C \sqsubseteq_R D \Rightarrow C \overset{\text{cl}}{\sqsubseteq} D$$

Proof We define:

$$\begin{array}{lcl} CY & = & (S_0\ ;\ \mathit{abort}) \sqcap (S_1\ ;\ C.m_1) \sqcap \ldots \sqcap (S_m\ ;\ C.m_m) \\ DY & = & (S_0\ ;\ \mathit{abort}) \sqcap (S_1\ ;\ D.m_1) \sqcap \ldots \sqcap (S_m\ ;\ D.m_m) \end{array}$$

We have to show that (a) to (f) above imply $K[C] \overset{\text{tr}}{\sqsubseteq} K[D]$ for any $K$, which means that for any $h_0$ and $S_i$:

$$\begin{array}{l} \textbf{var}\ h \mid h_0 \bullet \textbf{var}\ c \mid c_0 \bullet \textbf{do}\ CY\ [\!]\ CX\ \textbf{od} \overset{\text{tr}}{\sqsubseteq} \\ \quad \textbf{var}\ h \mid h_0 \bullet \textbf{var}\ d \mid d_0 \bullet \textbf{do}\ DY\ [\!]\ DX\ \textbf{od} \end{array}$$

We do so by applying Theorem 1 with $A_0 := C_0$, $A_\sharp := CY \sqcap CX_\sharp$, $A_\natural := CX_\natural$, $B_0 := D_0$, $B_\sharp := DY \sqcap DX_\sharp$, and $B_\natural := DX_\natural$ to the inner action system and get four conditions:

| | | |
|---|---|---|
| (1) | Initialization: | $C_0\ ;\ CX_\natural^*\ ;\ [R] \sqsubseteq D_0\ ;\ DX_\natural^*$ |
| (2) | Actions: | $(CY \sqcap CX_\sharp)\ ;\ CX_\natural^*\ ;\ [R] \sqsubseteq [R]\ ;\ (DY \sqcap DX_\sharp)\ ;\ DX_\natural^*$ |
| (3) | Exit Condition: | $R[\mathit{trm}\ (CY \sqcap CX) \wedge \mathit{grd}\ (CY \sqcap CX)] \leq \mathit{grd}\ (DY \sqcap DX)$ |
| (4) | Internal Convergence: | $R[\mathit{trm}\ (CY \sqcap CX) \wedge \mathit{trm}\ (\textbf{do}\ CX_\natural\ \textbf{od})] \leq \mathit{trm}\ (\textbf{do}\ DX_\natural\ \textbf{od})$ |

Condition (1) follows immediately from (a). For (2) we calculate, for any $S_i$:

$$(CY \sqcap CX_\sharp) \mathbin{;} CX_\natural^* \mathbin{;} [R] \sqsubseteq [R] \mathbin{;} (DY \sqcap DX_\sharp) \mathbin{;} DX_\natural^*$$
$$\equiv \quad \{\mathbin{;} \text{ distributes over } \sqcap\}$$
$$(CY \mathbin{;} CX_\natural^* \mathbin{;} [R]) \sqcap (CX_\sharp \mathbin{;} CX_\natural^* \mathbin{;} [R]) \sqsubseteq$$
$$([R] \mathbin{;} DY \mathbin{;} DX_\natural^*) \sqcap ([R] \mathbin{;} DX_\sharp \mathbin{;} DX_\natural^*)$$
$$\Leftarrow \quad \{\text{monotonicity}\}$$
$$(CY \mathbin{;} CX_\natural^* \mathbin{;} [R] \sqsubseteq [R] \mathbin{;} DY \mathbin{;} DX_\natural^*) \wedge$$
$$(CX_\sharp \mathbin{;} CX_\natural^* \mathbin{;} [R] \sqsubseteq [R] \mathbin{;} DX_\sharp \mathbin{;} DX_\natural^*)$$

The second conjunct follows from (c). We continue with the first conjunct:

$$CY \mathbin{;} CX_\natural^* \mathbin{;} [R] \sqsubseteq [R] \mathbin{;} DY \mathbin{;} DX_\natural^*$$
$$\equiv \quad \{\text{definition of } CY, DY \text{ and } \mathbin{;} \text{ distributes over } \sqcap\}$$
$$(S_0 \mathbin{;} abort \mathbin{;} CX_\natural^* \mathbin{;} [R]) \sqcap (S_1 \mathbin{;} C.m_1 \mathbin{;} CX_\natural^* \mathbin{;} [R]) \sqcap \ldots$$
$$\sqcap (S_m \mathbin{;} C.m_m \mathbin{;} CX_\natural^* \mathbin{;} [R]) \sqsubseteq$$
$$([R] \mathbin{;} S_0 \mathbin{;} abort \mathbin{;} DX_\natural^*) \sqcap ([R] \mathbin{;} S_1 \mathbin{;} D.m_1 \mathbin{;} DX_\natural^*) \sqcap \ldots$$
$$\sqcap ([R] \mathbin{;} S_m \mathbin{;} D.m_m \mathbin{;} DX_\natural^*)$$
$$\Leftarrow \quad \{abort \mathbin{;} S = abort \text{ and } S \mathbin{;} abort \sqsubseteq [R] \mathbin{;} S \mathbin{;} abort \text{ for independent } [R], S\}$$
$$(S_1 \mathbin{;} C.m_1 \mathbin{;} CX_\natural^* \mathbin{;} [R]) \sqcap \ldots \sqcap (S_m \mathbin{;} C.m_m \mathbin{;} CX_\natural^* \mathbin{;} [R]) \sqsubseteq$$
$$([R] \mathbin{;} S_1 \mathbin{;} D.m_1 \mathbin{;} DX_\natural^*) \sqcap \ldots \sqcap ([R] \mathbin{;} S_m \mathbin{;} D.m_m \mathbin{;} DX_\natural^*)$$
$$\Leftarrow \quad \{\text{monotonicity}\}$$
$$\forall i \in \{1, \ldots, m\} \bullet S_i \mathbin{;} C.m_i \mathbin{;} CX_\natural^* \mathbin{;} [R] \sqsubseteq [R] \mathbin{;} S_i \mathbin{;} D.m_i \mathbin{;} DX_\natural^*$$
$$\Leftarrow \quad \{\text{as } S_i \mathbin{;} [R] \sqsubseteq [R] \mathbin{;} S_i \text{ assumed}\}$$
$$\forall i \in \{1, \ldots, m\} \bullet S_i \mathbin{;} C.m_i \mathbin{;} CX_\natural^* \mathbin{;} [R] \sqsubseteq S_i \mathbin{;} [R] \mathbin{;} D.m_i \mathbin{;} DX_\natural^*$$
$$\Leftarrow \quad \{\text{monotonicity}\}$$
$$\forall i \in \{1, \ldots, m\} \bullet C.m_i \mathbin{;} CX_\natural^* \mathbin{;} [R] \sqsubseteq [R] \mathbin{;} D.m_i \mathbin{;} DX_\natural^*$$

