An Algebraic Semantics for an Abstract Language with Intra-Object-Concurrency*

Thomas Gehrke

Institut für Informatik, Universität Hildesheim
Postfach 101363, D-31113 Hildesheim, Germany
gehrke@informatik.uni-hildesheim.de

Object-oriented specification and implementation of reactive and distributed systems are of increasing importance in computer science. Therefore, languages like *Java* [1] and *Object REXX* [3,7] have been introduced which integrate object-orientation and concurrency. An important area in programming language research is the definition of semantics, which can be used for verification issues and as a basis for language implementations. In recent years several semantics for concurrent object-oriented languages have been proposed which are based on the concepts of process algebras; see, for example, [2,6,8,9]. In most of these semantics, the notions of processes and objects are identified: Objects are represented by sequential processes which interact via communication actions. Therefore, in each object only one method can be active at a given time. Furthermore, language implementations based on these semantics tend to be inefficient, because the simulation of message passing among local objects by communication actions is more expensive than procedure calls in sequential languages.

In contrast to this identification of objects with processes, languages like *Java* and *Object REXX* allow for intra-object-concurrency by distinguishing *passive* objects and *active* processes (in *Java*, processes are objects of a special class). Objects are data structures containing methods for execution, while system activity is done by processes. Therefore, different processes may perform methods of the same object at the same time. To avoid inconsistencies of data, the languages contain constructs to control intra-object-concurrency.

In this paper, we introduce an abstract language based on *Object REXX* for the description of the behaviour of concurrent object-based systems, i.e. we ignore data and inheritance. The operational semantics of this language is defined by a translation of systems into terms of a process calculus developed previously. This calculus makes use of process creation and sequential composition instead of the more common action prefixing and parallel composition. Due to the translation of method invocation into process calls similar to procedure calls in sequential languages, the defined semantics is appropiate as a basis for implementation.

The language $\mathcal{O}$: An $\mathcal{O}$-system consists of a set of objects and an additional sequence of statements, which describes the initial system behaviour. Each object is identified by a unique object identifier O and must contain at least one

* This work was supported by the DFG grant *EREAS* (Entwurf reaktiver Systeme).

```
object O1              object O2                         object O3
method m1              method m1 guarded                 method m1 guarded
    a; b                   c; reply; d                       a; reply; b
method m2              method m2 guarded
    a; reply; b            e; guard off; reply; f        object O4
method m3                                                method m1 guarded
    a □ reply; b                                             c; d
```

Fig. 1. $\mathcal{O}$ examples.

method. The special object identifier **self** can be used for the access of methods of the calling object (**self** is not allowed in the initial statement sequence). Methods are identified by method identifiers m and must contain at least one statement. Statements include single instructions and nondeterministic choices between sequences: $s_1 \Box s_2$ performs either s_1 or s_2. We distinguish five kinds of instructions: atomic actions a representing the visible actions of systems (e.g. access to common resources), methods calls $O.m$ and the three control instructions **reply**, **guard on** and **guard off**. ; denotes sequential composition; for syntactical convenience we assume that ; has a higher priority than $\Box$ (e.g., $a \Box b; c$ is $a \Box (b; c)$). System runs are represented by sequences $a_1...a_n$ of atomic actions, called *traces*.

During the execution of a called method, the caller is blocked until the called method has terminated. For example, the execution of the sequence $O1.m1; c$ in combination with the object $O1$ in Figure 1 generates the trace $a\,b\,c$. In some cases it is desirable to continue the execution of the caller before the called method has finished (e.g., the execution of the remaining instructions of the called method does not influence the result delivered to the caller). The instruction **reply** allows the caller to continue its execution concurrently to the remaining instructions of the called method. Therefore, the execution of $O1.m2; c$ leads to the traces $a\,b\,c$ and $a\,c\,b$. Furthermore, the execution of $O1.m3; c$ leads to the traces $a\,c$, $b\,c$ and $c\,b$.

