IOS Press

A Foundation for Refining Concurrent Objects

Martin Büchi
Turku Centre for Computer Science
Turku, Finland
Martin.Buechi@abo.fi

Emil Sekerinski
Department of Computing and Software
McMaster University
Hamilton, Ontario, Canada
emil@mcmaster.ca

Abstract. We study the notion of class refinement in a concurrent object-oriented setting. Our model is based on a combination of action systems and classes. An action system describes the behavior of a concurrent, distributed, or interactive system in terms of the atomic actions that can take place during the execution of the system. Classes serve as templates for creating objects. To express concurrency with objects, we add actions to classes.

We define class refinement based on trace refinement of action systems. Additionally, we give a simulation-based proof rule. We show that the easier to apply simulation rule implies the trace-based definition of class refinement.

Class refinement embraces algorithmic refinement, data refinement, and atomicity refinement. Atomicity refinement allows us to split large atomic actions into several smaller ones. Thereby, it paves the way for more parallelism. We investigate the special case of atomicity refinement by early returns in methods.

Keywords: concurrent objects, classes, inheritance, subtyping, action systems, atomicity refinement, class refinement, simulation, early return

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 im-

plementation. We view the specification as an abstract program P and the implementation as a concrete program Q. The task of ensuring that the implementation satisfies the specification is eased by introducing intermediate programs such that each program is a *refinement* of the previous one, formally expressed as:

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

In *algorithmic refinement* steps abstract (or more abstract) statements are replaced by concrete (or more concrete) statements whereas in *data refinement* steps abstract (or more abstract) data structures are replaced by concrete (or more concrete) data structures. For the development of concurrent programs, in *atomicity refinement* steps sequential (or less concurrent) parts are replaced 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 a cache for storage and would process write operations in the background, hence changing the state space and introducing concurrency. In any case, the illusion to the user of the write operation is maintained that the operation is executed *atomically*. In this example, concurrency is introduced in the implementation for allowing a better utilization of resources, which is an aspect we are interested in without formalizing it.

In this paper we propose a formal model for objects with attributes and methods, with self- and super-calls in methods, classes with inheritance, and action-based concurrency. Objects have 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. As in class-based programming languages, 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 programs with instances 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 in a setting with dynamic object structures.

The proposed class refinement extends class refinement as defined for sequential objects [27, 26] 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 [24]. 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 fixed point of [31, 16].

The action system model for parallel, distributed, and reactive systems was proposed by Back and Kurki-Suonio [7, 8]. The same basic approach has later been used in other models for distributed computing, notably UNITY [14] and TLA [21].

An action system describes the behavior of a concurrent system in terms of the atomic actions that can take place during the execution of the system. Action systems allow a succinct description of the overall behavior of a system. Furthermore, action-based approaches do not force us to fix the flow of control where doing so is unnecessary for an abstract specification (see e.g. [14]). Action systems can be used to express various forms of communication, e.g. shared variable, rendez-vous, and bounded channels, as well as different interaction mechanisms, e.g. semaphores, critical regions, and 4-phase handshake [8, 14].

Back and Sere [9] have added procedures to action systems. They, as well as Sere and Waldén [30] and Bonsangue et al [13], 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 [18] used aggregation rather than inheritance and overriding, based their semantics on TLA, and concentrated on superposition refinement. Back et al [6] concentrated on the design of a language. Bonsangue et al [13] developed a less formal model with an action-system-per-object semantics. Seuss [28] also combines objects with action-based concurrency. The catch in Seuss is that the set of objects (called clones) is static.

Atomicity refinement has first been proposed by Lipton [23]. Back studied input/output behavior preserving atomicity refinement in action systems [4, 5]. Sere and Waldén [30] and Bonsangue et al [13] have extended this to procedures and methods, still refining only input/output behavior. Lamport and Schneider [22] and Cohen and Lamport [15] have studied atomicity refinement in TLA considering liveness properties beyond termination. De Bakker and de Vink [17] 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 [19, 20] in a framework with explicit constructs for parallelism.

Our calculus for concurrent objects is meant to provide a design notation for programs to be implemented in concurrent object-oriented languages, such as POOL, Modula-3, and Java. Programs can be expressed more abstractly than in those languages. The synchronization and communication mechanisms of these programming languages can be expressed in our formalism and formally introduced in refinement steps.

Outline. In Section 2 we review the fundamentals of statements and action systems. 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 in terms of the externally observable behavior, gives a condition for class simulation using a relation, and proves that class simulation implies class refinement for a system with a single object of a given class. Section 5 introduces dynamic object structures and extends the discussion of class refinement and class simulation to that setting. 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

The refinement calculus, which provides the foundation for our work, is due to Back, Morgan, and von Wright [3, 29, 11]. We review the fundamentals of statements defined by predicate transformers following [11] and of action systems following [10].

2.1. Statements

State predicates of type $\mathcal{P}\Sigma$ are functions from elements of type Σ to $Bool$. Relations of type $\Delta \leftrightarrow \Omega$ are functions from Δ to (state) predicates over Ω. Predicate transformers of type $\Delta \mapsto \Omega$ are functions from predicates over Ω (the postconditions) to predicates over Δ (the preconditions):

$$\begin{array}{lcl} \mathcal{P}\Sigma & \hat{=} & \Sigma \rightarrow Bool \\ \Delta \leftrightarrow \Omega & \hat{=} & \Delta \rightarrow \mathcal{P}\Omega \\ \Delta \mapsto \Omega & \hat{=} & \mathcal{P}\Omega \rightarrow \mathcal{P}\Delta \end{array}$$

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 $R\ [p]\ y\ \hat{=}\ (\exists x \bullet R\ x\ y \wedge p\ x)$. 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:

$$(S\ ;\ T)\ q\ \hat{=}\ S\ (T\ q)$$

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

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

The demonic (nondeterministic) choice $\sqcap$ establishes a postcondition only if both alternatives do. The angelic choice $\sqcup$ establishes a certain postcondition if at least 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 $\Delta \mapsto \Omega$:

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

We generalize the binary demonic choice to the choice among a fixed set of statements:

$$(\sqcap i \in I \bullet S)\ q \ \widehat{=}\ (\forall i \in I \bullet S\ q)$$

As a variant, we allow the choice to be restricted by a state predicate:

$$(\sqcap i \mid p \bullet S) \ \widehat{=}\ (\sqcap i \bullet [p]\ ;\ S)$$

All of the above constructs are monotonic or preserve monotonicity. 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 and $i \in I$ form 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. For example, for any predicate transformers S, T, U we have that

$$(S \sqcap T)\ ;\ U = (S\ ;\ U) \sqcap (T\ ;\ U)$$

but only if U is positively conjunctive we have also that:

$$U\ ;\ (S \sqcap T) = (U\ ;\ S) \sqcap (U\ ;\ T)$$

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

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

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

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

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

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

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

The loop **do** S **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:

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

The loop **while** p **do** B is defined as **do** $p \rightarrow B$ **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 $R : \Sigma \leftrightarrow \Sigma'$:

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

Data refinement $S \sqsubseteq_R S'$ can be equivalently defined by $\{R^{-1}\}\ ;\ S \sqsubseteq S\ ;\ \{R^{-1}\}$, where R^{-1} is the relational inverse of R. 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:

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

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

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

The declaration of a local variable $y : \Delta$ with initialization predicate yi extends the state space and sets y to any value for which $yi\ y$ 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:

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

Leaving out the initialization predicate as in **var** $y \bullet S$ means initializing the variable arbitrarily, **var** $y \mid true \bullet S$. Where necessary, we also explicitly indicate the type Δ of the new variable as in **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 $\Gamma_1 \times \cdots \times \Gamma_n \times \Delta \mapsto \Gamma_1 \times \cdots \times \Gamma_n \times \Delta$. Assuming that variable names select the correct state space component, we can also commute state space components.

When writing state predicates, we usually leave out the lambda abstractions over the variables if they are evident from the context. For example, we write $x > c$ rather than $\lambda x, y \bullet x > c$ and similarly we would write **if** $x > c$ **then** S **else** T.

