International Journal of Parallel Programming, Vol. 22, No. 4, 1994

Using True Concurrency to Model Execution of Parallel Programs

Yosi Ben-Asher[1] and Eitan Farchi[2]

Received June 1993

Parallel execution of a progam R (intuitively regarded as a partial order) is usually modeled by sequentially executing one of the total orders (interleavings) into which it can be embedded. Our work deviates from this serialization principle by using *true concurrency*[1] to model parallel execution. True concurrency is represented via completions of R to *semi total orders*, called time diagrams. These orders are characterized via a set of conditions (denoted by Ct), yielding orders or time diagrams which preserve some degree of the intended parallelism in R. Another way to express semi total orders is to use re-writing or derivation rules (denoted by Cx) which for any program R generates a set of semi-total orders. This paper includes a classification of parallel execution into three classes according to three different types of Ct conditions. For each class a suitable Cx is found and a proof of equivalence between the set of all time diagrams satisfying Ct and the set of all terminal Cx derivations of R is devised. This equivalence between time diagram conditions and derivation rules is used to define a novel notion of correctness for parallel programs. This notion is demonstrated by showing that a specific asynchronous program enforces synchronous execution, which always halts, showing that true concurrency can be useful in the context of parallel program verification.

KEY WORDS: True-concurrency; partial-orders; parallel program verification.

1. INTRODUCTION

Let R be a parallel program, composed of *nested* parallel and sequential statements. Many parallel programming languages are of this type, such as

[1] Mathematics and Computer Science Department, Haifa University, Haifa.
[2] IBM Research Center, Haifa.

828/22/4-1

OCCAM,[2] ADA,[3] and others.[4–6] For instance, the following program spawns two processes, each of which splits into two additional processes.

```
R(){
    int x = 0;
    par for i = 1..2[
                    par for j = 1..2[x = i + j;]
    ]}
```

Such a syntax is equivalent to a partial order (also denoted by R):

$$R = (x = 0; ((i = 1; ((j = 1; x = i + j) \| (j = 2; x = i + j))) \| (i = 2; ((j = 1; x = i + j) \| (j = 2; x = i + j))))$$

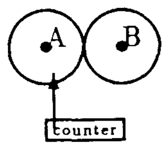
between all instructions of A and B.

It is now fairly common for parallel machines to adhere to the dictates of *sequential consistency* in order to facilitate our understanding of parallel programs.[7–9] This requirement states that the results of parallel execution of a program are the same as the results that would be obtained had the instructions from distinct parallel processes been interleaved and executed in some serial ordering. This is equivalent to a completion of R to a total order executed as a sequential program.

The problem is that if the interleaving is not completely defined, the program execution is indeterminate. As a consequence, distinct executions of the same program may lead to different results. For example, one execution may terminate with a result while another enters an infinite loop. The problem is especially severe in a shared memory model, where interactions are mediated by side effects. It is therefore important to develop a notion of correctness that enables the user to show that all execution orders of a program are correct *without explicitly generating them.* This idea can be expressed in terms of finding a "compact" representation to the set of all possible execution orders of a parallel program.

Modeling parallel execution as the set of all completions of R to a total order (interleavings) is rather arbitrary. Theoretically, parallel execution may be any set of partial orders which correctly augments (includes) R. Total completions of R actually represent a particular time model in which each instruction of R is assigned a unique time index. Another time model may assign the same time index to several instructions, forming *true concurrency.* As it turns out, true concurrency is essential in the context of parallel program verification, since it allows the user to use compound instructions in place of the original ones, leading to smaller size programs. [Note that for a given program, a true concurrency model may contain fewer execution orders than interleavings. This claim, although combinatorially false, may however hold in specific cases, which we believe to be the practical ones.] Compound statements or abstractions, (formally suggested in Ref. 7) may lead to true concurrency, e.g., when two or more compound statements are mutually dependent.

As an example of true concurrency, consider the following machine with two gears. It has four instructions, *Al*, *Ar*, *Bl*, *Br*, used to turn the gears *A* and *B* left or right. If *Al* and *Bl* are executed in parallel ($Al \| Bl$), then no gear can move and the counter will show zero. However, any sequential order of execution (*Al*; *Bl* or *Bl*; *Al*) will yield a positive counting. Hence, modeling parallel execution via interleavings fails to describe a parallel execution.



Theory of concurrency contains many relevant results, advocating different time models that can be applied to parallel execution of programs.[11] (such as comparing the expressiveness of interleavings versus true concurrency).[10] In particular, Pratt[1] constructs a mixed term of partial orders and temporal logic (Ref. 12) to model parallel processes. This work also contains motivations for true concurrency execution, and a short survey of relevant results. The system proposed here can be best represented through the formalism developed by Gaifman.[13] Unlike Gaifman's formalism, which is more general, this formalism is dedicated mainly to parallel programs and scheduling only. In this sense, Gaifman's formalism uses a set of partial orders to represent a device, while our formalism uses a single partial order which is the program. [In comparison to similar works such as that of Pratt,[1] this formalism excludes semantics operations like loops, choice, recursion, and communication. These issues are modeled via a set of external restrictions and thus are not part of the suggested formalism.] A more recent and general work demonstrating the power of true concurrency is the work of Janicki and Kountny.[14]

Gaifman describes a computation (of a device) as a triple (E, $<^c$, $<^t$), where E is a set of events, $<^c$ is a *causal* partial order representing inherent semantics, and $<^t$ is a *temporal* partial order describing a possible execution (completion or augmentation) of $<^c$ in time. For example, ($\{a, b\}$, $a \| b$, $a; b$) describes a sequential execution of the process $a \| b$. In addition, Gaifman defines a process as a set of partial orders $P = p_1, ..., p_n$. The core of a process is defined to be the maximal set of least constrained p_i's (those which do not include any other p_j). A process describes a specification of a device if its core is a correct specification of the device and the process is augment closed, i.e., every possible execution of the core is present in the process. Thus a process P describes a device if it can be divided into $\langle^c = core(P)$ and $\langle^t = P - core(P)$, and $\langle^t$ covers every possible computation of the device.

Gaifman's general notion requires that a specification should explicitly include every possible execution or computation $p_i \in P - core(P)$. Our work is based on the observation that in several cases the set of all possible executions (temporal orders of a program R) can be represented in a compact nonexplicit way (similar to a compact representation of all even numbers as all numbers whose mod 2 is zero).

Our representation deals with processes that describe execution of parallel programs rather than devices. Thus the core of every process consists of a single partial order, which represents a parallel program. The set of all possible computations of a program R (also referred to as executions or scheduling) is represented as a *restriction of a time model.* The notion of time models has been pursued in many papers (see Refs. 1 and 13 for relevance). In particular, let $P(R)$ be the set of all possible partial orders which are consistent with R (including completions to total or semi-total orders). A time model can be described as a selection function $f(P(R)) \in 2^{P(R)}$ which selects those partial orders which augment R in time.[7] For example, if the time model reflects serialization, then f will select all completions of R to a total order. A compact representation of f may turn out to be useful in showing that all execution orders of R satisfy some desired condition (such as halting).

We use two compact representations for a time model V: Ct^V, a set of conditions choosing a member of $2^{P(R)}$ and Cx^V, a set of derivation rules generating the desired member of $2^{P(R)}$. A process is a triple $\langle R, Ct, Cx\rangle$ where R is a program or a partial order and $Ct \subset Ct^V$, $Cx \subset Cx^V$ are restrictions of a time model V used to describe specific semantics of R. In Gaifman's notion, such a process should be represented as $P = R, Ct(P(R))$ or $P = R, Cx(P(R))$, where $R = core(P)$. Hence, we use an explicit compact representation of V rather than using a member of $2^{P(R)}$.

As explained before, in the context of parallel program verification[15] true concurrency is likely to be used. Hence, V should reflect true concurrency (also referred to as linear-time model in Ref. 13). It turns out that for some true concurrency time models, compact representation (Ct^V, Cx^V) can be devised. This can be done using the fact that true concurrency partial orders (referred as time diagrams) have a specific structure. Every possible time diagram of a time model should be of the form $S_1;...;S_n$ where $S_i = e_{i_1} \| \cdots \| e_{i_k}$ contains those instructions executed at time i. This structure is used in the proposed formalism, which includes the following components:

Ct- is a finite set of conditions specifying necessary relations between the states of every time diagram and the program.

Cx- is a finite set of derivation rules, such that every final derivation $R \xrightarrow{Cx} R'$ yields a time diagram selected by f.