The last line follows from (b). For (3) we calculate, for any $S_i$:

$$R[trm\ (CY \sqcap CX) \wedge grd\ (CY \sqcap CX)] \leq grd\ (DY \sqcap DX)$$
$$\equiv \quad \{\text{as } trm\ (S \sqcap T) = trm\ S \wedge trm\ T \text{ and } grd\ (S \sqcap T) = grd\ S \vee grd\ T\}$$
$$R[trm\ CY \wedge trm\ CX \wedge (grd\ CY \vee grd\ CX)] \leq grd\ DY \vee grd\ DX$$
$$\Leftarrow \quad \{\text{monotonicity}\}$$
$$(R[trm\ CY \wedge trm\ CX \wedge grd\ CY] \leq grd\ DY \vee grd\ DX) \wedge$$
$$(R[trm\ CX \wedge grd\ CX] \leq grd\ DX)$$

The second conjunct follows from (e). We continue with the first conjunct:

$$R[trm\ CY \wedge trm\ CX \wedge grd\ CY] \leq grd\ DY \vee grd\ DX$$
$$\Leftarrow \quad \{S \text{ universally conjunctive} \wedge S, T \text{ independent} \Rightarrow grd(S \mathbin{;} T) \leq grd\ T\}$$
$$R[trm\ CY \wedge trm\ CX \wedge grd\ CY] \leq$$
$$grd\ DY \vee grd\ (S_0 \mathbin{;} DX) \vee \ldots \vee grd\ (S_m \mathbin{;} DX)$$

$$
\begin{array}{ll}
\equiv & \{ grd\ (S \sqcap T) = grd\ S \vee grd\ T \text{ for any } S,T \} \\
& R[trm\ CY \wedge trm\ CX \wedge grd\ CY] \leq \\
& grd\ (DY \sqcap (S_0\ ;\ DX) \sqcap \ldots \sqcap (S_m\ ;\ DX)) \\
\equiv & \{ R[p] \leq q \equiv p \leq [R]\ q \text{ and } [R](grd\ S) = grd\ (\{R\}\ ;\ S)\ (*) \} \\
& trm\ CY \wedge trm\ CX \wedge grd\ CY \leq \\
& grd\ (\{R\}\ ;\ (DY \sqcap (S_0\ ;\ DX) \sqcap \ldots \sqcap (S_m\ ;\ DX))) \\
\equiv & \{ S\ ;\ (T \sqcap U) = (S\ ;\ T) \sqcap (S\ ;\ U) \text{ and } \\
& \quad abort \sqcap S = abort \text{ for any } S,T,U \} \\
& trm\ CY \wedge trm\ CX \wedge grd\ CY \leq grd\ ((\{R\}\ ;\ S_0\ ;\ abort) \sqcap \\
& \quad (\{R\}\ ;\ S_1\ ;\ (D.m_1 \sqcap DX)) \sqcap \ldots \sqcap (\{R\}\ ;\ S_m\ ;\ (D.m_m \sqcap DX))) \\
\Leftarrow & \{ \{R\}, S \text{ independent} \Rightarrow \{R\}\ ;\ S \sqsubseteq S\ ;\ \{R\} \text{ and } \\
& \quad T \sqsubseteq U \Rightarrow grd\ U \leq grd\ T \} \\
& trm\ CY \wedge trm\ CX \wedge grd\ CY \leq grd\ ((S_0\ ;\ \{R\}\ ;\ abort) \sqcap \\
& \quad (S_1\ ;\ \{R\}\ ;\ (D.m_1 \sqcap DX)) \sqcap \ldots \sqcap (S_m\ ;\ \{R\}\ ;\ (D.m_m \sqcap DX))) \\
\Leftarrow & \{ trm\ (S \sqcap T) = trm\ S \wedge trm\ T \text{ and } \\
& \quad grd\ (S \sqcap T) = grd\ S \vee grd\ T \text{ for any } S,T \} \\
& (trm\ (S_0\ ;\ abort) \wedge trm\ CX \wedge grd\ (S_0\ ;\ abort) \leq \\
& \quad grd\ (S_0\ ;\ \{R\}\ ;\ abort)) \wedge \\
& (\forall i \in \{1,\ldots,m\} \bullet trm\ (S_i\ ;\ C.m_i) \wedge trm\ CX \wedge grd\ (S_i\ ;\ C.m_i) \leq \\
& \quad grd\ (S_i\ ;\ \{R\}\ ;\ (D.m_i \sqcap DX))) \\
\Leftarrow & \{ \{R\}\ ;\ abort \sqsubseteq abort \text{ and } \\
& \quad T \sqsubseteq U \Rightarrow grd\ U \leq grd\ T \text{ for any } R,T,U \} \\
& \forall i \in \{1,\ldots,m\} \bullet trm\ (S_i\ ;\ C.m_i) \wedge trm\ CX \wedge grd\ (S_i\ ;\ C.m_i) \leq \\
& \quad grd\ (S_i\ ;\ \{R\}\ ;\ (D.m_i \sqcap DX)) \\
\Leftarrow & \{ S \text{ universally conjunctive} \wedge S,T \text{ independent} \Rightarrow \\
& \quad trm\ T \leq trm\ (S\ ;\ T) \} \\
& \forall i \in \{1,\ldots,m\} \bullet trm\ (S_i\ ;\ C.m_i) \wedge trm\ (S_i\ ;\ CX) \wedge grd\ (S_i\ ;\ C.m_i) \leq \\
& \quad grd\ (S_i\ ;\ \{R\}\ ;\ (D.m_i \sqcap DX)) \\
\Leftarrow & \{ trm\ (S \sqcap T) = trm\ S \wedge trm\ S \text{ for any } S,T \text{ and } \\
& \quad ;\ \text{distributes over } \sqcap \} \\
& \forall i \in \{1,\ldots,m\} \bullet trm\ (S_i\ ;\ (C.m_i \sqcap CX)) \wedge grd\ (S_i\ ;\ C.m_i) \leq \\
& \quad grd\ (S_i\ ;\ \{R\}\ ;\ (D.m_i \sqcap DX)) \\
\Leftarrow & \{ trm\ T \wedge grd\ U \leq grd\ V \Rightarrow \\
& \quad trm\ (S\ ;\ T) \wedge grd\ (S\ ;\ U) \leq grd\ (S\ ;\ V) \} \\
& \forall i \in \{1,\ldots,m\} \bullet trm\ (C.m_i \sqcap CX) \wedge grd\ C.m_i \leq \\
& \quad grd\ (\{R\}\ ;\ (D.m_i \sqcap DX)) \\
\equiv & \{ (*) \text{ above} \} \\
& \forall i \in \{1,\ldots,m\} \bullet R[trm\ (C.m_i \sqcap CX) \wedge grd\ C.m_i] \leq grd\ (D.m_i \sqcap DX) \\
\equiv & \{ trm\ (S \sqcap T) = trm\ S \wedge trm\ T \text{ and } \\
& \quad grd\ (S \sqcap T) = grd\ S \vee grd\ T \text{ for any } S,T \} \\
& \forall i \in \{1,\ldots,m\} \bullet R[trm\ C.m_i \wedge trm\ CX \wedge grd\ C.m_i] \leq \\
& \quad grd\ D.m_i \vee grd\ DX)
\end{array}
$$

