An Approach to Object-Orientation in Action Systems

Marcello M. Bonsangue[1], Joost N. Kok[1], and Kaisa Sere[2]

[1] Department of Computer Science, Rijks Universiteit Leiden
P.O. Box 9512, NL-2300 RA Leiden, The Netherlands
marcello@cs.leidenuniv.nl and joost@cs.leidenuniv.nl

[2] Department of Computer Science, Åbo Akademi University
Turku Centre for Computer Science, FIN-20520 Turku, Finland
Kaisa.Sere@abo.fi

Abstract. We extend the action system formalism with a notion of objects that can be dynamically created, active and distributed. With this extension we can model class-based systems as action systems. Moreover, as the introduced constructs can be translated into ordinary action systems, we can use the theory developed for action systems, especially the refinement calculus, even for class-based systems. We show how inheritance can be modelled in different ways via class refinement. Refining a class with an other class within the refinement calculus ensures that the original behavior of the class is maintained throughout the refinements. Finally, we show how to reuse refinements and entire class modules in a refinement step.

1 Introduction

Object-oriented programming offers a very clear way to structure large systems by encapsulating data and operations into objects, by grouping objects into classes, and by dynamically binding operations to program code. In contrast to the popularity reached by some object-oriented programming languages, there is no general agreement on their formal computational model.

We propose an approach to object-orientation based on action systems, so called OO-action systems. We take a class-based approach to object orientation: OO-action systems model classes and instances of classes, i.e. objects, are created at run time. The objects themselves can be *distributed* and *active*. Communication between objects takes place via remote procedure calls and shared variables.

We show how an OO-action system, i.e. a specification of some classes, is translated into an ordinary action system [BK83]: methods correspond to exportable procedures, attributes to shared and local variables, object variables to local variables, and classes to entire action systems. Moreover, object constructors correspond to initialization statements of an action system. A collection of classes is translated into a parallel composition of ordinary action systems. This translation allows us to use most of the theory built around action systems even when designing OO-action systems. As a side effect we get a mechanism to add processes to action systems, a desirable feature now missing from the formalism.

Action systems are designed stepwise within the refinement calculus [BS96]. We therefore define the notion of *refinement* for OO-action systems within the refinement calculus. The notion we want to preserve by refinement is total correctness if we are interested only on the input-output behaviour of the system, and trace correctness if we are interested on its reactive behaviour. We develop proof rules to be used when designing and reasoning about OO-action systems. The correctness of the rules is shown by appealing to the standard rules of the refinement calculus. The rules can be used to refine both methods, actions, and the constructors of a class.

When we refine a class with another class, the refined class can inherit methods and attributes from the original class. New attributes can be introduced in a refinement step and some methods can be overridden by the refined methods. As our objects are active, a refinement step can introduce additional autonomous behavior in the form of new actions in the class. Class refinement ensures that each set of successful computation with respect to the global variable satisfied by the original class is also satisfied by the refined class. Methods have the same interface throughout the refinement steps. Hence, so called interface refinement can only take place via shared variables.

One of the most useful features of object-orientation is the idea of reuse. OO-action systems employ this idea in two ways. Refinement via inheritance naturally models reuse of code. We additionally show that we can reuse refinement steps in our derivations.

Related work Action systems were originally proposed as a formalism for developing parallel and distributed systems [BK83]. They have similarities with e.g. UNITY-programs [CM88], in particular with the ImpUNITY formalism [UK95]. The action systems formalism has been recently extended from its original form in various ways, also towards object-orientation. Järvinen and Kurki-Suonio [JK91] develop their DisCo framework as a specification language for object-oriented systems. Back et al. [BBS97] show how objects are added into Action-Oberon, an Oberon-like language for parallel programming.

The OO-action system formalism is related with POOL [ABKR86,ABKR89] because objects are created dynamically, and their name can be assigned to variables. Furthermore, objects are active and distributed, hence, several objects are executed in parallel. The first model incorporating active objects was the actor model [Hew77,Agh86]. Moreover, CCS and the π-calculus, have been used to give a semantics to POOL-type parallel object-oriented languages [Jon93,Wal95]. Recently several formalisms and languages have been proposed that offer active objects, e.g. Oblique [Car95] which supports distributed object-oriented computation and Oblets [BN96] which are written in Oblique and which have a family of Web browsers capable of running Oblets.

We reason about OO-action systems in the framework of the refinement calculus. The refinement calculus and related calculi [Mor88,Mor87] have become popular foundations for program construction and for reasoning about specifications and implementations that meet their specifications. The refinement calculus uses weakest precondition predicate transformers as the semantic basis. Also Naumann [Nau94] uses predicate transformers to define a semantics for an Oberon-like object-oriented language with specification constructs. Moreover, Abadi and Leino develop a Hoare-style logic for reasoning about object-oriented programs [AL97]. Alternative frameworks for reasoning about object oriented systems include TLA [Lam91] used for reasoning about DisCo specifications, and coalgebras, used for automatic reasoning on CCSL and JAVA classes [HHJT98].

The way we define inheritance or sub-typing as class refinement is based on data refinement for action systems [Bac90,BS94,SW97]. Class refinement in the data refinement framework has also been studied by Mikhajlova and Sekerinski [MS97]. They construct new classes by inheritance and overriding, but do not consider the addition of new methods. Moreover, their objects are not active and distributed as ours are. Class refinement between Z specifications for object-oriented programs has also been reported in the literature [LH92,Str95].

Overview We proceed as follows. In Section 2 we describe the action system formalism and how they can be composed in parallel. We also briefly describe the refinement calculus for action systems. In Section 3 we introduce the OO-action systems, a formalism for the specification and development of object-oriented systems. To illustrate the formalism we present two examples of OO-action systems, one describing a phone company and another specifying the prime numbers sieve. In Section 4 we give a computational interpretation of OO-action systems in terms of standard action systems. Based on this interpretation in Section 5 we give rules for the refinement of OO-action systems. We show that we can reuse refinement step in Section 6. In Section 7 we give some examples of refinement steps for one of the two OO-action systems introduced in Section 3. We end in Section 8 with some concluding remarks.

2 Actions and action systems

Let *Var* be a countable set of *variables* and assume that each variable in *Var* is associated with a nonempty set of *values*. A *state* is a function mapping variables to values in their respective associated set of values. We denote by *true* the predicate on *Var* which holds for every state, and by *false* the predicate on *Var* which holds for no state. Given a predicate $\mathcal{P}$, a list of variables x, and a list of values v we denote by $\mathcal{P}[x/v]$ the predicate which holds for those states s such that $\mathcal{P}$ holds for $s[x/v]$, where $s[x/v]$ is the state which is defined as s but for the variables x which are mapped to values v.

A *conjunctive predicate transformer* is a function π mapping predicates to predicates such that, for every nonempty index set I,

$$\pi(\bigwedge\{\mathcal{P}_i \mid i \in I\}) = \bigwedge\{\pi(\mathcal{P}_i) \mid i \in I\}\,.$$

Conjunctive predicate transformers form the semantic domains for statements interpreted by means of a w*eakest precondition* semantics [Dij76].

We consider the following language of *actions* defined by the grammar

$$A ::= abort \mid skip \mid x := v \mid b \rightarrow A \mid p \mid A \; ; \; A \mid [\!]_I A_i \, .$$

Here x is a list of variables, v a list of values (possibly resulting from the evaluation of a list of expressions), b a predicate, p a procedure name, and I an index set ranged over by i. Intuitively, '*abort*' is the action which always deadlocks, '*skip*' is a stuttering action, '$x := v$' is a multiple assignment, '$b \rightarrow A$' is a guarded action, 'p' is a procedure call, '$A_1 \; ; \; A_2$' is the sequential composition of two actions 'A_1' and 'A_2', and '$[\!]_I A_i$' is the nondeterministic choice among actions 'A_i' for $i \in I$. Given a nonempty set V of values and a variable x, we denote by '$x :\in V$' an abbreviation for the nondeterministic assignment '$[\!]_{v \in V} \, x := v$'.

A procedure declaration $p = P$ consists of a header p and an action P, the body of the procedure. Given a declaration for each procedure, we define the weakest precondition semantics of the above language in a standard way [Dij76,Bac90] defined as the least conjunctive predicate transformer such that, for any predicate $\mathcal{P}$,

$$\begin{array}{ll} wp(abort, \mathcal{P}) = false & wp(skip, \mathcal{P}) \quad = \mathcal{P} \\ wp(x := v, \mathcal{P}) = \mathcal{P}[x/v] & wp(b \rightarrow A, \mathcal{P}) \; = b \Rightarrow wp(A, \mathcal{P}) \\ wp(p, \mathcal{P}) \quad\;\; = wp(P, \mathcal{P}) & wp(A_1 \; ; \; A_2, \mathcal{P}) = wp(A_1, wp(A_2, \mathcal{P})) \\ wp([\!]_I A_i, \mathcal{P}) \; = \forall i \in I . wp(A_i, \mathcal{P}) \, , & \end{array}$$

where P is the action denoting the body of the procedure p. The details of the definition of the above function is studied elsewhere [BK94].

In the sequel we will need the following notions. An action A is *enabled* in a given state if its *guard*

$$gd(A) = \neg \, wp(A, false)$$

holds in that state. An action A *cannot enable* another action B whenever

$$wp(A, true) = true \;\text{ and }\; \neg \, gd(B) \Rightarrow wp(A, \neg \, gd(B)) \, .$$

Moreover, A *cannot disable* B whenever

$$wp(A, true) = true \;\text{ and }\; gd(B) \Rightarrow wp(A, gd(B)) \, .$$

2.1 Refinement of actions

An action A is *refined* by another action A', denoted $A \leq A'$, if, whenever A establishes a certain postcondition, so does A':

$$A \leq A' \text{ if and only if } \forall \mathcal{P} . wp(A, \mathcal{P}) \; \Rightarrow wp(A', \mathcal{P}) \, .$$

Together with the monotonicity of the weakest precondition predicate transformers this implies that A' is less nondeterministic than A or it may behave miraculously, i.e. it may establish the postcondition *false*. Note that the refinement relation is reflexive and transitive.

