DECISION PROBLEMS CONCERNING PARALLEL PROGRAMMING

J.A. Bergstra

Institute of Applied Mathematics and Computer Science
University of Leiden
2300 R.A. Leiden Postbus 9512 Leiden the Netherlands

ABSTRACT. A notion of a correct (= deadlock free) scheduling of several recursive processes using common resources is introduced. The existence of correct schedulings is schown to be undecidable from recursive indices of the relevant processes. Further-more we isolate several cases where the mere existence of a correct scheduling does not imply the existence of a computable (recursive) correct scheduling.

§ 1 INTRODUCTION

A detailed account of (recursive) processes can be found in 1 . To be self-contained, however, we include the basic definitions.

We assume that L_Q, question language and L_A, answer language, are two decidable languages. L^* denotes the collection of finite strings of words in L.

1.1 Definition. The collection of prcess graphs w.r.t. L_Q and L_A is defined as follows:

$$PG(L_Q,L_A) = L_Q^* \longrightarrow L_A$$

If no confusion arises we use PG as an abbreviation. Words (sentenses) in L_Q are seen as questions on which processes give answers in L_A. If P is the name of a process then we denote its initial graph with P_ε. Further we maintain a function ext which associates at any instant to all relevant process name their current graph (= extension). Let $l_q \in L_Q$. Then $P(l_q)$ denotes the answer of P on question l_q. Formally $P(l_q) = l_a = ext(P)(l_q)$. After the answer has been given ext has been changed as follows: $ext(P)_{new} = \lambda\sigma_{\in L_Q^*} \cdot ext(P)(l_q{}^*\sigma)$

In this way P remembers that it has been asked l_q.

1.2 Definition. P is a recursive process if P_ξ is a recursive function, after coding of L_Q and L_A in the natural numbers.

1.3 Notation. $[P]$ is the process with $P_\xi = \phi_p$.
Here ϕ_p is the p-th (partial) recursive function. For this notation and for all other notations and result from recursion theory we refer to [2].

1.4 In 1.5 we will fix L_Q and L_A suitable for the description of parallel programming of sequential processes using common resources. To explain the definition it is useful first to define the cooperation of a scheduling system(s.s.) and processes $P^1 \ldots P^n$. A s.s. for n processes is a proces S with languages L_Q^S and L_A^S as follows:

$$L_Q^S = \{\text{START}\} \cup L_A$$
$$L_A^S = \{1,\ldots,n\} \times L_Q$$

The cooperation of S and $\vec{P}$ works as follows: START triggers S. It answers with $\langle i_1, l_q^1 \rangle$ for some $i_1 \leq n$. Then l_q^1 is given as a question to P^{i_1}. The answer l_a^1 is given as a question to S. The next answer $\langle i_2, l_q^2 \rangle$ of S question l_q^2 for P^{i_2} which in turn gives its answer back to S, and so on.

1.5 Now we define L_Q and L_A. We assume that there are common resources $A_1 \ldots A_k$. L_A contains sublanguage L plus the following reserved words:

words	meaning
request(i) (for $i \leq k$)	P asks acces to resource A_i
BLANK	P made no progress in its last step
release(i)	P releases resource A_i

L_Q consists of reserved words only

granted(i) (for $i \leq k$)	the request for A_i is granted (by S)
go on	evaluate the process one step further

Remark. Note that the s.s. speaks to the cooperating processes in L_Q and that the processes answer in L_A. We have not yet specified in what way the resources are used. With an exception in 2.7 and 3.6. we will assume that there is no communication between the processes and the resources. So the P_i do not change the A_j. P_i has used A_j during the scheduling if A_j has been granted to P_i for some period.

1.6 Definition. S is a correct scheduling for $P^1,\ldots,P^k$ if the cooperation between S and $\vec{P}$ satisfies the following properties:

i) every P^i is infinitely often on turn,

ii) every P^i gives infinitely often an answer not equal to BLANK,

iii) at any stage in the cooperation every resource is granted to at most one of the P^i (which has not yet released it).

iv) every resource is eventually released by all processes to which it has been granted.

We write $ECS(\vec{P})$ for: "there exists a correct scheduling for $\vec{P}$", and $ERCS(\vec{P})$ for: "there exists a recursive correct scheduling for $\vec{P}$".

§ 2 PROBLEMS AND RESULTS

We will define some classes of processes and state problems concerning $ECS(\vec{P})$ and $ERCS(\vec{P})$ for $\vec{P}$ in these classes. The proofs are postponed till § 3.

2.1 Definition. RCP is the collection of processes P over L_Q and L_A which are recursive and satisfy ERCS(P).
Motivation. This is the most general class we are interested in. Clearly if P alone has no correct scheduling it is useless to consider its cooperation with other processes.

2.2 Theorem. It is not possible to decide effectively $ECS(P^1,\ldots,P^n)$ from indices $p_1,\ldots,p_n$ of $P^1\ldots P^n$ RCP.
This theorem shows that if we are going to run $P^1\ldots P^n$ in parallel we must have some strong information on beforehand indicating that this is possible. We will then consider the following general problem for a class K of processes.

GP : Does $ECS(\vec{P})$ imply $ERCS(\vec{P})$ for $\vec{P} \in K$?
The motivation for GP is of course that we are not interested in the existence of a correct scheduling but in het possibility to generate such a scheduling effectively.

2.3 Theorem. In RCP the answer on GP in no.
We denote an initial segment of a cooperation by s. We define the predicate $E^{\vec{P}}(s)$ as follows: $E^{\vec{P}}(s) \Leftrightarrow$ "there exists a correct schedulin for $\vec{P}$ which extends s".

2.4 Definition. EDP is the class of processes in RCP for which $E^{\vec{P}}$ is a decidable predicate.

2.5 Theorem. In EDP the answer on GP is positive.
The proof of 2.3 depends on the possibility of the s.s. to communicate information to the processes during the cooperation.
This is possible by the fact that we consider processes which individualy admit infinitely many correct schedu - lings.
This motivates the following definition:

2.6 Definition. RPU is the class of recursive processes P satisfying the following condition:
A scheduling for P (alone) is correct if and only if it grants every request immediately and every grant is always preceded by the corresponding request.
We will present two variations on RPU :

2.7 Definition.

RPU^I Similar to RPU but there are infinitely many resources $A_0, A_1, \ldots$. (A straigthforward modification of L_Q and L_A is required).

RPU^M Similar to RPU but now the resources $A_1, \ldots, A_k$ can be changed by the processes. More precise they may contain one register for a natural number which can be read and changed by the processes using the resource. In this case we need a modification of L_Q, L_A to include a READ and WRITE statement. The idea is that S executes READ and WRITE instructions for the P^i in the A_j. Important is that if a scheduling for P RPU^{M^j} is to be correct its questions to P are all completely determined by the history of the cooperation.

2.8 Theorem. The answer on GP is:

i) unknown to us in RPU (open problem)

ii) negative in RPU^I

iii) negative in RPU^M

As to i) we note the following without proof. Suppose $\vec{P} \in RPU$ and $ECS(\vec{P}) \wedge \neg ERCS(\vec{P})$ holds then there must be uncountably many correct scheduling for $\vec{P}$.

2.9 Conclusions. There is no uniform method to decide whether or not a collection of computable processes admits a correct scheduling. Moreover, the existence of a correct scheduling on itself is of limited importance as it does not imply, in many cases, the existence of a computable correct scheduling.

Therefore apriori restrictions on the processes must ensure that they admit parallel programming.

