On the granularity of events when modeling program executions

Eric Leu[*] André Schiper[†]

[*] IBM Research Division
Almaden Research Center
650 Harry Road
San Jose, CA 95120, U.S.A.
e-mail: leu@almaden.ibm.com

[†] Ecole Polytechnique Fédérale de Lausanne
Département d'Informatique
IN-Ecublens
CH-1015 Lausanne, Switzerland
e-mail: schiper@lse.epfl.ch

July 9, 1996

Abstract

In the context of parallel programming, program execution models are required to get better understanding of the behavior, properties, and characteristics of parallel programs. The main disadvantages of most existing models are the pretty stiff definition of the events and the generally not clearly defined semantic associated to the model. The goal of the paper is (1) to define a program execution model that uniquely characterize a program execution in terms of program behavior, and (2) determine the sufficient conditions on the granularity of the events that are required to achieve (1).

One essential result of our analysis is that it defines the theoretical framework for reasoning about execution replay, whose goal is to circumvent the non-determinism during the debugging phase. Another essential result is the possibility to model *collective communication*, which is part of the *External User Interface (EUI)* on the

announced *IBM Scalable POWERparallel System 9076 SP1*, and which will be part of the future *Message Passing Interface (MPI)* standard.

1 Introduction

Due to the new complexity involved by parallel programming, modeling program executions is of primary concern. General and semantically well-defined models are required to get better understanding of the behavior, properties, and characteristics of parallel programs. Program execution models have mainly been used to analyze the causal relationships between the components of a computation [10, 14, 20, 23, 24], to detect data race in the context of shared memory architectures [3, 7, 9, 22], to analyze the notion of global states in a parallel computation [1, 4, 11, 13], and to deal with the notion of execution replay [6, 16, 17, 21].

One commonly adopted approach consists in representing an execution as a set of events and relations between these events. This paper addresses two key issues relating to this approach, (1) the definition of the events, i.e. how are the instructions grouped into events, and (2) the semantic associated to the model in terms of program behavior. Regarding the first issue, the events are in general defined in a pretty stiff way. In the case of message passing for example, each *send* and *receive* operation is usually defined as an individual event [1, 4, 10, 11, 14, 20, 23, 24]. However, this approach may be too restrictive. Consider for example collective communication such as *broadcast*, *reduce* (all-to-one reduction), *combine* (all-to-all reduction), or *concat* (all-to-all broadcast) [2, 8, 12]. When modeling such kind of communication, it may be desirable to group several *send* and/or *receive* operations in one event, since internal algorithms used to implement these operations are (or have to be) hidden to the user. Regarding the second issue, the semantic associated to the model is too often not clearly defined: what about two executions characterized by the same set of events and the same set of relations between events ? Will their behavior be identical, i.e. will each process go through the same sequence of local states ? This issue is particularly crucial in the context of parallel programming, since many programs are non-deterministic: two executions of the same program with the same input data may produce different results.

The goal of this paper is to define a program execution model that uniquely characterize a program execution, i.e. uniquely identifies the sequence of local states each process goes through. It will be shown that the whole difficulty to define such a model is linked to the granularity of the events. The sufficient conditions that the events have to satisfy in order to uniquely characterize an execution will be determined in a very formal way. The key ideas are the following:

- definition of the events in terms of *state transformations*;

- definition of two types of relations between events, a *bi-directional* relation B and a *precede* relation R;
- expression of the conditions on the granularity of the events in terms of *state transformations*.

The model will be in a first step defined in the context of message passing, and then extended to the communication through shared variables.

The remainder of this paper is organized as follows. In section 2, we present our program execution model in the context of distributed memory architectures. The conditions on the granularity of the events and the associated proofs are given in section 3. Extension of the results to the shared memory model are discussed in section 4. Finally, some concluding remarks are presented in section 5.

2 Program execution model

In this section, we introduce a program execution model that applies to distributed memory MIMD architectures.

We will consider the execution of a parallel program P. The program consists of N processes that operate asynchronously and communicate with one another solely by exchanging messages. The communication model consists of three operations:

1. $SEND\_RECV(MS, MD, Dst)$ sends message MS to process Dst and waits for reception of a message MD sent by Dst. We assume that this operation may only be used in conjunction with another $SEND\_RECV$ operation (rendez-vous semantic).[1]
2. $SEND(M, Dst)$ sends message M to process Dst. The operation does not return to the user until the message has been copied out of the user's buffer (blocking send).
3. $RECV(M, Src)$ receives a message M from process Src (*receive-by-source*). The value $S = DONTCARE$ may also be used (*receive-from-anybody*). The operation does not return to the user until the requested message has arrived and has been placed in the user's buffer (blocking receive).