If we need to change the state space in a refinement, we can use the following technique due to Hoare [Hoa72]. Let A be an action that refers to the variables x and y, and A' be an action referring to the variable x' and y. We say that the action A is *data refined* by the action A' using an *abstraction relation* $R(x, x', y)$ between the abstract variables x, the concrete variables x' and the common variables y, if

$$\forall \mathcal{P}.R \wedge wp(A, \mathcal{P}) \Rightarrow wp(A', \exists x.R \wedge \mathcal{P})\,,$$

where $\mathcal{P}$ is a predicate on the variables x and y, and $\exists x.R \wedge \mathcal{P}$ is a predicate on the variables x' and y. Note that the common variables y are not changed. We denote by $A \leq_R A'$, the refinement of A by A' using the abstraction relation R. Data refinement as described here is also called *downward simulation*. It may strengthen the guards of an action, decrease nondeterminism and increase termination. Note carefully that refinement and data refinement are related notions in the sense that the above data refinement relation can be defined in term of the ordinary refinement relation using special commands to encode abstraction relations [Wri94].

2.2 Action systems

An *action system* is a set of actions operating on local and global variables. First, the variables are created and initialized. Then, repeatedly, enabled actions are chosen and executed. Actions operating on disjoint sets of variables can be executed in parallel. The computation terminates if no action is enabled, otherwise it continues infinitely. Syntactically, an action system $\mathcal{A}$ is a statement of the form

$$\begin{array}{lll} \mathcal{A} = |[& \textbf{var} & y^* := y0 \;;\; x := x0 \\ & \textbf{proc} & p_1^* = P_1 \;;\; \cdots \;;\; p_n^* = P_n \;; \\ & & q_1 = Q_1 \;;\; \cdots \;;\; q_m = Q_m \\ & \textbf{do} & [\!]_I\, A_i \;\textbf{od} \\ ]| & & \end{array}$$

An action system provides a declaration section for variables and one for procedures. Here y is a list of *global variables*, marked with an asterisk $*$, that can be used locally by $\mathcal{A}$ and also by other action systems when put in parallel with $\mathcal{A}$. The variables in the list x are *local* to $\mathcal{A}$. The global variables y get initial values $y0$ and the local variables x get initial values $x0$. Local variables that are declared but not initialized get an arbitrary initial value in their respective sets of associated values. We assume that x and y are disjoint.

Procedures marked with an asterisk $*$ in a procedure declaration $p_i = P_i$ are *exportable*: they can be called by the actions of $\mathcal{A}$ or by actions of some other action systems when put in parallel with $\mathcal{A}$. A procedure declared in $q_i = Q_i$ is *local* and can be called only by actions of $\mathcal{A}$. A procedure p with body P is said to be *locally enabled* if only actions of $\mathcal{A}$ (possibly containing a procedure call p) can enable or disable the action P.

Actions of the action system $\mathcal{A}$ are the actions A_i for all $i \in I$ and all bodies of the procedures (both exportable and local) declared in $\mathcal{A}$. They can be any statement of our language of actions defined above, can refer to variables which are not declared in $\mathcal{A}$ but globally declared in an other action systems to be put in parallel with $\mathcal{A}$, and can contain procedure calls to procedure declared exportable in an other action systems to be put in parallel with $\mathcal{A}$. Since the nondeterministic choice $[\!]_I A_i$ is included as an operator on actions, we can confine ourselves to action systems with a body consisting of only a single action. Actions are atomic, meaning that if an action A_k, for $k \in I$, is chosen for execution then it is executed to completion without any interference from the other actions A_i of the system. This ensures that a parallel execution of an action system gives the same results of a sequential non-deterministic execution.

Next we define the parallel composition of action systems. Let I be a non-empty index set, and consider the action systems $\mathcal{A}_i$, for $i \in I$,

$$\begin{array}{lll} \mathcal{A}_i = |[& \textbf{var} & y_i^* := y0_i \ ; x_i := x0_i \\ & \textbf{proc} & p_{i,1}^* = P_{i,1} \ ; \cdots ; p_{i,n_i}^* = P_{i,n_i} ; \\ & & q_{i,1} = Q_{i,1} \ ; \cdots ; q_{i,m_i} = Q_{i,m_i} \\ & \textbf{do} \ A_i \ \textbf{od} & \\ ]| & & \end{array}$$

where the global variables, the local variables, the headers of the exportable procedures, and the headers of the local procedures declared in each action system $\mathcal{A}_i$ are required to be distinct. We define the the *parallel composition* $\|_I \mathcal{A}_i$ of the action systems $\mathcal{A}_i$ to be the action system

$$\begin{array}{lll} \|_I \mathcal{A}_i = |[& \textbf{var} & y^* := y0 \ ; x := x0 \\ & \textbf{proc} & p^* = P \ ; q = Q \\ & \textbf{do} \ [\!]_I \ A_i \ \textbf{od} & \\ ]| & & \end{array}$$

where y is the list obtained by concatenating all lists of global variables y_i, x is the list obtained by concatenating all lists of local variables x_i, $p = P$ is list containing the declarations of all the exportable procedures of every $\mathcal{A}_i$, and finally $q = Q$ is the list containing the declarations of all the local procedures of every $\mathcal{A}_i$. The initial values of the variables and the actions in $\|_I \mathcal{A}_i$ consist of the initial values and the actions of the original systems.

Since actions of an action system $\mathcal{A}$ may contain calls to procedure not declared in $\mathcal{A}$, in order to formally define the behaviour of $\mathcal{A}$ we need the following notion. An action system $\mathcal{E}$ is called a *full context* of an action system $\mathcal{A}$ if the actions of $\mathcal{E}$ contains only calls to procedures either declared in $\mathcal{E}$ or declared exportable in $\mathcal{A}$, and the actions of $\mathcal{A}$ contains only calls to procedure either declared in $\mathcal{A}$ or declared exportable in $\mathcal{E}$.

Action system in parallel with a full context can be considered as a command of the refinement calculus [Bac90]. This view allows for reasoning about the total correctness of the system. Indeed, data refinement for action systems, as described below, implies, for any full context, data refinement of the corresponding commands in the refinement calculus [SW97].

The behaviour of an action system in parallel with a full context considered as a reactive system is described by means of set of computations, that is, set of finite or infinite sequences of states restricted only to the global variables, without finite repetition of the same state (finite stuttering), and possibly terminating with a special symbol to denote abortion. A refinement relation for action systems as reactive systems can be defined as follows: an action system $\mathcal{A}$ is *trace refined* by an action system $\mathcal{A}'$ if for any full context $\mathcal{E}$ of $\mathcal{A}$ and $\mathcal{A}'$, whenever $\mathcal{A} \parallel \mathcal{E}$ cannot produce a computation which aborts then each computation of $\mathcal{A}' \parallel \mathcal{E}$ is also a computation of $\mathcal{A} \parallel \mathcal{E}$ [Bac90]. Data refinement for action systems, as described below, is justified via trace refinement.

2.3 Data refinement of action systems

Next we consider data refinement in the context of action systems. Let $\mathcal{A}$ be the action system

$$\begin{array}{lll} \mathcal{A} = \,|[& \mathbf{var} & y^* := y0 \,;\, x := x0 \\ & \mathbf{proc} & p_1^* = P_1 \,;\, \cdots \,;\, p_n^* = P_n \,; \\ & & q_1 = Q_1 \,;\, \cdots \,;\, q_m = Q_m \\ & \multicolumn{2}{l}{\mathbf{do}\ A\ \mathbf{od}} \\ ]| & & \end{array}$$

with every global procedure locally enabled, and let

$$\begin{array}{lll} \mathcal{A}' = \,|[& \mathbf{var} & y^* := y0 \,;\, x' := x0' \\ & \mathbf{proc} & p_1^* = P_1' \,;\, \cdots \,;\, p_n^* = P_n' \,; \\ & & r_1^* = R_1 \,;\, \cdots \,;\, r_h^* = R_h \,; \\ & & s_1 = S_1 \,;\, \cdots \,;\, s_k = S_k \\ & \multicolumn{2}{l}{\mathbf{do}\ A' \,[\!]\, H\ \mathbf{od}} \\ ]| & & \end{array}$$

We say that $\mathcal{A}$ is *data refined* by the action $\mathcal{A}'$ using an abstraction relation $R(x, x', y)$, denoted by $\mathcal{A} \preceq_R \mathcal{A}'$, if we have that

(1) Initialization: $R(x0, x0', y0)$,
(2) Procedures: $P_i \leq_R P_i'$,
(3) Enabledness: $R \wedge gd(P_i) \Rightarrow gd(P_i') \vee gd(A') \vee gd(H)$,
(4) Main actions: $A \leq_R A'$
(5) Continuation: $R \wedge gd(A) \Rightarrow gd(A') \vee gd(H)$,
(6) Auxiliary actions: $skip \leq_R H$,
(7) Internal convergence: $R \Rightarrow wp(\ \mathbf{do}\ H\ \ \mathbf{od}\ , true)$,
(8) Non-interference: $R \wedge wp(B, true) \Rightarrow wp(B, R)$ for every action B of another action system $\mathcal{B}$ which can occur in a parallel composition with A.

In case of procedures with parameters we require that the formal parameters of the procedures are not changed in the refinement step [SW97].

The idea is that local variables x in $\mathcal{A}$ are replaced by x' in $\mathcal{A}'$. Due to this replacement, the action A is changed into the action A', and the procedure bodies P_i of each exportable procedure p_i is changed into P_i'. Some new computations on the local variables x' may be added to the system in terms of the action H. Local procedures may be changed and new exportable procedures r_i may be added to the system. Clearly they may use global variables. However, by (4) the procedures called by the action A' use global variables essentially in the same way as some statements of the action A, whereas by (6) the procedures called by the action H do not change the content of any global variables.

