Atomic actions

Gérard Boudol

UNITÉ DE RECHERCHE
INRIA-SOPHIA ANTIPOLIS

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél.:(1) 39 63 55 11

Rapports de Recherche

N° 1026

Programme 1

ATOMIC ACTIONS

Gérard BOUDOL

Mai 1989

RR-1026

Atomic Actions

Actions Atomiques

Gérard Boudol
INRIA Sophia-Antipolis
06560-VALBONNE FRANCE

Abstract.
We give a formal specification of the semantics of atomic actions. We show that adding atomic action constructs to a low-level imperative language allows one to program higher-level synchronization mechanisms.

Résumé.
Nous présentons un formalisme pour décrire la sémantique des actions atomiques. Nous montrons que, partant d'un langage impératif très simple, on peut dériver des mécanismes de synchronisation de haut niveau en utilisant la notion d'action atomique.

Atomic Actions

(Note)

Gérard Boudol
INRIA Sophia-Antipolis
06560-VALBONNE FRANCE

Abstract.

We give a formal specification of the semantics of atomic actions. We show that adding atomic action constructs to a low-level imperative language allows one to program higher-level synchronization mechanisms.

1. Introduction.

This note intends to provide an alternative view on the so-called "action refinement" concept, which is by now widely studied (*cf.* for instance [4], and the references therein). Action refinement is formalized as the replacement of an action by a possibly complex process. Many recent papers on this topic advocate the idea that a sensible notion of process should be robust with respect to action refinement. For lack of an abstract mathematical notion, a process is understood as an equivalence class, and the problem is to find equivalences which are congruences with respect to the operation of substituting processes for actions.

It is not my intention to question the theoretical work that has been done in this direction. However, one may wonder whether this substitution operation really represents a programming concept. It is often argued that action refinement would help in the "top-down design" of programs. There one should have the ability to develop programs in several steps, each stage of development involving different levels of abstraction. In particular, one would have the ability to regard an action as atomic at an abstract level, even if it has to be implemented as a complex process at a more concrete level. To support this methodology, sequential programming languages usually provide features like procedure or data type definitions, and the semantics of these definitional mechanisms is by means of substitution.

Action refinement could be introduced as a construct def action $a = p$ in q (*cf.* [5]), but we claim that the "correct" interpretation is not by means of the substitution $[p/a]q$. More precisely, the idea which will be developed in this note is that one should not substitute a process for an action *at the wrong place*. We shall return to this point later, but we can note that this substitution operation does not preserve the notion of level of abtraction, since $[p/a]q$ may involve actions of various levels, without any syntactical difference between them.

It may be helpful for our discussion to briefly examine the concepts of *action* and *atomicity* from the programmers point of view. These notions have been introduced more than ten years ago

in the areas of databases and distributed systems. For further references, the reader can consult [3] for instance; in this paper we find the following definition of atomic action: "*to other activities, an atomic action is like a primitive operation which transforms the state of a system without having any intermediate states*". One usually distinguishes two important properties of atomic actions:

recoverability: an atomic action is an "all-or-nothing" activity. It should either complete or leave the state of the system unchanged. This corresponds to the idea that there is no "visible" intermediate state for the process performing the action.

non-interference: an atomic action should not interfere with any other one – there is no visible intermediate state for "other activities". Therefore the result of executing such actions concurrently should be the same as if they were executed sequentially.

We shall see that it is worth to formalize separately these two properties – more precisely we shall introduce two kinds of atomic actions, the first one representing recoverable actions, the second one involving both recoverability and non-interference.

There is another point which should be noted in the previous definition: an atomic action is an *operation* transforming the *state* of a system. Then to model atomic actions we have to formalize the idea of a process operating on data – typically the state of a system may be thought of as a memory, that is an assignment of values to variables(†). This is easily done using a transition relation like

$$\mathsf{state} \xmapsto{\mathsf{process}} \mathsf{state}'$$

The rules for specifying this part of the operational semantics will be such that a process operates by means of its *terminated* sequences of computations. Then an operation must complete to induce a state change.