$\langle R, Cx, Ct \rangle$- indicates that Ct and Cx are equivalent and yield the same set of time diagrams, hence representing the same set of time diagrams. $\langle R, Cx, Ct \rangle$ is referred to as an induction system for R.

Correctness. A program R is defined to be correct if it has an induction system such that every time diagram of R also halts. In this way, Ct actually describes every semantic aspect of executing R by a parallel computer.

For example, the interleavings set of R can be represented by the following induction system.

- Ct maintains that every $e_a <_R e_b$ should be in a different state, such that $e_a \in S_i$, $e_b \in S_j$ and $i < j$.
- Cx includes all possible ways to interleave R without violating $<_R$:

$$(A; B) \| C \to \begin{cases} (A \| C); B \\ A; (B \| C) \end{cases} (A \| B) \to \begin{cases} (B \| A) \\ (A; B) \\ (B; A) \end{cases} (A; B) \| (C; D) \to (A \| C); (B \| D)$$

In this work we define three main classes of true concurrency time models, using suitable Ct conditions and Cx rules. A proof that these Ct, Cx form an induction system for every type of time model is devised. These models are theoretically interesting, yet they serve as a mechanism to preserve the intended parallelism invoked in the construct $(A \| B)$. Thus, when true concurrency is used, the time model of a specific program is actually a restriction of the main time models or classes. A proof that a specific $\langle R, Ct, Cx \rangle$ system is an induction system can exploit the fact that both Ct and Cx are actually restrictions of the main Ct and Cx for which an induction system exists. This usage of natural classes of time models will be clarified in later sections.

Finally, an induction system $\langle R_k^l, Ct, Cx \rangle$ is constructed for special kinds of parallel programs, called parallel loop programs. Let R_k^l be a set of k loops executed by different processors, where each loop contains l instructions. It is shown that $\langle R_k^l, Ct, Cx \rangle$ yields exactly all synchronous executions of R_k^l. This is used to show that a specific asynchronous program is correct and can only be executed synchronously.

2. BASIC DEFINITIONS FOR A TRUE CONCURRENCY TIME MODEL

This section contains basic definitions of a novel time model called V. This time model reflects true concurrency via a special interpretation of the $\|$ operator, referred to as the "friction condition" (to be explained next).

Definition 2.1. A parallel program corresponds to the following regular expression:

$$R \rightarrow (R) \text{ or } (R \| R) \text{ or } (R; R) \text{ or } \|e_i$$

where sub-programs are denoted with capital letters, and atomic instructions with small letters, e.g., $R = (a; b) \| (c; d)$. The meaning of execution of programs will be the function that maps the Cartesian products of programs and states to states (ρ: $R \times S \rightarrow S$).

Let a parallel execution of a program R be a time diagram which assigns a time index to every instruction as described in Fig. 1. This implies that atomic instructions take one time unit and do not overlap.

For a given program there can be many and different time-diagrams which describe different possible executions (see Fig. 2). Our first step in defining what parallel execution of a program means is to formally define the set of all time-diagrams or executions of a program.

Definition 2.2. A program defines a partial order relation on the atomic instructions of the program (e.g., "before") in the following way: if a sub expression of the program is of the form $E_1; E_2$ then every atomic instruction in E_1 precedes every atomic instruction in E_2.

Definition 2.3. For the V time model, a time-diagram of a program (R) is a division of the atomic instructions of R into an order sequence of "steps" ($S_1, \ldots, S_T$) such that the following Ct^V conditions are met.

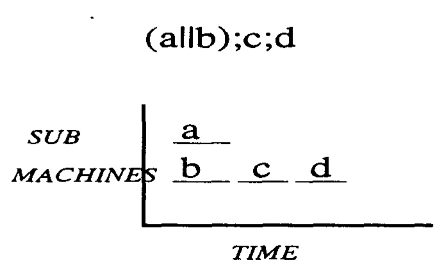


Fig. 1. Processors/time diagram of the program $(((a \| b); c); d)$.