The last line follows from (d). Condition (4) follows from (f) by monotonicity. □

A related theorem has first been given for action systems with remote procedures in [8] and in a revised form in [26], which is similar to the corresponding theorem for OO-action systems in [11]. The theorem given here generalizes those in four ways. First, we consider trace refinement and not just input/output refinement. Thus, class refinement also preserves reactive behavior and is meaningful for non-terminating systems. Second, removing abstract stuttering in refinement is explicitly considered. Third, the concrete stuttering action can be more general than a (data-) refinement of *skip*. Fourth, conditions (d) and (e) are weakened by including the termination conditions into the antecedents of the implications.

The case with no explicit abstract stuttering and the concrete stuttering actions being (data-) refinements of *skip* is obtained as a special case. Let $C$ and $D$ be classes and let $C_0$, $D_0$, $CX$, and $DX$ be defined as above. Assume there exists a decomposition $DX = DX_\sharp \sqcap DX_\natural$ such that $DX_\natural$ is a stuttering action. The conditions for this case are:

| | | |
|---|---|---|
| (a') | Initialization: | $C_0 \,;\, [R] \sqsubseteq D_0$ |
| (b') | Methods: | $C.m_i \,;\, [R] \sqsubseteq [R] \,;\, D.m_i$ for all $m_i$ in $m_1, \ldots, m_m$ |
| (c') | Main Actions: | $CX \,;\, [R] \sqsubseteq [R] \,;\, DX_\sharp$ |
| (d') | Internal Actions: | $[R] \sqsubseteq [R] \,;\, DX_\natural$ |
| (e') | Method Guards: | $R[\mathit{trm}\ C.m_i \wedge \mathit{trm}\ CX \wedge \mathit{grd}\ C.m_i] \leq \mathit{grd}\ D.m_i \vee \mathit{grd}\ DX$ for all $m_i$ in $m_1, \ldots, m_m$ |
| (f') | Exit Condition: | $R[\mathit{trm}\ CX \wedge \mathit{grd}\ CX] \leq \mathit{grd}\ DX$ |
| (g') | Internal Convergence: | $R[\mathit{trm}\ CX] \leq \mathit{trm}\ (\mathbf{do}\ DX_\natural\ \mathbf{od})$ |

Condition (d') is equivalent to $\mathit{skip} \sqsubseteq_R DX_\natural$, expressing that the concrete stuttering actions are data refinements of *skip*.

**Theorem 4** *Let C and D be classes and R a relation as above. If conditions (a') – (g') hold then $C \sqsubseteq_R D$.*

Proof We show that the above conditions (a') – (g') imply the conditions (a) – (f) of class simulation. We set $CX_\sharp := CX$ and $CX_\natural := \mathit{magic}$. Thus we have $CX_\natural^0 = \mathit{skip}$, $CX_\natural^i = \mathit{magic}$ for all $i > 0$, and, therefore, $CX_\natural^* = \mathit{skip}$ because $\mathit{skip} \sqcap \mathit{magic} = \mathit{skip}$. With this, (a) follows immediately from (a') and (d').

