DEVELOPING OBJECT-BASED DISTRIBUTED SYSTEMS

Marcello M. Bonsangue*
Joost N. Kok*
and Kaisa Sere†

Abstract: The OO-action systems formalism is a recent extension of action systems towards object-orientation. An OO-action system models an object-oriented system with active objects. In this paper we make the notion of a distributed object clear within this framework. Moreover, we show how object-based distributed systems are designed stepwise within a formal framework, the refinement calculus.

INTRODUCTION

The term *distributed object* can be understood in two ways: (1) a distributed object is an object that resides in some node of a network, or (2) it is an object which is itself distributed among a set of nodes in a network. Typically, the former interpretation is used and hence, a distributed object [25] is an object that can live anywhere in a network. Distributed objects are independent of each other and they communicate via method invocations. The latter approach gives an interesting and novel interpretation of a distributed object. In this paper we study a framework that supports both views in a unifying manner.

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

P. Ciancarini et al. (eds.), *Formal Methods for Open Object-Based Distributed Systems*

We propose an approach to the design of object-based distributed systems, so called OO-action systems, that is based on the action systems formalism. 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 are typically *active* having autonomous actions. Communication between objects takes place via method invocations and shared variables. Shared variables are of importance for OO-action systems since they determine the externally observable behavior of the system.

In our earlier work [10], we showed how an OO-action system is translated into an ordinary action system [8]: methods correspond to exportable procedures, attributes to shared and local variables, object variables to local variables, and classes to entire action systems. A collection of classes is translated into a parallel composition of action systems. This translation allows us to use most of the theory built around action systems even when designing OO-action systems.

In this paper we study the the OO-action systems without referring to the underlying translation. Hence we present a self-contained syntax and refinement rules for OO-action systems. Our goal is to make clear the notion of a distributed object within this framework supporting both views to object distribution described above. The second view where the objects themselves are distributed is faithful to the original action system approach to distributed computing [8]. Moreover, a somewhat similar interpretation is taken in the DisCo specification language [16] as well as in [7]. In both of these action systems related works, objects are active and spend their lives participating in enabled actions. Otherwise our approaches to object-orientation and reasoning about objects are different.

The OO-action system formalism is related with POOL [4] because objects are created dynamically, and their names 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 [15, 1]. Moreover, CCS and the π-calculus, have been used to give a semantics to POOL-type parallel object-oriented languages [17, 27]. Recently several formalisms and languages have been proposed that offer active objects, e.g. Java, Oblique [12] which supports distributed object-oriented computation and Oblets [11] which are written in Oblique and which have a family of Web browsers capable of running Oblets.

The refinement calculus and related calculi [23, 24] have become popular foundations for program construction and for reasoning about specifications and implementations that meet their specifications. Action systems are designed stepwise within the refinement calculus [9]. We therefore define the notion of *refinement* for OO-action systems within the refinement calculus. Our notion of refinement is based on observing the state of the system via its global, observable variables. The reasoning about parallel and distributed objects is carried out in a purely sequential manner, the usual way of reasoning about action systems. We present proof rules to be used when designing and

reasoning about OO-action systems. Due to space restriction we will be rather informal in presenting and justifying these rules. Our goal is to show how they can be used to refine an arbitrary OO-action system towards a system with distributed objects. Moreover, both methods, actions, and the constructors of a class can be refined and refinement effectively supports reuse of code.

When we refine a class with another class, the refined class can inherits 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. Class refinement (= inheritance) ensures that each set of successful computations w.r.t. the observable attributes with the original class is also satisfied by the refined class. Moreover, we show how object-based distributed systems are stepwise specified and developed within the refinement calculus via specialized class refinements.

Our notion of class refinement is based on data refinement for action systems [6, 26]. Class refinement in the data refinement framework has also been studied by Mikhajlova and Sekerinski [21]. 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 [19]. Alternative frameworks for reasoning about object oriented systems include TLA [18] used for reasoning about DisCo specifications, Hoare-style logic [2], and coalgebras, used for automatic reasoning on CCSL and JAVA classes [14].