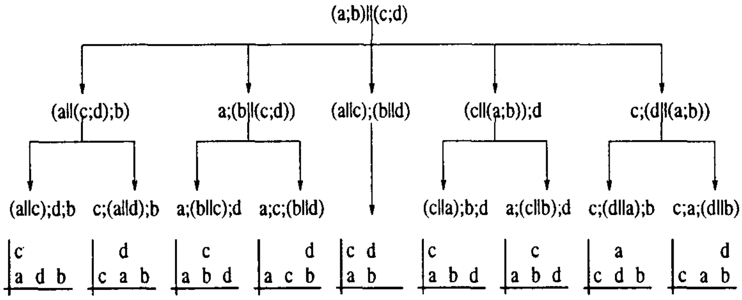


Fig. 2. Different normal forms of $(a; b) \| (c; d)$ and their time diagrams.

1. S_i is not empty.
2. $\forall (E_a \| E_b) \in R$ there exists $e_a \in E_a$, $e_b \in E_b$ such that $e_a, e_b \in S_i$. This condition is also referred to as the *friction condition* since it enforces minimum degree of true concurrency execution between parallel expressions.
3. For every e_a and e_b such that e_a is "before" e_b there are $j < i$, such that $e_a \in S_j$ and $e_b \in S_i$.

The set of well-defined parallel machines can be defined as follows:

Definition 2.4. A parallel machine M is well defined if any execution of a program R by M can be described by a legal time diagram of R. Such a machine is referred to as a "discrete" machine.

Obviously, a time-diagram $S_1, \ldots, S_T$ can be expressed by a program of the form $(\|E_1; \ldots; \|E_T)$ where $\|E_i$ contains all the atomic instructions of S_i. We refer to a program of this form as a program in "*normal form.*"

So far, the parallel execution was defined by a time trace on the instructions executed by a parallel machine (time diagrams). Another possibility to define or understand the parallel execution of a program is to imagine a virtual *term rewriting system* which reduces a program to normal form using a set of rewriting or derivation rules. These rewriting rules are not arbitrary and actually correspond to logical rules that parallel execution should fulfill. Thus the meaning of parallel execution is depicted in terms of its outcome (time diagrams) and its logical behavior (axioms or derivations).

Definition 2.5. The following set of axioms or rewriting rules Cx^V formally defines the execution of a parallel program R via the set of all normal forms obtained by a final derivation $R \xrightarrow{f*} R'$.

Commutativity. For every two programs $R_1 \| R_2 \equiv R_2 \| R_1$. This condition reflects a natural understanding that a parallel execution should be symmetrical.

Associativity. Sequential execution is oblivious to order of execution $(R_1; R_2);\ R_3 \equiv R_1;\ (R_2; R_3)$. However, for parallel execution associativity is allowed only at the instructions level, i.e., $(e_1 \| e_2) \| e_3 \equiv e_1 \| (e_2 \| e_3)$. Note that $\|$ between instructions indicates that all instructions should be executed at the same time; thus, we use the notation $e_1 \| \ldots e_k$ to indicate any placement of parentheses between parallel instructions.

Time division.

$$\overbrace{(R_{a1}; R_{a2})}^{R_a} \| (R_b; R_c) = \begin{cases} (R_a \| R_b);\ R_c \\ R_b;\ (R_a \| R_c) \\ ((R_{a1} \| R_b);\ (R_{a2} \| R_c)) \end{cases}$$

The restriction on associativity of complex terms reflects the requirement that $(E_1 \| E_2)$ denotes some true concurrency execution between E_1 and E_2. For example, consider the program $R = ((A \| B) \| (C \| D))$. The parentheses structure indicates that there should be a true-concurrency execution between A and B. However, if associativity is allowed, then R is also equivalent to $(A \| (B \| (C \| D)))$ and there is no longer need for a true-concurrency execution between A and B, but rather there could be true-concurrency execution between A and C or D or both.

Clearly, the associativity axiom enables any manipulations of parentheses in regular expressions with ';', as (for the sake of completeness) is proved next:

Lemma 2.1. Let R be a program of the form $R = R_1; \ldots; R_n$ and let $R^{(1}$ and $R^{(2}$ be any two legal (Def. 2.1) assignments of parentheses in R. By applying the associativity axiom, $R^{(1}$ can be transformed to $R^{(2}$.