Product Statements. For predicates $q_1 : \mathcal{P}\Sigma_1$ and $q_2 : \mathcal{P}\Sigma_2$ the product $q_1 \times q_2$ of type $\mathcal{P}(\Sigma_1 \times \Sigma_2)$ is defined as $(q_1 \times q_2)\ (\sigma_1, \sigma_2)\ \widehat{=}\ q_1\ \sigma_1 \wedge q_2\ \sigma_2$. 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:

$$(S_1 \times S_2)\ q\ (\delta_1, \delta_2)\ \widehat{=}\ \exists q_1, q_2 \mid q_1 \times q_2 \leq q \bullet S_1\ q_1\ \delta_1 \wedge S_2\ q_2\ \delta_2$$

Intuitively, this means that $S_1 \times S_2$ establishes the postcondition $q : \mathcal{P}(\Omega_1 \times \Omega_2)$ from initial state (δ_1, δ_2), if there is a “rectangular” subset $q_1 \times q_2$ of q such that independently S_1 establishes q_1 from δ_1 and S_2 establishes q_2 from δ_2 [12].

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 implies that there must exist S' and T' such that $S = S' \times skip$ and $T = skip \times T'$. 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 Γ 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 fixed point of statement tuples. For tuples $(s_1, \ldots, s_n)$ and $(s'_1, \ldots, s'_n)$, where s_i and s'_i are statements of the same type, the refinement ordering is defined elementwise:

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

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

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

defines p to be the least fixed point 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.

2.2. 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 consist of local variables, an initialization thereof, and a body, which 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 $AS = (ai, A)$ where $ai : \mathcal{P}\Sigma$ is the initializing predicate of the local state. Upon initialization, arbitrary values satisfying ai are chosen for the local variables. The global state space Γ 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 Σ and global state of type Γ. Because A is positively conjunctive, it can be written as $\{p\}\ ;\ [R]$. The next relation of A relates a state (u, v) in both the enabledness 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 AS 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 $ai\ v_0$, 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* AS can be thought of as the (disjoint) union of terminating, aborting, and non-terminating behaviors of AS.

We use the following syntax for an action system (ai, A) with local variables a:

$$\mathbf{var}\ a \mid ai \bullet \mathbf{do}\ A\ \mathbf{od}$$

Action systems are typically composed of a set of actions $A_1, \ldots, A_n$ operating on different parts of the state space, which we write as:

$$\mathbf{var}\ a \mid ai \bullet \mathbf{do}\ A_1 \parallel \ldots \parallel A_n\ \mathbf{od}$$

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:

$$\mathbf{var}\ a \mid ai \bullet \mathbf{do}\ A_1 \sqcap \ldots \sqcap A_n\ \mathbf{od}$$

We furthermore consider the case of an indexed set of actions and of set of actions where the possible choice depends on a state predicate:

$$\begin{array}{lcl} (\parallel i \in I \bullet A) & \widehat{=} & (\sqcap i \in I \bullet A) \\ (\parallel i \mid p \bullet A) & \widehat{=} & (\sqcap i \mid p \bullet A) \end{array}$$

To express various kinds of possibly parallel computations, we use also combinations of these notations, for example as in:

$$\mathbf{do}\ (\parallel i \mid p \bullet A) \parallel (\parallel j \mid q \bullet B)\ \mathbf{od}$$

Parallel Composition. The parallel composition of action systems $AS = (ai, A)$ and $BS = (bi, B)$ 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:

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

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. (We omit 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 $A \sqcap B$.) Parallel composition is commutative and associative, up to the order of state components.

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

$$\textbf{var}\ b \mid bi \bullet \textbf{var}\ a \mid ai \bullet \textbf{do}\ A\ \textbf{od}\ \ \widehat{=}\ \ \textbf{var}\ a, b \mid ai \wedge bi \bullet \textbf{do}\ A\ \textbf{od}$$

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 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 sequences of 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

- s is aborting and $tr\ s$ is a prefix of $tr\ t$, or
- $tr\ s = tr\ t$.

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

$$AS \preceq BS\ \ \widehat{=}\ \ \forall t \in beh\ BS \bullet \exists s \in beh\ AS \bullet s \preceq t$$

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

Simulation. Trace refinement can be shown to hold by simulation. Here we consider forward simulation between $AS = (ai, A)$ and $BS = (bi, B)$ 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 $S^* = (\sqcap\ n \in Nat \bullet S^n)$. Define $AI =$ **enter** $a \mid ai$ and $BI =$ **enter** $b \mid bi$. Action system AS is simulated by BS using R, written $AS \preccurlyeq_R BS$, if there are decompositions $A = A_\sharp \sqcap A_\natural$ and $B = B_\sharp \sqcap B_\natural$ such that $A_\natural$ and $B_\natural$ are stuttering actions and:

(a)	Initialization:	$AI\ ;\ A_\natural^*\ ;\ [R] \sqsubseteq BI\ ;\ B_\natural^*$
(b)	Actions:	$A_\sharp\ ;\ A_\natural^*\ ;\ [R] \sqsubseteq [R]\ ;\ B_\sharp\ ;\ B_\natural^*$
(c)	Exit Condition:	$R[trm\ A \wedge grd\ A] \leq grd\ B$
(d)	Internal Convergence:	$R[trm\ A \wedge trm\ (\textbf{do}\ A_\natural\ \textbf{od})] \leq trm\ (\textbf{do}\ B_\natural\ \textbf{od})$

Condition (a) expresses that after the initializations AI and BI, the states of AS and BS have to be in the refinement relation, provided that any number of stuttering actions $A_\natural^*$ and $B_\natural^*$ may follow the initializations, respectively. Condition (b) can be equivalently written as $A_\sharp$; $A_\natural^* \sqsubseteq_R B_\sharp$; $B_\natural^*$. It expresses that $A_\sharp$ is data refined by $B_\sharp$, provided that any number of stuttering actions $A_\natural^*$ and $B_\natural^*$ may follow the actions $A_\sharp$ and $B_\sharp$, respectively. Condition (c) expresses that BS must terminate whenever AS does. Condition (d) expresses that the stuttering action $B_\natural^*$ must terminate if the stuttering action $B_\natural^*$ does. The proof of condition (d) involves showing loop termination, which is typically done with a variant.

Theorem 2.1. *Let AS and BS be action systems and R a relation. Then:*

$$AS \preccurlyeq_R BS \Rightarrow AS \preceq BS$$

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 $ES = (ei, E)$ be an action system and let R be refinement relation for AS. Action system ES does not interfere with R if

$$trm\ E \wedge r \leq E\ r$$

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

Theorem 2.2. *Let AS, BS, and ES be action systems, let R be a relation. If ES does not interfere with R then:*

$$AS \preccurlyeq_R BS \Rightarrow AS \parallel ES \preceq BS \parallel ES$$

Figure 1 summarizes the various ordering relations on predicates, statements, traces, action systems, and classes.

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 to 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. Furthermore a subclass may add new attributes and overwrite 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 [31].

These principles are extended here: classes define additionally a set of actions, which are inherited in subclasses and may be overwritten. Subclasses may also introduce additional actions. Self-references are possible between both methods and actions. Self-references are resolved at the time when an object is created. Also, both methods and actions may contain super-references to methods and actions in the superclass.

$p \leq q$	entailment of predicates	Section 2.1
$S \sqsubseteq T$	algorithmic refinement of statements	Section 2.1
$S \sqsubseteq_R T$	data refinement of statements	Section 2.1
$s \preceq t$	approximation of traces	Section 2.2
$AS \preceq BS$	trace refinement of action systems	Section 2.2
$C \preceq^\circ D$	class refinement with single object	Section 4.1
$C \preceq^\uparrow D$	class refinement with dynamic object structures	Section 5.1
$AS \preccurlyeq_R BS$	simulation of action systems	Section 2.2
$C \preccurlyeq_R^\circ D$	simulation of classes with single object	Section 4.2
$C \preccurlyeq_R^\uparrow D$	simulation of classes with dynamic object structures	Section 5.1

Figure 1. Summary of ordering relations

3.1. Classes

Let Σ be the type of the attributes of some class C and let α 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, classes have several attributes and programs contain several global variables. Thus, elements of Σ and α are tuples. Attribute and variable names are used for accessing the corresponding components. The set of methods and actions of a class is represented 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

$$CM_i = \alpha \times \Sigma \times \Delta_i \times \Omega_i \mapsto \alpha \times \Sigma \times \Delta_i \times \Omega_i \qquad CA = \alpha \times \Sigma \mapsto \alpha \times \Sigma$$

where Δ_i and Ω_i are the types of the value, respectively result parameter of method m_i. Within a class, methods m_i and actions a_j of that class can be referred to 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. The usefulness of this generalization becomes clearer when considering inheritance. Let $self$ stand for the tuple of method and action names prefixed by $self$:

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

Let cm_i be the body of method m_i. Since cm_i may contain calls to other methods and actions of the same object, m_i is a function of $self$:

$$m_i = \lambda self \bullet cm_i$$

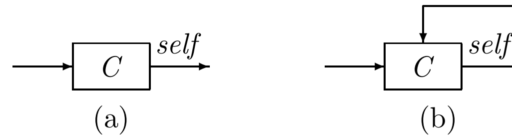

Figure 2 Illustration of (a) class C and of (b) taking the fixed point of C. The incoming arrow represents calls to C, the outgoing arrow stands for self-calls of C.