By reflexivity and transitivity of refinement, we get from condition (d') that $[R] \sqsubseteq [R] \,;\, DX_\natural^i$ for any $i \geq 0$. Since $[R]$ is refined by sequences of any length, it is also refined by their choice, $[R] \sqsubseteq [R] \,;\, DX_\natural^*$. Condition (b) then follows by a transitivity of the following calculation:

$$
\begin{array}{ll}
 & C.m_i \,;\, CX_\natural^* \,;\, [R] \\
\sqsubseteq & \qquad \{\text{as } [R] \sqsubseteq [R] \,;\, DX_\natural^*\} \\
 & C.m_i \,;\, [R] \,;\, DX_\natural^* \\
\sqsubseteq & \qquad \{\text{condition (b')}\} \\
 & [R] \,;\, D.m_i \,;\, DX_\natural^*
\end{array}
$$

Condition (c) follows analogously using (c'). The remaining conditions (d) to (f) follow directly from (e') to (g'). For (f) we observe that **do** $CX_\natural$ **od** $=$ *magic* and *trm magic* $=$ *true*. □

**Corollary 1** *Let C and D be classes and R a relation as above. If conditions (a') – (g') hold then* $C \overset{\text{cl}}{\sqsubseteq} D$.

As with action system refinement, class refinement is not compositional in the sense that refining the class of an object will not necessarily lead to a system with other objects running in parallel being refined. However, we get compositionality under the additional constraint of non-interference with the environment.

**Theorem 5** *Let* $C, D$ *be classes,* $ES = (e_0, E)$ *be an action systems, and R a relation. If E does not interfere with R then:*

$$C \sqsubseteq_R D \Rightarrow \forall K \cdot K[C] \parallel ES \overset{\text{tr}}{\sqsubseteq} K[D] \parallel ES$$

Proof By Theorems 2 and 3 and by the definition of $C \sqsubseteq_R D$. □

**Example.** We use an artificial aquarium as an example. Clearly, the observable sequences of states, denoting the position of the fishes, are the relevant aspect in such a system. A refinement of only the state transformation from initial to final states would be insufficient: a dedicated artificial aquarium has no final state and for a screen saver input/output refinement would only mean that at the end we are again guaranteed to get the original screen back.

The global variable $s$ : **array** $[0..w-1, 0..h-1]$ **of** *NAT* denotes the state (color) of each quadrant of the screen, with constants $w > 6$ and $h > 6$. The color value 0 stands for background water. The base class *Creature* of all objects in our aquarium is given by:

**class** *Creature*
  **attr** $x :\in \{0..w-1\}, y :\in \{0..h-1\}, col : NAT$ ,
  **meth** *move*( **val** *dx*, **val** *dy*) **is** $0 \leq x + dx < w \wedge 0 \leq y + dy < h \rightarrow$
    $skip \sqcap (s[x,y] := 0\ ;\ x, y := x + dx, y + dy\ ;\ s[x,y] := col$,
  **action** *newpos* **is** $s[x,y] := 0\ ;\ x :\in \{0..w-1\}\ ;\ y :\in \{0..h-1\}\ ;$
    $s[x,y] := col$
**end**

Fish described by class *Ray* are a refinement with a special form of movement. Rather than jumping wildly around the screen, they are always at the same vertical

position, have a horizontal speed *sx* and move at most 3 pixels at once:

$$
\begin{array}{l}
\textbf{class}\ \mathit{Ray} \\
\quad \textbf{attr}\ x := 0, y :\in \{0..h-1\}, \mathit{col} := 5, \mathit{sx} := 1, \\
\quad \textbf{meth}\ \mathit{move}(\textbf{val}\ \mathit{dx}, \textbf{val}\ \mathit{dy})\ \textbf{is}\ 0 \le x+\mathit{dx} < w \wedge -3 \le \mathit{dx} \le 3 \wedge \mathit{dy} = 0 \rightarrow \\
\quad\quad s[x,y] := 0\ ;\ x := x+\mathit{dx}\ ;\ s[x,y] := \mathit{col}), \\
\quad \textbf{action}\ \mathit{newpos}\ \textbf{is}\ 0 \le x+\mathit{sx} < w \rightarrow s[x,y] := 0\ ;\ x := x+\mathit{sx}\ ; \\
\quad\quad s[x,y] := \mathit{col}, \\
\quad \textbf{action}\ \mathit{bouncel}\ \textbf{is}\ x+\mathit{sx} < 0 \rightarrow \mathit{sx} :\in \{1..3\}, \\
\quad \textbf{action}\ \mathit{bouncer}\ \textbf{is}\ w \le x+\mathit{sx} \rightarrow \mathit{sx} :\in \{-3..-1\} \\
\textbf{end}
\end{array}
$$

Class *Ray* refines class *Creature* with the (local) refinement relation *R*:

$$
\begin{array}{l}
R\ (s,x,y,\mathit{col})\ (s',x',y',\mathit{col}',\mathit{sx}')\ \widehat{=} \\
\quad s = s' \wedge x = x' \wedge 0 \le x < w \wedge y = y' \wedge 0 \le y < h \wedge \\
\quad \mathit{col} = \mathit{col}' \wedge -3 \le \mathit{sx}' \le 3
\end{array}
$$

Since we have no explicit abstract stuttering, we can use Theorem 4 to prove $\mathit{Creature} \sqsubseteq_R \mathit{Ray}$. We set $\mathit{CX} := \mathit{Creature.newpos}$, $\mathit{DX}_\sharp := \mathit{Ray.newpos}$, $\mathit{DX}_\natural := \mathit{Ray.bouncel} \sqcap \mathit{Ray.bouncer}$, and $C_0$ and $D_0$ to the respective initialization. We show internal convergence (condition (f)) assuming that an access to *s* beyond the screen aborts:

$$
\begin{array}{ll}
 & R[\mathit{trm}\ \mathit{CX}] \\
= & \quad \{\text{definitions of } \mathit{trm} \text{ and } \mathit{CX}\} \\
 & R[\lambda s,x,y,\mathit{col} \bullet 0 \le x < w \wedge 0 \le y < h] \\
= & \quad \{\text{definition of } R\text{, relational image}\} \\
 & \lambda s',x',y',\mathit{col}',\mathit{sx}' \bullet 0 \le x' < w \wedge 0 \le y < h \wedge -3 \le \mathit{sx}' \le 3 \\
\le & \quad \{\text{universal implication}\} \\
 & \lambda s',x',y',\mathit{col}',\mathit{sx}' \bullet -1 \le x' \le w \vee 0 \le x'+\mathit{sx}' < w \\
= & \quad \{\text{definitions, calculus}\} \\
 & \mathit{trm}\ (\textbf{do}\ \mathit{DX}_\natural\ \textbf{od})
\end{array}
$$

The other conditions can also be proved by unfolding the definitions and simple calculus. By Corollary 1 we also get $\mathit{Creature} \overset{\mathrm{cl}}{\sqsubseteq} \mathit{Ray}$. Hence, replacing a *Creature* by a *Ray* in any context produces a trace refinement.