Proof. Let $R^{[} = (\ldots((R_1; R_2); R_3); \ldots); R_n)$ denote a left-most parentheses assignment of R. Let $\xrightarrow{(*}$ be a rapid application of "left" associativity on sub expressions in $R(R_a; (R_b; R_c)) \to ((R_a; (R_b); R_c)$ until no further application is possible. Clearly, there are no)) parentheses in R after $\xrightarrow{(*}$; hence, $\xrightarrow{(*}$ terminates in left-most parentheses assignments of R. Since $R^{(1} \xrightarrow{(*} R^{[}$, $R^{(2} \xrightarrow{(*} R^{[}$ and associativity can be applied in both directions, $R^{(1}$ can be transformed to $R^{[}$ and then back to $R^{(2}$. [This manipulation allows us to omit parentheses in any ; E expression.] □

3. PROVING THAT Ct^V AND Cx^V DESCRIBE THE SAME TIME MODEL

Until now two ways of evaluating parallel execution ($R: s_1 \to s_2$) have been presented:

- A derivation through the axioms $R \xrightarrow{f*} R'$, consisting of rapid application of the axioms of Def. 2.5, until no further derivation is possible.
- A time diagram $D(R)$ satisfying the conditions of Def. 2.3.

The connection between the axioms and the time-diagrams of a program is demonstrated in Fig. 2, where all legal time diagrams of $R = (a; b) \| (c; d)$ are derived using rapid applications of the "time-division" axiom.

In this section it is shown that both ways are equivalent, i.e., every derivation $R \xrightarrow{f*} R'$ yields a different time diagram, and every time diagram $D(R)$ can be derived by the axioms $R \xrightarrow{f*} D(R)$. Hence, both Ct^V and Cx^V define the same time model or class of parallel executions. From now on we omit ^{V} from both Ct^V and Cx^V, making the V time model the default one.

Definition 3.1. For a given program R let:

$T(R)$ be the set of all time diagrams of the program R satisfying the time diagram definitions (Def. 2.3).

$R \xrightarrow{f*} R'$ be a "forward" derivation of R' from R by applying the axioms of parallelism. Note that the time division axiom has three alternatives to replace $(a1; a2) \| (b; c)$. Thus $\xrightarrow{f*}$ is a forward derivation in which $(a1; a2) \| (b; c)$ is replaced by one of the three alternatives.

$R \xrightarrow{r*} R'$ be a "backward" derivation of R' from R in which one of the three alternatives of the time division axiom is replaced by $(a1; a2) \| (b; c)$. Clearly, if $R \xrightarrow{r*} R'$, then by reversing the $\xrightarrow{r*}$ derivation $R' \xrightarrow{f*} R$.

R^* be the set of all programs resulted by $\xrightarrow{f*}$, which can not be further reduced.

Further, in the following discussion we will use $D(R)$ both for a time diagram of R and the unique normal form that matches $D(R)$.

Theorem 3.1. There is a one to one mapping between the set of all time diagrams of a program and the set of programs derived by the axioms, hence $R^* = T(R)$. This equivalence shows that both conditions of Def. 2.3

and the derivation rules of Def. 2.5 define a class with true concurrency executions called V.

Proof. Follows from Th. 3.3, which shows that every number of R^* is in normal form and represents a time diagram, hence $R^* \subset T(R)$. From Th. 3.2 every time diagram $D(R) \in T(R)$ is in R^*, i.e., $R \xrightarrow{f*} D(R)$.

Corollary 3.1. In order to define a parallel machine M, it suffices to describe only the effect of executing s: $\|E \rightarrow s'$ (rather than describing $\forall R$ s: $R \rightarrow s'$).

Proof. Immediate from the normal form definition.

3.1. Proof for the Derivation of Time Diagrams

The fact that every time diagram of R can be derived by the axioms of parallelism is stated and then proved as follows:

Theorem 3.2. Every time diagram $D(R) \in T(R)$ is in R^*, i.e., for every $D(R)$, $R \xrightarrow{f*} D(R)$.

Proof. By induction on the structure of R. Note that for single instruction programs $T(R) = R^*$. Assume that $P1 \xrightarrow{f*} D(P1)$ and $P2 \xrightarrow{f*} D(P2)$. It is sufficient to show that $(P1; P2) \xrightarrow{f*} D(P1; P2)$ and that $(P1 \| P2) \xrightarrow{f*} D(P1 \| P2)$.

By the induction hypothesis $(P1; P2) \xrightarrow{f*} (D(P1); D(P2))$. Trivially, $(D(P1); D(P2)) = D((P1; P2))$, hence $(P1; P2) \xrightarrow{f*} D((P1; P2))$.