The $SEND\_RECV$ operation is synchronous, in the sense that the operation is blocking until a matching $SEND\_RECV$ operation has been issued. The $SEND$ and $RECV$ operations are asynchronous, in the sense that no bounds are assumed on the relative speeds of

[1] Such a primitive may be used to implement any kind of synchronous message passing communication, e.g. synchronous *send()* and *recv()*.

the processes or the time required to pass messages between processes. We further assume infinite system buffer space, and FIFO communication channels between the processes.

Consider now an execution X of a parallel program P. The execution of each process p_i of P is represented as a sequence of events, called local history of the process:

$$h_i^X = e_{i,0}^X, e_{i,1}^X, e_{i,2}^X, ... \tag{1}$$

where $e_{i,k}^X$ represents the k^{th} event of process p_i. The set of all events is noted:

$$E^X = \{e_{i,k}^X | 1 \leq i \leq N, 1 \leq k \leq |h_i^X|\} \tag{2}$$

Each event $e_{i,k}^X \in E^X$ represents the execution of a set of statements, and may be considered as a function ϕ bringing process p_i from local state $s_{i,j-1}^X$ to local state $s_{i,j}^X$ (the initial state of process p_i is defined as $s_{i,0}^X$) and eventually modifying the state of the communication channels. For the moment, we do not make any assumption about the granularity of the events.

We then introduce two relations between events, a *bi-directional* relation B and a *precede* relation R . A relation B will be defined between any two events $e_{i,j}$ and $e_{k,l}$ if information is "exchanged" by $e_{i,j}$ and $e_{k,l}$: some messages are sent by $e_{i,j}$ and received by $e_{k,l}$, other are sent by $e_{k,l}$ and received by $e_{i,j}$. A relation R will be defined between any two events $e_{i,j}$ and $e_{k,l}$ if (1) both of them are events of the same process, $e_{k,l}$ immediately following $e_{i,j}$, or (2) no B relation is defined between the two events, and a message is sent by $e_{i,j}$ and received by $e_{k,l}$.

Definition 2.1 *Given any two events $e_{i,j}^X$ and $e_{k,l}^X$, $B^X(e_{i,j}^X, e_{k,l}^X)$ and $B^X(e_{k,l}^X, e_{i,j}^X)$ iff one at least of the following two conditions is true:*

1. $\exists m_1, m_2 \mid SEND\_RECV(m_1, m_2, p_k) \in e_{i,j}^X \wedge SEND\_RECV(m_2, m_1, p_i) \in e_{k,l}^X$;
2. $\exists m_1, m_2 \mid (SEND(m_1, p_k) \in e_{i,j}^X \wedge RECV(m_1, p_i) \in e_{k,l}^X) \wedge$
   $(SEND(m_2, p_i) \in e_{k,l}^X \wedge RECV(m_2, p_k) \in e_{i,j}^X)$

Definition 2.2 *Given any two events $e_{i,j}^X$ and $e_{k,l}^X$, $R^X(e_{i,j}^X, e_{k,l}^X)$ iff one of the following two conditions is true:*

1. $e_{k,l}^X = e_{i,j+1}^X$;
2. $\exists m \mid SEND(m, p_k) \in e_{i,j}^X \wedge RECV(m, p_i) \in e_{k,l}^X \wedge \neg B^X(e_{i,j}^X, e_{k,l}^X)$.

Contrary to Lamport's *happened-before* relation [14], we do not consider the transitive closure of the relation when defining our *precede* relation R. Note also that due to the B relation, R^+ is always an order relation.

By defining S_0^X as the initial state of the program, an execution X may then be modeled as a tuple $< S_0^X, E^X, B^X, R^X >$.

3 Required granularity of the events

The goal of this section is to show that some assumptions about the granularity of the events are required to uniquely characterize a program execution using the model of section 2. We first illustrate the problem using an example (section 3.1). We then define the notion of *elementary events*, and prove that it corresponds to the sufficient conditions that the events have to satisfy in order to characterize a program execution (section 3.2).

3.1 Difficulties in characterizing program executions

Without any assumptions about the granularity of the events, the model of section 2 does not enable unique identification of the sequence of local states $s_{i,j}$ each process p_i goes through, as shown by the following example.

