Formal Techniques for
Parallel Object-Oriented Languages

Pierre America
Philips Research
P.O. Box 80.000
5600 JA Eindhoven
The Netherlands

Abstract

This paper is intended to give an overview of the formal techniques that have been developed to deal with the parallel object-oriented language POOL and several related languages. We sketch a number of semantic descriptions, using several formalism: operational semantics, denotational semantics, and a new approach to semantics, which we call layered semantics. Then we summarize the progress that has been made in formal proof systems to verify the correctness of parallel object-oriented programs. Finally we survey the techniques that we are currently developing to describe the behaviour of objects independently of their implementation, leading to linguistic support for behavioural subtyping.

1 Introduction

Over the last few years, object-oriented programming has gained widespread use and considerable popularity. Until now, the use of object-oriented techniques for parallel programming is mainly restricted to research environments, but nevertheless it holds considerable promises to contribute to the solutions of many problems associated with the programming of parallel computers.

At the Philips Research Laboratories in Eindhoven, the Netherlands, several research projects in this area have been carried out. The DOOM project (Decentralized Object-Oriented Machine) was a subproject of ESPRIT project 415: 'Parallel Architectures and Languages for Advanced Information Processing — a VLSI-directed Approach' [AHO+90]. This ESPRIT project aimed at improving the performance of computers in the area of symbolic applications by the use of large-scale parallelism. Several approaches were explored in different subprojects, which were tied together at a disciplinary level by working groups [Bak89, Tre90]. The DOOM subproject had chosen an object-oriented approach [AH90]. This subproject developed a parallel object-oriented programming language POOL in which applications can be written, together with a parallel machine architecture suitable to execute programs in this language. A number of example applications have been developed as well.

Another project, PRISMA (PaRallel Inference and Storage MAchine), built on the same object-oriented principles as the DOOM project. It aimed at developing a system that is able to handle very large amounts of knowledge and data, again using parallelism to reach a high performance. One of the concrete results of this project is a prototype relational database machine which can automatically exploit parallelism in evaluating the users' queries [AHH+90]. Together with the DOOM project, a prototype computer, POOMA (Parallel Object-Oriented Machine Architecture), has been built, on which the software developed in these projects is running. It comprises 100 processor nodes, each with its own local memory, and connected by a high-speed packet switching network.

This paper was written in the context of ESPRIT Basic Research Action 3020: Integration.
The language POOL (Parallel Object-Oriented Language) used in these projects has been the subject of extensive theoretical studies. In the present paper, we give a survey of the results of these studies, and we shall try to assess their influence on the design and the use of the language. In fact, the name POOL stands for a family of languages, developed over a period of seven years. The most important one is POOL2 [Ame88], the language that was implemented on the POOMA machine. Whenever the exact member of the POOL family does not matter, we shall just use the name POOL.

Section 2 gives an introduction to the language POOL itself. Section 3 gives an overview of the techniques that have been used to describe the semantics of POOL in a formal way. Then section 4 sketches the research that has been done in the area of formal verification of POOL programs. Finally, in section 5 we describe the typically object-oriented phenomena of inheritance and subtyping, and show how formal techniques can help to clarify many of the issues involved.

2 An overview of the language POOL

This section gives a summary of the most important ingredients of the POOL language. For more details, we refer the reader to [Ame89b], or to the official language definition [Ame88].

In POOL, a system is described as a collection of objects. An object can be thought of as a kind of box, containing some data and having the ability to perform some actions on these data. An object uses variables to store its data. A variable contains a reference to another object (or, possibly, to the object containing the variable itself). The object’s ability to perform operations on its internal data lies in two mechanisms: First, an object can have a set of methods, a kind of procedures, which can access and change the values of the variables. (Up to this point, the mechanisms that we have described are generally present in object-oriented languages.) Second, an object has a so-called body, a local process that can execute in parallel with the bodies of all the other objects in the system. This is specific for POOL; it constitutes the main source of parallelism in POOL programs.

A very important principle in object-oriented programming is encapsulation: The variables of one object are not directly accessible to other objects. In fact, the only way for objects to interact is by sending messages. A message is a request to the receiving object to execute one of its methods. The sending object explicitly mentions the receiver and the method name. It can also pass some parameters (again references to objects) to the method. The sender blocks until the receiver has answered its message. The receiver also explicitly states when it is prepared to answer a message. However, it does not specify the sender but only lists a set of possible method names. As soon as synchronization between sender and receiver takes place, the receiver executes the required method, using the parameters that the sender gave. The method returns a result (once again, a reference to an object), which is then passed back to the sender. After that, sender and receiver both continue their own processing in parallel.

Because of the above mechanisms, the only parallelism in the system is caused by the parallel execution of the bodies of the different objects. Inside each object everything happens sequentially and deterministically, so that the object is protected from the parallel and nondeterministic (and therefore ‘dangerous’) outside world. The interesting thing in POOL is that, like in other object-oriented languages, new objects can be created dynamically in arbitrary high numbers. In POOL, where as soon as an object is created, its body starts executing, this means that also the degree of parallelism can be increased dynamically. (Objects are never destroyed explicitly; rather, useless objects are removed by a garbage collector working behind the screens).

In order to describe these dynamically evolving systems of objects in a static program, the objects are grouped into classes. All the objects in one class (the instances of the class) have the same names and types for their variables (of course, each has its own private set of variables) and they execute the same methods and body. In a program, a class definition is used to describe this internal structure of the objects. Whenever a new object is to be created, a class is named which serves as a blueprint.
3 Semantics

A number of different techniques have been used to describe the semantics of POOL in a formal way. The following subsections sketch several of these approaches. In all of these descriptions, a syntactically simplified version of POOL is used. This is more convenient in the semantic definition but not very well readable in concrete programs. There is a straightforward translation from POOL2 or POOL-T (an older version) to this simplified notation.