Methods of different objects can always be executed in parallel. Concurrency within objects can be controlled by the notion of *guardedness*. If a method m of an object O is guarded, no other method of O is allowed to become active when m is taking place. If m is unguarded, other methods of O can be performed in parallel with m. The option **guarded** declares a method to be guarded. The instructions **guard on** and **guard off** allow to change the guardedness of a method during its execution. Consider the object $O2$ in Figure 1 in which both methods are declared as guarded. Although $O2.m1$ contains a **reply** instruction, the execution of the sequence $O2.m1; O2.m2$ can only generate the trace $c\,d\,e\,f$, because the guardedness of the methods prevents their concurrent execution. On the other hand the sequence $O2.m2; O2.m1$ leads to the traces $e\,f\,c\,d$ and $e\,c\,d\,f$

$$\frac{}{\mathbf{1} \xrightarrow{\delta} \mathbf{1}}\,\mathrm{T}_1 \qquad \frac{}{spawn(t) \xrightarrow{\delta} spawn(t)}\,\mathrm{T}_2 \qquad \frac{t \xrightarrow{\delta} t \quad u \xrightarrow{\delta} u}{(t;u) \xrightarrow{\delta} (t;u)}\,\mathrm{T}_3 \qquad \frac{t \xrightarrow{\delta} t}{(a:t) \xrightarrow{\delta} (a:t)}\,\mathrm{T}_4$$

$$\frac{}{\alpha \xrightarrow{\alpha} \mathbf{1}}\,\mathrm{R}_1 \qquad \frac{}{n \xrightarrow{\tau} \Theta(n)}\,\mathrm{R}_2 \qquad \frac{t \xrightarrow{\alpha} t'}{spawn(t) \xrightarrow{\alpha} spawn(t')}\,\mathrm{R}_3$$

$$\frac{t \xrightarrow{a\dagger} t' \quad a \neq b}{(b:t) \xrightarrow{a\dagger} (b:t')}\,\mathrm{R}_4 \qquad \frac{t \xrightarrow{\omega} t'}{t+u \xrightarrow{\omega} t'}\,\mathrm{R}_5 \qquad \frac{u \xrightarrow{\omega} u'}{t+u \xrightarrow{\omega} u'}\,\mathrm{R}_6 \qquad \frac{t \xrightarrow{\alpha} t'}{t;u \xrightarrow{\alpha} t';u}\,\mathrm{R}_7$$

$$\frac{t \xrightarrow{\delta} t' \quad u \xrightarrow{\alpha} u'}{t;u \xrightarrow{\alpha} t';u'}\,\mathrm{R}_8 \qquad \frac{t \xrightarrow{\delta} t' \quad t' \xrightarrow{\alpha} t'' \quad u \xrightarrow{\alpha'} u' \quad \{\alpha,\alpha'\} = \{a, a?\}}{t;u \xrightarrow{\tau} t'';u'}\,\mathrm{R}_9$$

Fig. 2. Transition rules.

(the trace $e\,c\,f\,d$ is prevented by the guardedness of $O2.m1$). Guardedness does not play a role in the initial statement sequence.

As a third example, consider the sequence $O3.m1; O4.m1$. This sequence leads to the possible traces $a\,b\,c\,d$, $a\,c\,b\,d$ and $a\,c\,d\,b$, because the called methods belong to different objects.

The process calculus $\mathcal{P}$: To model the behaviour of $\mathcal{O}$-systems, we introduce a process calculus $\mathcal{P}$, which is a slightly modified version of a calculus studied in [5]. We assume a countable set $\mathcal{C}$ of action names, ranged over by a, b, c. A name a can be used either for input, denoted $a?$, or for output, denoted with only the name a itself. We sometimes use $a\dagger$ as a "meta-notation" denoting either $a?$ or a. The set of actions is denoted $\mathcal{A} = \{a\dagger \mid a \in \mathcal{C}\} \cup \{\tau\}$, ranged over by α, β. τ is a special action to indicate internal behaviour of a process. $\Omega = \mathcal{A} \cup \{\delta\}$, ranged over by ω, is the set of transition labels, where δ signals successful termination. $\mathcal{N}$, ranged over by n, n', is a set of names of processes. The calculus $\mathcal{P}$, ranged over by t, u, v, is defined through the following grammar:

$$t ::= \mathbf{0} \mid \mathbf{1} \mid \alpha \mid n \mid t;t \mid spawn(t) \mid t+t \mid (a:t)$$