Informally, in order to prove a data refinement between two action systems we require that

(1) the abstraction relation is established by the initialization of the variables,
(2) the body P_i of each exportable procedure p_i is data refined as action by the respective body P_i' using R,
(3) if R holds then enabledness of the body of a global procedure in $\mathcal{A}$ is either preserved or an action in $\mathcal{A}'$ is enabled,
(4) the action A is data refined by A' using R,
(5) if R holds then enabledness of the action A is preserved by $A' \,[\!]\, H$,
(6) some new computations may be added by the action H but they do not involve global variables,
(7) if R holds then the new computations added by the action H terminates,
(8) any context preserves the relation R.

Relying on the definition of data refinement of action systems we have that for two action systems $\mathcal{A}$ and $\mathcal{A}'$ and abstraction relation R, if $\mathcal{A} \preceq_R \mathcal{A}'$ then $\mathcal{A}$ is trace refined by $\mathcal{A}'$. The correctness of this statement can be proved in a similar way as Theorem 2 of Sere and Walden [SW97], who also consider procedures with parameters.

3 OO-action systems

In this section we introduce our action based formalism for object-oriented systems, the so-called OO-action systems. An OO-action system consists of a finite set of classes, each class specifying the behaviour of objects that are dynamically created and executed in parallel. A class contains a description (attributes) of the local and shared variables of the object instances of the class, some pointers (object variables) to object instances of possibly other classes, the operations (methods) that can be applied to object instances of the class, the internal operations (procedure) of object instances of the class, and a description of the actions to be executed when an object instance of the class is created.

Before introducing the notion of class, we need to define some basic building blocks. Let *CName* be a fixed set of class names. We define the set *OName* of valid names for objects, ranged over by o, by the following grammar:

$$o ::= c(root) \;\mid\; c(o, i)\,,$$

where i runs over the natural numbers and c over the set *CName* of class names. Hence a name of an object is either $c(root)$, or a class name together with an object name and a number. The intended meaning is that the object $c(o, i)$ is an instance of a class with name c and it is the $(i+1)$-th object created by the object named o. The name $c(root)$ is the name of an object, instance of a class with name c, that starts the execution.

We will also consider a fixed set *Attr* of attributes and assume that each attribute is associated with a nonempty set of *values*; and a fixed set of object variables *OVar* assumed to be disjoint from *Attr*. The only valid values for object variables are the constant *self* and the names for objects in *OName*.

The actions of a class are defined by the following grammar:

$$\begin{aligned} A ::= {} & abort \mid skip \mid x := v \mid n := o \mid new(c) \mid n := new(c) \mid kill(n) \mid kill(self) \mid \\ & b \rightarrow A \mid p \mid n.m \mid self.m \mid A \,;\, A \mid [\!]_I A_i \,. \end{aligned}$$

Here x is a list of attributes, n is an object variable, v a list of values, o is an object name or the constant *self* (both possibly resulting from the evaluation of an expression), c is a class name, b a predicate (possibly on attributes and object variables and referring to the constant *self*), p is a procedure name and m is a method name. Intuitively, '$n{:=}o$' stores the object name o into the object variable n, '$n.m$' is a call of the method m of the object the name of which is stored in the object variable n, '$self.m$' is a call of the method m of declared in the same object. Note that method calls are always prefixed by an object variables or by the constant *self*. There are two object destructors '$kill(n)$' and '$kill(self)$', and two object constructors '$new(c)$' and '$n := new(c)$'. The latter assigns the name of the newly created instance of the class c to the object variable n.

A *class* is a pair $\langle c, \mathcal{C} \rangle$, where $c \in CName$ is the *name* of the class and $\mathcal{C}$ is its *body*, that is, a statement of the form

$$\begin{array}{rll} \mathcal{C} = [\![& \textbf{attr} & y^* := y0 \,;\, x := x0 \\ & \textbf{obj} & n \\ & \textbf{meth} & m_1 = M_1 \,;\, \cdots \,;\, m_h = M_h \\ & \textbf{proc} & p_1 = P_1 \,;\, \cdots \,;\, p_k = P_k \\ & \multicolumn{2}{l}{\textbf{do } A \textbf{ od}} \\ ]\!] & & \end{array}$$

A class body consists of an action of the above grammar and of four declaration sections. In the attribute declaration the *shared attributes* in the list y, marked with an asterisk $*$, describe the variables to be shared among all active objects. Therefore they can be used by instances of the class $\mathcal{C}$ and also by objects instances of other classes. Initially they get values $y0$. The *local attributes* in the list x describe variables that are local to an object instance of the class, meaning that they can only be used by the instance itself. The variables are initialized to the values $x0$. We assume that x and y are disjoint.

The list n of *object variables* describes a special kind of variables local to an object instance of the class. They contain names of objects and are used for calling methods of other objects. We assume that the lists x, y and n are pairwise disjoint.

A *method* $m_i = M_i$ describes a procedure of an object instance of the class. They can be called by actions of the object itself or by actions of another object instance of possibly another class. A method consists of an header 'm' and a body 'M'. The latter is an action of the above grammar.

A *procedure* $p_i = P_i$ describes a procedure that is local to the object instances of the class. It can be called only by actions of the object itself. Like a method, it consists of an header 'p' and an action forming the body 'P'.

An *action* of a class is a description of the actions to be executed when the object instance of the class is activated. It can refer to attributes which are declared shared in another class, and to the object variables and the local attributes declared within the class itself. It can contain procedure calls only to procedure declared in the class and method calls of the form $n.m$ to methods declared in another class. Methods calls $\mathit{self}.m$ are allow only if m is a method declared in the class. The constant *self* is used to indicate the name of the instance of the class. As for action systems actions are atomic.

An *OO-action system* OO consists of a finite set of classes

$$OO = \{\langle c_1, \mathcal{C}_1 \rangle, ..., \langle c_n, \mathcal{C}_n \rangle\}$$

such that the shared attributes declared in each $\mathcal{C}_i$ are distinct and actions in each $\mathcal{C}_i$ or bodies of methods and procedures declared in each $\mathcal{C}_i$ do not contain *new* statements referring to class names not used by classes in OO. Local attributes, object variables, methods, and procedures are not required to be distinct. There are some classes in OO, marked with an asterisk $*$, that serve as roots. Execution starts by the creation of one object instance of each of these classes. Each object, when created, chooses enabled actions and execute them. Actions operating on disjoint sets of local and shared attributes, and object variables can be executed in parallel. They can also create or kill other objects. Actions of different active objects can be executed in parallel if they are operating on disjoint sets of shared attributes. Objects can interact by means of the shared attributes and by executing methods of other objects.

3.1 An example: the prime number sieve

Our first example of OO-action systems is the (rather standard) prime number sieve which typically illustrates the dynamical binding of operations to program code. An object of the class $\langle \mathit{Driver}, \mathit{Dr} \rangle$ generates an infinite increasing sequence of natural numbers, which it feeds into a chain of object instances of the class $\langle \mathit{Prime}, \mathit{Pr} \rangle$. Each of those object stores in the variable $\mathit{prime}[n]$ the first number it receives (always a prime number greater than 1) and from the rest of the numbers it passes only those numbers that are not divisible by $\mathit{prime}[n]$. The OO-action system is defined as follows:

$$\{\langle \mathit{Driver}, \mathit{Dr} \rangle^*, \langle \mathit{Prime}, \mathit{Pr} \rangle\}\,.$$

The computation starts with an object instance of the class *Driver*.

Below we describe the body of the class *Driver*. It introduces two shared attributes $prime[1]$ and *counter*. The first one stores the first prime number, and will never be changed by any object. It is introduced only for sake of completeness. The other shared attribute is used to count the number of primes that are already found plus one. Since the system starts with the first prime number stored in the attribute $prime[1]$, *counter* is initially set to 2. The local attribute *number* is used to generate an infinite increasing sequence of natural numbers.

$$
\begin{array}{lll}
Dr = & |[& \textbf{attr} \;\; prime[1]^* := 1 \; ; \; counter^* := 2 \\
& & \qquad\;\; number := 1 \; ; \; created := false \\
& & \textbf{obj} \;\; first \\
& & \textbf{do} \\
& & \qquad \neg created \rightarrow first := new(Prime) \; ; \; created := true \\
& & \quad [\!] \; created \rightarrow number := number + 1 \; ; \; first.Sieve(number) \\
& & \textbf{od} \\
&]| &
\end{array}
$$

Next we define the body of the class *Prime*. It declares an infinite number of shared attributes $prime[n]$ where the n-th prime number will be stored. The class has only one method which is repeatedly executed by the object which creates an instance of *Prime*. The computation proceeds in a pipelined way: when a new prime number is found then a new object is created and pipelined.

$$
\begin{array}{llll}
Pr = & |[& \textbf{attr} & prime[2]^* := 0 \; ; \cdots ; \; prime[n]^* := 0 \; ; \cdots \\
& & & p := 0 \; ; \; created := false \\
& & \textbf{obj} & next \\
& & \textbf{meth} & Sieve(z) = (\neg created \rightarrow prime[counter] := z \; ; \\
& & & \qquad\qquad\qquad\qquad p := counter \; ; \; counter := counter + 1 \; ; \\
& & & \qquad\qquad\qquad\qquad next := new(Prime) \; ; \; created := true \\
& & & \qquad\qquad [\!] \, created \wedge z \;\textbf{mod}\; prime[p] = 0 \rightarrow skip \\
& & & \qquad\qquad [\!] \, created \wedge z \;\textbf{mod}\; prime[p] \neq 0 \rightarrow next.Sieve(z)) \\
&]| & &
\end{array}
$$

Here the local attribute p is used as a pointer to the prime array remembering where the objects has stored the prime number associated with its creation.

3.2 Another example: a phone company

As a second example of an OO-action system we describe a phone company with many phones that can call each other. The system consists of two classes, one named *PhoneCompany* and another one named *Phone*:

$$\{\langle PhoneCompany, PC\rangle^*, \langle Phone, Ph\rangle\}$$

The execution starts with the creation of one phone company (the class is marked with an asterisk) and then an arbitrary number of phones may start their activity.