Once formalized the notion of operation of processes on data, the next question is: what makes an operation *atomic*? We saw that a characteristic property is: an atom acts in an all-or-nothing manner, as if it were a primitive instruction. Then atomicity is reflected in another part of the semantics of processes, namely in the "fetch" transition system. Let us see this point in more detail: for any notion of program, we have an idea of what is the *next instruction* to execute (not necessarily unique, if concurrency is allowed), and what is the resulting state of the program. Then we can say that an action is atomic if it is fetched in a one-step, indivisible manner. For instance, "actions" like $(a\,;b)$ – sequential composition of a and b – or $(a \parallel b)$ – parallel composition – are not atomic since their execution may be decomposed in several steps. To describe the next-instruction/next-program behaviour we will use what we call the *fetch* transition relation:

$$\mathsf{process} \xrightarrow{\mathsf{action}} \mathsf{process}'$$

Then an action is what is above a fetch arrow ($\longrightarrow$), while a process appears at the left and right hand sides of such an arrow. To deal with refinement of actions we could now give the following rule, similar to the one of renaming in CCS:

$$q \xrightarrow{u} q' \;\Rightarrow\; \mathsf{def\ action}\ a = p\ \mathsf{in}\ q \xrightarrow{[p/a]\,u} \mathsf{def\ action}\ a = p\ \mathsf{in}\ q'$$

One can see that the substitution is performed within the action, not in the process; therefore the action stays atomic for the process performing it. This also shows that there is no specific notion

(†) by the way, we must say that such a "shared-memory" model is often rejected: it is widely recognized that this model is inadequate to support clear and maintainable concurrent programs. This is true for low-level languages, but the atomic action constructs (together with module definitions) may restore a clean way of programming. Moreover, shared memory is well-suited for implementation purposes.

of action: any process can be turned into an action, when it is lifted above the fetch arrow. As a matter of fact, the previous definitional construct is not really needed; this will be handled by standard process definition mechanisms, provided we have constructs converting a given process into an atom. Such constructs (atom p) will obey axioms like

$$(\text{atom } p) \xrightarrow{p} \mathbb{1}$$

Here $\mathbb{1}$ is a terminated process, and one can see that such an axiom asserts that (atom p) is indeed atomic: it performs an action in a one-step, indivisible manner.

Ultimately the operational semantics aims at defining the behaviour of systems made out of programs and data (*cf.* [6]). For such a system (p, s) we have an obvious rule saying that if the next instruction performed by p is u, with next state p', and if this instruction u operates on the state s, transforming it into s', then we can infer that the system may evolve into (p', s'). That is:

$$p \xrightarrow{u} p' \ , \ s \overset{u}{\longmapsto} s' \ \Rightarrow \ (p, s) \rightarrow (p', s')$$

The "fetch" semantics $p \xrightarrow{u} p'$ is purely symbolic: it merely consists in an unfolding of the syntactical structure of the process. On the other hand, the proof system for operation uses this next-instruction/next-program behaviour extensively: proving the operation $s \overset{u}{\longmapsto} s'$ generally requires to fetch in sequence the actions performed by u, up to termination; then u may have a "complex" operation. If u originates from the fetch of an atom, then the process p cannot access to intermediate states of this operation since the transition $p \xrightarrow{u} p'$ is indivisible in this case. The manipulation of data may induce some constraints on the behaviour of the program: if u cannot operate on the state s, then the process p cannot actually perform the action u within the system (p, s). We shall see examples showing that synchronization can be achieved by means of atomic actions, without any primitive notion of communication other than the manipulation of shared variables.

NOTE: this note evolves from previous work presented in [1,2]. There the notion of atomic action was embodied into an abstract data type mechanism. We introduce it here in a simpler framework.