Thus, the parameter *self* may be used inside cm_i. Actions are treated analogously. 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 $self.a_j : CA$. Note that *self* is here used to refer to methods and actions, but not to reference attributes (fields) of an object. Attributes are referenced with their unqualified names inside methods and actions.

A class also specifies possible initial values $ci : \mathcal{P}\Sigma$ of its attributes c. Hence a class C takes the form of a tuple:

$$C = (ci, cs)$$

Figure 2(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 $c \mid ci$,
  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 2(b)). Methods and actions of objects of class C, denoted by $C.m_i$ and $C.a_i$, respectively, are defined by taking the fixed point of the tuple of all methods and actions and then selecting the corresponding method or action:

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

Declaring a variable x to be of class C means declaring it to be of type Σ and initializing it with ci:

$$\textbf{var}\ x : C \bullet S \;\widehat{=}\; \textbf{var}\ x \mid ci \bullet S$$

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.

$$x.m_i \;\widehat{=}\; \textbf{var}\ c \bullet c := x\ ;\ C.m_i\ ;\ x := c$$

The name of the implicit 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 as for procedure calls. For convenience, we also use the same notation for selecting an action of an object:

$$x.a_i \;\widehat{=}\; \textbf{var}\ c \bullet c := x\ ;\ C.a_i\ ;\ x := c$$

Example. We illustrate the above definitions with a stylized example. Let class E be defined as follows:

class E
  attr $c \mid c = 0$,
  meth *change* **is** $c :\in NAT$,
  meth *inc* **is** $c := c + 1$,
  action a **is** $true \rightarrow self.change$
end

If $E = (ei, es)$, then $ei = \lambda c \bullet (c = 0)$ and es is given by:

$$es = \lambda(self.change, self.inc, self.a) \bullet (c :\in NAT, c := c + 1, true \rightarrow self.change)$$

Taking the fixed point of es results in the substitution of the call to *change* by the definition of *change* in E:

$$\mu\ es = (c :\in NAT, c := c + 1, true \rightarrow c :\in NAT)$$

The use of fixed points becomes clear when we consider overriding in inheritance.

3.2. Inheritance

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 graph as implemented in most object-oriented languages [16]. 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 Λ. We consider only modifiers that 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 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:

$$
\begin{array}{l}
\textbf{modifier } L \\
\quad \textbf{attr } l \mid li, \\
\quad \textbf{meth } m_1(\textbf{val } v_1, \textbf{res } r_1) \textbf{ is } lm_1, \\
\quad \ldots, \\
\quad \textbf{meth } m_m(\textbf{val } v_1, \textbf{res } r_m) \textbf{ is } lm_m, \\
\quad \textbf{action } a_1 \textbf{ is } la_1, \\
\quad \ldots, \\
\quad \textbf{action } a_b \textbf{ is } la_b \\
\textbf{end}
\end{array}
$$

For the types of methods m_i and actions a_j of L we define

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

where β is the type variable for global variables and further attributes in subclasses of D. Thus, we instantiate α 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, that is Δ_i and Ω_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 $super.m_i$ 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. We let $self$ and $super$ stand for:

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

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 $li : \Lambda$ of the new attributes l. Hence a modifier L takes the form of a tuple:

$$L = (li, ls)$$

The modification of C by L binds super-calls in L to C and leaves the self-calls in L and C unresolved for possible further modification (Figure 3(b)):

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

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. Self-calls in L **mod** C, including those in methods and action of C, are bound to L when an object is instantiated (Figure 3(c)).

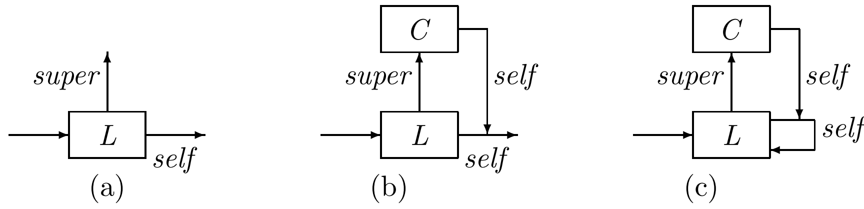

Figure 3 Illustration of (a) modifier L, of (b) L **mod** C, and of (c) taking the fixed point of L **mod** C

Example. We illustrate inheritance by extending class E of Section 3.1. Modifier F overrides method *change* and adds action b:

$$\begin{array}{l} \textbf{modifier } F \\ \quad \textbf{meth } \mathit{change} \textbf{ is } \mathit{super}.\mathit{inc}(), \\ \quad \textbf{action } b \textbf{ is } c < 10 \rightarrow \mathit{self}.\mathit{inc}() \\ \textbf{end} \end{array}$$

If $F = (\mathit{fi}, \mathit{fs})$, then $\mathit{fi} = \mathit{true}$ and fs is given by:

$$\begin{aligned} \mathit{fs} = {} & \lambda(\mathit{self}.\mathit{change}, \mathit{self}.\mathit{inc}, \mathit{self}.a, \mathit{self}.b) \bullet \\ & \quad \lambda(\mathit{super}.\mathit{change}, \mathit{super}.\mathit{inc}, \mathit{super}.a) \bullet \\ & \qquad (\mathit{super}.\mathit{inc}(), \mathit{super}.\mathit{inc}(), \mathit{super}.a, c < 10 \rightarrow \mathit{self}.\mathit{inc}()) \end{aligned}$$

The second and third component are the implicit supercalls of not explicitly redefined method *inc* and action a. The application F **mod** E gives the following:

$$\begin{array}{lcl} F \textbf{ mod } E & = & (gi, gs) \\ gi & = & \lambda c \bullet (c = 0) \\ gs & = & \lambda(\mathit{self}.\mathit{change}, \mathit{self}.\mathit{inc}, \mathit{self}.a, \mathit{self}.b) \bullet (c := c + 1, c := c + 1, \\ & & \quad \mathit{true} \rightarrow \mathit{self}.\mathit{change}(), c < 10 \rightarrow \mathit{self}.\mathit{inc}()) \end{array}$$

This illustrates that the super-calls are bound to the definitions in E. On the other hand, the self-calls in both E and F are still unresolved. This makes it possible to add another modifier to F **mod** E. The self-calls are again bound when an instance of F **mod** E is created:

$$\mu\ gs = (c := c + 1, c := c + 1, \mathit{true} \rightarrow c := c + 1, c < 10 \rightarrow c := c + 1)$$

4. Class Refinement and Class Simulation

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.

4.1. Class Refinement

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:

$$\mathcal{A}[x] = \mathbf{do}\ x.a_1 \parallel \ldots \parallel x.a_a\ \mathbf{od}$$

Let $\mathcal{O}[x]$ be an action system observing object x only through method calls: we represent $\mathcal{O}[x]$ as the (guarded) choice of either aborting or calling a method of x, where additionally local variables may be updated between method calls. Let $SA, S_1, \ldots, S_m$ be universally conjunctive statements that are independent of the global state, i.e. they access only local variables h:

$$\mathcal{O}[x] = \mathbf{var}\ h \mid hi \bullet \mathbf{do}\ SA\ ;\ abort \parallel S_1\ ;\ x.m_1 \parallel \ldots \parallel S_m\ ;\ x.m_m\ \mathbf{od}$$

Let $\mathcal{K}[C]$ be a 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. We describe $\mathcal{K}[C]$ by an interleaving of method calls to x and of actions of x:

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

Class D is a refinement of class C, written $C \preceq^\circ D$, if using an object of class D instead of C in all possible programs yields a trace refinement of the original program:

$$C \preceq^\circ D \ \mathrel{\widehat{=}}\ \forall \mathcal{K} \bullet \mathcal{K}[C] \preceq \mathcal{K}[D]$$

Class refinement between two classes is independent of how the classes are constructed using inheritance. However, it is considered good practice if a class refines all its superclasses, particularly in languages in which inheritance leads to subtyping (i.e. substitutability).