## 5 Dynamic Object Structures

We model the heap as an array and pointers as indices into this array [23]. We first describe the basic ideas using only one class and then generalize it to multiple classes with subtypes. Let $C = (c_0, cs)$ denote a class and $\Sigma$ the type of its attributes. We declare a program variable *heap* to contain the whole dynamic data structure:

$$\textbf{var}\ \mathit{heap} : \textbf{array}\ \mathit{NAT}\ \textbf{of}\ \Sigma$$

Pointers to instances of $C$ are then simply natural numbers, that is the declaration $p$ : **pointer to** $C$ stands for $p : NAT$. We use 0 to denote *nil*, that is the pointer not referencing any object. We use a separate counter *next*, initialized to 1, to generate new pointer values. Thus for $p$ : **pointer to** $C$,

$$p := new\ C \ \widehat{=}\ p := next\ ;\ (\mathbf{var}\ c \mid c_0 \bullet heap[p] := c)\ ;\ next := next + 1$$

where $heap[p]$ denotes an access of the $p$th array element.

To correctly handle nested method calls, we need to pass references rather than the values of the referenced attributes for the implicit receiver argument. We denote the receiver by *this* and let $c$, the variable denoting the receiver's attributes in methods and actions, stand for $heap[this]$. This global replacement formalization assumes that all object structures are dynamic. A simple extension using flags could handle a combination of static and dynamic object structures. A method call $p.m$ is defined as:

$$\mathbf{var}\ this : NAT \bullet this := p\ ;\ C.m$$

In our formalization, *this* is used to reference the receiver object whereas *self* and *super* are used in classes to reference methods and actions.

This formalization easily extends to multiple classes with subtyping. We declare for each class $C_i$ with attribute type $\Sigma_i$ a separate $heap_i$ : **array** $NAT$ **of** $\Sigma_i$. Pointers are extended to tuples with one index indicating the heap and one index indicating the element within the heap. Thus with a class declaration **class** $C_i \ldots$ **end** we associate:

$$\mathbf{var}\ heap_i : \mathbf{array}\ NAT\ \mathbf{of}\ \Sigma_i,$$
$$\mathbf{var}\ next_i : NAT := 1$$

A pointer variable declaration $p$ : **pointer to** $C_i$ stands for $p : NAT \times NAT := (0,0)$. The *nil* value is always represented by $(0,0)$ to make it unique. The type of *this* is now also $NAT \times NAT$. Attribute access occurrences of $c$ in the method body stand for $heap_{fst\ this}[snd\ this]$. In the receiver role of method and action calls ($c.m$), $c$ stands for *this*. Assuming that $C_k, \ldots, C_l$ are all subtypes of $C_i$ (including $C_i$), object creation, method calls with dynamic dispatch, and type test are defined by:

$$\begin{array}{lcl}
p := new\ C_i & \widehat{=} & p := (i, next_i)\ ;\ (\mathbf{var}\ c \mid c_i \bullet heap_i[next_i] := c)\ ; \\
 & & next_i := next_i + 1 \\
p.m() & \widehat{=} & \mathbf{if}\ p = (0,0)\ \mathbf{then}\ abort \\
 & & \mathbf{elsif}\ fst\ p = k\ \mathbf{then}\ (\mathbf{var}\ this \bullet this := p\ ;\ C_k.m) \\
 & & \ldots \\
 & & \mathbf{elsif}\ fst\ p = l\ \mathbf{then}\ (\mathbf{var}\ this \bullet this := p\ ;\ C_l.m) \\
 & & \mathbf{end} \\
p\ \mathbf{instanceof}\ C_i & \widehat{=} & fst\ p \in \{k, \ldots, l\}
\end{array}$$

We assume that the heaps and the next counters are implicitly declared global variables that do not appear in the traces. With each class declaration $C_i$, we associate an action system $\mathcal{A}[C_i]$ which represents all actions of all objects of that class:

$$
\begin{array}{rl}
\mathcal{A}[C_i] = & \textbf{do} \\
& \quad (\, \textbf{var}\ \mathit{this} : \mathit{NAT} \times \mathit{NAT} \bullet \mathit{this} :\in \{i\} \times \{1..\mathit{next}_i - 1\} \ ; \\
& \quad\quad (C_i.a_1 \sqcap \ldots \sqcap C_i.a_a)) \\
& \textbf{od}
\end{array}
$$

For a program with classes $C_1, \ldots, C_n$ we take the parallel composition of the action systems for objects of each class. This composition is then to be combined with further action systems containing normal actions and procedures:

$$
\mathcal{A}[C_1] \parallel \ldots \parallel \mathcal{A}[C_n] \parallel \mathit{BS}
$$

**Example.** Let classes *Creature* and *Ray* be as in the aquarium example of Section 4 and let $\mathit{Creature} \sqsubseteq_Q \mathit{Turtle}$ for some relation $Q$. Let $G$ be the following specification of an artificial aquarium, in which new objects may constantly be added and where the most recently created object may be influenced through its *move* method:

$$
\begin{array}{rcl}
G & = & (\, \textbf{var}\ p : \mathit{Creature} := \mathit{nil} \bullet \textbf{do}\ p := \mathit{new}\ \mathit{Creature}\ [\!] \\
& & p \neq \mathit{nil} \rightarrow p.\mathit{move}(2,0)\ \textbf{od}\,) \parallel \mathcal{A}[\mathit{Creature}]
\end{array}
$$