Consider the program P in Fig. 1 with the three events $e_{1,1}, e_{2,1}, e_{3,1}$. Process p_2 receives two messages; the length of the first one is stored into variable i, and the length of the second one is stored into variable j. Suppose that during an execution X of that program P, p_2 first receives message m_1 from p_1, then message m_2 from p_3 (Fig. 2.a). The length of m_1 is stored into variable i, whereas the length of m_2 is stored into variable j. Consider now another execution Y of the same program (Fig. 2.b), where p_2 receives message m_2 before m_1. The length of m_1 is now stored into variable j, and the length of m_2 is stored into variable i. For both executions, we have $S_0^X = S_0^Y, E^X = E^Y, B^X = B^Y = \{\}, R^X = R^Y = \{(e_{1,1}, e_{2,1}), (e_{3,1}, e_{2,1})\}, s_{2,0}^X = s_{2,0}^Y$, but $s_{2,1}^X \neq s_{2,1}^Y$ (see the state of the variables i and j).

Figure 1: Program P with message passing communication.

Figure 2: Executions X and Y of P.

This identification problem is linked to the granularity of the events. If we consider Fig. 1, two messages are received by a same event $e_{2,1}$. Thus we have $R(e_{1,1}, e_{2,1})$ and $R(e_{3,1}, e_{2,1})$ whatever the order of reception of m_1 and m_2. However, p_2's state after occurrence of $e_{2,1}$ depends on this order.

3.2 Elementary events and equivalence of executions

The required assumptions about the granularity of the events are expressed in terms of state transformations. We first introduce the following notations:

- $M_{e_{i,j}}$ is the set of messages m received by event $e_{i,j}$;
- $M'_{e_{i,j}}$ is the set of messages m sent by event $e_{i,j}$;

Definition 3.1 *An elementary event $e_{i,j}$ is an event satisfying the following conditions:*

C1) *If $e_{i,j}$ includes no message sendings and no message receptions, state $s_{i,j}$ of process p_i after occurrence of $e_{i,j}$ only depends on $s_{i,j-1}$ and $e_{i,j}$: $(s_{i,j-1}, e_{i,j}) \rightarrow s_{i,j}$.*

C2) *If messages are received by $e_{i,j}$, then $s_{i,j}$ depends on $s_{i,j-1}$, $e_{i,j}$, and the set of messages being received: $(s_{i,j-1}, M_{e_{i,j}}, e_{i,j}) \rightarrow s_{i,j}$.*

C3) *If messages are sent by $e_{i,j}$, then $s_{i,j}$ and the messages being sent depend on $s_{i,j-1}$, the set of messages being received, and $e_{i,j}$: $(s_{i,j-1}, M_{e_{i,j}}, e_{i,j}) \rightarrow (s_{i,j}, M'_{e_{i,j}})$.*

C4) *If information is exchanged between $e_{i,j}$ and another event $e_{k,l}$ (messages are sent to and received from $e_{k,l}$), then $s_{i,j}$ and the messages being sent depend on $s_{i,j-1}$, $s_{k,l-1}$, $e_{i,j}$ and $e_{k,l}$: $(s_{i,j-1}, s_{k,l-1}, e_{i,j}, e_{k,l}) \rightarrow (s_{i,j}, M'_{e_{i,j}})$.*

Conditions C1) to C4) are sufficient to uniquely characterize a program execution, as proved by the following proposition:

Proposition 3.2 *Consider two executions $X =< S_0^X, E^X, B^X, R^X >$ and $Y =< S_0^Y, E^Y, B^Y, R^Y >$ such that all events $e \in E^X \cup E^Y$ are elementary events;*

If $S_0^X = S_0^Y, E^X = E^Y, B^X = B^Y$, and $R^X = R^Y$, then the sequence of local states $s_{i,j}$ each process p_i goes through is the same in X and Y.

Proof. Consider an event $e_{i,j}$ and let $seq(e_{i,j}, l) =< e_{n,1}, e_{n_2}, ..., e_{n,l} = e_{i,j} >$ be a sequence of l different events (not necessary belonging to the same process) such that $\forall i, R(e_{n,i}, e_{n,i+1}) \vee B(e_{n,i}, e_{n,i+1})$ [2]. Then let $max\_seq(e_{i,j})$ be the sequence $seq(e_{i,j}, l)$ such that l is maximum. This sequence exists since R^+ is always an order relation and since the sequence is by definition composed of different events. Finally, define $long(e_{i,j})$ as the length of the sequence $max\_seq(e_{i,j})$, and $E_l = \{e_{i,j} \in E \mid long(e_{i,j}) = l\}$. We prove by induction on l that for each event $e_{i,j} \in E_l$, $s_{i,j}^X = s_{i,j}^Y$, $M_{e_{i,j}}^X = M_{e_{i,j}}^Y$, and $M'^X_{e_{i,j}} = M'^Y_{e_{i,j}}$.