Our theory of refinement applies to classes with inheritance and self- and super-calls as introduced above. Because self- and super-calls in methods and actions are resolved before refinement is considered, there is no textually explicit resolution with fixed points here. Therefore, our treatment of refinement is independent of the model for inheritance and self- and super-calls and is also applicable to models lacking these concepts. In summary, our notion of refinement is targeted at the model of classes introduced in Section 3, but is independent enough to be applicable to other models as well.

4.2. Class Simulation

For proving refinement between classes $C = (ci, cs)$ and $D = (di, ds)$ we use a simulation with a refinement relation R. Define $CI = \mathbf{enter}\ c \mid ci$, $DI = \mathbf{enter}\ d \mid di$, 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 $C \preccurlyeq^\circ_R D$, 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:	$CI\ ;\ CX_\natural^*\ ;\ [R] \sqsubseteq DI\ ;\ DX_\natural^*$
(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:	$R[\mathit{trm}\ CX \wedge \mathit{trm}\ (\mathbf{do}\ CX_\natural\ \mathbf{od})] \leq \mathit{trm}\ (\mathbf{do}\ DX_\natural\ \mathbf{od})$

Theorem 4.1. *Let C and D be classes and R a relation. Then:*

$$C \preccurlyeq_R^\circ D \Rightarrow C \preceq^\circ D$$

Proof:
By the subordinate lemma below and Theorem 2.1. □

Lemma 4.1. *Let C and D be classes and R a relation. Then:*

$$C \preccurlyeq_R^\circ D \Rightarrow \forall \mathcal{K} \bullet \mathcal{K}[C] \preccurlyeq_R \mathcal{K}[D]$$

Proof:
We define:

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

We have to show that (a) to (f) above imply $\mathcal{K}[C] \preccurlyeq_R \mathcal{K}[D]$ for any $\mathcal{K}$ as above, which means that for any hi, SA, and $S_1, \ldots, S_m$:

$$\begin{array}{l} \mathbf{var}\ h \mid hi \bullet \mathbf{var}\ c \mid ci \bullet \mathbf{do}\ CY \parallel CX\ \mathbf{od} \preccurlyeq_R \\ \mathbf{var}\ h \mid hi \bullet \mathbf{var}\ d \mid di \bullet \mathbf{do}\ DY \parallel DX\ \mathbf{od} \end{array}$$

We note that R is independent of h, hence h is not involved in the refinement. According to the definition of action system simulation (Section 2.2) with $AI := CI$, $A_\sharp := CY \sqcap CX_\sharp$, $A_\natural := CX_\natural$, $BI := DI$, $B_\sharp := DY \sqcap DX_\sharp$, and $B_\natural := DX_\natural$ we get four conditions:

(1)	Initialization:	$CI\ ;\ CX_\natural^*\ ;\ [R] \sqsubseteq DI\ ;\ DX_\natural^*$
(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:	$R[\mathit{trm}\ (CY \sqcap CX) \wedge \mathit{trm}\ (\mathbf{do}\ CX_\natural\ \mathbf{od})] \leq \mathit{trm}\ (\mathbf{do}\ DX_\natural\ \mathbf{od})$

Condition (1) follows immediately from (a). For (2) we calculate, for any SA and $S_1, \ldots, S_m$:

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

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

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

The last line follows from (b). For (3) we calculate, for any SA and $S_1, \ldots, S_m$:

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

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

$$
\begin{array}{ll}
& R[trm\ CY \wedge trm\ CX \wedge grd\ CY] \leq grd\ DY \vee grd\ DX \\
\Leftarrow & \quad \{grd(S \ ; \ T) \leq grd\ T \text{ if } S \text{ universally conjunctive and } S, T \text{ independent}\} \\
& R[trm\ CY \wedge trm\ CX \wedge grd\ CY] \leq \\
& \quad grd\ DY \vee grd\ (SA \ ; \ DX) \vee \ldots \vee grd\ (S_m \ ; \ DX) \\
\equiv & \quad \{grd\ (S \sqcap T) = grd\ S \vee grd\ T \text{ for any } S, T\} \\
& R[trm\ CY \wedge trm\ CX \wedge grd\ CY] \leq grd\ (DY \sqcap (SA \ ; \ DX) \sqcap \ldots \sqcap (S_m \ ; \ DX)) \\
\equiv & \quad \{R[p] \leq q \equiv p \leq [R]\ q \text{ and } [R](grd\ S) = grd\ (\{R\} \ ; \ S)\ (*)\} \\
& trm\ CY \wedge trm\ CX \wedge grd\ CY \leq \\
& \quad grd\ (\{R\} \ ; \ (DY \sqcap (SA \ ; \ DX) \sqcap \ldots \sqcap (S_m \ ; \ DX))) \\
\equiv & \quad \{; \text{ distributes over } \sqcap \text{ and } abort \sqcap S = abort \text{ for any } S\} \\
& trm\ CY \wedge trm\ CX \wedge grd\ CY \leq grd\ ((\{R\} \ ; \ SI \ ; \ abort) \sqcap \\
& \quad (\{R\} \ ; \ S_1 \ ; \ (D.m_1 \sqcap DX)) \sqcap \ldots \sqcap (\{R\} \ ; \ S_m \ ; \ (D.m_m \sqcap DX))) \\
\Leftarrow & \quad \{\{R\} \ ; \ S \sqsubseteq S \ ; \ \{R\} \text{ if } R, S \text{ independent and } grd\ U \leq grd\ T \text{ if } T \sqsubseteq U\}
\end{array}
$$

$$
\begin{array}{ll}
& trm\ CY \wedge trm\ CX \wedge grd\ CY \leq grd\ ((SA\ ;\ \{R\}\ ;\ abort) \sqcap \\
& \quad (S_1\ ;\ \{R\}\ ;\ (D.m_1 \sqcap DX)) \sqcap \ldots \sqcap (S_m\ ;\ \{R\}\ ;\ (D.m_m \sqcap DX))) \\
\Leftarrow & \quad \{ 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\ (SA\ ;\ abort) \wedge trm\ CX \wedge grd\ (SI\ ;\ abort) \leq grd\ (SI\ ;\ \{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 & \quad \{\{R\}\ ;\ abort = abort \text{ for any } R\} \\
& \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 & \quad \{ trm\ T \leq trm\ (S\ ;\ T) \text{ if } S \text{ universally conjunctive and } S, T \text{ independent}\} \\
& \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 & \quad \{ trm\ (S \sqcap T) = trm\ S \wedge trm\ S \text{ for any } S, T \text{ and } ;\ \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 & \quad \{(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 grd\ (\{R\}\ ;\ (D.m_i \sqcap DX)) \\
\equiv & \quad \{(*) \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 & \quad \{ 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 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 [9] and in a revised form in [30], which is similar to the corresponding theorem for OO-action systems in [13]. 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 refinements of *skip* is obtained as a special case. Let C and D be classes and let CI, DI, 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:	$CI\ ;\ [R] \sqsubseteq DI$	
(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 4.2. *Let C and D be classes and R a relation as above. If conditions (a') – (g') hold then $C \preccurlyeq^\circ_R D$.*

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 from the following calculation:

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

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

Corollary 4.1. *Let C and D be classes and R a relation as above. If conditions (a') – (g') hold then $C \preceq^\circ D$.*

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. The environment is expressed as an action system that can access the global variables, but cannot access the (single) object of the class in question.