By applying Theorem 3 twice and Theorem 5 we can show that this specification is trace refined, $G \overset{\mathrm{tr}}{\sqsubseteq} H$, by implementation $H$ with rays and turtles:

$$
\begin{array}{rcl}
H & = & (\, \textbf{var}\ p : \mathit{Creature} := \mathit{nil} \bullet \textbf{do}\ p := \mathit{new}\ \mathit{Ray}\ [\!]\ p := \mathit{new}\ \mathit{Turtle} \\
& & [\!]\ p \neq \mathit{nil} \rightarrow p.\mathit{move}(2,0)\ \textbf{od}\,) \parallel \mathcal{A}[\mathit{Ray}] \parallel \mathcal{A}[\mathit{Turtle}]
\end{array}
$$

# 6 Early Return

Atomicity refinement is used to increase concurrency by decreasing the granularity of atomic actions. Consider method *rnd* that computes random numbers and for later reference stores them in a time ordered sequence:

$$
\textbf{meth}\ \mathit{rnd}(\,\textbf{res}\ y)\ \textbf{is}\ y :\in \mathit{NAT}\ ;\ \text{‘store } y \text{ in sequence’}
$$

Using atomicity refinement, we could split up *rnd* so that it returns control to the caller after assigning $y$ and schedules the —if the sequence is kept on secondary storage— time consuming insertion operation for later. Thereby, the execution time of any action $a$ calling *rnd* is reduced. Thus, other actions accessing the same resources as $a$ can be started earlier, thereby increasing concurrency.

We introduce a **release** statement, which facilitates the above type of atomicity refinement. A **release** returns control to the caller of a method and schedules

| **class** $C$ | **class** $C$ |
|---|---|
| **attr** $c \mid c_0$, | **attr** $c, lck \mid c_0 \wedge lck = 0$, |
| **meth** $m$ **is** $S$ ; **release** ; $T$, | **meth** $m$ **is** $lck = 0 \rightarrow S$ ; $lck := 1$, |
| **meth** $n$ **is** $U$, | **meth** $n$ **is** $lck = 0 \rightarrow U$, |
| **action** $a$ **is** $V$ | **action** $a$ **is** $lck = 0 \rightarrow V$, |
| | **action** $r$ **is** $lck = 1 \rightarrow T$ ; $lck := 0$ |
| **end** | **end** |
| a) Method with **release** | b) Equivalent without **release** |

Figure 3: Definition of **release** as enabling a remainder action

the remainder to be executed later on. If the method containing the **release** statement has result parameters, they must be assigned before executing **release**. For example, we could rewrite method *rnd* as follows:

**meth** $rnd($**res** $y)$ **is** $y :\in NAT$ ; **release** ; 'store $y$ in sequence'

Figure 3 defines **release** as enabling an action $r$ that performs the remainder. The object is locked, that is none of its other methods or actions can be executed, until the remainder action is completed. For simplicity, we do not allow self and reentrant calls and parameter and local variable access in the remainder. These generalizations are made below.

Introducing **release** leads to class refinement:

**Theorem 6** *Let C and D be classes which are identical except that method m in C and m in D, referred to as C :: m and D :: m, are defined by:*

**meth** $C :: m$ **is** $S$ ; $T$,
**meth** $D :: m$ **is** $S$ ; **release** ; $T$

*If T does not modify global variables and does not access parameters, then* $C \overset{\mathrm{cl}}{\sqsubseteq} D$ *holds.*

Proof We apply Theorem 3 with $R\ (u,c)\ (u',c',lck') := u' = u \wedge ((l' = 0 \wedge c' = c) \vee (l' = 1 \wedge nxt\ S\ c\ c'))$, $C_0 :=$ **enter** $c \mid c_0$, $CX_\sharp := V$, $CX_\natural := magic$, $D_0 :=$ **enter** $c, lck \mid c_0 \wedge lck = 0$, $DX_\sharp := lck = 0 \rightarrow V$, $DX_\natural := lck = 1 \rightarrow T$ ; $lck := 0$. The theorem follows by simplifications of the conditions (a) – (f).
Note that Theorem 4 cannot be used for the proof since the remainder action $lck = 1 \rightarrow T$ ; $lck := 0$ is a concrete stuttering action which is not a (data-) refinement of *skip*.

| | |
|---|---|
| **class** $C$<br>  **attr** $c := c_0$,<br>  **meth** $m($ **val** $v$, **res** $r)$ **is**<br>    **var** $x \bullet S$ ; **release** ; $T$,<br>  **meth** $n($ **val** $w$, **res** $s)$ **is** $U$<br>  **action** $a$ **is** $V$<br><br><br>**end** | **class** $C$<br>  **attr** $c, lck, m_v, m_r, m_x \mid c_0 \wedge lck = 0$,<br>  **meth** $m($ **val** $v$, **res** $r)$ **is** $lck = 0 \rightarrow$<br>    **var** $x \bullet S$ ; $lck, m_v, m_r, m_x := 1, v, r, x$,<br>  **meth** $n($ **val** $w$, **res** $s)$ **is** $lck = 0 \rightarrow U$,<br>  **action** $a$ **is** $lck = 0 \rightarrow V$,<br>  **action** $r$ **is** $lck = 1 \rightarrow$<br>    **var** $v, r, x := m_v, m_r, m_x \bullet T$ ; $lck := 0$<br>**end** |
| a) Method with **release** | b) Equivalent without **release** |

Figure 4: Definition of **release** with Remainder Accessing Parameters and Local Variables

The **release** statement can be generalized to allow the remainder to access the value parameter and the local variables of the method and also read the result parameter (Figure 4). The values of the parameters and local variables are stored in additional attributes for use by the remainder.