The body PC of the class *PhoneCompany* is described below. It models a company that can create new phones (when the variable *allow_new_phones* is true). When a new phone is created, it is added to the phonebook (a set of integers represented by the shared attribute *phonebook*) that contains phone numbers. The link between a phone number i and the name of the phone is kept in the object variable $names[i]$. After a phone is created, the name of the phone company where it has been registered in is told to the phone. This is done by the phone company by calling the method *where* of the created phone. Each phone company offers a service *give_name* to the phones: given a phone number, it will give the name of the phone (in a result parameter).

$$
\begin{array}{lll}
PC = \;[\![& \textbf{attr} & \mathit{phones}^* := 0 \;;\; \mathit{phonebook}^* := \emptyset; \\
& & \mathit{allow\_new\_phones} := \mathit{true} \\
& \textbf{obj} & \mathit{entry} \;;\; \mathit{names}[1] \;;\; \cdots \;;\; \mathit{names}[n] \;;\; \cdots \\
& \textbf{meth} & \mathit{Give\_name}(x, n) = n := \mathit{names}[x] \\
& \textbf{proc} & \mathit{Update}(n) = (\,\mathit{phones} := \mathit{phones} + 1; \\
& & \qquad\qquad\qquad \mathit{phonebook} := \mathit{phonebook} \cup \{\mathit{phones}\}; \\
& & \qquad\qquad\qquad \mathit{names}[\mathit{phones}] := n\ ) \\
& \textbf{do} & \\
& & \mathit{allow\_new\_phones} \rightarrow \mathit{entry} := \mathit{new}(\mathit{Phone}); \\
& & \qquad\qquad\qquad\qquad \mathit{Update}(\mathit{entry}); \\
& & \qquad\qquad\qquad\qquad \mathit{entry}.\mathit{Where}(\mathit{self}) \\
& & [\!]\ \mathit{allow\_new\_phones} :\in \{\mathit{true}, \mathit{false}\} \\
& \textbf{od} & \\
]\!] & &
\end{array}
$$

Here the variable *phones* stores the number of phones registered at the company, and *entry* is the name of the new phone to be register in the phonebook.

Next we describe the body *Ph* of the class *Phone*. Each phone has a variable *number* which stores for the caller the number of the phone to which it is currently connected, or, if the phone is not connected, the number of the last phone it called.

$$
\begin{array}{lll}
Ph = \;[\![& \textbf{attr} & \mathit{number} := -1; \\
& & \mathit{idle} := \mathit{true} \mathit{registered} := \mathit{false} \\
& \textbf{obj} & \mathit{company} \;;\; \mathit{callee} \\
& \textbf{meth} & \mathit{Accept\_call\_from}(n) = (\,\mathit{idle} \wedge n \neq \mathit{self} \rightarrow \mathit{idle} := \mathit{false}; \\
& & \qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{callee} := n\ ) \\
& & \mathit{Where}(y) = (\mathit{company} := y \;;\; \mathit{registered} := \mathit{true}) \\
& & \mathit{Return} = \mathit{idle} := \mathit{true} \\
& \textbf{proc} & \mathit{Call} = (\,\mathit{number} :\in \mathit{phonebook}; \\
& & \qquad\quad \mathit{company}.\mathit{Give\_name}(\mathit{number}, \mathit{callee}); \\
& & \qquad\quad \mathit{callee}.\mathit{Accept\_call\_from}(\mathit{self}) \;;\; \mathit{idle} := \mathit{false}) \\
& \textbf{do} & \\
& & \mathit{idle} \wedge \mathit{registered} \rightarrow \mathit{Call} \\
& & [\!]\ \neg \mathit{idle} \rightarrow \mathit{callee}.\mathit{Return} \;;\; \mathit{idle} := \mathit{true} \\
& \textbf{od} & \\
]\!] & &
\end{array}
$$

Phones have two actions, one for calling other phones and another one for ending a call. Only a registered phone in an idle state may call another phone. In this case a number is selected from the phonebook and the name of the corresponding object is obtained by the company via the method *Give_name* and stored in the object variable *callee*. If the phone *callee* is idle then both phones are connected and enter in a not-idle state. One of the two phones can now break the connection and both phones return in their idle state.

3.3 Composing OO-action systems

Let OO_1 and OO_2 be two OO-action systems where the shared attributes declared in each class in OO_1 are distinct from the shared attributes declared in each class in OO_2. Furthermore, assume that the class names used by the classes in OO_1 are distinct from the class names used by the classes in OO_2. We define the *parallel composition* $OO_1 \parallel OO_2$ of OO_1 and OO_2 to be the OO-action system $OO_1 \cup OO_2$, where a class in $OO_1 \parallel OO_2$ is marked by an asterisk $*$ if it is marked by an asterisk $*$ in either OO_1 or OO_2.

4 The computational model

Computationally, an OO-action system can be represented by a parallel composition of infinitely many action systems, each of them used either for initializing the global variables of a class, or for representing instances of classes. The action and procedure bodies of each of these action systems are guarded by a local boolean variable *start*. It is set to true only when that instance becomes an active object and it is set to false when the object is deactivated. The constructors $new(c)$ and $n := new(c)$ and the destructors $kill(n)$ and $kill(self)$ are replaced by remote procedure calls that act on the variable *start*.

We define the semantic model of an OO-action system in three steps. In the first step we translate every action A of a class $\langle c, \mathcal{C}\rangle$ into an action A_o of an action system. This translation is parametric with respect to the name of the object o where the action has to be executed. In the second step we associate to each class $\langle c, \mathcal{C}\rangle$ an action system $\mathcal{A}(c)$ obtained as the parallel composition of an action system $\mathcal{A}(c(root))$ with all the action systems $\mathcal{A}(c(o,i))$ for $o \in OName$ and $i \in \mathbb{N}$. The first action system initializes the global variables of the class and, if the class is marked by an asterisk, starts executing an action of the first object instance of the class. The action systems $\mathcal{A}(c(o,i))$ for $o \in OName$ and $i \in \mathbb{N}$ represent all possible object instances of the class $\langle c, \mathcal{C}\rangle$. Finally, in the third step we translate an OO-action system OO into an single action system $\mathcal{A}(OO)$ obtained by putting in parallel the action systems $\mathcal{A}(c)$ for each class $\langle c, \mathcal{C}\rangle \in OO$.

4.1 Translating class-actions

For fixed and disjoint sets *Attr* and *OVar* of attributes and object variables let us consider a set *Var* of variables containing x, $o.x$, and $o.n$ for each $x \in Attr$,

$n \in OVar$ and object name $o \in OName$. Furthermore, let us assume that $o.start$ and $o.count$ are in Var but not in $Attr$ and $OVar$ for each object name $o \in OName$. The set of values associated with the variables x and $o.x$ are the same as for the set of values associated with the attribute x, while $OName$ is the set of value associated with every variable of the form $o.n$.

We start with the translation of an action of a class. Let o be an object name in $OName$, and A be an action of some class with local attributes x, object variables n and local procedures p. Define the action A_o as the result of consecutively applying the following steps:

1. replace the constant *self* by the name o,
2. replace every local attribute x by $o.x$,
3. replace every call to a local procedure p by $o.p$,
4. replace every method call $n.p$, for some object variable n, by the statement
$$[\!]_{u \in OName}(n = u) \rightarrow u.p \, ,$$
5. replace every action $new(c)$ by the statement
$$([\!]_{i \in \mathbb{N}}(o.count = i) \rightarrow c(o,i).init) \; ; \; o.count := o.count + 1 \, ,$$
6. replace every $n := new(c)$ by the statement
$$([\!]_{i \in \mathbb{N}}(o.count = i) \rightarrow n := c(o,i) \; ; \; c(o,i).init) \; ; \; o.count := o.count + 1 \, ,$$
7. replace every $kill(n)$ by the statement
$$[\!]_{u \in OName}(n = u) \rightarrow u.stop \, ,$$
8. replace every $kill(o)$ by the procedure call $o.stop$
9. replace every object variable n by $o.n$.

The idea is that the name o is the name of an object instance of a class where the action A_o has to be executed. Since this action must be an action in the format required by the syntax of action systems, we substitute the constant *self* by the name of the object. Local attributes and object variables become ordinary variables. Locality of these variables is obtained by prefixing them with the name of the object o. The same is applied also to local procedures. Every method call becomes a call to a remote procedure. Since the name of the object where the procedure is declared is known only at run-time, we need to quantify the procedure call over all possible object names. Note that, in the first step, a method call prefixed by the constant *self* is replaced by a procedure call declared exportable in the object itself. The actions $new(c)$ and $n := new(c)$ are both replaced by a call to the remote procedure $c(o,i).init$ followed by an increment of the local variable $o.count$. The procedure $c(o,i).init$ is declared in the i-th object instance of the class c created by the object named o, where i is a number stored in $o.count$. Since this number is known only at run-time, we quantify over all possible values of $o.count$, that is, over all natural numbers.

The procedure $c(o,i).\mathit{init}$ has the task of setting the boolean flag $c(o,i).\mathit{start}$ to true, so that the computation of the object named $c(o,i)$ may start. The action $\mathit{kill}(n)$ is replaced by a call to the remote procedure $u.\mathit{stop}$ declared in the object named u, where u is the object name stored in n. This name is known only at run-time. Similarly, the action $\mathit{kill}(\mathit{self})$ is replaced by a call to the procedure $o.\mathit{stop}$. The effect of the procedure $u.\mathit{stop}$ is to set the flag $u.\mathit{start}$ of the object named u to false interrupting its computation.