We proceed as follows. In the next section we introduce the OO-action systems, a formalism for the specification and development of object-oriented systems. Then we discuss ways in which distributed objects are modeled as OO-action systems. To illustrate the formalism we present an example of distributed OO-action systems. In the subsequent section, we first describe some rules for the refinement of distributed OO-action systems and then we give some examples of refinement steps that aim towards distribution. We end in the last section with some concluding remarks.

OO-ACTION SYSTEMS

In this section we introduce OO-action systems. An OO-action system consists of a finite set of classes, each class body describing how instances of the class (objects) behave. Objects are dynamically created and executed in parallel.

Actions. We will consider a fixed set *Attr* of attributes (variables) and assume that each attribute is associated with a nonempty set of *values*. Also, we consider a set *Act* of actions defined by the following grammar

$$A \quad ::= \quad \mathit{abort} \mid \mathit{skip} \mid x := v \mid x :\in V \mid b \rightarrow \mid \{b\} \mid A\,;A \mid A \parallel A\,.$$

Here x is a list of attributes, v a list of values (possibly resulting from the evaluation of a list of expressions), V a nonempty set of values, b is a predicate over attributes. Intuitively, '*abort*' is the action which always deadlocks, '*skip*'

is a stuttering action, '$x := v$' is a multiple assignment, '$x :\in V$' is a random assignment, '$b \rightarrow$' is a guard of an action, '$\{b\}$' is an assertion, '$A_1 \; ; \; A_2$' is the sequential composition of the actions 'A_1' and 'A_2', and '$A_1 \; [\!] \; A_2$' is the nondeterministic choice between the actions 'A_1' and 'A_2'. For simplicity, we will often write '$b \rightarrow A$' in place of '$b \rightarrow \; ; A$'.

The semantics of the above language of actions is defined in a standard way using weakest preconditions [13]. For any predicate $\mathcal{P}$,

$$
\begin{array}{ll}
wp(abort, \mathcal{P}) = false & wp(b \rightarrow, \mathcal{P}) \;\; = b \Rightarrow \mathcal{P} \\
wp(skip, \mathcal{P}) \;\; = \mathcal{P} & wp(\{b\}, \mathcal{P}) \;\; = b \wedge \mathcal{P} \\
wp(x := v, \mathcal{P}) = \mathcal{P}[x/v] & wp(A_1 \; ; \; A_2, \mathcal{P}) = wp(A_1, wp(A_2, \mathcal{P})) \\
wp(x :\in V, \mathcal{P}) = \forall v \in V . \mathcal{P}[x/v] & wp(A_1 \; [\!] \; A_2, \mathcal{P}) = wp(A_1, \mathcal{P}) \wedge wp(A_2, \mathcal{P}) \, .
\end{array}
$$

We say that an action A is *enabled* in a given state if its *guard* $gd(A) \equiv \neg \, wp(A, false)$ holds in that state.

Classes and objects. Let *CName* be a fixed set of class names and *OName* a countable set of valid names for objects. We will also consider a fixed set of object variables *OVar* assumed to be disjoint from *Attr*. The only valid values for object variables are the names for objects in *OName*. The set of object actions *OAct* is defined by extending the grammar of actions as follows:

$$
\begin{array}{rcl}
O & ::= & A \mid q \rightarrow \mid n := o \mid new(c) \mid n := new(c) \mid p \mid n.m \mid \\
 & & self.m \mid super.m \mid O \; ; \; O \mid O \; [\!] \; O \; .
\end{array}
$$

Here $A \in Act$, q is a predicate over attributes and object variables, n is an object variable, o is either an object name or the constants *self* or *super* (all three possibly resulting from the evaluation of an expression), c is a class name, p a procedure name, and m is a method name. Intuitively, '$n := o$' stores the object name o into the object variable n, '$new(c)$' creates a new object instance of the class c, '$n := new(c)$' assigns the name of the newly created instance of the class c to the object variable n, 'p' is a procedure call, '$n.m$' 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 declared in the same object, and '$super.m$' is a call of the method m declared in the object that created the calling object. Note that method calls are always prefixed by an object variable or by the constants *self* or *super*. Procedures and methods with call-by-value and call-by-results parameters can be handled by substitution (see [5] for a detailed study).