Finally, we consider the case where an action contains multiple calls to methods of the same object. If a method of an object that has an outstanding remainder is called then the latter is executed as part of the call. Otherwise, the guard of the methods called after performing a **release** would be false and, therefore, such actions never enabled. Consider action $b$ where $o$ references an object of type $C$ as in Figure 5:

**action** $b$ **is** ( **var** $z : U \bullet o.m(0, z)$ ; $o.n(0, z)$)

If we simply locked $o$, that is, defined the implicit guard of $n$ to be $lck = 0$, then $b$ would never be enabled.

We illustrate this with a random number class that stores a sequence of already computed numbers:

**class** $C$
  **attr** $l := 0, s$ : **array** $NAT$ **of** $NAT$ ,
  **meth** $rnd($ **res** $y)$ **is** $y :\in NAT$ ; $s[l], l := y, l + 1$,
  **meth** $get($ **val** $i$, **res** $y)$ **is** $i < l \rightarrow y := s[i]$
**end**

| | |
|---|---|
| **class** $C$ | **class** $C$ |
| **attr** $c \mid c_0$, | **attr** $c, lck, m_v, m_r, m_x \mid c_0 \wedge lck = 0$ |
| **meth** $m($ **val** $v$, **res** $r)$ **is** | **meth** $m($ **val** $v$, **res** $r)$ **is** |
| **var** $x \bullet S$ ; **release** ; $T$, | $p$ ; **var** $x \bullet S$ ; $lck, m_v, m_r, m_x := 1, v, r, x$, |
| **meth** $n($ **val** $w$, **res** $s)$ **is** $U$, | **meth** $n($ **val** $w$, **res** $s)$ **is** $p$ ; $U$, |
| | **meth** $p$ **is if** $lck = 1$ **then** |
| | **var** $v, r, x := m_v, m_r, m_x \bullet T$ ; $lck := 0$ |
| | **end** , |
| **action** $a$ **is** $V$ | **action** $a$ **is** $lck = 0 \rightarrow V$, |
| | **action** $r$ **is** $lck = 1 \rightarrow p$ |
| **end** | **end** |
| a) Method with **release** | b) Equivalent without **release** |

Figure 5: Definition of **release** supporting multiple calls to an object within an action

Class $C$ is refined by $D$, where a **release** is introduced in method *rnd* after the assignment of $y$. We show directly the expansion according to Figure 5:

> **class** $D$
>   **attr** $l := 0, s$ : **array** $NAT$ **of** $NAT$ $, lck := 0, rnd_y$,
>   **meth** $rnd($**res** $y)$ **is** $p$ ; $y :\in NAT$ ; $lck, rnd_y := 1, y$,
>   **meth** $get($**val** $i$, **res** $y)$ **is** $p$ ; $i < l \rightarrow y := s[i]$,
>   **meth** $p$ **is if** $lck = 1$ **then var** $y := rnd_y \bullet s[l], l, lck := y, l+1, 0$ **end** ,
>   **action** $r$ **is** $lck = 1 \rightarrow p$
> **end**

We have $C \sqsubseteq_R D$ for the following $R$:

$$R\ (l,s)\ (l',s',lck',rnd_y') \equiv lck' \in \{0,1\} \wedge$$
$$(lck' = 0 \Rightarrow l = l' \wedge (\forall i \in \{0..l-1\} \bullet s[i] = s'[i])) \wedge$$
$$(lck' = 1 \Rightarrow l = l'+1 \wedge (\forall i \in \{0..l-2\} \bullet s[i] = s'[i]) \wedge s[l-1] = rnd_y')$$

The proof is a simple verification of the six conditions of class simulation with $CX_\sharp = magic$, $CX_\natural = magic$, $DX_\sharp = magic$, $DX_\natural = r$, and $C_0$ and $D_0$ the respective initializations.

# 7 Conclusions and Discussion

We have given a model for action-based concurrency with objects. Classes with attributes, methods, and actions serve as templates for objects. Class refinement supporting algorithmic, data, and atomicity refinement is defined based on trace

refinement. Class refinement can be proved by a simulation rule. Early returns are a special form of atomicity refinement. Dynamic data structures allow objects to run concurrently.

Class refinement for concurrent objects is defined here as an extension of class refinement defined in [24], following the general model of classes as self-referential structures with a delayed taking of the fixpoint of [27, 14]. As known from [24], inheritance is not monotonic with respect to the refinement of the base class, leading to the so called fragile base class problem. This problem persists in the concurrent setting here as well. With the possibility of self- and super-references between actions, it extends to actions.

For expressing symmetric communication and synchronization among several objects, multi-party actions have been studied in [5]. They can be introduced here without further difficulties.

Many interesting questions are connected with early returns. The remainder of a method into which we introduce a **release** statement cannot modify global variables. Otherwise, multiple changes that were previously executed in one atomic step could now be performed in multiple steps. The definition of trace refinement does not permit this. Making intermediate states visible and even making modifications to other global variable before the remainder's changes to global variables are performed are not legal refinements.

However, introducing in a refinement step changes to other objects in the remainder is a useful concept studied in [18]. This is allowed if there are no other references to those objects and hence those changes are not observable to the remaining program. For this [18] uses the concept of unique references. Spinning the idea of non-observability even further, the global state could also be updated in multiple steps if parts of it could be locked and be guaranteed not to be observed until the remainder has been executed. Incorporating such refinement steps here is an open issue.

The main advantage of a **release** statement over a "manual" atomicity refinement are the readability (no need to syntactically split method into parts, syntactically clutter guards and the split method with synchronization and variable save statement) and the automatic resource locking. A version without resource locking would be possible and would allow additional interleavings, but would lead to practically rather strong proof conditions, making it less attractive.

The **release** statement could also be introduced into action systems without objects, for example within procedures. Objects, however, have the advantage that they encapsulate tightly coupled state components and, thereby, make it in practice easier to lock resources accessed by the remainder.

**Acknowledgments** We would like to thank Ralph Back and Marina Waldén for a number of clarifying discussions.

## References


[1] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer Verlag, 1996.