3.1 Operational Semantics

The simplest semantic technique is the use of transition systems to define an operational semantics. This technique has been introduced by Hennessy and Plotkin [HP79, Plo81, Plo83]. It describes the behaviour of a system in terms of sequences of transitions between configurations. A configuration describes the system at one particular moment during the execution. Apart from a component describing the values of the variables, it typically contains as a component that part of the program that is still to be executed. The possible transitions are described by a transition relation, a binary relation between configurations (by having a relation instead of a function, it is possible to model nondeterminism). This transition relation is defined by a number of axioms and rules. Because of the presence of (the rest of) the program itself in the configurations, it is possible to describe the transition relation in a way that is closely related to the syntactic structure of the language.

The term 'operational' can now be understood as follows: The set of configurations defines a (very abstract) model of a machine, and the transition relation describes how this machine operates: each transition corresponds to an action that the machine can perform. The fact that the semantic description follows the syntactic structure of the language so closely (as we shall see below) is a definite advantage of the transition system approach to operational semantics.

In the operational semantics of POOL [ABKR86] uses configurations having four components:

\[ \text{Conf} = \mathcal{P}_{\text{Fin}}(\text{LStat}) \times \Sigma \times \text{Type} \times \text{Unit} \]

The first component is a finite set of labelled statements:

\[ \{(\alpha_1, s_1), \ldots, (\alpha_n, s_n)\} \]

Here each \( \alpha_i \) is an object name and the corresponding \( s_i \) is the statement (or sequence of statements) that the object is about to execute. This models the fact that the objects \( \alpha_1, \ldots, \alpha_n \) are executing in parallel. The second component is a state \( \sigma \in \Sigma \), which records the values of the instance variables and temporary variables of all the objects in the system. The third component is a typing function \( \tau \in \text{Type} \), assigning to each object name the class of which the object is an instance. Finally, the last component is the complete POOL program or unit, which is used for looking up the declarations of methods (whenever a message is sent) and bodies (when new objects are created).

The transition relation \( \rightarrow \) between configurations is defined by axioms and rules. In general, an axiom describes the essential operation of a certain kind of statement or expression in the language. For example, the axiom describing the assignment statement looks as follows:

\[ \langle X \cup \{\alpha, x := \beta\}, \sigma, \tau, U \rangle \rightarrow \langle X \cup \{\alpha, \beta\}, \sigma[\beta/\alpha, x], \tau, U \rangle \]

Here, \( X \) is a set of labelled statements which are not active in this transition, \( \beta \) is another object name, a special case of the expression that can in general appear at the right-hand side of an assignment, and \( \sigma[\beta/\alpha, x] \) denotes the state that results from changing in the state \( \sigma \) the value of the variable \( x \) of the object \( \alpha \) into the object name \( \beta \).

Rules are generally used to describe how to evaluate the components of a composite statement or expression. For example, the following rule describes how the (general) expression at the right-hand side of an assignment is to be evaluated:

\[ \langle X \cup \{\alpha, x := e\}, \sigma, \tau, U \rangle \rightarrow \langle X' \cup \{\alpha, e'\}, \sigma', \tau', U \rangle \]

where \( \langle X \cup \{\alpha, x := e\}, \sigma, \tau, U \rangle \rightarrow \langle X' \cup \{\alpha, x := e'\}, \sigma', \tau', U \rangle \)
According to this rule, if the transition above the line is a member of the transition relation, then so is the transition below the line. In this way the rule reduces the problem of evaluating the expression in an assignment to evaluating the expression on its own. The latter is described by specific axioms and rules dealing with the several kinds of expressions in the language. Note that as soon as the right-hand side expression has been evaluated completely, so that an concrete object name β results, the assignment axiom above applies and the assignment proper can be performed.

The semantics of a whole program can now be defined as the set of all maximal sequences of configurations \( (c_1, c_2, c_3, \ldots) \) that satisfy \( c_i \rightarrow c_{i+1} \). Each of these sequences represents a possible execution of the program.

### 3.2 Denotational semantics

The second form of semantic description that has been used to describe POOL is denotational semantics. Whereas operational semantics uses an abstract machine that can perform certain actions, denotational semantics assigns a mathematical value, a ‘meaning’, to each individual language construct. Here, the most important issue is compositionality: the meaning of a composite construct can be described in terms of only the meanings of its syntactic constituents.

For sequential languages, it is very natural that the value associated with a statement is a function from states to states: when applied to the state before the execution of the statement, this function delivers the state after the execution. However, for parallel languages this is no longer appropriate: Not only does the presence of nondeterminism lead to a set of possible final states, in addition, information on the intermediate states is required to be able to compose a statement in parallel with other statements. This leads us to the concept of resumptions (introduced by Plotkin [Plo76]). Instead of delivering the final state after the execution of the statement has completed, we divide the execution of the statement into its atomic (indivisible) parts, and we deliver a pair \( (\sigma', r) \), where \( \sigma' \) is the state after the execution of the first atomic action and \( r \) is the resumption, which describes the execution from this point on. In this way, it is possible to put another statement in parallel with this one: the execution of the second statement can be interleaved with the original one in such a way that between each pair of subsequent atomic actions of the first statement an arbitrary number of atomic actions of the second one can be executed. Each atomic action can inspect the state at the beginning of its execution and possibly modify it.

For a very simple language (not yet having the power of POOL) we get the following equation for the set \( P \) (the domain) in which the values reside that we want to assign to our statements:

\[
P \cong \{p_0\} \cup \left( \Sigma \rightarrow \mathcal{P}(\Sigma \times P) \right).
\]

The intended interpretation of this equation is the following: Let us call the elements of the set \( P \) processes and denote them with letters \( p, q, \) and \( r \). Then a process \( p \) can either be the terminated process \( p_0 \), which cannot perform any action, or it is a function which, when provided with an input state \( \sigma \), delivers a set \( X \) of possible actions. Each element of this set \( X \) is a pair \( (\sigma', q) \), where \( \sigma' \) is the state after this action, and \( q \) is a process that describes the rest of the execution.