2. A Simple Language.

2.1 Syntax.

To set up the afore-introduced ideas, we use Scott's language for flow diagrams ([7]), extended with parallel composition. We assume given a set A of *primitive instructions*, ranged over by a, $b \ldots$ Typically such instructions may be thought of as assignments of values to variables, but we do not analyze them further. We also assume given a set T of *boolean tests*, closed under negation $\neg t$, and a set X of program variables, ranged over by x, y, $z \ldots$ The syntax of processes is given by the following grammar, where a is a primitive instruction, x a program variable, and t a boolean test:

$$p \ ::= \ \mathbb{1} \mid a \mid x \mid (t \rightarrow p, p) \mid (p \,;\, p) \mid (p \parallel p) \mid \mu x.p$$

The set of processes will be denoted by P. Some comments: $\mathbb{1}$ is an identity process, representing also termination; $(t \rightarrow p, q)$ is the abstract syntax for conditional branching – you may read it as (if t then p else q). As usual, $(p \,;\, q)$ and $(p \parallel q)$ represent sequential and parallel composition and $\mu x.p$ is the fixpoint construct, which could be written (let rec $x = p$ in x); the substitution of p for x in q is denoted $[p/x]q$.

EXAMPLE. The (while t do p) loop – in Scott's notation $(t \star p)$ – is defined by $\mu x.(t \rightarrow (p \,;\, x), \mathbb{1})$; then for instance "busy-waiting" is represented by $(t \star \mathbb{1})$, while an interruptible "test-then-set" operation can be written $((t \star \mathbb{1}) \,;\, a)$. This example will be used again in the following.

In order to give the "fetch" semantics of the language, especially for sequential composition, we need a notion of termination. A process p is said to be *terminated*, in notation $p\dagger$, if $p \equiv \mathbb{1}$ where $\equiv$ is the congruence generated by the equations

$$(p\,;\mathbb{1}) \equiv p \equiv (\mathbb{1}\,;p)$$
$$(p \parallel \mathbb{1}) \equiv p \equiv (\mathbb{1} \parallel p)$$
$$\mu x.\mathbb{1} \equiv \mathbb{1}$$

Note that a parallel composition terminates when both its components terminate.

2.2 Execution of Processes.

Now giving the rules for the fetch transition relation is an easy exercise in structural operational semantics. The first item is an axiom for primitive instructions:

$$\boxed{a \xrightarrow{a} \mathbb{1}}$$

Next we have two axioms for conditional branching: we regard $(t \rightarrow p, q)$ as a guarded sum, the first action consisting in evaluating the test.

$$\boxed{(t \rightarrow p, q) \xrightarrow{t} p \qquad (t \rightarrow p, q) \xrightarrow{\neg t} q}$$

The rules for sequential and parallel composition may be regarded as structural rules, saying where the next instruction is to be fetched:

$$\boxed{\frac{p \xrightarrow{u} p'}{(p\,;q) \xrightarrow{u} (p'\,;q)} \qquad p\dagger\,\frac{q \xrightarrow{v} q'}{(p\,;q) \xrightarrow{v} q'}}$$

$$\boxed{\frac{p \xrightarrow{u} p'}{(p \parallel q) \xrightarrow{u} (p' \parallel q)} \qquad \frac{p \xrightarrow{u} p',\ q \xrightarrow{v} q'}{(p \parallel q) \xrightarrow{(u \parallel v)} (p' \parallel q')} \qquad \frac{q \xrightarrow{v} q'}{(p \parallel q) \xrightarrow{v} (p \parallel q')}}$$

Note that for parallel composition we have the two usual rules of interleaving, but also a rule allowing for cooperation between the actions performed by the concurrent components. Finally we have the standard rule for recursive definitions:

$$\boxed{\frac{[\mu x.p/x]p \xrightarrow{u} p'}{\mu x.p \xrightarrow{u} p'}}$$

One can see from these rules that the syntax of *actions* (what can appear above the arrow) is given by:

$$u \;::=\; a \mid t \mid (u \parallel u)$$

Therefore in this simple language an action is either a primitive instruction, or the evaluation of a test, or the cooperation of two actions.

It is easy to see that the syntactic equality $\equiv$ is a bisimulation with respect to the fetch transition relation, that is:

$$p \equiv q \;\&\; p \xrightarrow{u} p' \;\Rightarrow\; \exists q' \equiv p'.\ q \xrightarrow{u} q'$$

Therefore we can deal with transitions up to $\equiv$, simplifying a term r such that $p \xrightarrow{u} r$.