[2] Pierre America. Designing an object-oriented programming language with behavioral subtyping. In *Foundations of Object-Oriented Languages, REX School/Workshop*, Lecture Notes in Computer Science 489, pages 60–90, 1991.

[3] Ralph Back. Refining atomicity in parallel algorithms. In *PARLE Conference on Parallel Architectures and Languages Europe*, Eindhoven, June 1989. Springer-Verlag.

[4] Ralph Back. Atomicity refinement in a refinement calculus framework. Technical Report on Computer Science & Mathematics, Ser. A. No 141, Åbo Akademi, 1993.

[5] Ralph Back, Martin Büchi, and Emil Sekerinski. Action-based concurrency and synchronization for objects. In T. Rus and M. Bertran, editors, *Transformation-Based Reactive System Development, Fourth AMAST Workshop on Real-Time Systems, Concurrent, and Distributed Software*, Lecture Notes in Computer Science 1231, pages 248–262, Palma, Mallorca, Spain, 1997. Springer-Verlag.

[6] Ralph Back and Reino Kurki-Suonio. Decentralization of process nets with centralized control. In *2nd ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 131–142. ACM Press, 1983.

[7] Ralph Back and Reino Kurki-Suonio. Distributed co-operation with action systems. *ACM Transactions on Programming Languages and Systems 10:513–554*, 1988.

[8] Ralph Back and Kaisa Sere. Action systems with synchronous communication. In E.-R. Olderog, editor, *IFIP Working Conference on Programming Concepts, Methods, Calculi*, pages 107–126, San Miniato, Italy, 1994. North-Holland.

[9] Ralph Back and Joakim von Wright. Trace refinement of action systems. In B. Jonsson and J. Parrow, editors, *CONCUR '94: Concurrency Theory*, Lecture Notes in Computer Science 836. Springer-Verlag, 1994.

[10] Ralph Back and Joakim von Wright. *Refinement Calculus – A Systematic Introduction*. Springer-Verlag, 1998.

[11] Marcello M. Bonsangue, Joost N. Kok, and Kaisa Sere. An approach to object-orientation in action systems. In *Mathematics of Program Construction*, Lecture Notes in Computer Science 1422, Marstrand, Sweden, 1998. Springer-Verlag.

[12] K. M. Chandy and J. Misra. *Parallel Program Design – A Foundation*. Addison Wesley, 1988.

[13] Ernie Cohen and Leslie Lamport. Reduction in TLA. In *Proceedings of CONCUR'98*, Lecture Notes in Computer Science 1466, pages 317–331. Springer-Verlag, 1998.

[14] William Cook and Jens Palsberg. A denotational semantics of inheritence and its correctness. In *ACM Conference Object Oriented Programming Systems, Languages and Applications*, ACM SIGPLAN Notices, Vol 14, No 10, pages 433–443, 1989.

[15] J.W. de Bakker and E.P. de Vink. Bisimulation semantics for concurrency with atomicity and action refinement. *Fundamenta Informaticae*, 20(1):3–34, 1994.

[16] H.-M. Järvinen and R. Kurki-Suonio. DisCo specification language: Marriage of action and objects. In *Proceedings of 11th International Conference on Distributed Computing Systems*, pages 142–151, Arlington, Texas, 1991. IEEE Computer Society Press.

[17] Cliff B. Jones. An object-based design method for concurrent programs. Technical report, University of Manchester, Department of Computer Science, December 1992.

[18] Cliff B. Jones. Accomodating interference in the formal design of concurrent object-based programs. *Formal Methods in System Design*, 8(2):105–122, March 1996.

[19] Leslie Lamport. The temporal logic of actions. *ACM Transactions of Programming Languages and Systems*, 16(3):872–923, 1994.

[20] Leslie Lamport and Fred B. Schneider. Pretending atomicity. Technical Report Research Report 44, Compaq Systems Research Center, May 1989.

[21] Richard J. Lipton. Reduction: A method of proving properties of parallel programs. *Communications of the ACM*, 18(12):717–721, December 1975.

[22] Barbara H. Liskov and Jeanette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.

[23] David C. Luckham and Norihisa Suzuki. Verification of array, record, and pointer operations in pascal. *ACM Transactions on Programming Languages and Systems*, 1(2):226–244, October 1979.

[24] Leonid Mikhajlov and Emil Sekerinski. A study of the fragile base class problem. In Eric Jul, editor, *ECOOP'98 – 12th European Conference on*

*Object-Oriented Programming*, Lecture Notes in Computer Science 1445, pages 355–382, Brussels, Belgium, 1998. Springer-Verlag.

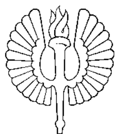
[25] Anna Mikhajlova and Emil Sekerinski. Class refinement and interface refinement in object-oriented programs. In John Fitzgerald, Cliff Jones, and Peter Lucas, editors, *Formal Methods Europe'97*, Lecture Notes in Computer Science 1313, pages 82–101, Graz, Austria, 1997. Springer-Verlag.

[26] Kaisa Sere and Marina Waldén. Data refinement of remote procedures. In *Proceedings of TACS 97*, Lecture Notes in Computer Science 1281, pages 267–294. Springer-Verlag, 1997.

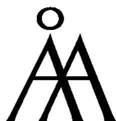
[27] Peter Wegner and Stanley B. Zdonik. Inheritance as an incremental modification mechanism or what like is and isn't like. In S. Gjessing and K. Nygaard, editors, *European Conference on Object Oriented Programming*, Lecture Notes in Computer Science 322, pages 55–77. Springer-Verlag, 1988.

**Turku Centre for Computer Science**
**Lemminkäisenkatu 14**
**FIN-20520 Turku**
**Finland**

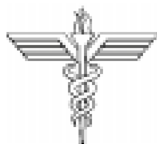
http://www.tucs.fi



**University of Turku**
- **Department of Mathematical Sciences**



**Åbo Akademi University**
- **Department of Computer Science**
- **Institute for Advanced Management Systems Research**



**Turku School of Economics and Business Administration**
- **Institute of Information Systems Science**