We define the guard $gd(O)$ of an object action O to be the guard of the action in *Act* obtained by substituting every atomic object action of O with the action *skip*, where an atomic object action is

$$
q \rightarrow , \;\; n := o , \;\; new(c) , \;\; n := new(c) , \;\; p , \;\; n.m , \;\; self.m , \;\; super.m \, .
$$

The resulting statement is an action in *Act* and hence its guard is well-defined.

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

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

A class body consists of an object action O 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 object 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 $x0$.

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 a method name 'm' and an object action 'M'. If $m1(n) = n := n1$ is a method which assign to the formal parameter n an object name $n1$, then we write $n0.m1.m2$ in an object action O as an abbreviation for $n0.m1(n2)$; $n2.m2$, where $n2$ is an object variable not used in O.

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 a procedure name 'p' and an object action forming the body 'P'.

The *class body* is a description of the actions to be executed repeatedly when the object instance of the class is activated. It can refer to attributes which are declared to be 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 procedures declared in the class and method calls of the form $n.m$ or $super.m$ to methods declared in other classes. Method calls $self.m$ are allowed only if m is a method declared in the same class. As for action systems, the execution of an object action is atomic.

OO-action systems. 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 $*$. Execution starts by the creation of one object instance of each of these classes. Each object, when created, chooses enabled actions and executes them. Actions within an object operating on disjoint sets of local and shared attributes, and object variables can be executed in parallel. They can also create other objects. Actions of different active objects can be executed in parallel if they are operating on disjoint sets of shared attributes. Objects interact by means of the shared attributes and by executing methods of other objects.

DISTRIBUTED OO-ACTION SYSTEMS

Let us now consider in what ways we can model distributed objects with the OO-action systems formalism. Computationally, an OO-action system is a parallel composition of many objects, each of them representing an instance with of a class [10]. We can have parallel activity within an object as well as between different objects. Initially, only the objects generated from the classes marked with an asterisk are active, i.e. their actions are potentially enabled. These actions might make other objects active by executing a *new* statement in their action bodies.

Let us consider a network of nodes and edges. We associate each object with some node in the network where it is executed. In a purely distributed system the objects communicate only by sending and receiving messages, i.e. via method invocations. In an OO-action system this means that no shared variables are used as means of communication. In case an object communicates with another object it does this by calling a method of the other object.

From the above we have that a set of classes models a distributed object-based system when between every pair of classes referring to each others attributes there needs to be a communication medium for the underlying communicating objects. When objects do communicate via shared variables we can still consider the whole system to be distributed if the objects that communicate via the same shared variables are associated with a single node. The implementation again may provide a broadcasting network or a point-to-point network.

The above interpretation follows approach (1) to distributed objects as described in the introduction to this paper. Let us now consider approach (2) where the objects themselves can be distributed. For this we again have a network of nodes and edges, but now we associate each attribute with some node in the network. In this way the objects become distributed among the nodes. The actions of an object referring to the distributed attributes are now executed in co-operation by any number of processes in a distributed manner. Two or more actions of an object can be executed in parallel, if they do not refer to common attributes.

An example: a phone company. As an 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 *PhoneCmp* and another one named *Phone*:

$$\{\langle PhoneCmp, PC\rangle^*, \langle Phone, Ph\rangle\}\,. \tag{1.1}$$

The execution starts with the creation of one phone company (the class is marked with an asterisk). This phone company creates an arbitrary number phones (objects instance of the class named *Phone*) which may call each other.

The body *PC* of the class *PhoneCmp* 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 phone directory (a set of integers represented by the attribute *phonedir*) 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. Moreover, a copy of the current phone directory is handed over to the phone. Every now-and-then the phone company updates the individual phone directoriess of the customers via the *Update* method offered by the phones. 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).

```
PC=  ||[ attr   phones* := 0 ; phonedir := ∅ ; allow_new_phones := true
         obj    entry;
                names[1] ; ··· ; names[n] ; ···
         meth   Give_name(x, n) = (n := names[x])
         proc   Update(n) = ( phones := phones + 1;
                              phonedir := phonedir ∪ {phones};
                              names[phones] := n )
         do
              allow_new_phones →  entry := new(Phone);
                                  Update(entry);
                                  entry.Where(self);
                                  entry.Update(phonedir)
           [] ∀i ∈ {1..phones}:names[i].Update(phonedir)
           [] allow_new_phones :∈ {true, false}
         od
      ]| .
```

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

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.