It is clear that equation (1) cannot be solved in the framework of sets, because the cardinality of the right-hand side would always be larger than that of the left-hand side. In contrast to many other workers in the field of denotational semantics of parallelism, who use the framework of complete partial orders (cpo's) to solve this kind of equations (see, e.g., [Plo76]), we have chosen to use the framework of complete metric spaces. (Readers unfamiliar with this part of mathematics are referred to standard topology texts like [Dug66, Eng89] or to [BZ82].) The most important reason for this choice is the possibility to use Banach's fixed point theorem:

Let \( M \) be a complete metric space with distance function \( d \) and let \( f : M \rightarrow M \) be a function that is contracting, i.e., there is a real number \( \epsilon \) with \( 0 < \epsilon < 1 \) such that for all \( x, y \in M \) we have \( d(f(x), f(y)) < \epsilon.d(x, y) \). Then \( f \) has a unique fixed point.
This ensures that whenever we can establish the contractivity of a function we have a **unique** fixed point, whereas in cpo theory mostly we can only guarantee the existence of a **least** fixed point.

Another reason for using complete metric spaces is the naturalness of the power domain construction. Whereas in cpo theory there are several competing definitions (see, e.g., [Plo76, Smy78]) all of which are somewhat hard to understand, in complete metric spaces there is a very natural definition:

If $M$ is a metric space with distance $d$, then we define $\mathcal{P}(M)$ to be the set of all **closed** subsets of $M$, provided with the so-called **Hausdorff distance** $d_H$, which is defined as follows:

$$d_H(X, Y) = \max\left\{\sup_{x \in X} d(x, Y), \sup_{y \in Y} d(y, X)\right\}$$

where $d(\hat{x}, Z) = \inf_{z \in Z} d(x, z)$ (with the convention that $\sup \emptyset = 0$ and $\inf \emptyset = 1$).

(Minor variations on this definition are sometimes useful, such as taking only the nonempty subsets of $M$ or only the compact ones. The metric is the same in all cases.)

The domain equation that we use for the denotational semantics of POOL (see [ABKR89]) is somewhat more complicated than equation (1), because it also has to accommodate for communication among objects. For POOL, the domain $P$ of processes is defined as follows:

$$P \cong \{p_0\} \cup \left(\Sigma \to \mathcal{P}(\text{Step}_P)\right)$$

where the set $\text{Step}_P$ of **steps** is given by

$$\text{Step}_P = (\Sigma \times P) \cup \text{Send}_P \cup \text{Answer}_P,$$

with

$$\text{Send}_P = \text{Obj} \times \text{MName} \times \text{Obj}^* \times (\text{Obj} \to P) \times P$$

and

$$\text{Answer}_P = \text{Obj} \times \text{MName} \times (\text{Obj}^* \to (\text{Obj} \to P) \to P).$$

The interpretation of these equations (actually, they can be merged into one large equation) is as follows: As in the first example, a process can either terminate directly, or it can take one out of a set of steps, where this set depends on the state. But in addition to internal steps, which are represented by giving the new state plus a resumption, we now also have communication steps. A send step gives the destination object, the method name, a sequence of parameters, and two resumptions. The first one, the **dependent** resumption, is a function from object names to processes. It describes what should happen after the message has been answered and the result has been returned to the sender. To do that, this function should be applied to the name of the result object, so that it delivers a process that describes the processing of that result in the sending object. The other resumption, also called the independent resumption, describes the actions that can take place in parallel with the sending and the processing of the message. These actions do not have to wait until the message has been answered by the destination object. (Note that for a single object the independent resumption will always be $p_0$, because a sending object cannot do anything before the result has arrived. However, for the correct parallel composition of more objects, the independent resumption is necessary to describe the actions of the objects that are not sending messages.) Finally we have an answer step: This consists of the name of the destination object and the method name, plus an even more complicated resumption. This resumption takes as input the sequence of parameters in the message plus the dependent resumption of the sender. Then it returns a process describing the further execution of the receiver and the sender together.

Equations like (1) can be solved by a technique explained in [BZ82]: An increasing sequence of metric spaces is constructed, its union is taken and then the metric completion of the union space satisfies the equation. The equation for POOL processes cannot be solved in this way, because the domain variable $P$ occurs at the left-hand side of the arrow in the definition of answer steps. A more general, category-theoretic technique for solving this kind of domain equations has been developed to solve this problem. It is described in [AR89]. Let us only remark here that it is necessary to restrict ourselves to the set of non-distance-increasing functions (satisfying $d(f(x), f(y)) \leq d(x, y)$), which is denoted by $\to_1$ in the above equation.
Let us now give more details about the semantics of statements and expressions. These are described by the following two functions:

\[
\begin{align*}
&[\ldots]_S : \text{Stat} \rightarrow \text{Env} \rightarrow A\text{Obj} \rightarrow \text{Cont}_S \rightarrow P \\
&[\ldots]_E : \text{Exp} \rightarrow \text{Env} \rightarrow A\text{Obj} \rightarrow \text{Cont}_E \rightarrow P.
\end{align*}
\]

The first argument of each of these function is a statement (from the set \text{Stat}) or an expression (from \text{Exp}), respectively. The second argument is an \text{environment}, which contains the necessary semantic information about the declarations of methods and bodies in the program (for more details, see [ABKR89]). The third argument is the name of the (active) object executing the statement/expression. The last argument is a \text{continuation}. This is explained in more detail below. Continuations are necessary to describe the sequential composition of statements that can create processes and for dealing with expressions that can have arbitrary side-effects (such as sending messages).

Continuations work as follows: The semantic function for statements is provided with a continuation, which is just a process \((\text{Cont}_S = P)\), describing the execution of all the statements following the current one. The semantic function then delivers a process that describes the execution of the current statement plus the following ones. Analogously, the semantic function for expressions is fed with a continuation, which in this case is a function which maps object names to processes \((\text{Cont}_E = \text{Obj} \rightarrow P)\). This function, when applied to the name of the object that is the result of the expression, gives a process describing everything that should happen in the current object after the expression evaluation. Again, the semantic function delivers a process describing the expression evaluation plus the following actions.

Let us illustrate this by giving some examples of clauses that appear in the definition of the semantic functions \([\ldots]_S \text{ and } [\ldots]_E\). Let us start with a relatively simple example, the assignment statement:

\[
[x := e]_S(\gamma)(\alpha)(p) = [e]_E(\gamma)(\alpha)(\lambda \beta. \lambda \sigma \{ (\sigma', p) \}).
\]

This equation says that if the statement \(x := e\) is to be executed in an environment \(\gamma\) (recording the effect of the declarations), by the object \(\alpha\), and with continuation \(p\) (describing the actions to be performed after this assignment), then first the expression \(e\) is to be evaluated, with the same environment \(\gamma\) and by the same object \(\alpha\), but its resulting object is to be fed into an expression continuation \(\lambda \beta. \lambda \sigma \{ (\sigma', p) \}\) that delivers a process of which the first action is an internal one leading to the new state \(\sigma'\) and having the original continuation \(p\) as its resumption. Here, of course, the new state \(\sigma'\) is equal to \(\sigma(\beta/\alpha, x)\).

The semantic definition of sequential composition is easy with continuations:

\[
[s_1; s_2]_S(\gamma)(\alpha)(p) = [s_1]_S(\gamma)(\alpha)([s_2]_S(\gamma)(\alpha)(p)).
\]

Here the process describing the execution of the second statement \(s_2\) just serves as the continuation for the first statement \(s_1\).

As a simple example of a semantic definition of an expression let us take an instance variable:

\[
[x]_E(\gamma)(\alpha)(f) = \lambda \sigma . \{(\sigma, f(\sigma(\alpha)(x)))\}.
\]

Evaluating the expression \(x\) takes a single step, in which the value \(\sigma(\alpha)(x)\) of the variable is looked up in the state \(\sigma\). The resumption of this first step is obtained by feeding this value into the expression continuation \(f\) (which is a function that maps object names into processes).

As a final example of a semantic definition, let us take object creation: The expression \text{new}(C)\) creates a new object of class \(C\) and its value is the name of this object. Its semantics is defined as follows:

\[
[\text{new}(C)]_E(\gamma)(\alpha)(f) = \lambda \sigma . \{(\sigma', \gamma(C)(\beta) \parallel f(\beta))\}.
\]

Here \(\beta\) is a fresh object name, determined from \(\sigma\) in a way that does not really interest us here, and \(\sigma'\) differs from \(\sigma\) only in that the variables of the new object \(\beta\) are properly initialized. We see that
execution of this new-expression takes a single step, of which the resumption consists of the parallel composition of the body $\gamma(C)(\beta)$ of the new object with the execution of the creator, where the latter is obtained by applying the expression continuation $f$ to the name of the new object $\beta$ (which is, after all, the value of the new-expression). The parallel composition operator $\parallel$ is a function in $P \times P \rightarrow P$, which can be defined as the unique fixed point of a suitable contracting higher-order function $\Phi_{PC} : (P \times P \rightarrow P) \rightarrow (P \times P \rightarrow P)$ (an application of Banach's fixed point theorem).

From the above few equations it can already be seen how the use of continuations provides an elegant solution to the problems that we have mentioned.

There are a number of further steps necessary before we arrive at the semantics of a complete program. One interesting detail is that in the denotational semantics, sending messages to standard objects is treated in exactly the same way as sending messages to programmer-defined objects. The standard objects themselves (note that there are infinitely many of them!) are represented by a (huge) process $p_{ST}$, which is able to answer all the messages sent to standard objects and immediately return the correct results. This process $p_{ST}$ is composed in parallel with the process $p_U$, which describes the execution of the user-defined objects in order to give the process describing the execution of the whole system.

Despite the fact that the two forms of semantics described above, the operational and the denotational one, are formulated in widely different frameworks, it turns out that it is possible to establish an important relationship between them: $O = abstr \circ D$, which in some sense says that the different forms of semantics of POOL are equivalent. Here $D$ is the function that assigns a process to a POOL program according to the denotational semantics and $O$ assigns to each program a set of (finite or infinite) sequences of states, which can be extracted from the sequences of configurations obtained from the operational semantics. Finally, $abstr$ is an abstraction operator that takes a process and maps it into the set of sequences of states to which the process gives rise. The complete equivalence proof can be found in [Rut90].

3.3 Layered semantics

The denotational semantics described above has one significant disadvantage: it does not describe the behaviour of a single object. In order to get a better grip on this fundamental concept of object-oriented programming, a layered form of denotational semantics for POOL has been developed [AR90]. In this formalism, there are different semantic domains (again complete metric spaces) for the meanings of statements, objects, and systems.

The semantic domain $SProc$ of statement processes is used for the semantic descriptions of POOL statements and expressions. It reflects precisely the different ways in which statements and expressions can interact with their environment:

$$SProc \cong \{p_0\} \cup (\Sigma \times SProc) \cup (CName \times (Obj \rightarrow SProc)) \cup (Obj \times MName \times Obj^* \times (Obj \rightarrow SProc)) \cup (MName \xrightarrow{\mathcal{F}} (Obj^* \rightarrow SProc)) \times SProc \cup (Obj \times SProc)$$

(With $A \xrightarrow{\mathcal{F}} B$ we denote the set of finite partial maps from $A$ to $B$.)

A statement (or expression) can do nothing, it can do an internal step, it can create a new object, it can send a message, it can answer a message, it can conditionally answer a message, or it can send the result of a message back to the original sender. In all these cases, the exact mechanism to perform the task (e.g., determining the name for a new object or synchronizing on communication) is not described here, but only the information necessary to perform it (e.g., the class of the new
object). We should also note that a state $\sigma \in \Sigma$ only describes the variables of a single object. Semantic functions like the ones in section 3.2 deliver for each statement or expression the statement process that describes its meaning.

The domain $OProc$ of object processes is very similar to $SProc$, but it does not include the internal steps:

$$OProc \equiv \{q_0\} \cup (CName \times (Obj \rightarrow OProc))$$
$$\cup (Obj \times MName \times Obj^* \times (Obj \rightarrow OProc))$$
$$\cup (MName \rightarrow (Obj^* \rightarrow OProc))$$
$$\cup (MName \rightarrow (Obj \rightarrow OProc)) \times OProc$$
$$\cup (Obj \times OProc)$$

The semantics of an object is obtained by applying an abstraction operator $\text{abstr} : SProc \rightarrow OProc$ to the meaning of the body of this object. This operator $\text{abstr}$ removes all the internal steps. It turns out that this operator is not continuous (in the classical metric/topological meaning), since it has to transform an arbitrarily long sequence of internal steps followed by a non-internal step into this non-internal step, but an infinite sequence of internal steps by $q_0$. For the mathematical treatment, this non-continuity presents no problem.

Finally, the domain $GProc$ of global processes is determined by the following domain equation:

$$GProc = \{r_0\} \cup \mathcal{P}(GStep)$$
$$GStep = Obj \times MName \times Obj^* \times Obj \times GProc$$
$$\cup Obj \times (Obj \rightarrow GProc)$$
$$\cup MName \times Obj \times (Obj \rightarrow Obj^* \rightarrow GProc)$$
$$\cup Obj \times Obj \times Obj \times GProc$$
$$\cup \text{Comm}^+ \times GProc$$

where

$$\text{Comm}^+ = \text{Comm} \cup \{\ast\}$$
$$\text{Comm} = Obj \times MName^+ \times Obj^* \times Obj$$
$$MName^+ = MName \cup \{\ast\}$$

The possible steps in a global process are either one-sided communication attempts (sending or receiving a message or result) or completed communications, reflected by a communication record $c \in \text{Comm}$.

An object process in $OProc$ is ‘globalized’, i.e., transformed into a global process, by an operator $\omega$. This operator $\omega$ also takes care of the naming of new objects and it remembers to which object the result of a method should be returned. There is a parallel composition operator $\mid\mid : GProc \times GProc \rightarrow GProc$, which takes care of synchronization between sender and receiver of a message or result. The global process describing the whole execution of a program can be obtained by applying $\omega$ to the object process that belongs to the root object (the first object in the system).

If desired, the set of traces of all successful communications in program executions can be obtained from this global process. However, it is much more interesting to determine the real observable input/output behaviour of a program. This is done by distinguishing a special object $\text{world}$ to which input and output messages can be sent. By concentrating on the interactions of the rest of the system with this object, one can view a program as a nondeterministic transformation from an input stream of standard objects (integers, booleans) to an output stream of standard objects.

The advantage of this layered approach to POOL semantics is not only that it really gives a semantic interpretation of a single object, but also that it develops a framework in which issues about full abstractness can be studied (a denotational semantics is called fully abstract if it does not give more information than necessary to determine the observable behaviour of a program construct in context). These issues are present on two levels: the statement/expression level and the object level. They are subject of ongoing research.

In addition to the different forms of semantics described above, POOL has been the subject of a number of other semantic studies. In [Vaa86], the semantics of POOL is defined by means of
process algebra [BK84, BK85]. In [ELR90], a simplified version of POOL is used as an example to illustrate POTs (Parallel Object-based Transition systems), an extended version of Petri nets. Finally, in [DD86, DDH87] a description is given of an abstract POOL machine. In contrast to the ‘abstract machine’ employed in the operational semantics described above, this abstract POOL machine is intended to be the first step in a sequence of refinements which ultimately lead to an efficient implementation on real parallel hardware. This abstract POOL machine is described formally in AADL, an Axiomatic Architecture Description Language.

4 Verification

Developing a formal proof system for verifying the correctness of POOL programs is an even harder task than giving a formal semantics for this language. Therefore this work has been done in several stages. A more detailed account of the research on verification of POOL programs is given in [Boe91].

4.1 A sequential version

First the proof theory of SPOOL, a sequential version of POOL, has been studied (see [Ame86]). This language is obtained by omitting the bodies (and the possibility to return a result before a method ends) from POOL, such that now at any moment there is only one active object and we have a sequential object-oriented language. For this language a Hoare-style [Apt81, Hoa69] proof system has been developed. The main contribution from the proof theory of SPOOL was a formalism to deal with dynamically evolving pointer structures. This reasoning should take place at an abstraction level that is at least as high as that of the programming language. More concretely, this means the following:

1. The only operations on ‘pointers’ (references to objects) are
   - testing for equality
   - dereferencing (determining the value of an instance variable of the referenced object)

2. In a given state of the system, it is only possible to reason about the objects that exist in that state, i.e., an object that does not exist (yet) cannot play a role.

Requirement 1 can be met by only admitting the indicated operations to the assertion language (however, this excludes the approach where pointers are explicitly modelled as indices in a large array that represents the 'heap'). In order to satisfy requirement 2, variables are forbidden to refer to nonexisting objects and the range of quantifiers is restricted to the existing objects. (The consequence is that the range of quantification depends on the state!)

In reasoning about pointer structures, first-order predicate logic is not enough to express interesting properties, and therefore several extensions have been explored. The first extension is the possibility to use recursively defined predicates. One variant of this formalism uses a so-called \( \mu \)-notation to express these predicates. More precisely, the phrase

\[
\mu X(z_1, \ldots, z_n)(P)
\]

is used to denote the smallest predicate \( X \) satisfying

\[
\forall z_1, \ldots, z_n \quad X(z_1, \ldots, z_n) \leftrightarrow P.
\]

(In order to be sure that such a predicate exists, we require that \( X \) must not occur negatively in \( P \).)

In this notation, the assertion

\[
\mu X(y, z) \left( y \equiv z \lor X(y \cdot x, z) \right)(v, w)
\]

(2)
can be used to express the property that the object denoted by the variable \( w \) is a member of the linked list starting with \( v \) and linked by the variable \( z \), or in other words, that starting from \( v \) one can get to \( w \) by following the reference stored in the variable \( z \) an arbitrary number of times.

Another notational variant for using recursively defined predicates is to name and declare them explicitly. The declaration

\[
q(x_1, \ldots, x_n) \leftarrow P
\]

defines \( q \) to be the smallest predicate satisfying

\[
\forall x_1, \ldots, x_n \quad q(x_1, \ldots, x_n) \leftarrow P.
\]

The property expressed in equation (2) can now simply be written as \( q(v, w) \), where the predicate \( q \) is declared by

\[
q(y, z) \leftarrow \ y \triangleq z \lor q(y, x, z).
\]

The \( \mu \)-notation is easier to deal with in reasoning about the proof system, but the version with declared predicates is more convenient in actual proofs.

Since an assertion language with recursively defined predicates does not admit the standard techniques for establishing the completeness of the proof system, we have also explored a different extension to first-order logic: quantification over finite sequences of objects. In this formalism, the property in equation (2) can be expressed as follows:

\[
\exists z \left( z \cdot 1 \triangleq v \land z \cdot |z| \triangleq w \land \forall n \left( 0 < n \land n < |z| \right) \rightarrow (z \cdot n).x \triangleq z \cdot (n + 1) \right)
\]

where \( z \) ranges over finite sequences of objects, \( z \cdot n \) denotes the \( n \)th element of the sequence \( z \), and \( |z| \) denotes its length.

It is somewhat surprising that even with the restrictions on pointer operations, mentioned above as items 1 and 2, it is possible to describe, e.g., the creation of a new object. This is done by an axiom that is similar in form to the traditional axiom of assignment:

\[
\{P[new/u]\} \ u \leftarrow new \ \{P\}.
\]

The trick is in the definition of the substitution operation \([\text{new/u}]\), which is not ordinary substitution, but fulfills the same goal: replacing any expression or assertion by another one that, when evaluated in the state before the assignment, has the same value as the original one in the state after the assignment. In the case of object creation, this is not possible for every expression, because the variable \( u \) will refer to the new object after the statement and this object cannot be denoted by any expression before the statement. However, this variable \( u \) can only in two contexts in an assertion: either it is compared for equality with another expression, or it is dereferenced. In both cases we know what the result will be. The precise definition of the substitution \([\text{new/u}]\) is somewhat complicated, so here we just give a few examples (see Ame86 for full details):

\[
\begin{align*}
\ u \triangleq y.z[new/u] &= \ false \\
\ u \triangleq u[new/u] &= \ true \\
\ u.z \triangleq z[new/u] &= \ \text{nil} \triangleq z \\
\ u.z \triangleq u[new/u] &= \ false
\end{align*}
\]

In the case of quantification, the substituted assertion contains one component that ranges over the old objects and one component that talks about the new one, for example:

\[
(\forall z \ P)[\text{new/u}] = (\forall z \ (P[new/u])) \land P[u/z][\text{new/u}].
\]

In the presence of recursively defined predicates or quantification over finite sequences, similar measures are taken to ensure correct functioning of the substitution operation, so that the axiom for object creation can be shown to be sound, i.e., everything that can be proved is actually true.
If the left-hand side of the assignment is an instance variable instead of a temporary variable, then certain other modifications to the substitution used in the assignment axiom are necessary to deal with aliasing, i.e., the possibility that two different expressions denote the same variable. For example, applying the substitution \([l/z]\) to the assertion

\[
\forall z (z \neq \text{self} \to z \cdot x \neq 0)
\]

yields the assertion

\[
\forall z (z \neq \text{self} \to \text{if } z = \text{self} \text{ then } 1 \text{ else } z \cdot x \text{ fi } \neq 0),
\]

which can be simplified to \(\forall z (z \neq \text{self} \to z \cdot x = 0)\).

Another contribution of the SPOOL proof system is a proof rule for message passing and method invocation (in a sequential setting). In this rule the context switching between sending and receiving object and the transmission of parameters and result are representing by appropriate substitution operations. For details, see [Ame86].

In the version with quantification over finite sequences, the SPOOL proof system has been proved to be not only sound but also complete [AB89], i.e., every correctness formula that is true can be proved in this system.

### 4.2 Dealing with parallelism

Along a different track a proof theory was developed to deal with parallelism, in particular with dynamic process creation. In [Boe86] a proof system was given for a language called P, which essentially only differs from POOL in that message passing only consists of transmitting a single value from the sender to the receiver (like in CSP [Hoa78]).

Whereas the proof system in [Boe86] uses an explicit coding of object references by numbers, an integration with the work on SPOOL has led to a more abstract proof system for the same language P [AB88, AB90].

To deal with parallelism, this proof system uses the concepts of cooperation test, global invariant, bracketed section, and auxiliary variables, which have been developed in the proof theory of CSP [AFR80]. Described very briefly, the proof system for the language P consists of the following elements:

- **A local stage.** This deals with all statements that do not involve communication or object creation. These statements are proved correct with respect to pre- and postconditions in the usual manner of sequential programs [Apt81, Hoa69]. At this stage, assumptions are used to describe the behaviour of the communication and creation statements. These will be verified in the next stage.

- **An intermediate stage.** In this stage the above assumptions about communication and creation statements are verified. For each creation statement and for each pair of possibly communicating send and receive statements it is verified that the specification used in the local proof system is consistent with the global behaviour.

- **A global stage.** Here some properties of the system as a whole can be derived from a kind of standard specification that arises from the intermediate stage.

In the local stage, a local assertion language is employed, which only talks about the current object in isolation. In this assertion language, the variables of the current object can be named directly, but the variables of other objects cannot be named at all. Likewise, quantification over integers (and booleans) is possible, but quantification over all (existing) objects is not available. In the intermediate and global stages, a global assertion language is used, which reasons about all the objects in the system. Here quantification over all existing objects and even over finite sequences of existing objects is possible. An assertion \(p\) in the local assertion language can be transformed to
a global assertion $p \downarrow z$ by applying a syntactic substitution (denoted by $\downarrow z$, where $z$ is a global expression that denotes the object that $p$ should talk about). For example, if the local assertion $p$ is

$$b \neq \text{nil} \rightarrow \forall i \ (i < m \rightarrow \exists j \ i \times j \models b)$$

then $p \downarrow z$ is

$$z \cdot b \neq \text{nil} \rightarrow \forall i \ (i < z \cdot m \rightarrow \exists j \ i \times j \models z \cdot b).$$

Whereas in the local stage the axioms and rules are the same as in traditional Hoare logic, the axioms and rules of the intermediate stage make use of the same techniques as the ones developed for SPOOL to deal with object creation and aliasing.