EXAMPLE (CONTINUED). For the while loop we have (up to $\equiv$):

$$(t \star p) \xrightarrow{t} (p\,;(t \star p)) \qquad \text{and} \qquad (t \star p) \xrightarrow{\neg t} \mathbb{1}$$

Let us see another example: the behaviour of the "test-then-set" term $((t \star \mathbb{1})\,;a)$ is given by:

$$((t \star \mathbb{1})\,;a) \xrightarrow{t} ((t \star \mathbb{1})\,;a) \xrightarrow{t} \cdots \qquad \text{and} \qquad ((t \star \mathbb{1})\,;a) \xrightarrow{\neg t} a \xrightarrow{a} \mathbb{1}$$

Thus if the test t is invariably true the term $((t \star \mathbb{1})\,;a)$ does not terminate.

2.3 Operation of Processes on Data.

Now we turn to the operation of processes on data. Obviously the transition relation $\longmapsto$ depends on the meaning of primitive instructions and tests. Therefore to define the operation we assume given an *interpretation*, that is a transition system $I = (S, \theta)$ where S is the set of *states* (or data), while $\theta \subseteq S \times (A \cup T) \times S$ describes the semantics of the primitives, and satisfies the axiom:

$$(s, \neg\neg t, s') \in \theta \;\Leftrightarrow\; (s, t, s') \in \theta$$

Let us explain our interpretation of tests: here $(s,t,s') \in \theta$ means that the test t is true at s *and* the evaluation of t transforms the state into s' – on the other hand, the test t is false at s if there is no transition $(s,t,s') \in \theta$. Then we allow evaluation of tests to yield side-effects. There is no actual side-effect if $s' = s$, whereas an atomic "test-and-set" could be represented by a test with side-effect.

An interpretation $I = (S, \theta)$ generates an operation of processes on data, that is a transition relation $\underset{I}{\longmapsto} \subseteq S \times P \times S$, by means of the following rules (where for simplicity the subscript I is omitted):

$$(s,c,s') \in \theta \;\frac{}{\;s \xmapsto{c} s'\;} \qquad p\dagger \;\frac{}{\;s \xmapsto{p} s\;} \qquad \frac{p \xrightarrow{u} q \;,\; s \xmapsto{u} s'' \xmapsto{q} s'}{s \xmapsto{p} s'}$$

The second axiom asserts that a terminated process acts as an identity. The third rule looks perhaps more complicated; let us read it: to infer that p operates on s one must fetch an action u of p and chain an operation of u on s to an operation of the resulting process q. Clearly this must be repeated until we reach axioms of the proof system, that is given primitive operations or terminated processes. For instance, to prove an operation of $((a\,;b) \parallel c)$ one must break this process down to an execution sequence like

$$((a\,;b) \parallel c) \xrightarrow{a} (b \parallel c) \xrightarrow{c} b \xrightarrow{b} \mathbb{1}$$

or *abc* or *cab*. Then from an operation of this execution sequence, given by θ, we can decuce an operation of the process, that is:

$$s \underset{\theta}{\xmapsto{a}} \underset{\theta}{\xmapsto{c}} \underset{\theta}{\xmapsto{b}} s' \;\Rightarrow\; s \xmapsto{((a\,;b) \parallel c)} s'$$

(the reader is invited to build a full proof out of the rules). Formally, we have:

PROPOSITION (SEQUENTIALIZATION of OPERATION). *A process p operates on a state s, resulting in s', that is $s \xmapsto{p} s'$, if and only if there exists a sequence $c_1, \ldots, c_n$ of elements of $A \cup T$ and a process q such that:*

$$p \xrightarrow{c_1} \cdots \xrightarrow{c_n} q\dagger \;\&\; s \xmapsto{c_1} \cdots \xmapsto{c_n} s'$$

we can see that $s_0 \xmapsto{P} s_1$, but neither $(P \parallel P)$ nor $P\,;P$ can operate on s_0. Therefore we may regard this process, together with $V = \lceil (t \star \mathbb{1})\,;a \rceil$ as providing the basic operations on a *boolean semaphore*, satisfying the specification:

$$s_0 \xmapsto{P} s_1 \qquad \text{and} \qquad s_1 \xmapsto{V} s_0$$

Now assume given two boolean semaphores, with operations P, V and P', V' respectively. For lack of an explicit naming of data, we formalize this as follows: the set of data is

$$S = \{s_0, s_1\} \times \{s'_0, s'_1\}$$

and we assume that $\{a, b, a', b'\} \subseteq A$ and $\{t, t'\} \subseteq T$. Then a, b, t operate as before, acting as an identity on the second component of the data, that is for instance $(s_1, s') \xmapsto[\theta]{a} (s_0, s')$, and so on. Similarly a', b' and t' do not change the first component of the state. Let us define $S = \langle P\,;V' \rangle$ (for "send") and $R = \langle P'\,;V \rangle$ (for "receive"). We let the reader convince him/herself (*cf.* [1,2]) that S or R alone cannot operate on (s_0, s'_0), since S needs the cooperation of R to complete, whereas

$$(s_0, s'_0) \xmapsto{(S \parallel R)} (s_0, s'_0)$$

Therefore we could say that we have "implemented" a rendez-vous mechanism by means of shared variables, using the atomic action constructs. Some other examples (multiple rendez-vous, broadcasting) may be found in [1,2].

4. Conclusion.

We saw that in our framework there is no real difference between actions and processes. Then "action refinement" may be handled in our language by means of the usual process definition mechanism, that is using for instance (def $x = \langle p \rangle$ In q) or (def $x = \lceil p \rceil$ In q). The semantics of such terms is that of $[\langle p \rangle / x]q$ and $[\lceil p \rceil / x]q$ respectively.

The reader has noted that we put the emphasis on the *operation* of processes, that is on the usual input/output semantics. This contrasts with the bisimulation semantics, widely adopted when dealing with concurrency. Then one may wonder what is the right notion of equivalence to use with respect to our proposal. We could obviously define an extensional equivalence by:

$$p \sim q \;\Leftrightarrow_{\text{def}}\; \forall I\; \forall s.\; s \xmapsto[I]{p} s' \;\Leftrightarrow\; s \xmapsto[I]{q} s'$$

However this seems to be too weak; alternatively we could use a contextual preorder given by:

$$p \sqsubseteq q \;\Leftrightarrow_{\text{def}}\; \forall I\; \forall s\; \forall C.\; (C[p], s) \rightarrow_I^* (\mathbb{1}, s') \;\Rightarrow\; (C[q], s) \rightarrow_I^* (\mathbb{1}, s')$$

(using transitions up to $\equiv$). We leave this question for further investigations. Another interesting problem is the following: could we define a class of interpretations and a notion of *implementation* (in a lower-level language, by means of atomic action constructs), so that we would be able to prove that an implementation is correct with respect to a specification?

REFERENCES

[1] G. BOUDOL, *Communication is an Abstraction*, Actes du Second Colloque C^3 (1987) 45-63, and INRIA Res. Rep. 636.

[2] G. BOUDOL, I. CASTELLANI, *Concurrency and Atomicity*, Theoretical Comput. Sci. 59 (1988) 25-84.

[3] R.H. CAMPBELL, P. JALOTTE, *Atomic Actions in Concurrent Systems*, 5th Intern. Conf. on Distributed Computing Systems (1985) 184-191.

[4] R. van GLABBEEK, U. GOLTZ, *Equivalence Notions for Concurrent Systems and Refinement of Actions*, GMD Res. Rep. 366 (1989).

[5] D.B. LOMET, *Process Structuring, Synchronization, and Recovery using Atomic Actions*, SIGPLAN Notices 12 (1977) 128-137.

[6] G. PLOTKIN, *A Structural Approach to Operational Semantics*, Daimi FN-19, Aarhus University (1981).

[7] D. SCOTT, *The Lattice of Flow Diagrams*, Symposium on Semantics of Algorithmic Languages, Lecture Notes in Mathematics 188 (1971) 311-366.

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique