IMPLEMENTATIONS OF NONDETERMINISTIC PROGRAMS

Pedro Guerreiro

Departamento de Informática
Faculdade de Ciências e Tecnologia
Universidade Nova de Lisboa
2825 Monte da Caparica, Portugal

Abstract
Non-determinism is an important concept for program development. Nevertheless, at some low level of detail, the existing machines work in a deterministic manner. Therefore, it is necessary to know how to suppress the non-determinism of a given program, without disturbing its behaviour. This paper treats that problem formally, within the framework of the relational semantics of non-deterministic programs, and provides the justification of the validity of some usual techniques which are based on an intuitive understanding of the concept.

1. INTRODUCTION

Non-determinism is widely recognized as a valuable tool for program development [Dijkstra 1976]. It appears also as the natural environment for the evolution of systems of parallel processes; sequential implementation of such systems through non-deterministically simulating their distributed behaviour can be an effective programming technique [Guerreiro 1981, 1983b]. Furthermore, the concept of non-determinism itself arises many interesting theoretical questions [Plotkin 1976, Schmidt 1979, Apt and Plotkin 1981, Back 1981, Hennessy 1982, Poigné 1982, etc.].

Nevertheless, if we are interested in obtaining an implementation for a program that describes a non-deterministic situation, sooner or later we are bound to consider, implicitly or explicitly, a deterministic version of it, since, at some level of detail, the existing machines operate in a deterministic fashion. Therefore, the problem arises of determining exactly in what way can the non-determinism be suppressed, or, at least, under which conditions is the deterministic program a legitimate version of the non-deterministic one. More generally, we want to have a means of deciding whether a given program \( P_1 \) can be replaced by another program \( P_2 \), in such a way that it is impossible by looking only at the results of the execution of \( P_2 \) to find out that the executed program was not \( P_1 \) after all.

This is the question we discuss in this paper. It was suggested to us by our work with programming systems of communicating processes. We use a language which resembles Hoare's CSP [Hoare 1978], but incorporates the notion of port and the bidirectional communication as in [Milne and Milner 1979]. A system of processes written in the language can be transformed systematically into a non-deterministic sequential program, by use of a few rules, operating on the syntax of the component processes. In order to obtain an executable version, we suppress the non-determinism in the resulting program, turning it into a regular Pascal program. Although the first transformation is justified semantically [Guerreiro 1981], the second one has been accomplished, up to now, in an informal (but quite reasonable) basis. The reflection that led to this paper has helped us to understand better the underlying theoretical validity of our approach.

The presentation is carried out in the framework of the relational semantics of non-deterministic programs [de Roever 1976, Guerreiro 1980]. In order to make it self-contained, we start with some basic definitions and results. Then we introduce...
a notion of observation of the behaviour of a (non-deterministic) program. Using it, we define an implementation relation for programs such that a program $P_2$ implements another program $P_1$ if by observing $P_2$ we cannot discover that we are not dealing with $P_1$. In fact, we will consider three such relations, each corresponding to a different knowledge about the original programs. One of those relations turns out to be a partial-order, while the remaining two lack the antisymmetry property, which makes them pre-orders only.

Once we have decided that a program $P_2$ is an implementation of a program $P_1$, the next question is the following: is it legitimate to replace $P_1$ by $P_2$ in all possible situations? More precisely: if we substitute $P_1$ by $P_2$ in an arbitrary program $P$ where $P_1$ appears as a component, do we get an implementation of $P$? We will use the first of the mentioned implementation relations, together with the relational semantics of the language of guarded commands [Dijkstra 1975] to show that in that context the answer is yes.

Note: This paper is a shortened version of [Guerreiro 1983a], where the proofs missing here can be found.

2. RELATIONAL SEMANTICS OF NON-DETERMINISTIC PROGRAMS

Usually, a deterministic program is viewed as a representation, or description, in a given language of a certain partial function from a suitable set of states into itself. Putting forward the semantics of the program consists essentially in presenting the associated function, or relevant properties of it, in some, hopefully sufficiently widespread, standard formalism. Generalizing this idea, a non-deterministic program, that is, a program that can produce several outputs for some of its inputs, can be considered to be a representation of a function from the set of states to the set of sets of states, or, alternatively, of a binary relation over that set of states.

The operational meaning of the semantic relation must reflect the input-output behaviour of the program: whenever there exists a possibility that an execution of the program initialized in a state $a$ terminates in a state $b$ the pair $(a,b)$ must appear in the relation. However, special attention must be paid to those cases in which the execution may not terminate. This situation of non-termination must be recorded somehow in the semantics, otherwise it will be impossible to distinguish, for example, the two following programs (written in the language of guarded commands):

\[
\text{r1 :: if true --> skip fi} \quad \text{r2 :: if true --> skip}\]

\[
\text{[]} \quad \text{true --> abort fi}
\]

If we consider terminating computations only, the input-output behaviours of $r_1$ and $r_2$ are the same. However, $r_1$ and $r_2$ definitely should not be regarded as "equivalent". In order to distinguish them two techniques can be used:

1. to exclude from the relation all pairs whose first elements are also starting points of non-terminating computations. This is justified if we are interested only in total correctness semantics, as in this case we regard the possibility of non-termination as "bad" as guaranteed non-termination. Therefore, the loss of information caused by deleting those pairs appears to be of no consequence. Using this technique, the semantic relation of program $r_1$ is the identity relation, whereas that of $r_2$ is the empty relation.

ii. to enlarge the state space with a special element to be used as a "final" state for non-terminating computations. If we choose this alternative, the