One final remark should be made concerning the importance of computable correct scheduling. Suppose we admit scheduling mechanisms which are computable from a random generator $\hat{P}$ which may assume values 0 and 1. Then the following situation exists: There are P^1, P^2 in RCP such that:

i) $ECS(P^1,P^2)$

ii) $\neg ERCS(P^1,P^2)$

iii) for every $1 > \delta > 0$ there is a scheduling mechanism S^δ recursive in $\hat{P}$ such that the probability that the cooperation of S^δ and P^1 and P^2 is a correct scheduling exceeds $1 - \delta$.

The proof is along the lines of 3.3. The point to make, however, is that the above fact suggests a strong argument not to restrict to recursive schedulings (and processes). It should be noted that the complexity of a description of S_δ increases with $1/\delta$.

§ 3 PROOFS

As the ideas involved are rather simple we will give very sketchy proofs using algorithmic descriptions of processes in an antropomorphic style.

3.1 Lemma. There are processes P^1 and P^2 in $RCP \cap RPU$ such that $ECS(P^1,P^2)$ does not hold.

Proof. We can take P^1 and P^2 identical to P, where P works as follows:
It asks acces to A_0. As soon as acces to A_0 has been granted it requests for A_1. (ST). As soon as A_1 has been granted it releases A_0 and immediately afterwards requests for A_0 again. Then if A_0 is granted A_1 is released and immediately afterwards it is requested again. Now the process is in situation (ST) again.
ERCS(P) holds as it is possible to allow P to use alternatively A_0 and A_1. However, $ECS(P^1,P^2)$ is false. Clearly in any correct scheduling of P_1 and P_2, P_1 and P_2 are (after they have been granted acces to A_0 for the first time) both using at least A_0 or A_1. If, however, P^1 wants to proceed it needs both A_0 and A_1 for some time.

3.2 Proof of theorem 2.2
Let P be a recursive function such that for all n $B^n = \lceil P(n) \rceil$ is a process in RCP with the following description:
B^n maintains a counter C which counts the number of turns it has had. As long as the predicate $\exists\, m \leq C\ T(n,n,m)$ is false it answers questions "go on" with the answer 1 (for some $1 \in L$). As soon as $\exists\, m \leq C\ T(n,n,C)$ has become true B^n transforms itself into P as introduced in the previous lemma.
Now we see:

i) If $\neg \exists\ m\ T(n,n,m)$ then $ECS(B^n,B^n)$ (T is Kleene s, T - predicate)
ii) If $\exists m\ T(n,n,m)$ then eventually both copies of B^n behave like P and hence $ECS(B^n,B^n)$ is false (see lemma 3.1).

We may conclude that it is not possible do decide $ECS(\lceil P(n) \rceil, \lceil P(n) \rceil)$ as it would lead to a decision method for the halting problem (i.e. $\exists m\ T(n,n,m)$).

3.3 Proof of theorem 2.3
As in the proof of 2.2 we define a process $R \in RCP$ which reacts on a certain event by transforming it-

self into P. Then ECS(R,P) is true but ERCS(R,P) is false as during all recursive scheduling, R is transformed into P. To obtain this situation we define R in such a way that, at infinitely many places, it allows the s.s. exactly two ways to proceed. In this way information in the form of an infinite 0 - 1 sequence can be transferred from the s.s. to R. Let f be this function and let at any stage t $f(0),\ldots,f(r_t)$ be the intial segment of f that has been transferred to R up to stage t. Before proceeding R tests a predicate SEC on $<f(0),\ldots,f(r_t)>$. We will define SEC below. If SEC holds then from that stage R behaves like P otherwise it gives answers which are used for the mechanism to read f.
For SEC we take a recursive predicate which has the following properties:

i) $\forall f_{REC}\ \exists n\ SEC(<f(0),\ldots f(n-1)>)$

ii) $SEC(\sigma) \rightarrow SEC(\sigma * \tau)$

iii) $\exists f\ \forall n \neg SEC(<f(0),\ldots,f(n-1)>)$

We assume that the mechanism feeding in f can be run in parallel with P. Then S is a correct scheduling if it transfers a function f such that $\forall n \neg SEC(<f(0),\ldots,f(n-1)>)$. Such an f exists but cannot be recursive. If S is recursive then so is the corresponding f. From some stage R will behave like P and a deadlock will occur.