4.2 Translating classes

We now translate a single class $\langle c, \mathcal{C} \rangle$ into an action system $\mathcal{A}(c)$. Let $\mathcal{C}$ be the class body

$$
\begin{array}{ll}
|[& \mathbf{attr} \;\; y^* := y0 \; ; \; x := x0 \\
 & \mathbf{obj} \;\; n \\
 & \mathbf{meth} \;\; m_1 = M_1 \; ; \cdots ; \; m_h = M_h \\
 & \mathbf{proc} \;\; p_1 = P_1 \; ; \cdots ; \; p_k = P_k \\
 & \mathbf{do} \; A \; \mathbf{od} \\
]| &
\end{array}
$$

We begin with the definition of the action system $\mathcal{A}(c(\mathit{root}))$ containing the declaration and initialization of all global variables corresponding to the shared attributes of the class c and the actions to be initially executed in case the class $\langle c, \mathcal{C} \rangle$ is marked with an asterisk $*$. If the class is not marked by an asterisk then

$$\mathcal{A}(c(\mathit{root})) = \; |[\; \mathbf{var} \; y^* := y0 \;]| \; ,$$

otherwise

$$
\begin{array}{lll}
\mathcal{A}(c(\mathit{root})) = \; |[& \mathbf{var} & y^* := y0; \\
 & & c(\mathit{root}).\mathit{start} := \mathit{true} \; ; \; c(\mathit{root}).\mathit{count} := 0; \\
 & & c(\mathit{root}).x := x0 \; ; \; c(\mathit{root}).n \\
 & \mathbf{proc} & c(\mathit{root}).\mathit{stop}^* = (c(\mathit{root}).\mathit{start} := \mathit{false}); \\
 & & c(\mathit{root}).m_1^* = (c(\mathit{root}).\mathit{start} \rightarrow M_{1_{c(\mathit{root})}}); \\
 & & \quad \vdots \\
 & & c(\mathit{root}).m_h^* = (c(\mathit{root}).\mathit{start} \rightarrow M_{h_{c(\mathit{root})}}); \\
 & & c(\mathit{root}).p_1 = P_{1_{c(\mathit{root})}} \; ; \cdots ; \; c(\mathit{root}).p_k = P_{k_{c(\mathit{root})}} \\
 & \mathbf{do} & c(\mathit{root}).\mathit{start} \rightarrow A_{c(\mathit{root})} \; \mathbf{od} \\
]| & &
\end{array}
$$

In case the class $\langle c, \mathcal{C} \rangle$ is not marked by an asterisk, the action system $\mathcal{A}(c(\mathit{root}))$ is used only to transform the shared attributes of the class into global variables. Therefore $\mathcal{A}(c(\mathit{root}))$ has no declaration section for procedures and no action in its body. If the class $\langle c, \mathcal{C} \rangle$ is marked by an asterisk $*$, the shared attributes of the class are transformed into global variables of the action system $\mathcal{A}(c(\mathit{root}))$, and every action A, method body M and procedure body P of the class are translated into actions $A_{c(\mathit{root})}$, $M_{c(\mathit{root})}$ and $P_{c(\mathit{root})}$, respectively. Actions and methods are also guarded by the local variable $c(\mathit{root}).\mathit{start}$ which is initially set to true; it can be set to false only when the object is deactivated by means of a call of the exportable procedure $c(\mathit{root}).\mathit{stop}$. Local attributes and object variables of the class are treated as local variables. There is an extra local variable $c(\mathit{root}).\mathit{count}$ which is used to store the number of objects created by $\mathcal{A}(c(\mathit{root}))$.

Next we define, for each $o \in \mathit{OName}$ and $i \in \mathbb{N}$, the action system $\mathcal{A}(c(o,i))$ that describes the behaviour of the i-th object instance of the class $\langle c, \mathcal{C}\rangle$ created by the object named o:

$$
\begin{array}{lll}
\mathcal{A}(c(o,i)) = & [\![\;\; \mathbf{var} & c(o,i).\mathit{start} := \mathit{false} \; ; \; c(o,i).\mathit{count} := 0; \\
& & c(o,i).x := x0 \; ; \; c(o,i).n \\
& \;\;\;\;\; \mathbf{proc} & c(o,i).\mathit{init}^* = (c(o,i).\mathit{start} := \mathit{true}); \\
& & c(o,i).\mathit{stop}^* = (c(o,i).\mathit{start} := \mathit{false}); \\
& & c(o,i).m_1^* = (c(o,i).\mathit{start} \to M_{1_{c(o,i)}}); \\
& & \quad \vdots \\
& & c(o,i).m_h^* = (c(o,i).\mathit{start} \to M_{h_{c(o,i)}}); \\
& & c(o,i).p_1 = P_{1_{c(o,i)}} \; ; \cdots ; \; c(o,i).p_k = P_{k_{c(o,i)}} \\
& \;\;\;\;\; \mathbf{do} \; c(o,i).\mathit{start} \to A_{c(o,i)} \; \mathbf{od} & \\
&]\!] &
\end{array}
$$

The action system $\mathcal{A}(c(o,i))$ is defined in a similar way as for the action system $\mathcal{A}(c(\mathit{root}))$ when the class c is marked by an asterisk. There are no global variables corresponding to the shared attributes, because they are declared only in the action system $\mathcal{A}(c(\mathit{root}))$. Moreover, the local variable $c(o,i).\mathit{start}$ is initially set to false: it will be set to true only when the object is created and set to false when the object is deactivated. The value of the variable $c(o,i).\mathit{start}$ can only be changed by a remote call to the exportable procedures $c(o,i).\mathit{init}$ and $c(o,i).\mathit{stop}$.

The action system $\mathcal{A}(c)$ describing the behaviour of the class $\langle c, \mathcal{C}\rangle$ is defined by

$$\mathcal{A}(c) = \mathcal{A}(c(\mathit{root})) \parallel (\parallel_{\mathit{OName} \times \mathbb{N}} \mathcal{A}(c(o,i)))\,.$$

Clashes of variables and procedure names are avoided in the above definition by declaring global variables only in the action system $\mathcal{A}(c(\mathit{root}))$ and by prefixing each local variable and procedure name by the name of the object instance of the class.

4.3 Translating OO-action systems

In the last step we define the action system $\mathcal{A}(OO)$ for an OO-action system $OO = \{\langle c_1, \mathcal{C}_1\rangle, ..., \langle c_n, \mathcal{C}_n\rangle\}$:

$$\mathcal{A}(OO) = (\mathcal{A}(c_1) \parallel \cdots \parallel \mathcal{A}(c_n))\,.$$

The action system $\mathcal{A}(OO)$ is a parallel composition of all possible instance of the classes composing the OO-action system OO. The execution starts with an action from an action system $A(c(root))$ for some class $\langle c, \mathcal{C} \rangle \in OO$ marked by an asterisk. Initially, only the actions generated from the marked classes are potentially enabled.

The following theorem is an immediate consequence of the translation above and the definition of parallel composition between OO-action systems.

Theorem 1. *Let $OO_1 \parallel OO_2$ be the OO-action system obtained as the parallel composition of two OO-action systems OO_1 and OO_2. Then $\mathcal{A}(OO_1 \parallel OO_2) = \mathcal{A}(OO_1) \parallel \mathcal{A}(OO_2)$.*

5 Refining OO-action systems

A refinement relation between OO-action systems can be obtained by translating them into ordinary action systems. For two OO-action systems OO and OO' we say that OO is refined by OO', denoted by

$$OO \sqsubseteq OO'$$

if there exists an abstraction relation R such that

$$\mathcal{A}(OO) \preceq_R \mathcal{A}(OO') \,.$$

The idea is that in an OO-action system only the values of the shared attributes are visible. Since shared attributes are translated into global variables, refinement between two OO-action systems preserves the behavior of the original system with respect to these variables.

In this section we give a number of rules that can be applied directly between OO-action system without translating them into action systems. The justification for all the refinement rules below is based on this translation.

Class refinement We start by a simple rule which allows to substitute a class in an OO-action system by a refinement of that class. We say that a class $\langle c, \mathcal{C} \rangle$ with

$$\begin{array}{lll} \mathcal{C} & = |[& \textbf{attr} \;\; y^* := y0 \; ; \; x := x0 \\ & & \textbf{obj} \;\; n \\ & & \textbf{meth} \;\; m_1 = M_1 \; ; \cdots ; \; m_h = M_h \\ & & \textbf{proc} \;\; p_1 = P_1 \; ; \cdots ; \; p_k = P_k \\ & & \textbf{do} \; A \; \textbf{od} \\ &]| & \end{array}$$

is *class-refined* by a class $\langle c', \mathcal{C}' \rangle$ with

$$\begin{array}{lll} \mathcal{C}' & = |[& \textbf{attr} \;\; y^* := y0 \; ; \; x' := x0' \\ & & \textbf{obj} \;\; n \\ & & \textbf{meth} \;\; m_1 = M_1' \; ; \cdots ; \; m_h = M_h' \\ & & \textbf{proc} \;\; q_1 = Q_1 \; ; \cdots ; \; q_j = Q_j \\ & & \textbf{do} \; A' \; \textbf{od} \\ &]| & \end{array}$$

if $c = c'$ and there exists a relation $R(x, x', y)$ between the attributes x and x' and the shared attributes y such that,

1. $\mathcal{A}(c(root)) \preceq_{R_{c(root)}} \mathcal{A}(c'(root))$, and
2. for each $o \in OName$ and $i \in \mathbb{N}$, $\mathcal{A}(c(o,i)) \preceq_{R_{c(o,i)}} \mathcal{A}(c'(o,i))$.

Here, for $o \in OName$, R_o is the relation obtained by translating each attribute x and x' in $o.x$ and $o.x'$, respectively. If a class $\langle c, \mathcal{C}\rangle$ is refined by another class $\langle c, \mathcal{C}'\rangle$ then we call the class $\langle c, \mathcal{C}'\rangle$ a *subclass* of $\langle c, \mathcal{C}\rangle$ and $\langle c, \mathcal{C}\rangle$ a *superclass* of $\langle c, \mathcal{C}'\rangle$.

Because the relation R does not refer to the object variables n and the variables $c(o,i).start$ and $c(o,i).count$, and because all relations R_o are distinct, class-refinement implies that $\mathcal{A}(c) \preceq_{R'} \mathcal{A}(c')$, where R' is the union of all relations $R_{c(root)}$ and $R_{c(o,i)}$, for $o \in OName$ and $i \in \mathbb{N}$. This justifies the following rule.