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

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 phonedir 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.

The OO-action system above models a distributed system of objects: there are no shared attributes among the generated objects which only communicate via method invocations. The class *PhoneCmp* is a centralized resource with the responsibility of keeping every copy of the phone directory up-to-date. Hence, an object of this class must be able to communicate with every phone object. Moreover, the underlying communication network must be able to establish a communication channel between each pair of phones as a single phone can potentially call any other phone. Within the body *PC* we observe that out of the three actions the two last ones can be executed in parallel in an object as they share no attributes. An object of this class could thus be executed in a distributed manner.

REFINING OO-ACTION SYSTEMS

In this section we give a number of refinement rules that can be applied between OO-action systems. The rules are such that the observable behavior of objects in execution w.r.t the shared attributes is preserved during the refinement steps. Formally, the behavior of an OO-action system can be described by a set of traces: finite or infinite sequences of states describing only the shared attributes, without finite repetition of the same state (finite stuttering), and possibly terminating with a special symbol to denote abortion. A refinement relation between two OO-action systems preserving traces (when the system

to be refined does not abort) is presented in [10], where a set of rules for this relation is also given. Soundness of these rules is obtained via an infinitary translation of OO-action systems into action system. In this section we present some of these rules in a restricted version and a new rule (Rule 4) useful for generating new distributed objects. All these rules can be presented without the above infinitary translation, and are useful when working with distributed OO-action systems.

Action refinement. Refinement between actions in our framework is based on the weakest precondition predicate transformers semantics for actions [6]. Let A be an action that refers to the local attributes x and to the shared attributes y, and A' be an action referring to the local attributes x' and to y. We say that the action A is *data refined* by the action A' using an *abstraction relation* $R(x, x', y)$ between the local attributes x and x' and the shared attributes y if for all predicates $\mathcal{P}$ on x and y

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

where $\exists x.R \wedge \mathcal{P}$ is a predicate on x' and y. In other words, whenever A establishes a certain postcondition with respect to the variables y, so does A'. We denote by $A \leq_R A'$, the refinement of A by A' using R.

We cannot extend the above definition to object actions because we did not give a semantics for statements manipulating object variables in terms of predicate transformers. However we can define a notion of refinement between object actions piecewise, by refining only those parts of the action that do not refer to object variables and class names. For this purpose, we define a *context* to be an object action with a finite number of holes in it. Formally, we extend the grammar defining the set $OAct$ with a 'hole' symbol '$-$'.

Every object action O has a maximal context $C[-, \cdots, -]$ obtained by substituting each action in Act (of maximal size) of O by an hole $-$. Hence each object action O is equivalent to its maximal context applied to the actions which have been substituted by the holes. For example, the object action

$$n \neq self \rightarrow x \neq 0 \rightarrow new(c) \; ; \; (x := 0 \; [\!] \; x := 1)$$

is equivalent to $C[A_1, A_2]$, where

$$\begin{aligned} C[-,-] &\equiv n \neq self \rightarrow \; ; - \; ; new(c) \; ; - \\ A_1 &\equiv x \neq 0 \rightarrow \\ A_2 &\equiv (x := 0 \; [\!] \; x := 1). \end{aligned}$$

For each context $C[-, \cdots, -]$ we say that an object action $C[A_1, \cdots, A_n]$ is *data refined* by another object action $C[A'_1, \cdots, A'_n]$ via an abstract relation R if $A_i \leq_R A'_i$ for $i \in \{1, \cdots, n\}$.

Since every object action is a context applied to some ordinary actions, we denote the above refinement relation by $O \leq_R O'$. Informally, an object action O is data refined by another object action O' whenever every action in O

not referring to an object variable or class name is data refined via R by a corresponding action in O'. The actions referring to object variables and class names are left unrefined.