[2] We require the sequence $seq(e_{i,j}, l)$ to be composed of *different* events in order to avoid cycling through the B relation!

i) l = 1. First, note that as $E^X = E^Y$ and $R^X = R^Y$, $e^X_{i,j} = e^Y_{i,j}$ $\forall i,j$. Let us show that the proposition holds for all states $s_{i,j}$ such that $e_{i,j} \in E_1$. By definition of $max\_seq(e_{i,j})$, there are no $e_{k,l}$ such that $R(e_{k,l}, e_{i,j}) \vee B(e_{k,l}, e_{i,j})$; thus $e_{i,j}$ corresponds necessarily to the first event $e_{i,1}$ of p_i and cannot be composed of message receptions. As $S^X_0 = S^Y_0$ implies $s^X_{i,0} = s^Y_{i,0}$, it follows from conditions C1) and C3) that $s^X_{i,1} = s^Y_{i,1}$, $M'^X_{e_{i,1}} = M'^Y_{e_{i,1}}$, and $M^X_{e_{i,1}} = M^Y_{e_{i,1}} = \{\}$.

ii) Inductive step. Suppose that $\forall e_{o,p} \in E_k \mid k \leq l$, we have $s^X_{o,p} = s^Y_{o,p}$, $M^X_{e_{o,p}} = M^Y_{e_{o,p}}$ and $M'^X_{e_{o,p}} = M'^Y_{e_{o,p}}$. Let us prove that these equalities are also verified for each event $e_{i,j} \in E_{l+1}$[3].

Case 1. Consider first the set of events $e_{o,p} \mid R(e_{o,p}, e_{i,j})$. By definition of max_seq, $\exists (k \leq l) \mid e_{o,p} \in E_k$. Thus, $\forall e_{o,p} \mid R(e_{o,p}, e_{i,j})$, we have $s^X_{o,p} = s^Y_{o,p}$ and $M'^X_{e_{o,p}} = M'^Y_{e_{o,p}}$. Among these events $e_{o,p}$ must be $e_{i,j-1}$, thus $s^X_{i,j-1} = s^Y_{i,j-1}$. Two cases have to be considered:

- messages are sent by $e_{i,j}$. As $s^X_{i,j-1} = s^Y_{i,j-1}$, it follows from condition C3) that $M'^X_{e_{i,j}} = M'^Y_{e_{i,j}}$;
- messages are received by $e_{i,j}$. For all events $e_{o,p} \mid SEND(m, p_i) \in e_{o,p} \wedge RECV(m, DONTCARE) \in e_{i,j}$, $s^X_{o,p} = s^Y_{o,p}$ and $M'^X_{e_{o,p}} = M'^Y_{e_{o,p}}$ from the induction hypothesis, which means that the same messages are sent by these events $e_{o,p}$. Since $R^X = R^Y$, and since $e^X_{o,p} = e^Y_{o,p}$ and $e^X_{i,j} = e^Y_{i,j}$, the same messages are received by $e_{i,j}$, i.e. $M^X_{e_{i,j}} = M^Y_{e_{i,j}}$.

Since $s^X_{i,j-1} = s^Y_{i,j-1}$ and $M^X_{e_{i,j}} = M^Y_{e_{i,j}}$, it follows from conditions C1) - C3) that $s^X_{i,j} = s^Y_{i,j}$.

Case 2. Consider then the set of events $\{e_{o,p} \mid B(e_{o,p}, e_{i,j}) \wedge \exists (k \leq l) \mid e_{o,p} \in E_k\}$. From 1), we have $s^X_{i,j-1} = s^Y_{i,j-1}$. From the induction hypothesis, $s^X_{o,p} = s^Y_{o,p}$, and thus $s^X_{o,p-1} = s^Y_{o,p-1}$. Since $e^X_{o,p} = e^Y_{o,p}$ and $e^X_{i,j} = e^Y_{i,j}$, it follows from condition C4) that $s^X_{i,j} = s^Y_{i,j}$ and $M'^X_{e_{i,j}} = M'^Y_{e_{i,j}}$.