Rule 1. *Let OO be an OO-action system such that $\langle c, \mathcal{C}\rangle \in OO$. Let also $\langle c, \mathcal{C}'\rangle$ be another class (which cannot be in OO because a class with the name c is already in it) such that $\langle c, \mathcal{C}\rangle$ is class-refined by $\langle c, \mathcal{C}'\rangle$. Then*

$$OO \sqsubseteq OO' = OO \setminus \{\langle C, \mathcal{C}\rangle\} \cup \{\langle C, \mathcal{C}'\rangle\},$$

where the class $\langle c, \mathcal{C}'\rangle$ is marked with an asterisk $*$ in OO' if the class $\langle c, \mathcal{C}\rangle$ is marked with an asterisk $*$ in OO.

Introducing new classes Next we give a rule which allows an OO-action system to duplicate one of its classes. The class which is duplicated can then be refined according to the previous rule obtaining a subclass. This may be useful because the superclass can have variables that are used by the refined class or some other class might share them. Moreover, some clients might use the superclass, and some others the subclass as will be made clear later in this section.

Rule 2. *Let OO be an OO-action system such that $\langle c, \mathcal{C}\rangle \in OO$ and assume $c' \in CName$ is a class name not used in OO. Let $\mathcal{C}'$ be a copy of $\mathcal{C}$ without declaration of shared attributes. Then*

$$OO \sqsubseteq OO \cup \{\langle c', \mathcal{C}'\rangle\},$$

where the class $\langle c', \mathcal{C}'\rangle$ is not marked by an asterisk.

This rule can be justified as follows. The idea is that we refine within an iterative command an action A by the action $A \mathbin{[\!]} g \to A'$, where g is a boolean variable initially set to false and which cannot be set to true by A. Informally, introducing a new class $\langle c', \mathcal{C}'\rangle$ means that new actions and new exportable procedures are now introduced in the action system $\mathcal{A}(OO)$. However, these actions and procedure bodies are guarded by a boolean variable which is initially set to false. Only if the environment calls the procedure $c'(o,i).init$ for some $o \in OName$ and $i \in \mathbb{N}$ can this guard be set to true. But this procedure can only be called if a statement $new(c')$ or $n := new(c')$ is present in the action of some other class in OO. Since the class name c' is not used in OO such *new* statements cannot be present in any class of OO. For a similar reason, no full context of $\mathcal{A}(OO)$ can call any procedure $c'(o,i).init$.

Removing a class A converse of the above rule is possible. A class that has no declaration of shared attributes, that is not marked by an asterisk, and such that no other class refers to it can be safely removed from an OO-action system.

Rule 3. *Let $\langle c, \mathcal{C}\rangle$ be a class of an OO-action system OO such that no shared attributes are declared in $\mathcal{C}$, it is not marked by an asterisk, and for all other classes $\langle c', \mathcal{C}'\rangle \in OO$ with $c' \neq c$ no statement $new(c)$ or $n := new(c)$ is present in the actions of $\mathcal{C}'$. Then*

$$OO \sqsubseteq OO \setminus \{\langle c, \mathcal{C}\rangle\}\,.$$

This rule can be justified in a similar way as the Rule 2. Note that each class introduced by Rule 2 can be safely removed by the above rule obtaining the original OO-action system. Similarly, each class removed by the above rule can be re-introduced by Rule 2 obtaining the original OO-action system.

Refining new statements In a refinement we are interested only in the values of the shared attributes. It is of no importance which object acts on which attributes. Hence it is unobservable if an object is an instance of a class named c or of another class d if both classes have the same body (except for the list of shared attributes declared, which must be disjoint).

Rule 4. *Let OO be an OO-action system and assume the $\langle c, \mathcal{C}\rangle$ and $\langle c', \mathcal{C}'\rangle$ are two classes in OO such that $\mathcal{C}'$ is a copy of $\mathcal{C}$ except for declaration of the shared attributes which are distinct. Let $\langle d, \mathcal{D}\rangle$ be a class in OO and define $\mathcal{D}'$ to be as $\mathcal{D}$ but where some of the occurrences of $new(c)$ have been replaced either by $new(c')$ or by $new(c) \,[\!]\, new(c')$, and some of the of the occurrences of $n := new(c)$ have been replaced either by $n := new(c')$ or by $n := new(c) \,[\!]\, n := new(c')$. Then*

$$OO \sqsubseteq OO \setminus \{\langle d, \mathcal{D}\rangle\} \cup \{\langle d, \mathcal{D}'\rangle\}\,.$$

The class $\langle d, \mathcal{D}'\rangle$ is marked by an asterisk only if $\langle d, \mathcal{D}\rangle$ is marked by an asterisk in OO.

The above rule can be justified by an argument similar to the following. Assume p_1 and p_2 are two procedures with the same body. If A is an action containing a call to the procedure p_1 then it can be refined by an action A' where p_2 or $p_1 \,[\!]\, p_2$ substitutes p_1.

Client refinement Rule 4 can be used in combination with the previous rules to model explicit client refinement as follows: we change the name of the class to be refined by creating a new class (Rule 2), then change the appropriate clients to refer to the newly created class by refining the *new* statements (Rule 4). Finally, we can refine the newly created class (Rule 1). In this case in an OO-action system we have both the superclass and the subclasses of the same original class.

To model implicit client refinement we can use Rule 1: we replace in an OO-action system a class with another class refining it without changing the name of the class. In this way every client of the superclass becomes implicitly a client of the subclass.

Inheritance and overriding Rule 1 for class refinement models inheritance in a limited manner: the new class $\langle c, \mathcal{C}' \rangle$ inherits all the methods m as well as the local procedures q from its superclass $\langle c, \mathcal{C} \rangle$. Method overriding takes place via refinement, as the bodies of the procedures declared in $\mathcal{C}$ might change in the refinement step. Using this rule no new global methods are provided by the subclass $\mathcal{C}'$. The formal parameters of the methods are not to be changed in a refinement step.

A special case of Rule 1, the superposition rule [BS92,BS94], could be used instead of the above, more general data refinement rule. In superposition we do not remove old attributes, only new attributes are added into a class. Hence, a subclass would work with all attributes declared in the superclass and with some new, added attributes.

Introducing new methods Next we give a rule for the introduction of new methods. The idea is similar as for the Rule 2 for the introduction of new classes. Indeed a local procedure p of a class $\langle c, \mathcal{C} \rangle$ can be made a method without any observable change of the behaviour of the system, because only actions of $\mathcal{C}$ can call the new method p.

Rule 5. *Let $\langle c, \mathcal{C} \rangle$ be a class in an OO-action system OO, and let $\mathcal{C}'$ be as $\mathcal{C}$ except that some local procedures of $\mathcal{C}$ are now declared as methods. Then*

$$OO \sqsubseteq OO \setminus \{\langle c, \mathcal{C} \rangle\} \cup \{\langle c, \mathcal{C}' \rangle\}\,.$$

The class $\langle c, \mathcal{C}' \rangle$ is marked by an asterisk only if $\langle c, \mathcal{C} \rangle$ is marked by an asterisk in OO.

This rule can be easily justified by the fact that an action system is always refined by an equivalent one where we make some local procedures global.

Merging classes Let OO be an OO-action system, and let $\langle c_1, \mathcal{C}_1 \rangle$ and $\langle c_2, \mathcal{C}_2 \rangle$ be two classes in OO where

$$\mathcal{C}_1 = |[\; \begin{array}{ll} \textbf{attr} & y^* := y0 \;;\; x := x0 \\ \textbf{obj} & n \\ \textbf{meth} & m_1 = M_1 \;; \cdots ;\; m_h = M_h \\ \textbf{proc} & p_1 = P_1 \;; \cdots ;\; p_k = P_h \\ \multicolumn{2}{l}{\textbf{do}\ A_1\ \textbf{od}} \end{array} \\ \quad]| \qquad \mathcal{C}_2 = |[\; \begin{array}{ll} \textbf{attr} & z^* := z0 \;;\; w := w0 \\ \textbf{obj} & n' \\ \textbf{meth} & r_1 = R_1 \;; \cdots ;\; r_i = R_i \\ \textbf{proc} & q_1 = Q_1 \;; \cdots ;\; q_j = Q_j \\ \multicolumn{2}{l}{\textbf{do}\ A_2\ \textbf{od}} \end{array} \\ \quad]|$$

Since both classes are in OO it follows that the list of shared attributes y and z are disjoint. Furthermore, we assume that the lists of local attributes x and object variables n are disjoint from w and n', and that the methods m and procedures p are distinct from the methods r and the procedures q.

Then we have the following rule that allows us to introduce a class into an OO-action system by merging two classes together.

Rule 6. *Let OO be an OO-action system, and let $\langle c_1, \mathcal{C}_1\rangle$ and $\langle c_2, \mathcal{C}_2\rangle$ be two classes in OO as above. Let also c be a class name not used by the classes in OO. Then*

$$OO \sqsubseteq OO' \cup \{\langle c, \mathcal{C}\rangle\}$$

where

$$\begin{array}{rll}\mathcal{C} = |[& \textbf{attr} & x := x0 \ ; \ w := w0 \\ & \textbf{obj} & n \ ; \ n' \\ & \textbf{meth} & m_1 = M_1 \ ; \cdots ; \ m_h = M_h \ ; \\ & & r_1 = R_1 \ ; \cdots ; \ r_i = R_i \\ & \textbf{proc} & p_1 = P_1 \ ; \cdots ; \ p_k = P_h \ ; \\ & & q_1 = Q_1 \ ; \cdots ; \ q_j = Q_j \\ & \multicolumn{2}{l}{\textbf{do } A_1 \ [\!] \ A_2 \textbf{ od}} \\ ]| & & \end{array}$$

and OO' is obtained by replacing some occurrences of $new(c_1)$; $new(c_2)$ by $new(c)$ in the actions of the classes in OO.

The converse rule for splitting a class in two is more problematic: a local procedure of one of the classes may need to become a method in order for an action of the other class to call it; but this introduces the problem of which object instance should be considered. Also conditions on the constant *self* should be given in order to preserve its intended meaning.

6 Reusing refinements and classes

In the above rules for class refinement the goal was to reuse existing classes and use them via inheritance. Let us now look at how to reuse classes in a different way, namely reuse refinements between classes.