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. Let A be an action in Act. We say that a class $\langle c, C\rangle$ with

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

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

$$\begin{array}{lcll} C' & = & [\![& \textbf{attr} & y^* := y0\ ;\ x' := x'0 \\ & & & \textbf{obj} & n \\ & & & \textbf{meth} & m_1 = M_1'\ ;\ \cdots\ ;\ m_h = M_h' \\ & & & \textbf{proc} & p_1 = P_1'\ ;\ \cdots\ ;\ p_k = P_k' \\ & & & \textbf{do } O' \parallel A \textbf{ od} & \\ & &]\!] & & \end{array}$$

if there is a relation $R(x, x')$ between the local attributes x and x' such that,

1. Attributes: $R(a0, c0)$,
2. Methods: for each $i \in \{1, \cdots, h\}$, $M_i \leq_R M_i'$ and $R \wedge gd(M_i) \Rightarrow gd(M_i')$,
3. Procedures: for each $i \in \{1, \cdots, k\}$, $P_i \leq_R P_i'$ and $R \wedge gd(P_i) \Rightarrow gd(P_i)$,
4. Class Bodies: $O \leq_R O'$ and $R \wedge gd(O) \Rightarrow gd(O') \vee gd(A)$,
5. New actions: $skip \leq_R B$ and $R \Rightarrow wp(\textbf{ do } B \textbf{ od }, true)$.

Class-refinement between classes guarantees that the observable behavior of the objects generated from class $\langle c, C\rangle$ w.r.t to shared attributes y is preserved by the objects of the class $\langle c, C'\rangle$. Observe that in a more general setting the relation R would also depend on the shared attributes y [10].Since class-refinement preserves the behavior w.r.t. these attributes, class-refinement can be lifted to a refinement between OO-action system as follows.

Rule 1 *If OO be an OO-action system such that $\langle c, C\rangle \in OO$ and $\langle c, C\rangle$ is class-refined by $\langle c, C'\rangle$ then OO is* refined *by $OO \setminus \{\langle c, C\rangle\} \cup \{\langle c, C'\rangle\}$, where the class $\langle c, C'\rangle$ is marked with an asterisk if the class $\langle c, C\rangle$ is marked with an asterisk in OO.*

The soundness of the above rule, as well as for all other rules given below, is shown in [10] with respect to subset inclusion of finite or infinite sequences of states restricted only the shared attributes, without finite stuttering, and possibly terminating with a special symbol to denote abortion.

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.

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

Refining *new* statements. The following refinement rule allows us to make reference to classes that are refinements of existing classes in an OO-action system.

Rule 3 *Let OO be an OO-action system and assume that $\langle c, \mathcal{C}\rangle$ 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 is refined by $OO \setminus \{\langle d, \mathcal{D}\rangle\} \cup \{\langle d, \mathcal{D}'\rangle\}$. 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.*

Distributing a class. Using the next rule we can split a class into two separate classes both of which can generate new distributed objects.

Rule 4 *Let OO be an OO-action system such that $\langle c, \mathcal{C}\rangle \in OO$ with the class body:*

$$
\begin{array}{lcll}
\mathcal{C} & = & [\![& \textbf{attr} \quad y^* := y0 \,;\, x1, x2 := x10, x20 \\
 & & & \textbf{obj} \quad n1, n2 \\
 & & & \textbf{meth} \quad m1 = M1 \,;\, m2 = M2 \\
 & & & \textbf{proc} \quad p1 = P1 \,;\, p2 = P2 \\
 & & & \textbf{do } A1 \,[\!]\, A2 \textbf{ od} \\
 & &]\!] \,. &
\end{array}
$$