$\mathbf{0}$ denotes the inactive process, $\mathbf{1}$ denotes a successfully terminated term. A process name n is interpreted by a function $\Theta : \mathcal{N} \to \mathcal{P}$, called *process environment*, where n denotes a process call of $\Theta(n)$. $t;u$ denotes the sequential composition of t and u, i.e. u can perform actions when t has terminated. $spawn(t)$ creates a new process which performs t concurrently to the spawning process: $spawn(t);u$ represents the concurrent execution of t and u. The choice operator $t+u$ performs either t or u. $(a:t)$ restricts the execution of t to the actions in $\mathcal{A} \setminus \{a, a?\}$. The semantics of $\mathcal{P}$ is given by the transition rules in Figure 2.

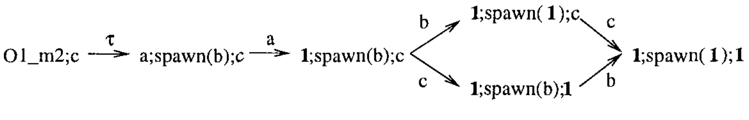
Translation: An object system is translated into a process environment Θ and a term representing the initial statement sequence. The set $\mathcal{N}$ of process names is defined as $\mathcal{N} = \{O\_m \mid O \text{ object}, m \text{ method of } O\} \cup \{O\_mutex \mid O \text{ object}\}$. For each method m of an object O, we include a process name O_m into the

$$\begin{array}{ll}
O1\_m1 & \mapsto a; b \\
O1\_m2 & \mapsto a; spawn(b) \\
O1\_m3 & \mapsto \tau; a + \tau; spawn(b) \\
\\
O2\_m1 & \mapsto o2\_l; c; spawn(d; o2\_u) \\
O2\_m2 & \mapsto o2\_l; e; o2\_u; spawn(o2\_l; f; o2\_u) \\
O2\_mutex & \mapsto o2\_l?; o2\_u?; O2\_mutex \\
\\
O3\_m1 & \mapsto o3\_l; a; spawn(b; o3\_u) \\
O3\_mutex & \mapsto o3\_l?; o3\_u?; O3\_mutex \\
\\
O4\_m1 & \mapsto o4\_l; c; d; o4\_u \\
O4\_mutex & \mapsto o4\_l?; o4\_u?; O4\_mutex
\end{array}$$

Fig. 3. Process semantics of the objects in Figure 1.

set $\mathcal{N}$; the process term $\Theta(O\_m)$ models the behaviour of the body of m. For example, the method $m1$ of object $O1$ in Figure 1 is translated into $O1\_m1 \mapsto a; b$ (see Figure 3). The additional O_mutex processes are used for synchronizing methods. Method calls are translated into process calls of the corresponding process definitions. Therefore, the statement sequence $O1.m1; c$ is translated into $O1\_m1; c$, which is able to perform the following transitions: $O1\_m1; c \xrightarrow{\tau} a; b; c \xrightarrow{a} \mathbf{1}; b; c \xrightarrow{b} \mathbf{1}; \mathbf{1}; c \xrightarrow{c} \mathbf{1}; \mathbf{1}; \mathbf{1}$. Note that the sequence of the transition labels without τ corresponds to the trace for $O1.m1; c$.

The translation of the **reply**-instruction is realized by enclosing the remaining instructions of the method in a *spawn*-operator. For example, the method $m2$ of $O1$ is translated into the process $O1\_m2 \mapsto a; spawn(b)$. The sequence $O1.m2; c$ is translated into $O1\_m2; c$, which leads to the following transition system:


$$\begin{tikzcd}
& & & \mathbf{1};spawn(\mathbf{1});c \arrow[dr, "c"] & \\
O1\_m2;c \arrow[r, "\tau"] & a;spawn(b);c \arrow[r, "a"] & \mathbf{1};spawn(b);c \arrow[ur, "b"] \arrow[dr, "c"'] & & \mathbf{1};spawn(\mathbf{1});\mathbf{1} \\
& & & \mathbf{1};spawn(b);\mathbf{1} \arrow[ur, "b"'] &
\end{tikzcd}$$