Reusing refinements Consider a class $\langle c, \mathcal{C}\rangle$ with

$$\begin{array}{rll}\mathcal{C} \ = \ |[& \textbf{attr} & y^* := y0 \ ; \ x := x0 \\ & \textbf{obj} & n \\ & \textbf{meth} & m_1 = M_1 \ ; \cdots ; \ m_h = M_h \\ & \textbf{proc} & p_1 = P_1 \ ; \cdots ; \ p_k = P_k \\ & \multicolumn{2}{l}{\textbf{do } A \textbf{ od}} \\ ]| & & \end{array}$$

and assume that $\langle c, \mathcal{C}\rangle$ is *class-refined* by a class $\langle c, \mathcal{C}'\rangle$ with

$$\begin{array}{rll}\mathcal{C}' \ = \ |[& \textbf{attr} & y^* := y0 \ ; \ x' := x0' \\ & \textbf{obj} & n \\ & \textbf{meth} & m_1 = M_1' \ ; \cdots ; \ m_h = M_h' \\ & \textbf{proc} & p_1' = Q_1' \ ; \cdots ; \ p_j' = P_j' \\ & \multicolumn{2}{l}{\textbf{do } A' \textbf{ od}} \\ ]| & & \end{array}$$

This refinement step can be reused when the body of the class $\langle c, \mathcal{C}\rangle$ is 'inside' another class $\langle d, \mathcal{D}\rangle$ where

$$
\begin{array}{lll}
\mathcal{D} = |[& \textbf{attr} & y^* := y0 \; ; \; x := x0; \\
& & z^* := z0 \; ; \; w := w0 \\
& \textbf{obj} & n; \\
& & n' \\
& \textbf{meth} & m_1 = M_1 \; ; \cdots ; \; m_h = M_h \, ; \\
& & r_1 = R_1 \; ; \cdots ; \; r_i = R_i \\
& \textbf{proc} & p_1 = P_1 \; ; \cdots ; \; p_k = P_k \, ; \\
& & q_1 = Q_1 \; ; \cdots ; \; q_l = Q_l \\
& \multicolumn{2}{l}{\textbf{do}\ A \ [\!] \ B \ \textbf{od}} \\
]| & &
\end{array}
$$

Hence, the class $\langle d, \mathcal{D}\rangle$ extends the class $\langle c, \mathcal{C}\rangle$ because it has new shared attributes z, local attributes w, name variables n', methods r, local procedures q, and a new action B.

Because the class $\langle c, \mathcal{C}\rangle$ is *class-refined* by the class $\langle c, \mathcal{C}'\rangle$, the class $\langle d, \mathcal{D}\rangle$ extending $\langle c, \mathcal{C}\rangle$ is *class-refined* by the class $\langle d, \mathcal{D}'\rangle$, where

$$
\begin{array}{lll}
\mathcal{D}' = |[& \textbf{attr} & y^* := y0 \; ; \; x' := x0'; \\
& & z^* := z0 \; ; \; w := w0 \\
& \textbf{obj} & n; \\
& & n' \\
& \textbf{meth} & m_1 = M_1' \; ; \cdots ; \; m_h = M_h' \, ; \\
& & r_1 = R_1 \; ; \cdots ; \; r_i = R_i \\
& \textbf{proc} & p_1' = P_1' \; ; \cdots ; \; p_j' = P_j' \, ; \\
& & q_1 = Q_1 \; ; \cdots ; \; q_l = Q_l \\
& \multicolumn{2}{l}{\textbf{do}\ A' \ [\!] \ B \ \textbf{od}} \\
]| & &
\end{array}
$$

provided that the action B does not contain calls to procedures $p_1 \cdots p_k$. Hence, we can reuse the proof of a refinement of a class $\langle c, \mathcal{C}\rangle$ for classes $\langle d, \mathcal{D}\rangle$ with $\mathcal{D}$ syntactically including $\mathcal{C}$. In particular this can be used to prove that class refinement does not depend on class names.

Reusing a set of classes Let OO and OO' be two OO-action systems such that $OO \sqsubseteq OO'$ and let $\langle c, \mathcal{C}\rangle$ be a class in OO marked with an asterisk such that there exists a class $\langle c, \mathcal{C}'\rangle \in OO'$ with the same name c and also marked with an asterisk $*$. Let now $\langle d, \mathcal{D}\rangle$ be another class such that the name d is not used in both OO and OO'. Then

$$\{\langle d, \mathcal{D}\rangle^*\} \cup OO \sqsubseteq \{\langle d, \mathcal{D}\rangle^*\} \cup OO' \, ,$$

where the class $\langle c, \mathcal{C}\rangle \in OO$ and $\langle c, \mathcal{C}'\rangle \in OO'$ are not marked with an asterisk. Notice that actions of the class $\langle d, \mathcal{D}\rangle$ may or may not contain the statements '$new(c)$' or '$n := new(c)$'. Hence, the behaviour of the new OO-action system can be completely different from the behaviour of the action systems OO and OO'.

In the above case we have reused an entire OO-action system OO and made it part of a bigger OO-action system. We can interpret the above so that entire class specifications are modules that can be (re)used by other class specifications.

7 Examples of refinement

In this section we give some examples of refinement of the OO-action systems modelling a simple phone company introduced in Section 3.2. We want to refine the system in order to allow for two kind of phones: ordinary phones and payphones. The latter must be managed by a separate company. All phone companies start their activity only after the government creates them. For sake of brevity we omit the details of the refinement steps and present only the results.

We begin with refining the body of the class $\langle Phone, Ph\rangle$. We introduce a new type of phone PPh, a so-called payphone. It has a *credit* variable, which can recharge itself. If there is no credit, then the phone is not working in the sense that it cannot initiate phone calls. However, when there is no credit, then it is still possible to receive calls. In the description below of PPh we do not write the bodies of procedures and methods of the class which are not changed with respect to their definition in Ph.

$$
\begin{array}{lll}
PPh = |[& \textbf{attr} & number := -1; \\
& & idle := true \; ; \; registered := false \; ; \; credit := 0 \\
& \textbf{obj} & company \; ; \; callee \\
& \textbf{meth} & Accept\_call\_from(n) = \cdots \\
& & Where(y) = \cdots \\
& & Return = \cdots \\
& \textbf{proc} & Call = \cdots \\
& \textbf{do} & \\
& & \quad idle \wedge registered \wedge credit > 0 \rightarrow Call \; ; \; credit := credit - 1 \\
& & [\!] \; \neg idle \rightarrow callee.Return \; ; \; idle := true \\
& & [\!] \; credit := credit + 1 \\
& \textbf{od} & \\
]| & &
\end{array}
$$

It is not hard to see that the class $\langle Phone, PPh\rangle$ refines the class $\langle Phone, Ph\rangle$. Hence, using Rule 1 we have

$$
\begin{gathered}
\{\langle PhoneCompany, PC\rangle^*, \langle Phone, Ph\rangle\} \\
\sqsubseteq \\
\{\langle PhoneCompany, PC\rangle^*, \langle Phone, PPh\rangle\}\,.
\end{gathered}
$$

In order to have two kinds of phones (normal phones and payphones) in the same system we can use Rules 2 and 1. Indeed first we can introduce a class $\langle PayPhone, Ph\rangle$ using Rule 2 and then we can refine the system replacing Ph with the above PPh in the class named $PayPhone$ obtaining the refinement

$$
\begin{gathered}
\{\langle PhoneCompany, PC\rangle^*, \langle Phone, Ph\rangle\} \\
\sqsubseteq \\
\{\langle PhoneCompany, PC\rangle^*, \langle Phone, Ph\rangle, \langle PayPhone, PPh\rangle\}\,.
\end{gathered}
$$

Note that the payphones are not (yet) connected to the rest of the system, in the sense that no instances of the class *PayPhone* may be created by the above system. To solve this problem we introduce in a similar way a second phone company $\langle PhoneCompany2, PC2\rangle$, where $PC2$ is a copy of PC but without the declaration of the shared attributes *phones* and *phonebook* and where phones are replaced by payphones in the object constructor *new*. As before, we do not write down the bodies of procedures and methods of $PC2$ which are not changed with respect to their definition in PC:

$$
\begin{array}{lll}
PC2 = \;|[& \textbf{attr} & allow\_new\_phones := true \\
& \textbf{obj} & entry \; ; \; names[1] \; ; \cdots ; \; names[n] \; ; \cdots \\
& \textbf{meth} & Give\_name(x, n) = \cdots \\
& \textbf{proc} & Update(n) = \cdots \\
& \textbf{do} & \\
& & \quad allow\_new\_phones \rightarrow \; entry := new(PayPhone); \\
& & \qquad\qquad\qquad\qquad\quad Update(entry); \\
& & \qquad\qquad\qquad\qquad\quad entry.Where(self) \\
& & [\!] \; allow\_new\_phones :\in \{true, false\} \\
& \textbf{od} & \\
]| & &
\end{array}
$$

Using Rules 2, 1, and 4 we have obtained the following refinement:

$$
\begin{gathered}
\{\langle PhoneCompany, PC\rangle^*, \langle Phone, Ph\rangle\} \\
\sqsubseteq \\
\{\langle PhoneCompany, PC\rangle^*, \langle PhoneCompany2, PC2\rangle, \\
\langle Phone, Ph\rangle, \langle PayPhone, PPh\rangle\} \; .
\end{gathered}
$$

Now object instance of the class *PayPhone* may be created by instances of the class *PhoneCompany2*. In order to create an instance of this class we make the following two steps. First we apply the rule for reusing a set of classes and add a new class *Government* which creates two instances of the same phone company *PhoneCompany*. Then we refine the government in order to create an instance of both phone companies. Define the body of the class *Government* as follows

$$
Gvt = |[\; \textbf{do} \; new(PhoneCompany) \; ; \; new(PhoneCompany) \; ; \; kill(self) \; \textbf{od} \;]|
$$

Basically, a *Government* creates two instances of the same kind of phone company and then terminates its activity. By reuse of a set of classes we have

$$
\begin{gathered}
\{\langle Government, Gvt\rangle^*, \langle PhoneCompany, PC\rangle, \langle Phone, Ph\rangle\} \\
\sqsubseteq \\
\{\langle Government, Gvt\rangle^*, \langle PhoneCompany, PC\rangle, \langle PhoneCompany2, PC2\rangle \\
\langle Phone, Ph\rangle, \langle PayPhone, PPh\rangle\}
\end{gathered}
$$

In the second step we refine Gvt by Gvt' replacing one $new(PhoneCompany)$ by $new(PhoneCompany2)$:

$$
Gvt' = |[\; \textbf{do} \; new(PhoneCompany) \; ; \; new(PhoneCompany2) \; ; \; kill(self) \; \textbf{od} \;]|
$$

Using Rule 4, we conclude with the last refinement

$$\{\langle Government, Gvt\rangle^*, \langle PhoneCompany, PC\rangle, \langle Phone, Ph\rangle\}$$
$$\sqsubseteq$$
$$\{\langle Government, Gvt'\rangle^*, \langle PhoneCompany, PC\rangle, \langle PhoneCompany2, PC2\rangle$$
$$\langle Phone, Ph\rangle, \langle PayPhone, PPh\rangle\}$$

8 Concluding remarks

We have extended the action systems framework with objects and defined a new language, OO-action systems. The formal semantics of the new language is given by translating the OO-action systems into ordinary action systems. Moreover, we defined a refinement relation between OO-action systems and gave a number of refinement rules expressing inheritance and reuse of code and refinements.