Let now $\langle c, \mathcal{C}'\rangle$ have the class body:

$$
\begin{array}{lcll}
\mathcal{C}' & = & [\![& \textbf{attr} \quad y^* := y0 \,;\, b1 := \mathit{true} \,;\, b2 := \mathit{true} \\
 & & & \textbf{obj} \quad n1', n2' \\
 & & & \textbf{meth} \quad m1 = (\neg b1 \rightarrow n1'.m1) \,;\, m2 = (\neg b2 \rightarrow n2'.m2); \\
 & & & \quad get\_n1'(n) = n := n1' \,;\, get\_n2'(n) = n := n2' \\
 & & & \textbf{do} \\
 & & & \quad b1 \rightarrow n1' := new(c1) \,;\, b1 := \mathit{false} \\
 & & & \quad [\!]\; b2 \rightarrow n2' := new(c2) \,;\, b2 := \mathit{false} \textbf{ od} \\
 & &]\!] \,. &
\end{array}
$$

Then OO is refined by $OO \setminus \{\langle c, \mathcal{C}\rangle\} \cup \{\langle c, \mathcal{C}'\rangle, \langle c1, \mathcal{C}1\rangle, \langle c2, \mathcal{C}2\rangle\}$ where $\langle c1, \mathcal{C}1\rangle$ is the class with the class body that we get when we remove $A2$ and the declarations of $y, x2, n2, m2, p2$ from $\mathcal{C}$ and add to it two methods per local attribute $x1$ and object variable $n1$, namely get_x1, get_n1 to read its value and

set_x1, set_n1 to assign a new value. Thereafter, every remaining read reference to x2 in C1 is replaced by super.get_n2'.get_x2 and every remaining write reference to it by super.get_n2'.set_x2. Every remaining reference to the object variable n2 is replaced by a reference to an object variable via super.get_n2'.get_n2. Moreover, every procedure p1 becomes a method and a call to it in C1 is replaced by self.p1. A remaining call to p2 is replaced by a method call super.get_n2'.p2. The class body of the class $\langle c2, C2\rangle$ is received similarly from C. The class $\langle c, C'\rangle$ is marked by an asterisk only if $\langle c, C\rangle$ is marked by an asterisk in OO.

We can informally justify this rule as follows. The original class body C is replaced by three class bodies $C', C1, C2$. The first of these, C', will redirect every call to a method $m1, m2$ to the respective method in one of the new classes $C1, C2$. If the two classes $C1, C2$ need to refer to each others local attributes or object variables, this is done via the *get* and *set* methods by first asking for the identity of the corresponding object from the class C' (*super.get_n1'*, *super.get_n2'*). The atomicity of actions guarantees that a read of the value of an attribute x via a method call *get_x* followed by a write of the attribute via a method call *set_x* will not be interfered by other actions.

Refining the phone company. In this section we give some examples of refinement of the OO-action systems modelling a simple phone company introduced earlier. We want to refine the system in order to allow for two kinds of phones: ordinary phones and pay phones. The latter must be managed by a separate company. Thereafter the work of the phone company is distributed among different departments. Moreover, the system of phones and pay phones must form a collection of independently working distributed objects.

We begin with refining the body of the class $\langle Phone, Ph\rangle$. We introduce a new type of phone *PPh*, a so-called pay phone. It has a *credit* variable, which can recharge itself up to a certain limit L. If there is no credit, the phone is not working in the sense that it cannot initiate phone calls. However, when there is no credit it is still possible to receive calls. It is easily proved that $\langle Phone, Ph\rangle$ is class-refined by $\langle Phone, PPh\rangle$ using the relation

$$a = c \wedge 0 \leq credit \leq L$$

where a denotes the local attributes of *Ph*, c the corresponding local attributes of *PPh*, and

$$\begin{array}{lll} PPh = [\![& \textbf{attr} & number := -1 \ ; \ idle := true; \\ & & phonedir := \emptyset \ ; \ registered := false \ ; \ credit := 0 \\ & \textbf{obj} & company \ ; \ callee \\ & \textbf{meth} & Accept\_call\_from(n) = \cdots \\ & & Where(y) = \cdots \\ & & Return = \cdots \\ & & Update(p) = \cdots \\ & \textbf{proc} & Call = \cdots \end{array}$$

```
        do
              idle ∧ registered ∧ credit > 0 → Call ; credit := credit − 1
          ▯   ¬idle → callee.Return ; idle := true
          ▯   credit < L → credit := credit + 1
        od
    ]] .
```

Hence, using Rule 2 and Rule 1 we have that the OO-action system given in (1.1) is refined by the OO-action system

$$\{\langle PhoneCmp, PC\rangle^*, \langle Phone, Ph\rangle, \langle PayPhone, PPh\rangle\} \tag{1.2}$$

Note that the pay phones are not connected to the rest of the system, because nobody creates such objects.