Theorem 4.3. *Let C and D be classes, ES be an action systems, and R be a relation. If ES does not interfere with R then:*

$$C \preccurlyeq^\circ_R D \Rightarrow \forall \mathcal{K} \bullet \mathcal{K}[C] \parallel ES \preceq \mathcal{K}[D] \parallel ES$$

Proof:
By Lemma 4.1 and Theorem 2.2. □

4.3. 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. For its use as 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 $x, y, col \mid 0 \leq x < w \land 0 \leq y < h \land col \neq 0$,
  meth $move($**val** dx, **val** $dy)$ **is**
    $0 \leq x + dx < w \land 0 \leq y + dy < h \rightarrow$
      $skip \sqcap (s[x, y] := 0 \; ; \; x := x + dx \; ; \; y := y + dy \; ; \; s[x, y] := col)$,
  action *newpos* **is**
    $s[x, y] := 0 \; ; \; x :\in \{0..w-1\} \; ; \; y :\in \{0..h-1\} \; ; \; s[x, y] := col$
end

Creatures described by class *Ray* are a refinement with a special form of movement. Rather than jumping wildly around the screen, rays are always at the same vertical position, have a horizontal speed sx, and move at most 3 pixels at once:

class *Ray*
  attr $x, y, col, sx \mid x = 0 \land 0 \leq y < h \land col = 5 \land sx = 1$,
  meth $move($**val** dx, **val** $dy)$ **is**
    $0 \leq x + dx < w \land -3 \leq dx \leq 3 \land dy = 0 \rightarrow$
      $s[x, y] := 0 \; ; \; x := x + dx \; ; \; s[x, y] := col$,
  action *newpos* **is**
    $0 \leq x + sx < w \rightarrow s[x, y] := 0 \; ; \; x := x + sx \; ; \; s[x, y] := col$,
  action *bouncel* **is** $x + sx < 0 \rightarrow sx :\in \{1..3\}$,
  action *bouncer* **is** $w \leq x + sx \rightarrow sx :\in \{-3..-1\}$
end

Class *Ray* refines class *Creature* with refinement relation R:

$$R \; (s, x, y, col) \; (s', x', y', col', sx') \;\; \equiv \;\; \begin{array}{l} s = s' \land x = x' \land 0 \leq x < w \land y = y' \land \\ 0 \leq y < h \land col = col' \land -3 \leq sx' \leq 3 \end{array}$$

We can use Theorem 4.2 to prove $Creature \preccurlyeq^{\circ}_{R} Ray$ because we have no explicit abstract stuttering. We set $CX := Creature.newpos$, $DX_{\sharp} := Ray.newpos$, $DX_{\natural} := Ray.bouncel \sqcap Ray.bouncer$, and CI and DI to the respective initialization. Internal convergence (condition (g')) follows by

transitivity from the calculation below (assuming that an access to s outside the screen aborts):

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

The other conditions can also be proved by unfolding the definitions and refinement rules. By Corollary 4.1 we also get $Creature \preceq^\circ Ray$. Hence, replacing a *Creature* by a *Ray* in any context $\mathcal{K}$ produces a trace refinement.

5. Dynamic Object Structures

In this section we introduce dynamic object structures, which allow multiple objects to run concurrently. Furthermore, we extend the discussion of class refinement and class simulation to this setting.

We model the heap as an array and pointers as indices into this array [25]. We first describe the basic ideas using only one class and then generalize it to multiple classes with subtypes.

5.1. Single Class

For a class C with attributes of type Σ we declare a program variable *heap* to contain all dynamically created objects:

$$\mathbf{var}\ heap : \mathbf{array}\ NAT\ \mathbf{of}\ \Sigma$$

Pointers to objects 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. If *ci* is the initialization of the attributes of C and p is a pointer, p : **pointer to** C, then the creation of a new object is defined by:

$$p := new\ C \ \mathrel{\widehat{=}}\ p := next\ ;\ (\sqcap c \mid ci \bullet heap[p] := c)\ ;\ next := next + 1$$

To handle the way how attributes of objects on the heap are referenced, we have to introduce an indirection for each attribute reference via the receiver (the current object). We denote the receiver by *this* and introduce the shorthand *this.c* for referencing the attribute c of the object *heap*[*this*]:

$$this.c \ \mathrel{\widehat{=}}\ heap[this].c$$

We use this shorthand in both expressions and for assignments in methods. A method call $p.m$ is then defined as (We use the restricted choice rather than the variable notation for *this* because the latter is a constant rather than a program variable.):

$$p.m \;\widehat{=}\; \{p \neq nil\} \;;\; (\sqcap this \mid this = p \bullet C.m)$$

Parameter passing is handled as for procedures. In our formalization, *this* is used to reference the receiver object whereas *self* and *super* are used in classes to reference methods and actions.

Formally, a class C with dynamically created objects is given by $C = (ci, cs)$ as previously, except that *heap* is now necessarily part of the global state and all references in cs to attributes go via *heap*. The selection $C.m_i$ and $C.a_i$ are defined as previously and we use the same syntax:

class C
  attr $c \mid ci$,
  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

With the declaration of class C as above, we associate an action system $\mathcal{A}[C]$ which consists of actions operating on all objects of that class:

$$\mathcal{A}[C] = \textbf{do}\ ([\![this \mid 1 \leq this < next \bullet C.a_1 [\!] \ldots [\!] C.a_a)\ \textbf{od}$$

This action system is composed in parallel with any other action system using objects of class C.

Example. A class *Creature* with dynamically created objects could be defined by:

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

This declaration stands for:

$$
\begin{array}{l}
\textbf{var } \mathit{heap} : \textbf{array } \mathit{NAT} \textbf{ of } \mathit{NAT} \times \mathit{NAT} \times \mathit{NAT} \\
\textbf{var } \mathit{next} \mid \mathit{next} = 1 \\
\textbf{class } \mathit{Creature} \\
\quad \textbf{meth } \mathit{move}(\textbf{val } \mathit{dx}, \textbf{val } \mathit{dy}) \textbf{ is} \\
\quad\quad 0 \le \mathit{heap}[\mathit{this}].x + \mathit{dx} < w \wedge 0 \le \mathit{heap}[\mathit{this}].y + \mathit{dy} < h \rightarrow \\
\quad\quad\quad \mathit{skip} \sqcap (s[\mathit{heap}[\mathit{this}].x, \mathit{heap}[\mathit{this}].y] := 0 \ ; \\
\quad\quad\quad\quad \mathit{heap}[\mathit{this}].x := \mathit{heap}[\mathit{this}].x + \mathit{dx} \ ; \\
\quad\quad\quad\quad \mathit{heap}[\mathit{this}].y := \mathit{heap}[\mathit{this}].y + \mathit{dy} \ ; \\
\quad\quad\quad\quad s[\mathit{heap}[\mathit{this}].x, \mathit{heap}[\mathit{this}].y] := \mathit{heap}[\mathit{this}].\mathit{col}), \\
\quad \textbf{action } \mathit{newpos} \textbf{ is} \\
\quad\quad s[\mathit{heap}[\mathit{this}].x, \mathit{heap}[\mathit{this}].y] := 0 \ ; \\
\quad\quad \mathit{heap}[\mathit{this}].x :\in \{0..w-1\} \ ; \ \mathit{heap}[\mathit{this}].y :\in \{0..h-1\} \ ; \\
\quad\quad s[\mathit{heap}[\mathit{this}].x, \mathit{heap}[\mathit{this}].y] := \mathit{heap}[\mathit{this}].\mathit{col} \\
\textbf{end}
\end{array}
$$

If cr is a pointer to a *Creature* object, cr : **pointer to** *Creature*, then $cr := \mathit{new}\ \mathit{Creature}$ is defined by:

$$
\begin{array}{l}
\mathit{cr} := \mathit{next} \ ; \\
(\sqcap x, y, \mathit{col} \mid 0 \le x < w \wedge 0 \le y < h \wedge \mathit{col} \ne 0 \bullet \mathit{heap}[\mathit{cr}] := (x, y, \mathit{col})) \ ; \\
\mathit{next} := \mathit{next} + 1
\end{array}
$$

A method call $\mathit{cr}.\mathit{move}(2,7)$ stands for:

$$
\{\mathit{cr} \ne \mathit{nil}\} \ ; \ (\sqcap \mathit{this} \mid \mathit{this} = \mathit{cr} \bullet \mathit{Creature}.\mathit{move}(2,7))
$$

The method selection $\mathit{Creature}.\mathit{move}(2,7)$ stands for:

$$
\begin{array}{l}
\quad \textbf{var } \mathit{dx}, \mathit{dy} \bullet \mathit{dx}, \mathit{dy} := 2, 7 \ ; \\
0 \le \mathit{heap}[\mathit{this}].x + \mathit{dx} < w \wedge 0 \le \mathit{heap}[\mathit{this}].y + \mathit{dy} < h \rightarrow \\
\quad \mathit{skip} \sqcap (s[\mathit{heap}[\mathit{this}].x, \mathit{heap}[\mathit{this}].y] := 0 \ ; \ \mathit{heap}[\mathit{this}].x := \mathit{heap}[\mathit{this}].x + \mathit{dx} \ ; \\
\quad \mathit{heap}[\mathit{this}].y := \mathit{heap}[\mathit{this}].y + \mathit{dy} \ ; \\
\quad s[\mathit{heap}[\mathit{this}].x, \mathit{heap}[\mathit{this}].y] := \mathit{heap}[\mathit{this}].\mathit{col})
\end{array}
$$