Case 3. Consider finally the set of events $e_{o,p} \mid B(e_{o,p}, e_{i,j})$ and $\neg(\exists (k \leq l) \mid e_{o,p} \in E_k)$. For such events, $\exists (k \leq l) \mid e_{o,p-1} \in E_k$, since if this would not be the case, $e_{i,j}$ would not belong to E_{l+1}, i.e. $max\_seq(e_{i,j}) = (..., e_{o,p-1}, e_{o,p}, e_{i,j}) > l + 1$! From the induction hypothesis, $s^X_{o,p-1} = s^Y_{o,p-1}$, and from 1), $s^X_{i,j-1} = s^Y_{i,j-1}$. Since $e^X_{o,p} = e^Y_{o,p}$ and $e^X_{i,j} = e^Y_{i,j}$, it follows from condition C4) that $s^X_{i,j} = s^Y_{i,j}$ and $M'^X_{e_{i,j}} = M'^Y_{e_{i,j}}$.

□

In our previous example (Fig. 1), $e_{2,1}$ is not an elementary event, since condition C2) is not satisfied: initial state $s_{2,0}$, event $e_{2,1}$ and the two messages m_1 and m_2 do not determine

[3] By definition, $E_l \cap E_{l+1} = \{\}$.

the order in which both messages will be received, and thus state $s_{2,1}$ after occurrence of $e_{2,1}$.

This is no more the case in the example of Fig. 3: as *receive-by-source* are used instead of *receive-from-anybody*, messages m_1 and m_2 will always be received in the same order. Event $e_{2,1}$ is thus an elementary event.

Figure 3: Program P decomposed in elementary events.

As another example of elementary events, consider a collective communication operation like a *combine*[4] for example, where the result is known by all group members [8, 12]. A clean and correct implementation of such kind of operation is such that the final result only depends on the set of input data (one piece of data being specified by each process), i.e. on the set of states $s_{i,j-1}$ for each group member p_i assuming the operation occurs in $e_{i,j}$. In that case, it is sufficient to consider a collective communication operation as one single event (and not each message sending and message reception that occurs inside the operation)! In the case of Fig. 4, we assume that the three processes p_1, p_2, p_3 belong to group *gid*, and that the *combine* operation is blocking until all members get the final result. An execution of this program is illustrated in Fig. 5, a B relation being defined between the following pair of events: $(e_{1,j}, e_{2,k}), (e_{1,j}, e_{3,l}), (e_{2,k}, e_{3,l})$.

Figure 4: SPMD program Q using collective communication primitives.

Figure 5: An execution of program Q.

This fundamental result expressed by Proposition 3.2 may be used to define formally the equivalence of executions:

Definition 3.3 *Consider two executions* $X =< S_0^X, E^X, B^X, R^X >$ *and* $Y =< S_0^Y, E^Y, B^Y, R^Y >$ *such that all events* $e \in E^X \cup E^Y$ *are elementary events;*

X and Y are equivalent ($X \approx Y$) iff $S_0^X = S_0^Y, E^X = E^Y, B^X = B^Y, R^X = R^Y$.

If two executions are equivalent according to Definition 3.3, each process of the program went through the same sequence of local states.

This notion of equivalence of executions is essential in the context of parallel program debugging. As many parallel programs are non-deterministic, a technique called *execution replay* has been suggested [16, 17, 21, 23] to enable cyclic debugging[5]. The goal of execution

[4] A *combine* operation applies an associative reduction operation on all the processes in a group (e.g. global sum), and places the result in each of the processes in the group.

[5] Cyclic debugging consists in rerunning the program a certain number of times, until all errors have been located, e.g. using a symbolic debugger.

replay is to provide the user with a set of identical executions in the sense that if an error occurred during an initial execution I of a program, it will be reproduced during subsequent replays R of that program. Proposition 3.2 and Definition 3.3 define the theoretical framework for reasoning about execution replay. Conditions C1) - C4) may be used to define the minimal amount of information that has to be collected during the initial execution in order to enforce equivalent executions [19].

4 Extension of the results to the shared memory model

Until now, we only considered parallel programs in the context of distributed memory architectures (e.g. based on message passing). All the discussion regarding the granularity of the events may be extended to the shared memory model.