Choice operators $s_1 \Box s_2$ are translated into terms $\tau; t_1 + \tau; t_2$ where t_1, t_2 are the translations of s_1 and s_2. The initial τ-actions simulate the internal nondeterminism of the $\Box$-operator. If an instruction sequence contains subterms of the form $(s_1 \Box s_2); s_3$, we have to distribute sequential composition over choice, i.e. $(s_1 \Box s_2); s_3$ has to be transformed into $s_1; s_3 \Box s_2; s_3$ before the translation into the process calculus can be applied. This transformation is necessary for correct translation of the **reply**-instruction. For example, the sequence $(a; b \Box \textbf{reply}; c); d$ is transformed into $a; b; d \Box \textbf{reply}; c; d$, and then translated into $\tau; a; b; d + \tau; spawn(c; d)$.

Guardedness is implemented by mutual exclusion with semaphores. In the absence of data, we have to simulate semaphores by communication. For each object O using guardedness, we introduce a special semaphore process $O\_mutex \mapsto o\_l?; o\_u?; O\_mutex$. Communication on channel o_l means locking the semaphore, communication on o_u the corresponding release (unlock). To ensure that only one guarded method of an object can be active at the same time, methods of objects using guardedness have to interact with the corresponding semaphore process. For example, consider the translation of object $O2$ in Figure 3. In order to integrate unguarded actions into the synchronization mechanism, we have to enclose every unguarded action by lock and unlock actions. Therefore, in $O2.m2$ the instruction f is translated into $o2\_l; f; o2\_u$. Without the synchronization actions, unguarded actions could be performed although a guarded method has locked the semaphore. In objects without guardedness, the insertion of lock and unlock actions is not necessary and therefore omitted (see Figure 3). To enforce communication over the l and u channels, we have to restrict these actions to the translation of the initial instruction sequence. Furthermore, the semaphore process has to be spawned initially. Therefore, the initial statement sequence $O2.m1; O2.m2$ is translated into $(\{o2\_l, o2\_u\} : spawn(O2\_mutex); O2\_m1; O2\_m2)$. The semaphore process is spawned initially and runs concurrently to the method calls. $o2\_l$ and $o2\_u$ are restricted, therefore they can only be performed in communications between $O2\_m1$, $O2\_m2$ and the semaphore process. It is easy to see that the only possible trace is $c\,d\,e\,f$ (with τ-actions omitted).

In the full paper [4], the translation of $\mathcal{O}$-systems is defined via translation functions. Furthermore, *weak bisimulation* is used as a equivalence relation on object systems.

References

1. K. Arnold and J. Gosling. *The Java Programming Language.* Addison-Wesley, 1996.
2. P. Di Blasio and K. Fisher. A Calculus for Concurrent Objects. In *Proceedings of CONCUR '96*, LNCS 1119. Springer, 1996.
3. T. Ender. *Object-Oriented Programming with REXX.* John Wiley and Sons, Inc., 1997.
4. T. Gehrke. An Algebraic Semantics for an Abstract Language with Intra-Object-Concurrency. Technical Report HIB 7/98, Institut für Informatik, Universität Hildesheim, May 1998.
5. T. Gehrke and A. Rensink. Process Creation and Full Sequential Composition in a Name-Passing Calculus. In *Proceedings of EXPRESS '97*, vol. 7 of *Electronic Notes in Theoretical Computer Science.* Elsevier, 1997.
6. K. Honda and M. Tokoro. An Object Calculus for Asynchronous Communication. In *Proceedings of ECOOP '91*, LNCS 512. Springer, 1991.
7. C. Michel. Getting Started with Object REXX. In *Proceedings of the SHARE Technical Conference*, March 1996.
8. E. Najm and J.-B. Stefani. Object-Based Concurrency: A Process Calculus Analysis. In *Proceedings of TAPSOFT '91 (vol. 1)*, LNCS 493. Springer, 1991.
9. D. Walker. Objects in the π-Calculus. *Information and Computation*, 116:253–271, 1995.