The action system $\mathcal{A}[\mathit{Creature}]$ associated with *Creature* is:

$$
\begin{array}{l}
\textbf{do} \\
\quad ([\!] \mathit{this} \mid 1 \le \mathit{this} < \mathit{next} \bullet \\
\quad\quad s[\mathit{heap}[\mathit{this}].x, \mathit{heap}[\mathit{this}].y] := 0 \ ; \ \mathit{heap}[\mathit{this}].x :\in \{0..w-1\} \ ; \\
\quad\quad \mathit{heap}[\mathit{this}].y :\in \{0..h-1\} \ ; \ s[\mathit{heap}[\mathit{this}].x, \mathit{heap}[\mathit{this}].y] := \mathit{heap}[\mathit{this}].\mathit{col}) \\
\textbf{od}
\end{array}
$$

5.2. Class Refinement and Class Simulation

We show that with the above definitions the notion of class refinement carries over analogously to dynamic object structures. With the declaration of a class C, we associate an action system $\mathcal{O}[C]$, which observes all objects of class C by calling their methods. We represent $\mathcal{O}[C]$ as the (guarded) choice of either aborting or calling a method of x, where additionally local variables may be updated between method calls. Let $SA, S_1, \ldots, S_m, SC$ be universally conjunctive statements that are independent of the global state, i.e. they access only local variables h:

$$\begin{array}{l}\mathcal{O}[C] = \\ \quad \textbf{var}\ h \mid hi \bullet \\ \quad \textbf{do}\ SA\ ;\ abort \\ \quad [\!](\,[\!]\,this \mid 1 \leq this < next \bullet S_1\ ;\ C.m_1 \,[\!]\, \ldots \,[\!]\, S_m\ ;\ C.m_m) \\ \quad [\!]\,SC\ ;\ p := new\ C \\ \quad \textbf{od}\end{array}$$

Here we assume that p is part of the local variables h. Let $\mathcal{K}[C]$ be a program operating on objects of class C such that $\mathcal{K}$ is the full context of objects of class C, in the sense that no other program accesses the attributes of objects of C or creates new objects of C. We describe $\mathcal{K}[C]$ by an interleaving of method calls to instances of C, creation of new instances of C, and actions of instances of C:

$$\mathcal{K}[C] = \textbf{var}\ heap, next \mid next = 1 \bullet \mathcal{O}[C] \parallel \mathcal{A}[C]$$

Class D is a refinement of class C, written $C \preceq^{\uparrow} D$, if using objects of class D instead of C in all possible programs yields a trace refinement of the original program:

$$C \preceq^{\uparrow} D \ \ \widehat{=}\ \ \forall \mathcal{K} \bullet \mathcal{K}[C] \preceq \mathcal{K}[D]$$

The conditions for simulation between two classes with dynamically created objects are like those for simulation with a single object, except that all objects on the heap are in the refinement relation. Let R be a refinement relation between classes $C = (ci, cs)$ and $D = (di, ds)$ such that

$$next = 1 \Rightarrow R(u, heap, next)(u', next' heap')$$

where u are the global variables. That is, if the heap is empty the refinement relation must hold. Furthermore we define $CC = p := new\ C$, $DC = p := new\ D$, and

$$\begin{array}{l}CX = (\sqcap this \mid 1 \leq this < next \bullet C.a_1 \sqcap \ldots \sqcap C.a_a) \\ DX = (\sqcap this \mid 1 \leq this < next \bullet D.a_1 \sqcap \ldots \sqcap D.a_b)\end{array}$$

Class C is simulated by D using R, written $C \preccurlyeq^{\uparrow}_{R} D$, 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)	Creation:	$CC\ ;\ CX_\natural^*\ ;\ [R] \sqsubseteq [R]\ ;\ DC\ ;\ DX_\natural^*$
(b)	Methods:	$C.m_i\ ;\ CX_\natural^*\ ;\ [R] \sqsubseteq [1 \leq \mathit{this} < \mathit{next}]\ ;\ [R]\ ;\ D.m_i\ ;\ DX_\natural^*$ 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:	$R[1 \leq \mathit{this} < \mathit{next} \wedge \mathit{trm}\ C.m_i \wedge \mathit{trm}\ CX \wedge \mathit{grd}\ C.m_i] \leq \mathit{grd}\ D.m_i \vee \mathit{grd}\ DX$ 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:	$R[\mathit{trm}\ CX \wedge \mathit{trm}\ (\mathbf{do}\ CX_\natural\ \mathbf{od})] \leq \mathit{trm}\ (\mathbf{do}\ DX_\natural\ \mathbf{od})$

Theorem 5.1. *Let C and D be classes and R a relation. Then:*

$$C \preccurlyeq_R^{\uparrow} D \Rightarrow C \preceq^{\uparrow} D$$

Proof:
By the subordinate lemma below and Theorem 2.1. □

Lemma 5.1. *Let C and D be classes and R a relation. Then:*

$$C \preccurlyeq_R^{\uparrow} D \Rightarrow \forall \mathcal{K} \bullet \mathcal{K}[C] \preccurlyeq_R \mathcal{K}[D]$$

Proof:
We define:

$$\begin{array}{lcl} CY & = & (SA\ ;\ \mathit{abort}) \sqcap \\ & & (\sqcap \mathit{this} \mid 1 \leq \mathit{this} < \mathit{next} \bullet (S_1\ ;\ C.m_1) \sqcap \ldots \sqcap (S_m\ ;\ C.m_m)) \sqcap \\ & & (SC\ ;\ CC) \\ DY & = & (SA\ ;\ \mathit{abort}) \sqcap \\ & & (\sqcap \mathit{this} \mid 1 \leq \mathit{this} < \mathit{next} \bullet (S_1\ ;\ D.m_1) \sqcap \ldots \sqcap (S_m\ ;\ D.m_m)) \sqcap \\ & & (SC\ ;\ DC) \\ CI & = & \mathbf{enter}\ \mathit{heap}, \mathit{next} \mid \mathit{next} = 1 \\ DI & = & \mathbf{enter}\ \mathit{heap}, \mathit{next} \mid \mathit{next} = 1 \end{array}$$

Leaving out the types, we note that *heap* in CI is an array of C attributes and *heap* in DI is an array of D attributes. We have to show that (a) to (f) above imply $\mathcal{K}[C] \preccurlyeq_R \mathcal{K}[D]$ for any $\mathcal{K}$ as above, which means that for any hi, SA, $S_1, \ldots, S_m$, and SC:

$$\begin{array}{l} \mathbf{var}\ h \mid hi \bullet \mathbf{var}\ \mathit{heap}, \mathit{next} \mid \mathit{next} = 1 \bullet \mathbf{do}\ CY \parallel CX\ \mathbf{od} \preccurlyeq_R \\ \mathbf{var}\ h \mid hi \bullet \mathbf{var}\ \mathit{heap}, \mathit{next} \mid \mathit{next} = 1 \bullet \mathbf{do}\ DY \parallel DX\ \mathbf{od} \end{array}$$

We note that R is independent of h, hence h is not involved in the refinement. According to the definition of action system simulation (Section 2.2) with $AI := CI$, $A_\sharp := CY \sqcap CX_\sharp$, $A_\natural := CX_\natural$, $BI := DI$, $B_\sharp := DY \sqcap DX_\sharp$, $B_\natural := DX_\natural$, and $R := R$ we get four conditions:

(1)	Initialization:	$CI\ ;\ CX_\natural^*\ ;\ [R] \sqsubseteq DI\ ;\ DX_\natural^*$
(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:	$R[\mathit{trm}\ (CY \sqcap CX) \wedge \mathit{trm}\ (\mathbf{do}\ CX_\natural\ \mathbf{od})] \leq \mathit{trm}\ (\mathbf{do}\ DX_\natural\ \mathbf{od})$

Condition (1) expands to:

$$\textbf{enter}\ heap, next \mid next = 1\ ;\ CX_\natural^*\ ;\ [R] \sqsubseteq \textbf{enter}\ heap, next \mid next = 1\ ;\ DX_\natural^*$$

First we note that after the initialization of $next$ by 1, neither $CX_\natural$ nor $DX_\natural$ is enabled, as $(\sqcap i \mid false \bullet S) = magic$. Therefore, $CX_\natural^* = skip$ and $DX_\natural^* = skip$. As $next$ is set to 1, the refinement relation is true by the assumption, and the refinement holds vacuously.