4.1 Program execution model for shared memory communication

For the purpose of modeling, we again consider a parallel program composed of N processes that operate asynchronously, but we now suppose that these processes communicate with one another by *reading* and *writing* shared variables. In order to define a *bi-directional* relation B and a *precede* relation R in the context of the communication through shared variables, we'll assume that all accesses to a memory unit [6] are totally ordered [15]. Let's introduce the following notation:

Definition 4.0.a *Consider a memory unit m and a set of events accessing m, where each event may access the same unit more than once.*

The string MACCESS(m) is defined as the ordered sequence of accesses to unit m, each access being identified by the event accessing m.

As an example, consider three events $e_{1,1}, e_{1,2}, e_{2,1}$ accessing unit m in the following order: $e_{1,1}, e_{2,1}, e_{1,1}, e_{1,2}$. We have $MACCESS(m) = \ 'e_{1,1}, e_{2,1}, e_{1,1}, e_{1,2}\ '$, whereas $'e_{1,1}, e_{2,1}\ '$, $'e_{2,1}, e_{1,1}, e_{1,2}\ '$ are substrings of *MACCESS(m)*.

Using the same principle as for message passing, a bi-directional relation B will be defined between any two events $e_{i,j}$ and $e_{k,l}$ if they access common shared memory units "at the same time", one access at least being a *write* access. Similarly, a relation R will be defined

[6] A memory unit is such that each *read* or *write* access to that unit is *atomic* (in the sense of indivisible). The granularity of the units depends on the hardware.

between any two events $e_{i,j}$ and $e_{k,l}$ if (1) both of them are events of the same process, $e_{k,l}$ immediately following $e_{i,j}$, or (2) no B relation is defined between the two events, and $e_{k,l}$ accesses a shared memory unit immediately after $e_{i,j}$, one at least being a *write* access.

In order to simplify the formal definitions of B and R, we introduce the following notation:

Definition 4.0.b *Consider any two events* $e^X_{i,j}$ *and* $e^X_{k,l}$*;*

FOLLOWS($e^X_{i,j}, e^X_{k,l}$*) iff there exists a memory unit m such that:*

a) *'*$e^X_{i,j}, e^X_{k,l}$*' is a substring of MACCESS(m), both accesses being write accesses, or*

b) *'*$e^X_{i,j}, \{e^X_{o,p}\}, e^X_{k,l}$*' is a substring of MACCESS(m), only one of the two accesses being a write access,* $\{e^X_{o,p}\}$ *representing a set of read accesses.*

7

Definition 4.1 *Given any two events* $e^X_{i,j}$ *and* $e^X_{k,l}$*,* $B^X(e^X_{i,j}, e^X_{k,l})$ *and* $B^X(e^X_{k,l}, e^X_{i,j})$ *iff* $FOLLOWS(e^X_{i,j}, e^X_{k,l}) \wedge FOLLOWS(e^X_{k,l}, e^X_{i,j})$.

Definition 4.2 *Given any two events* $e^X_{i,j}$ *and* $e^X_{k,l}$*,* $R^X(e^X_{i,j}, e^X_{k,l})$ *iff a)* $e^X_{k,l} = e^X_{i,j+1}$*, or b)* $FOLLOWS(e^X_{i,j}, e^X_{k,l}) \wedge \neg B^X(e^X_{i,j}, e^X_{k,l})$.

4.2 Elementary events

The difficulties in characterizing a program execution in the context of the communication through shared variables are very similar to the ones discussed in the context of message passing (see section 3). Consider as an example the three programs in Fig. 6. Events $e_{1,1}$, $e_{1,2}$, and $e_{2,1}$ in program a) are too coarse-grained, and the execution cannot be uniquely characterized: two executions of this program may have the same set of events, the same set of B and R relations, but different end values for T (e.g (3,8,0), (3,0,0), (3,0,8), ...). In the case of program b), the local state of both processes p_1 and p_2 will always be uniquely characterized, but not the state of shared variable d (e.g. *d='1' or d='2'*). Finally, the event decomposition in program c) enable unique characterization of the program execution, though each event correspond to several accesses to different shared memory units!

Figure 6: Example of three programs based on a communication through shared variables.

In order to define the notion of *elementary events* in the context of the communication through shared variables, we introduce the following notations:

[7]Note that such a function could also have been defined in the context of message passing and used for defining B and R.