Also Back et al. [BBS97] give the formal semantics for their Action-Oberon by reducing them to ordinary action systems. Instead of translating the system into an infinite number of objects as we do here (although they mention this possibility), they use a data structure which keeps information about each object, inheritance history and focus on the n-ary communication between objects which participate in actions.

Bailes and Duke [BD91] define a notion of ecological refinement between pairs of classes and their environments. Our refinement notion between classes conforms to theirs in the sense that both define the refinement as a relation between the externally visible behaviours of the systems. Moreover, also our environment (the clients) can change during refinement steps as allowed by the ecological refinement.

There seems to be a close connection between our computational model of OO-action systems and the way action systems are modeled [Wal97] within the B Method [Abr96]. Both formalisms give names to action systems and their refinements as well as their global variables are treated in a similar way. This connection might prove useful as it potentially allows us to use the tool support of the B-Method on OO-action systems. This is, however, left for future studies.

Observe that we do not consider interface refinement in this paper. Hence, all the methods of the subclass appear the same as those of the superclass to a specific client class the formal parameters of the procedures being identical. Interface refinement in the context of class refinement has been studied e.g. by Mikhajlova and Sekerinski [MS97]. This problem is left for future research.

Finally, it would be interesting to study both an operational and a denotational semantics of OO-action systems without translating them into action systems, using, for example, the work of America et al. [ABKR86,ABKR89].

Acknowledgments The authors are grateful to Robin Milner for his suggestion to use the theory of remote procedures of action systems for object-oriented systems. We would also like to thank Martin Büchi, Marina Waldén and Jockum von Wright for their comments and several discussions on the contents of this paper. We are very much indebted to the anonymous referees, who have written detailed reports, pointing out some errors and suggesting ways of improving the presentation. The work of Kaisa Sere is supported by the Academy of Finland and the work of Marcello Bonsangue is supported by the *Stichting Informatica Onderzoek in Nederland* within the context of the project no. 612-33-007 'Formal methods and refinement for *Co*ordination *La*nguages'.

References

[ABKR86] P. America, J.W. de Bakker, J.N. Kok, and J.J.M.M. Rutten. Operational semantics of a parallel object-oriented language. In *Proceedings of the 13th Annual ACM Symposium on Principles of Programming Languages*, pages 194–208, 1986.

[ABKR89] P. America, J.W. de Bakker, J.N. Kok, and J.J.M.M. Rutten. Denotational semantics of a parallel object-oriented language. *Information and Computation*, 83(2):152–205, 1989.

[Abr96] J.-R. Abrial. *The B-Book.* Cambridge University Press, 1996.

[Agh86] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, 1986.

[AL97] M. Abadi and K.R.M. Leino. A logic of object-oriented programs. In *Theory and Practice of Software Development, TAPSOFT'97*, volume 1214 of *Lecture Notes in Computer Science*, Springer-Verlag, 1997.

[Bac80] R.-J.R. Back. *Correctness Preserving Program Refinements: Proof Theory and Applications.* Volume 131 of *Mathematical Centre Tracts.* Mathematical Centre, Amsterdam, 1980.

[Bac87] R.J.R. Back. Procedural abstraction in the refinement calculus. Technical Report, Åbo Akademi University, Department of Computer Science, Turku, Finland 1987.

[Bac90] R.J.R. Back. Refinement calculus, part II: parallel and reactive programs. In J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*, volume 430 of *Lecture Notes in Computer Science*, pages 67–93, Springer-Verlag, 1990.

[BBS97] R.J.R. Back, M. Büchi, and E. Sekerinski. Action-based concurrence and synchronization for objects. In *Proceedings of the 4th AMAST Workshop on Real-Time Systems, Concurrent, and Distributed Software, Ciutad de Mallorca*, Springer-Verlag, 1997.

[BD91] C. Bailes and R. Duke. The Ecology of class refinement. In J.M. Morris and C. Shaw, editors, *Proceedings of the 4th Refinement Workshop*, January 1991, Cambridge, UK. *Workshops in Computing*, pp. 185–196, Springer Verlag.

[BK83] R.J.R. Back and R. Kurki-Suonio. Decentralization of process nets with centralized control. *Distributed Computing*, 3(2):73–87, 1983.

[BK94] M.M. Bonsangue and J.N. Kok. The weakest precondition calculus: recursion and duality. *Formal Aspects of Computing*, 6A:788–800, 1994.

[BN96] M.H. Brown and M.A. Najork. Distributed active objects. SRC Research Report 141a, DEC Palo Alto CA, 1996.

[BS92] R.J.R. Back and K. Sere. Superposition refinement of parallel algorithms. In Parker, K.R. and Rose, editors, *Proceedings of the IFIP Working Conference on Formal Description Techniques - IV*, pages 475–494. North-Holland Publishing Company, 1992.

[BS94] R.J.R. Back and K. Sere. Action systems with synchronous communication. In E.-R. Olderog, editor, *Proceedings of the IFIP Working Conference on Programming Concepts, Methods and Calculi*, IFIP Transactions A-56, pages 107–126, North-Holland Publishing Company, 1994.

[BS96] R.J.R. Back and K. Sere. From action systems to modular systems. *Software - Concepts and Tools*, 17, Springer Verlag, 1996.

[Car95] L. Cardelli. A language with distributed scope. *Computing Systems* 8(1):27–29, 1995.

[CM88] K.M.Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.

[Dij76] E.W. Dijkstra. *A Discipline of Programming*. Prentice–Hall International, 1976.

[Hew77] C. Hewitt. Viewing control structures as patterns of passing messages. *Artificial Intelligence* 8(3), 1977.

[HHJT98] U. Hensel, M. Huisman, B. Jacobs, and H. Tews. Reasoning about classes in object-oriented languages: logical models and tools. To appear in *Proceedings of ESOP/ETAPS 1998*, *Lecture Notes in Computer Science*, Springer-Verlag, 1998.

[Hoa72] C.A.R. Hoare. Proofs of correctness of data representation. *Acta Informatica*, 1(4):271–281, 1972.

[JK91] H.-M. Järvinen and R. Kurki-Suonio. DisCo specification language: marriage of actions and objects. In *Proceedings of the 11th International Conference on Distributed Computing Systems*, IEEE Computer Society Press, pages 142-151, 1991.

[Jon93] C.B. Jones. A π-calculus semantics for an object-based design notation. In *Proceedings of CONCUR'93*, volume 715 of *Lecture Notes in Computer Science*, Springer-Verlag, 1993.

[Lam91] L. Lamport. The temporal logic of actions. Research report 79, DEC Systems research Center, 1991. To appear in *ACM Transactions on Programming Languages and Systems*.

[LH92] K. Lano and H. Haughton. Reasoning and refinement in object-oriented specification languages. In *European Conference on Object-Oriented Programming'92*, volume 615 of *Lecture Notes in Computer Science*, Springer-Verlag 1992.

[MS97] A. Mikhajlova and E. Sekerinski. Class refinement and interface refinement in object-oriented programs. In *Proceedings of the Fourth International Formal Methods Europe Symposium (FME'97)*, volume 1313 of *Lecture Notes in Computer Science*, Springer-Verlag, 1997.

[Mor88] C. Morgan. The specification statement. *ACM Transactions on Programming Languages and Systems* 10:3, pages 403–419, 1988.

[Mor87] J.M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming*, 9:287–306, 1987.

[Nau94] D.A. Naumann. Predicate transformer semantics of an Oberon-like language. In E.-R. Olderog, editor, *Proceedings of the IFIP Working Conference on Programming Concepts, Methods and Calculi*, IFIP Transactions A-56, North-Holland Publishing Company, 1994.

[Str95] B. Strulo. How firing conditions help inheritance. In J.P. Bowen and M.G. Hinchey, editors, *ZUM'95: The Z Formal Specification Notation*, volume 967 of *Lecture Notes in Computer Science*, Springer-Verlag 1995.

[SW97] K. Sere and M. Waldén. Data refinement of remote procedures. In M. Abadi and T. Ito, editors, *Proceedings of the International Symposium on Theoretical Aspects of Computer Software (TACS'97), Sendai, Japan*, volume 1281 of *Lecture Notes in Computer Science*, pages 267–294, Springer-Verlag, 1997.

[UK95] R.T. Udink and J.N. Kok. ImpUNITY: UNITY with Procedures and Local Variables. In *Proceedings of Mathematics of Program Construction '95*, volume 947 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.

[Wal95] D.J. Walker. Objects in the π-calculus. *Information and Computation*, 116(2):253–271, 1995.

[Wal97] M. Waldén. Layering distributed algorithms. Technical report No 121, Turku Centre for Computer Science, Turku, Finland, 1997. To appear in *Proceedings of the 2nd B conference*, France, April 1998.

[Wri94] J. von Wright. The lattice of data refinement. *Acta Informatica*, 31(2):105–135, 1994.