Next we use Rule 2 to introduce in a similar way, a second phone company PC' which is a copy of PC but for the shared attribute *phone* which is missing. Thus we obtain that the OO-action system described in (1.2) is refined by

$$\{\langle PhoneCmp, PC\rangle^*, \langle PhoneCmp2, PC'\rangle, \langle Phone, Ph\rangle, \langle PayPhone, PPh\rangle\}\,.$$

Thereafter we use Rule 3 to replace the phones of the new phone company by pay phones substituting the object action $new(Phone)$ by $new(PayPhone)$ in the first object action of the class. Hence the above OO-action system is refined by the OO-action system

$$\{\langle PhoneCmp, PC\rangle^*, \langle PhoneCmp2, PC2\rangle, \langle Phone, Ph\rangle, \langle PayPhone, PPh\rangle\}\,,$$

where $PC2 = PC'[new(PayPhone)/new(Phone)]$.

Finally, we want to distribute the work of the $PhoneCmp2$ by splitting its class body $PC2$ into three parts, $PC2', Mgt, PhD$, where $PC2'$ redirects the method calls of the original class $PC2$ to the new classes, Mgt models a management department that decides if new pay phones can be created, and PhD models a department that creates the phones and keeps them up-to-date. The class bodies are as follows:

```
PC2' = ||[  attr    b1, b2 := true, true
            obj     m, p
            meth    Give_name(x, n) = p.Give_name(x, n)
                    get_m(n) = n := m ; get_p(n) = n := p
            do
                  b1 → m := new(Management) ; b1 := false
              ▯   b2 → p := new(PhoneDept) ; b2 := false
            od
       ]|

Mgt  = ||[  attr    allow_new_phones := true
            meth    get_allow_new_phones = allow_new_phones
            do allow_new_phones :∈ {true, false} od
       ]|
```

PhD= ||[**attr** $phonedir := \emptyset$
obj $entry$; $names[1]$; $\cdots$; $names[n]$; $\cdots$
meth $Give\_name(x, n) = \cdots$
proc $Update(n) = \cdots$
do
$super.get\_m.get\_allow\_new\_phones \rightarrow$ $entry := new(PayPhone)$;
$Update(entry)$;
$entry.Where(self)$;
$entry.Update(phonedir)$
[] $\forall i \in \{1..phones\}$:$names[i].Update(phonedir)$
od
]| .

Observe that because the attribute *phonedir* and procedure *Update* are not referenced in the class *Mgt* they can be kept local to the class *PhD*

By Rule 4 we obtain that the above OO-action system is refined by the OO-action system

$$\{\langle PhoneCmp, PC\rangle^*, \langle PhoneCmp2, PC2'\rangle, \langle Management, Mgt\rangle,$$
$$\langle PhoneDept, PhD\rangle, \langle Phone, Ph\rangle, \langle PayPhone, PPh\rangle\}.$$

We now have that the phone network is completely distributed as the phone classes have only local attributes. The phone companies, however, share the attribute *phones*. Hence, the system is not truly distributed. However, within the pay phone company, the two departments can work in parallel and independently of each other. Further refinement steps could be used to separate the company numbers (in a similar manner as the *phonedir* is distributed). Alternatively, if we want the companies to be able to share the numbers in a distributed manner we could generate a higher level organization, a government, to maintain the *phone* attribute. Moreover, the pay phone company is not active. Refinements to solve this problem are carried out elsewhere [10].

CONCLUDING REMARKS

An OO-action system is a description of a set of classes. At run time this set has an interpretation as a network of concurrently executing, active, distributed objects. Moreover, we can have parallelism in the system both within an object and between objects depending on the view we take on distribution.