- $Sv_{e_{i,j}}$ is the state before occurrence of event $e_{i,j}$ of all shared memory units accessed by $e_{i,j}$;
- $Sv'_{e_{i,j}}$ is the state after occurrence of event $e_{i,j}$ of all shared memory units accessed by $e_{i,j}$;

Definition 4.3 *In the context of the communication through shared variables, an elementary event $e_{i,j}$ is an event satisfying the following conditions:*

C1') *If $e_{i,j}$ includes no accesses to shared memory units, state $s_{i,j}$ of process p_i after occurrence of $e_{i,j}$ only depends on $s_{i,j-1}$ and $e_{i,j}$: $(s_{i,j-1}, e_{i,j}) \rightarrow s_{i,j}$.*

C2') *If shared memory units are read by $e_{i,j}$, then $s_{i,j}$ depends on $s_{i,j-1}$, $e_{i,j}$, and the state of all shared memory units accesses by $e_{i,j}$: $(s_{i,j-1}, Sv_{e_{i,j}}, e_{i,j}) \rightarrow s_{i,j}$.*

C3') *If shared memory units are written by $e_{i,j}$, then $s_{i,j}$ and the resulting state of these units $Sv'_{e_{i,j}}$ depend on $s_{i,j-1}$, the state of the shared memory units being read, and $e_{i,j}$: $(s_{i,j-1}, Sv_{e_{i,j}}, e_{i,j}) \rightarrow (s_{i,j}, Sv'_{e_{i,j}})$.*

C4') *If $\exists e_{k,l} \mid FOLLOWS(e_{i,j}, e_{k,l}) \wedge FOLLOWS(e_{k,l}, e_{i,j})$, then $s_{i,j}$ and the state of the shared memory units accessed by $e_{i,j}$ depend on $s_{i,j-1}$, $s_{k,l-1}$, $e_{i,j}$ and $e_{k,l}$: $(s_{i,j-1}, s_{k,l-1}, e_{i,j}, e_{k,l}) \rightarrow (s_{i,j}, Sv'_{e_{i,j}})$.*

Consider again the three programs in Fig. 6. In the case of program a), condition C2') is not satisfied for event $e_{2,1}$. In the case of program b), conditions C2') and C3') are satisfied for both events $e_{1,1}$, $e_{2,1}$, but not condition C4') (the final state of shared variable d is not determined by $s_{1,0}, e_{1,1}, s_{2,0}$ and $e_{2,1}$. Finally, in program c), both $e_{1,1}$ and $e_{2,1}$ satisfy conditions C1') - C4'), and are thus elementary events.

When modeling program executions in the context of a communication through shared variables, elementary events may be obtained by defining each access to a shared memory unit as an event. A coarse-grain granularity can be obtained if a protocol that ensures a valid linearization, such as a *Concurrent-Read-Exclusive-Write* (CREW) protocol [5] is used to access the shared variables. Each access to a shared variable can in that case be assimilated as one elementary event. Depending on the application and/or the communication library, still more coarse-grain events can be defined (e.g. Fig. 6).

The proof of Prop. 3.2 extended to the shared memory model is very similar to the one given for the message passing model (section 3.2), and is thus not presented in this paper.

5 Conclusion

We have discussed two key issues related to modeling parallel program executions, (1) the definition of the events, and (2) the semantic associated to the model in terms of program behavior. We have shown that the difficulty in defining a model that uniquely characterize a program execution is linked to the granularity of the events. The model we propose is composed of a set of events, two types of relations between events, a *bi-directional* relation B and a *precede* relation R, and a set of conditions on the granularity of the events. Based on that model, a semantically clean and precise definition of the notion of equivalence of executions has been given.

One essential result of our analysis is that our model defines the theoretical framework for reasoning about execution replay. By modeling non-blocking message passing communication as a combination of both standard message passing (blocking) and communication through shared variables, we developped a very efficient execution replay technique base on that model [17, 19]. The other essential result is the possibility to efficiently model collective communication, which is part of the *External User Interface (EUI)* on the announced *IBM Scalable POWERparallel System 9076 SP1*, and which will be part of the future *Message Passing Interface (MPI)* standard. These two aspects suggests for our model interesting avenues for further learning and research.

References

[1] M. Adam, M. Hurfin, N. Plouzeau, M. Raynal, *Distributed Debugging Techniques*, IRISA/IFSIC, 35042 Rennes Cedex, April 1991.

[2] J. Bruck, R. Cypher, H. Ho et al., *CCL: A Portable and Tuneable Collective Communication Library for Scalable Parallel Computers*, IBM Research Report, RC-xxx, March 1993.