The existence of predicates like SEC is well-known in recursion theory.

3.4 Proof of theorem 2.5

Let $P^1\ldots P^n$ be in EDP. We want to define a recursive correct scheduling S for $\vec{P}$ given the fact that $ECS(\vec{P})$ is true. Suppose, for simplicity, that n = 2. The cooperation of S and P^1,P^2 has to satisfy the conditions i)...iv) from 1.6. This is done as follows: S can be in two overall states, I and II. In both states it performs a finite number of steps. Let s be the initial segment of the cooperation at a stage that S enters state I. In state I S will extend s to

$s * s^1$ where s^1 is minimal (in length) such that

i) $\vec{E^P}(s * s^1)$ holds

ii) P^1 has given at least one non BLANK answer during the steps coded in s^1

iii) after $s * s^1$ P^1 has released all resources that it had been granted after s at least once.

The description of state II is similar with P^2 instead of P^1. Note that s^1 must exist if $E^P(s)$ is true and that the search for a minimal such s^1 is effective due to the decidability of $\vec{E^P}$.

3.5 Proof of theorem 2.8 ii)

Again we construct a counterexample using two processes. Let P^i be the process which behaves like P but uses A_{2i} instead of A_0 and A_{2i+1} instead of A_1. Let Let Q^i be as follows: it counts the number of steps in a counter C. At any stage it evaluates $T(i,i,C)$. If the value is false and has been false before then an answer $1 \in L$ is given. If, however, $T(i,i,C)$ holds this invokes Q^i to request for A_{2i} and A_{2i+1}. Then after both have been granted simultaneously for one step both are released and Q^i answers 1 for ever. Now consider a scheduling S for P^i, Q^i. There are two cases if $\forall C \neg T(i,i,C)$ then S must just run Q^i and P^i in turns. However, if $\exists C\, T(i,i,C)$ then S must first get Q^i through the stage where it needs both A_{2i} and A_{2i+1}. Only there after P^i may get permission to start using A_{2i}. So we conclude that no uniform method exists to find S given i, although S exists. Using the fact that there are infinitely many resources it is now possible to glue together the P^i and the Q^i to processes P and Q in such a way that a scheduling S for P, Q exists but requires solution of the halting problem (and hence cannot be recursive).

3.6 Proof of theorem 2.8 iii)

Again we have a counter using two processes R^0, R^1. We use a resource A which is in fact a register for natural numbers. In A a numeral is stored which codes a

finite 0-1 sequence. The activities of the R^i are divided into blocks which must be executed separately. This is regulated by the acces to A. Execution of a block of R^i amounts to attaching an i to the end of the list stored in A. After any block execution the s.s. is free to choose whether R^0 or R^1 will be on turn for execution of the next block. In this way s.s. can communicate a function f to A. A will contain sequence numbers $< f(0),\ldots,f(n-1) >$.
Now we let the R^i evaluate $SEC(< f(0),\ldots,f(n-1) >)$ at any stage, where SEC is as in 3.3.
If the values are never true then S is a correct scheduling and must be nonrecursive. On the other hand if S is recursive. On the other hand if S is recursive both R^0 and R^1 will see that $SEC(< f(0),\ldots f(n-1) >)$ holds at some stage. Now they will transform themselves to P from that stage thus making deadlock unavoidable.

Acknowledgement. Henh Goeman has made several helpful remarks after reading a previous draft of this paper.

REFERENCE

[1] J.A. Bergstra, Recursion theory on processes, Techn. Report Leiden, July 1977

[2] H. Rogers, The theory of recursive functions and effective computability.