The global proof system makes use of the fact that the initial state is precisely known: there is one single object and all its variables have the value nil. Therefore the precondition can have the form $(p \land x_1 \models \text{nil} \land \cdots \land x_n \models \text{nil}) \downarrow z$, where $p$ is an assertion talking only about logical variables (variables that do not occur in the program).

Again this proof system has been proved to be sound and complete (for details see [AB88]). Moreover, using the same basic ingredients, but with the addition of an assumption/commitment mechanism, a sound and complete proof system has been developed for the full language POOL, with its rendezvous communication [Boe90, Boe91]. Nevertheless, this proof system also has its shortcomings. The most important problem is that it involves global reasoning: the invariant incorporates all the objects in the system. For even slightly complicated programs, this leads to an unmanageably complex proof. The only way out here seems to be the use of compositional proof techniques (see [HR87] for a survey). Unfortunately, since the configuration of objects in POOL is not static but dynamic, it is not so clear how the decomposition should work here.

5 Inheritance and subtyping

Inheritance is a mechanism that is tightly bound to the notion of object-oriented programming. Its basic idea is that in defining a new class it is often very convenient to start with all the ingredients (variables, methods) of an existing class and to add some more and possibly redefine some in order to get the desired new class. The new class is said to inherit the variables and methods of the old one. This can repeated several times and one can even allow a class to inherit from more than one class (multiple inheritance). In this way a complete inheritance hierarchy arises. By sharing code among classes in this way, the total amount of code in a system can sometimes be drastically reduced and its maintenance can be simplified.

This inheritance relationship between classes also suggests another relationship: If a class $B$ inherits from a class $A$, each instance of $B$ will have at least all the variables and methods that instances of $A$ have. Therefore it seems that whenever we require an object of class $A$, an instance of class $B$ would do equally well, or in other words, that we can regard the class $B$ as a specialized version of $A$. Note, however, that inheritance is concerned with the internal structure of the objects (variables and methods), whereas this specialization phenomenon is about the possible use of the objects, characterized by their externally observable behaviour. For a long time the idea has prevailed in the object-oriented community that the inheritance hierarchy coincides completely with the specialization hierarchy. However, recently it is becoming clear that identifying these two hierarchies leads to several problems and that it is useful to separate them (see also [Ame87, Sny86]).

In order to get a clear conceptual view, we proposed the following definitions [Ame89a]: Whereas a class is a collection of objects that have (exactly) the same internal structure (variables, method code, body), a type is a collection of objects that share the same externally observable behaviour. In this context we can see that inheritance forms a relationship between classes, while subtyping is a relationship between types. A language called POOL-I has been designed that works according to these principles [AL90]. Now inheritance is adequately described by the syntax for writing code in the programming language under consideration (which does not mean that it is a simple task to
design an inheritance mechanism that works well in practice), but in order to get a good formal grip on subtyping we need to do some more work.

Let us start with realizing that a type is in essence a specification of object behaviour. The important point here is that the specification should only consider those aspects of this behaviour that can be observed from outside the object, independently of its internal structure. Ideally, the specification should only talk about the messages that are received and sent by the object. Now specifying objects in terms of possible sequences of messages is certainly possible, but for most kinds of objects, this is not the best way to characterize them. For example, consider the following specification, which is written in English to avoid notational problems:

The object will accept put and get messages, but it will accept a get message only if the number of put messages already received exceeds the number of get messages. A put message contains one integer as an argument and does not return a result. A get message contains no arguments and it returns as its result the integer that was the argument of the last put message that has preceded an equal number of put and get messages.

It is not immediately clear that this specification characterizes something as simple and well-known as a stack of integers. The most important reason for this difficulty is that the intuitive view most people have of such an object is that inside it stores a certain amount of information (a sequence of integers) and that the messages interact with this information. A more technical disadvantage is that reasoning in terms of sequences of message fails to make explicit that different sequences may lead to the same end result: a stack into which three elements have been inserted and subsequently removed is equivalent to an empty stack.

Therefore we propose to specify object behaviour in terms of an abstract state, which is an abstraction from the object's concrete internal state. For our stack, the abstract state would just be a sequence of integers. In a sequential setting, methods can now be described by pre- and postconditions formulated in terms of the abstract state. For example, the method get of the above stack is specified by

\[ \{s \neq \emptyset\} \text{get}() \{s_0 = s \ast (r)\}. \]

Here \(s\) is the sequence representing the current state of the stack, \(s_0\) in the postcondition stands for the value of \(s\) before the method execution, and \(r\) stands for the result of the get method. Furthermore, the operator \(\ast\) denotes concatenation of sequences, \(\emptyset\) is the empty sequence, and \(\langle n \rangle\) is the sequence having \(n\) as its only element.

In general, a specification of a type \(\sigma\) consists of a domain \(\Sigma\), representing the set of possible abstract states of objects of type \(\sigma\), plus a set of method specifications of the form \(\{P\}m(\vec{p})\{Q\}\), where the precondition \(P = P(s, \vec{p})\) describes the state of affairs before the method execution and the postcondition \(Q = Q(s, s_0, \vec{p}, r)\) describes the situation after its execution (\(s\) always stands for the current abstract state, \(s_0\) for the abstract state before the method execution, \(\vec{p}\) for the method parameters, and \(r\) for the result). The meaning of such a method specification is that each object of type \(\sigma\) should have a method with name \(m\) available such that if the method is executed in a state where the precondition \(P\) holds, then after the method execution \(Q\) holds.

The next important question is under what conditions the objects of a given class \(C\) are members of a type \(\sigma\), in which case we say that the class \(C\) implements the type \(\sigma\). We do this as follows: We require a representation function \(f : C \rightarrow \Sigma\), where \(C\) is the set of possible concrete states of objects of class \(C\), i.e., the set of possible values of the variables \(\vec{v}\) of such an object, and \(\Sigma\) is the set of abstract states associated with the type \(\sigma\). The representation function \(f\) maps the values \(\vec{v}\) of the variables of an object of class \(C\) to an element \(s\) of the mathematical domain \(\Sigma\) that is used in the specification of the type \(\sigma\). We also need a representation invariant \(I\), which is a logical formula involving the values of the variables of the class \(C\). This invariant will describe the set of values of these variables that can actually occur (in general this is a proper subset of the set \(C\)). The representation function \(f\) should at least be defined for all concrete states for which the invariant \(I\) holds.
For the class \( C \) to implement the type \( \sigma \) the following conditions are required to hold:

1. The invariant \( I \) holds initially, i.e., just after the creation and initialization of each new object.

2. Every method \( m \) of the class \( C \) (the ones that are mentioned in \( \sigma \)'s specification as well as the ones that are not mentioned) should maintain the invariant.

3. For every method specification \( \{ P \} m(p) \{ Q \} \) occurring in the specification of \( \sigma \), the class \( C \) should also have a method \( m \) with parameters \( \bar{p} \) of the right number and types. Furthermore this method should satisfy

\[
\{ P \circ f \land I \} m(\bar{p}) \{ Q \circ f \land I \}.
\]

Here \( P \circ f \) stands for the formula \( P \) where every occurrence of the abstract state \( s \) is replaced by the function \( f \) applied to the variables and analogously with \( s_0 \): \( P \circ f = P[f(\bar{u})/s,f(\bar{u}_0)/s_0] \).

Now we can say that a type \( \sigma \) is a subtype of a type \( \tau \) if all objects belonging to \( \sigma \) also belong to \( \tau \) (note that nontrivial subtyping relationships are indeed possible, because one can write specifications that leave some degree of freedom in behaviour so that other specifications can be more specific).

Of course, we would like to be able to conclude such a subtyping relationship from the specifications of \( \sigma \) and \( \tau \). At a first glance, it seems sufficient to require that for every method specification \( \{ P \} m(p) \{ Q \} \) occurring in \( \tau \)'s specification there should be a method specification \( \{ P' \} m(\bar{p}) \{ Q' \} \) in the specification of \( \sigma \) such that the latter implies the former, which can be expressed by \( P \rightarrow P' \) and \( Q' \rightarrow Q \). Under these circumstances we can indeed use any element of \( \sigma \) whenever an element of \( \tau \) is expected: When we send such an object a message listing the method \( m \), using it as an element of \( \tau \) guarantees that initially the precondition \( P \) will hold. By the implication \( P \rightarrow P' \) we can conclude that the precondition \( P' \) in \( \sigma \)'s specification also holds. Then after the method execution the postcondition \( Q' \) from \( \sigma \) will hold and this again implies the postcondition \( Q \) that is required by \( \tau \).

However, in general we must assume that the type \( \tau \) has been specified using a different mathematical domain \( T \) than the domain \( \Sigma \) used in \( \sigma \)'s specification. Therefore in order to show that a type \( \sigma \) is a subtype of the type \( \tau \), we require the existence of a function \( \phi : \Sigma \rightarrow T \), called transfer function, that maps the mathematical domain \( \Sigma \) associated with \( \sigma \) to \( T \), the one associated with \( \tau \). This time we do not need an extra invariant, because we can assume that \( \Sigma \) has been chosen small enough to exclude all the values that cannot actually occur. We now require that for every method specification \( \{ P \} m(p) \{ Q \} \) occurring in \( \tau \)'s specification there should be a method specification \( \{ P' \} m(\bar{p}) \{ Q' \} \) in the specification of \( \sigma \) such that

1. \( P \circ \phi \rightarrow P' \).
2. \( Q' \rightarrow Q \circ \phi \).

Again \( P \circ \phi \) can be obtained from \( P \) by replacing the abstract state of \( \tau \) by \( \phi \) applied to the abstract state of \( \sigma \) and analogously for the old values of the abstract states.

On the basis of the above definitions one can easily prove the desirable property that whenever a class \( C \) implements a type \( \sigma \) and \( \sigma \) is a subtype of \( \tau \), then \( C \) implements \( \tau \).

In order to make these ideas more formal, a language, called SPOOL-S, has been defined that allow the formal expression of the abovementioned ingredients: classes and types, connected by representation and transfer functions [DK91]. The abstract domains, in which the abstract states reside, can be specified in any formalism in which mathematical entities can be specified (in [DK91] the Larch Shared Language [GHM90] was chosen as an example). Method specifications can be written using a formalism that is slightly different from the one above. In addition to a pre- and a postcondition, a method specification also contains a so-called modifies-list, in which all the objects are mentioned that could be modified by the method. This is necessary to be able to describe an object on its own, because ordinary Hoare logic does not provide a mechanism to state that 'all the rest stays the same'.
The language also allows for the situation where one abstract object is implemented by a collection of concrete objects. For example, a stack can be implemented by a linked list of objects that each store a single item. This can be expressed in the representation function, which maps the concrete state of an object into its abstract state, possibly using the abstract states of other objects as well. In our example, if the modifies-list allows a method to change a stack object, then all the concrete objects implementing this stack could possibly be modified.

Up to now, a formal proof system for verifying SPOOL-S programs has not yet been developed, but it seems certainly possible with the techniques that have been used for SPOOL (see section 4.1). It is a much greater challenge, however, to generalize them to deal with parallelism. Here the techniques described in section 4.2 definitely fall short, because they involve global reasoning on an abstraction level that is as high as that of the programming language, but not higher. Probably a viable approach to this problem will still include the use of abstract internal states, but in the presence of parallelism it will be hard to avoid reasoning about sequences of messages. Perhaps a judicious combination of compositional proof techniques and the techniques describe above will lead to a satisfactory solution.

References