[3] D. Callahan, K. Kennedy, J. Subhlok, *Analysis of Event Synchronization in a Parallel Programming Tool*, Proc. of 2nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, March 1990.

[4] K. Chandy, L. Lamport, *Distributed Snapshots: Determining Global States of Distributed Systems*, ACM Transactions on Computer Systems, Vol. 3 No. 1, Feb. 1985.

[5] P. Courtois, F. Heymans, D. Parnas, *Concurrent Control with Readers and Writers*, Communications of the ACM 14(10), Oct. 1971.

[6] R. Curtis, L. Wittie, *BugNet: A Debugging System for Parallel Programming Environments*, Proc. 3rd Int. Conf. on Distributed Computing Systems, Hollywood, FL, Oct. 1982.

[7] A. Dinning, E. Schonberg, *Detecting Anomalies in Programs with Critical Sections*, ACM SIGPLAN Notices, Vol. 26 No. 12, Dec. 1991.

[8] J. Dongarra, R. Hempel, A. Hey, D. Walker, *A Proposal for a User-Level, Message-Passing Interface in a Distributed Memory Environment*, ORNL/TM-12231, March 1993.

[9] P. Emrath, D. Padua, *Automatic Detection of Nondeterminacy in Parallel Programs*, SIGPLAN Notices, Vol. 24 No. 1, Jan. 1989.

[10] C. Fidge, *Timestamps in Message Passing Systems that Preserve the Partial Ordering*, Proc. of 11th Australian Computer Science Conf., Univ. of Queensland, Feb. 1988.

[11] J. Fowler, W. Zwaenepoel, *Causal Distributed Breakpoints*, Proc. 10th IEEE Int. Conf. on Distributed Computing Systems, Paris, May 1990.

[12] D. Frye, R. Bryant, H. Ho, R. Lawrence, M. Snir, *An External User Interface for Scalable Parallel Systems*, Technical Report, IBM Highly Parallel SuperComputing Systems Laboratory, Nov. 1992.

[13] A. Goldberg, A. Gopal, A. Lowry, R. Strom, *Restoring Consistent Global States of Distributed Computations*, ACM SIGPLAN Notices, Vol. 26 No. 12, Dec. 1991.

[14] L. Lamport, *Time, Clocks, and the Ordering of Events in a Distributed System*, Communications of the ACM 21(7), 1978.

[15] L. Lamport, *How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs*, IEEE Trans. on Computers C-28(9), Sept. 1979.

[16] T. Leblanc, J. Mellor-Crummey, *Debugging Parallel Programs with Instant Replay*, IEEE Transactions on Computers C-36(4), April 1987.

[17] E. Leu, A. Schiper, A. Zramdini, *Efficient Execution Replay Technique for Distributed Memory Architectures*, Proc. of 2nd European Distributed Memory Computing Conf., LNCS 487, Springer-Verlag, Munich, April 1991.

[18] E. Leu, A. Schiper, *Execution Replay: A Mechanism for Integrating a Visualization Tool with a Symbolic Debugger*, Proc. of CONPAR 92 - VAPP V Conf., LNCS 634, Springer Verlag, Lyon, Sept. 1992.

[19] E. Leu, *La réexécution, pierre angulaire de la mise au point des programmes parallèles*, Ph.D. thesis Nb. 1049, Swiss Federal Institute of Technology, Computer Science Department, CH-1015 Lausanne, Switzerland.

[20] F. Mattern, *Virtual Time and Global States of Distributed Systems*, Proc. Int. Workshop on Parallel and Distributed Algorithms (Cosnard et al. eds), North Holland, 1989.

[21] J. Mellor-Crummey, *Debugging and Analysis of Large-Scale Parallel Programs*, Univ. of Rochester, Technical Report 312, Sept. 1989.

[22] R. Netzer, B. Miller, *Detecting Data Races in Parallel Program Executions*, Proc. 3rd Workshop on Programming Languages and Compilers for Parallel Computing, Irvine CA, Aug. 1990.

[23] R. Netzer, B. Miller, *Optimal Tracing and Replay for Debugging Message-Passing Parallel Programs*, Proc. of SuperComputing 92 Conf., Minneapolis, Nov. 1992.

[24] R. Schwarz, F. Mattern, *Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grail*, Technical Report, 1992, University of Kaiserslautern, P.O. Box 3049, D - 6750 Kaiserslautern.