One way of looking at the two approaches to distributed objects as described in the introductory section within our framework is that in initial stages of designing OO-action systems we can take view (2): given a class that give rise to objects that refer to many attributes and contain many parallel active actions, we can split it into set of classes where every class gives rise to objects that basically refer to attributes local only to the corresponding objects. Hence, during the refinement steps, classes that communicate via shared attributes are split into classes which communicate via method calls only. The refinement concerning the phone company followed this pattern: the original class body $PC2$ contained parallel activity while the split class bodies *Mgt* and *PhD* in the refined system localize this activity to distributed objects.

Distributed objects typically move around [25]. A straightforward way to introduce *mobile* objects into the OO-action systems formalism is to add an extra *location* attribute into every class in the style of Mobile UNITY [20] and mobile process calculi [3, 22]. This would denote the current location of the generated object. Moreover, the behaviors of the objects could be dependent on the value of the location attributes. This direction is left for future research.

Acknowledgments

We want to thank Wolfgang Weck for clarifying discussions concerning different views of distributed objects. The work of Kaisa Sere is supported by the Academy of Finland.

References

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

[2] M. Abadi and K.R.M. Leino. A logic of object-oriented programs. In *Theory and Practice of Software Development, TAPSOFT'97*, LNCS 1214, Springer-Verlag, 1997.

[3] R.M. Amadio. An asynchronous model of locality, failure, and process mobility. In *Coordination Languages and Models*, LNCS 1282, pages 374–391, Springer-Verlag, 1997.

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

[5] R.J.R. Back. Procedural abstraction in the refinement calculus. Technical report A-55, Department of Computer Science, Åbo Akademi Univeristy, Turku, Finland, 1987.

[6] R.J.R. Back. Refinement calculus, part II: parallel and reactive programs. In *Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*, LNCS 430, pages 67–93, Springer-Verlag, 1990.

[7] R.J.R. Back, M. Büchi, and E. Sekerinski. Action-based concurrence and synchronization for objects. In *Proc. 4th AMAST Workshop on Real-Time Systems, Concurrent, and Distributed Software*, Springer-Verlag, 1997.

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

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

[10] M.M. Bonsangue, J.N. Kok, and K. Sere. An approach to object-orientation in action systems. In *Proc. 4th Mathematics of Program Construction*, LNCS 1422, pages 68–95, Springer-Verlag, 1998.

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

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

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

[14] U. Hensel, M. Huisman, B. Jacobs, and H. Tews. Reasoning about classes in object-oriented languages: logical models and tools. In *European Symposium on Programming*, LNCS 1381, pages 105-121, Springer-Verlag, 1998.

[15] C. Hewitt. Viewing control structures as patterns of passing messages. *Artificial Intelligence* 8(3), 1997.

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

[17] C.B. Jones. A π-calculus semantics for an object-based design notation. In *Proc. of CONCUR'93*, LNCS 715, Springer-Verlag, 1993.

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

[19] K. Lano and H. Haughton. Reasoning and refinement in object-oriented specification languages. In *European Conference on Object-Oriented Programming'92*, LNCS 615, Springer-Verlag, 1992.

[20] P.J. McCann and G.-C. Roman. Compositional programming abstractions for mobile computing. *IEEE Transactions on Software Engineering 24* 2, pages 97–110, 1998.

[21] A. Mikhajlova and E. Sekerinski. Class refinement and interface refinement in object-oriented programs. In *Proc. 4th Formal Methods Europe Symposium*, LNCS 1313, Springer-Verlag, 1997.

[22] R. Milner, J. Parrow, and D.J. Walker. A calculus of mobile processes. *Information and Computation 100* 1, pages 1–40, 1992.

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

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

[25] R. Orfali, D. Harkey, and J. Edwards. *Distributed Objects: Survival Guide.* John Wiley and Sons, Inc., 1996.

[26] K. Sere and M. Waldén. Data refinement of remote procedures. In *Proc. of the Symposium on Theoretical Aspects of Computer Software*, LNCS 1281, pages 267–294, Springer-Verlag, 1997.

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