For (2) we calculate, for any SA, $S_1, \ldots, S_m$, and SC:

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

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

$$\begin{array}{ll}
& CY\ ;\ CX_\natural^*\ ;\ [R] \sqsubseteq [R]\ ;\ DY\ ;\ DX_\natural^* \\
\equiv & \quad \{\text{definitions of } CY \text{ and } DY \text{ and } ;\ \text{distributes over } \sqcap\} \\
& (SA\ ;\ abort\ ;\ CX_\natural^*\ ;\ [R]) \sqcap \\
& (\sqcap this \mid 1 \le this < next \bullet (S_1\ ;\ C.m_1\ ;\ CX_\natural^*\ ;\ [R]) \sqcap \ldots \\
& \quad \sqcap (S_m\ ;\ C.m_m\ ;\ CX_\natural^*\ ;\ [R])) \sqcap \\
& (SC\ ;\ CC\ ;\ CX_\natural^*\ ;\ [R]) \sqsubseteq \\
& ([R]\ ;\ SA\ ;\ abort\ ;\ DX_\natural^*) \sqcap \\
& (\sqcap this \mid 1 \le this < next \bullet ([R]\ ;\ S_1\ ;\ D.m_1\ ;\ DX_\natural^*) \sqcap \ldots \\
& \quad \sqcap ([R]\ ;\ S_m\ ;\ D.m_m\ ;\ DX_\natural^*)) \sqcap \\
& ([R]\ ;\ SC\ ;\ DC\ ;\ DX_\natural^*) \\
\Leftarrow & \quad \{\text{monotonicity}\} \\
& (SA\ ;\ abort\ ;\ CX_\natural^*\ ;\ [R] \sqsubseteq [R]\ ;\ SA\ ;\ abort\ ;\ DX_\natural^*) \wedge \\
& ((\sqcap this \mid 1 \le this < next \bullet (S_1\ ;\ C.m_1\ ;\ CX_\natural^*\ ;\ [R]) \sqcap \ldots \\
& \qquad \sqcap (S_m\ ;\ C.m_m\ ;\ CX_\natural^*\ ;\ [R])) \sqsubseteq \\
& \quad (\sqcap this \mid 1 \le this < next \bullet ([R]\ ;\ S_1\ ;\ D.m_1\ ;\ DX_\natural^*) \sqcap \ldots \\
& \qquad \sqcap ([R]\ ;\ S_m\ ;\ D.m_m\ ;\ DX_\natural^*))) \wedge \\
& (SC\ ;\ CC\ ;\ CX_\natural^*\ ;\ [R] \sqsubseteq [R]\ ;\ SC\ ;\ DC\ ;\ DX_\natural^*) \\
\Leftarrow & \quad \{S\ ;\ [R] \sqsubseteq [R]\ ;\ S \text{ for independent } R, S \text{ and } abort\ ;\ S = abort \text{ for any } S\} \\
& ((\sqcap this \mid 1 \le this < next \bullet (S_1\ ;\ C.m_1\ ;\ CX_\natural^*\ ;\ [R]) \sqcap \ldots \\
& \qquad \sqcap (S_m\ ;\ C.m_m\ ;\ CX_\natural^*\ ;\ [R])) \sqsubseteq \\
& \quad (\sqcap this \mid 1 \le this < next \bullet ([R]\ ;\ S_1\ ;\ D.m_1\ ;\ DX_\natural^*) \sqcap \ldots \\
& \qquad \sqcap ([R]\ ;\ S_m\ ;\ D.m_m\ ;\ DX_\natural^*))) \wedge \\
& (SC\ ;\ CC\ ;\ CX_\natural^*\ ;\ [R] \sqsubseteq [R]\ ;\ SC\ ;\ DC\ ;\ DX_\natural^*) \\
\Leftarrow & \quad \{\text{definition of } \sqcap i \mid p \bullet S \text{ and} \\
& \qquad (\forall i \bullet S \sqsubseteq T) \Rightarrow (\sqcap i \bullet S) \sqsubseteq (\sqcap i \bullet T) \text{ for any } S, T\}
\end{array}$$

$$
\begin{array}{ll}
 & (\forall \mathit{this} \bullet \forall i \in \{1, \ldots, m\} \bullet \\
 & [1 \leq \mathit{this} < \mathit{next}] \ ; \ S_i \ ; \ C.m_i \ ; \ CX_\natural^* \ ; \ [R] \sqsubseteq \\
 & \quad [1 \leq \mathit{this} < \mathit{next}] \ ; \ [R] \ ; \ S_i \ ; \ D.m_i \ ; \ DX_\natural^*) \wedge \\
 & (SC \ ; \ CC \ ; \ CX_\natural^* \ ; \ [R] \sqsubseteq [R] \ ; \ SC \ ; \ DC \ ; \ DX_\natural^*) \\
\Leftarrow & \quad \{S_i \text{ and } R \text{ and } SC \text{ and } R \text{ independent, refinement calculus}\} \\
 & (\forall \mathit{this} \bullet \forall i \in \{1, \ldots, m\} \bullet \\
 & \quad S_i \ ; \ C.m_i \ ; \ CX_\natural^* \ ; \ [R] \sqsubseteq [1 \leq \mathit{this} < \mathit{next}] \ ; \ S_i \ ; \ [R] \ ; \ D.m_i \ ; \ DX_\natural^*) \wedge \\
 & (SC \ ; \ CC \ ; \ CX_\natural^* \ ; \ [R] \sqsubseteq SC \ ; \ [R] \ ; \ DC \ ; \ DX_\natural^*)
\end{array}
$$

The first conjunct follows from (b) and the second from (a). The proof of (3) is similar to the one of the corresponding condition in Theorem 4.1 and is left out for brevity. Condition (4) follows from (f) by monotonicity. □

As for the case with a single object, class refinement with dynamic object structures is compositional only under the additional constraint of non-interference with the environment. The environment takes the form of an action system that can access the global variables, but cannot access the heap with the objects of the class in question.

Theorem 5.2. *Let C and D be classes, ES be an action systems, and R be a relation. If ES does not interfere with R then:*

$$C \preccurlyeq_R^{\uparrow} D \Rightarrow \forall \mathcal{K} \bullet \mathcal{K}[C] \parallel ES \preceq \mathcal{K}[D] \parallel ES$$

Proof:
By Lemma 5.1 and Theorem 2.2. □

5.3. Multiple Classes and Subtyping

This formalization easily extends to multiple classes with subtyping. We declare for each class C_i with attribute type Σ_i a separate heap_i : **array** NAT **of** Σ_i. Thus with a class declaration **class** $C_i \ldots$ **end** we associate:

$$
\begin{array}{l}
\textbf{var } \mathit{heap}_i : \textbf{array } NAT \textbf{ of } \Sigma_i, \\
\textbf{var } \mathit{next}_i \mid \mathit{next}_i = 1
\end{array}
$$

Pointers are extended to tuples with one index indicating the heap and one index indicating the element within the heap. A pointer variable declaration p : **pointer to** C_i stands for $p : NAT \times NAT$. The first component of a pointer p is selected by $p.\mathit{class}$, the second component by $p.\mathit{ref}$. The nil value is always represented by $(0, 0)$ to make it unique.

Assuming that $C_k, \ldots, C_l$ are all subtypes of C_i (including C_i), object creation, method calls

with dynamic dispatch, type tests, and attribute access are defined by:

$$
\begin{array}{lcl}
p := \mathit{new}\ C_i & \widehat{=} & p := (i, \mathit{next}_i)\ ;\ (\sqcap c \mid ci_i \bullet \mathit{heap}_i[\mathit{next}_i] := c)\ ; \\
 & & \mathit{next}_i := \mathit{next}_i + 1 \\
p.m & \widehat{=} & \{p \neq \mathit{nil}\}\ ; \\
 & & (\sqcap \mathit{this} \mid \mathit{this} = p \bullet p.\mathit{class} = k \rightarrow C_k.m \sqcap \ldots \sqcap \\
 & & \qquad\qquad p.\mathit{class} = l \rightarrow C_l.m) \\
p\ \textbf{instanceof}\ C_i & \widehat{=} & p.\mathit{class} \in \{k, \ldots, l\} \\
x := \mathit{this}.c & \widehat{=} & \mathit{this}.\mathit{class} = k \rightarrow x := \mathit{heap}_k[\mathit{this}.\mathit{ref}].c \sqcap \ldots \sqcap \\
 & & \quad \mathit{this}.\mathit{class} = l \rightarrow x := \mathit{heap}_l[\mathit{this}.\mathit{ref}].c
\end{array}
$$

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:

$$
\begin{array}{ll}
\mathcal{A}[C_i] = & \textbf{do} \\
 & ([\!] \mathit{this} \mid \mathit{this}.\mathit{class} = i \wedge 1 \leq \mathit{this}.\mathit{ref} < \mathit{next}_i \bullet C_i.a_1 \sqcap \ldots \sqcap C_i.a_a) \\
 & \textbf{od}
\end{array}
$$

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:

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

Example. Let class *Creature* be as defined previously in this section and *Ray* be defined by:

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

We further assume that a similar class *Turtle* is defined. Let *Aquarium* be the main program of an aquarium, expressed as an action system, in which new rays and turtles are constantly added and where the most recently created creature is influenced through its *move* method:

$$
\begin{array}{lcl}
\mathit{Aquarium} & = & \textbf{var}\ p : \textbf{pointer to}\ \mathit{Creature} \mid p = \mathit{nil} \bullet \\
 & & \textbf{do}\ p := \mathit{new}\ \mathit{Ray}\ [\!]\ p := \mathit{new}\ \mathit{Turtle}\ [\!]\ p \neq \mathit{nil} \rightarrow p.\mathit{move}(2, 0)\ \textbf{od}
\end{array}
$$

Then the whole system becomes the parallel composition of the action systems associated with all classes and the main program:

$$\mathcal{A}[\mathit{Creature}] \parallel \mathcal{A}[\mathit{Ray}] \parallel \mathcal{A}[\mathit{Turtle}] \parallel \mathit{Aqaurium}$$

Note that $\mathcal{A}[\mathit{Creature}]$ is only going to affect objects of class *Creature* (of which there are none), $\mathcal{A}[\mathit{Ray}]$ is only going to affect objects of class *Ray*, and similarly for *Turtle*.

6. Early Return

Early returns are a syntactically simple way of increasing concurrency by splitting an action in two parts. In this section, we show how early returns can be defined and how they can be introduced as a special case of atomicity refinement.

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 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:

$$\textbf{meth}\ \mathit{rnd}(\textbf{res}\ y)\ \textbf{is}\ y :\in \mathit{NAT}\ ;\ \textbf{release}\ ;\ \text{`store } y \text{ in sequence'}$$

Figure 4 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. Introducing a **release** in m leads to an earlier completion of the action calling m and allows other actions to be executed in parallel with the remainder T, thus increasing concurrency. For simplicity, we do not allow self-calls in the remainder.

Introducing **release** leads to class refinement under certain conditions. We give a theorem for the case of a single object:

Theorem 6.1. *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:*

$$\textbf{meth}\ C :: m\ \textbf{is}\ S\ ;\ T,$$
$$\textbf{meth}\ D :: m\ \textbf{is}\ S\ ;\ \textbf{release}\ ;\ T$$

We assume that the classes do not contain any self-calls. If T is always enabled, is always terminating, and does not access global variables, then $C \preceq^{\circ} D$ holds.

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

Figure 4. Definition of **release** as enabling a remainder action

Proof:
Without loss of generality we assume that class D is as in Figure 4 and class C is analogously. As the methods and actions do not contain any self-calls, taking their fixpoint is not going to change them, i.e. $C.m = S \; ; \; T$, $C.n = U$, $C.a = V$, $D.m = (lck = 0 \to S \; ; \; lck := 1)$, $D.n = (lck = 0 \to U)$, $D.a = (lck = 0 \to V)$, and $D.r = (lck = 0 \to T \; ; \; lck := 0)$. We apply Theorem 5.1 with $R\ (u, c)\ (u', c', lck') := u' = u \wedge (lck' = 0 \Rightarrow c' = c)$ and $CI :=$ **enter** $c \mid ci$, $CX_\sharp := V$, $CX_\natural := magic$, $DI :=$ **enter** $c, lck \mid ci \wedge lck = 0$, $DX_\sharp := lck = 0 \to V$, $DX_\natural := lck = 1 \to T \; ; \; lck := 0$. The theorem follows by simplifications of the conditions (a) – (f). □

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 5). 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 6:

$$\textbf{action}\ b\ \textbf{is}\ (\textbf{var}\ z : U \bullet o.m(e, z) \; ; \; o.n(e, z))$$

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** $i < l \to y := s[i]$
end

class C
  attr $c := ci$,
  meth $m($**val** v, **res** $r)$ **is**
    var $x \bullet S$; **release** ; T,
  meth $n($**val** w, **res** $s)$ **is**
    U,
  action a **is**
    V
end
	class C
	  attr $c, lck, m_v, m_r, m_x \mid ci \wedge lck = 0$,
	  meth $m($**val** v, **res** $r)$ **is**
	    $lck = 0 \rightarrow$ **var** $x \bullet S$; $lck, m_v, m_r, m_x := 1, v, r, x$,
	  meth $n($**val** w, **res** $s)$ **is**
	    $lck = 0 \rightarrow U$,
	  action a **is**
	    $lck = 0 \rightarrow V$,
	  action r **is**
	    $lck = 1 \rightarrow$ **var** $v, r, x := m_v, m_r, m_x \bullet T$; $lck := 0$
	end

a) Method with **release**	b) Equivalent without **release**

Figure 5. Definition of **release** with remainder accessing parameters and Local Variables

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 6:

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 $C \preccurlyeq^{\circ}_{R} D$ for the following R:

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

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 CI and DI 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,

class C	**class** C
attr $c \mid ci$,	**attr** $c, lck, m_v, m_r, m_x \mid ci \wedge lck = 0$
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 6. Definition of **release** supporting multiple calls to an object within an action

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.

The refinement rules have been developed in a most general form without considering some useful special cases. For example, for the refinement of classes with dynamically created objects each attribute reference goes via the heap. If aliasing can be excluded, the rule could be simplified. Another special case is superposition refinement. When a subclass is created by superposition, the original computation on the inherited attributes is left unchanged. Additional functionality is provided through new attributes. Deriving rules for such special cases is left as future work.

Another point about refinement can be illustrated with the example of Section 4.3: Class *Creature* can be refined by a class that is identical, except that the method *move* is never enabled, i.e. defined as *magic*. All conditions for class simulation hold with *Id* as the refinement relation and no stuttering actions. In particular condition (d) holds as the action *newpos* is always enabled. While our notion of refinement in a sense preserves liveness of the whole system, it allows that certain methods calls become impossible. A stronger notion of refinement preserving the possibility of method calls is worth further study.

Class refinement for concurrent objects is defined here as an extension of class refinement defined in [26, 27], following the general model of classes as self-referential structures with a delayed taking of the fixed point of [31, 16]. As known from [26], inheritance is not monotonic with respect to the refinement of the base class: if C is refined by D, then L **mod** C is not

necessarily refined by L **mod** D. If D is supposed to be a revision of C and L an independently developed extension of C, then this leads to the fragile base class problem, a problem plaguing independent class development and evolution. This problem persists in the concurrent setting. 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 [6]. They can be introduced here without further difficulties.

Many interesting, open questions are connected with early returns. So far we disallowed self-calls in classes with early returns. Also, 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.

Modifications to other objects in the remainder of a method is a useful concept studied by Jones [20]. This is allowed if there are no other references to those objects and hence those changes are not observable to the remaining program. To this aim, Jones uses 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 guaranteed not to be observed until the remainder has been executed. The incorporation of such refinement steps into our formalism is an open issue.

The main advantage of a **release** statement over a "manual" atomicity refinement are the readability (no need to syntactically split the method into parts and to syntactically clutter all 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. The insightful comments of the anonymous referees are also gratefully acknowledged.

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. *Correctness Preserving Program Refinements: Proof Theory and Applications*, volume 131 of *Mathematical Center Tracts*. Mathematical Centre, Amsterdam, 1980.

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

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

[6] 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.

[7] 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.

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

[9] 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.

[10] 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.

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

[12] Ralph Back and Joakim von Wright. Products in the refinement calculus. Technical Report 235, Turku Centre for Computer Science, February 1999.

[13] 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.

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

[15] 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.

[16] 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.

[17] 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.

[18] 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.

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

[20] 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.

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

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

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

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] Jayadev Misra. A discipline of multiprogramming. *ACM Computing Surveys*, 28A(4), December 1996.

[29] Caroll C. Morgan. *Programming from Specifications*. Prentice Hall, 1990.

[30] 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.

[31] 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.