The second case is more complicated and uses several sub claims and a "roll-back" process, which restores $D((P1 \| P2))$ to a program of the form $(D(P1) \| D(P2))$. Lemma 3.2 shows that the effect of the roll-back process is to transform $D(P1 \| P2)$ to

$$D(P1 \| P2) \xrightarrow{r*} (\|\alpha_1;...; \|\alpha_n) \| (\|\beta_1;...; \|\beta m) \qquad \|\alpha_i \in P1 \wedge \|\beta i \in P2$$

Lemma 3.3 shows that the roll-back process yields separate time diagrams of $P1$ and $P2$. This is done by showing that

$$(\|\alpha_1;...; \|\alpha_n) \in T(P1) \wedge (\|\beta_1;...; \|\beta m) \in T(P2)$$

The induction hypothesis yields that $D(P1 \| P2) \xrightarrow{r*} D(P1) \| D(P2) \xrightarrow{r*} P1 \| P2$, hence $P1 \| P2 \xrightarrow{f*} D(P1 \| P2)$, as required. □

The following notation is used to describe the roll-back processes of $D(P1 \| P2) = S_1, \ldots, S_k$ where:

$$S_i = \begin{cases} \alpha_i & \text{if } S_i \text{ contains instructions from } P1 \text{ only} \\ \beta_i & \text{if } S_i \text{ contains instructions from } P2 \text{ only} \\ \gamma_i = (\alpha_1; \ldots; \alpha_n) \| (\beta_1; \ldots; \beta m) & \text{mixed term of } \alpha \text{ and } \beta \text{ sub expressions} \end{cases}$$

Note that initially in γ_i both $m, n = 1$, while during the roll-back process γ_i accumulates states. Clearly, up to commutativity and associativity of '$;E$' and $\|e_i$ every $D(P1 \| P2)$ is of this form. Hence any $D(P1 \| P2)$ is a sequence $\ldots; \alpha; \ldots; \gamma; \ldots; \beta; \ldots; \gamma; \ldots$. Initially every γ in $D(P1 \| P2)$ contains one α and one β. During the roll-back process, any γ expression collects more α, β terms. However, the structure $\ldots; \alpha; \ldots; \gamma; \ldots; \beta; \ldots; \gamma; \ldots$ is maintained throughout the process.

Definition 3.2. A roll-back step is a step where the time division axiom is applied backwards to reduce sub-programs of the form $\langle S_i; \gamma \rangle$ or $\langle \gamma; S_i \rangle$ to γ:

$$\alpha_x; \overbrace{(\alpha_y \| \beta_z)}^{\gamma} \xrightarrow{r*} \overbrace{(\alpha_x; \alpha_y) \| \beta_z}^{\gamma}$$

$$\beta x; \overbrace{(\alpha_y \| \beta_z)}^{\gamma} \xrightarrow{r*} \overbrace{\alpha_y \| (\beta x; \beta z)}^{\gamma}$$

$$\overbrace{(\alpha_y \| \beta_z)}^{\gamma}; \alpha_x \xrightarrow{r*} \overbrace{(\alpha_y; \alpha_x) \| \beta_z}^{\gamma}$$

$$\overbrace{(\alpha_y \| \beta_z)}^{\gamma}; \beta_x \xrightarrow{r*} \overbrace{\alpha_y \| (\beta z; \beta x)}^{\gamma}$$

$$\overbrace{(\alpha_x \| \beta_x)}^{\gamma}; \overbrace{(\alpha_y \| \beta_y)}^{\gamma} \xrightarrow{r*} \overbrace{(\alpha_x; \alpha_y) \| (\beta x; \beta y)}^{\gamma}$$

Lemma 3.1. $D(P1 \| P2)$ contains at least one S_i in a γ form.

Proof. $D(P1 \| P2)$ is a time diagram and the second condition of Def. 2.3 implies that there exists $e_1 \in P1$, $e_2 \in P2$ such that $e_1, e_2 \in S_i$. Clearly this S_i is in γ form. □

Definition 3.3. The roll-back process consists of repeating applications of the roll-back step on $D(P1 \| P2)$ until no further steps are possible.

Lemma 3.2. The roll-back process reduces $D(P1 \| P2)$ into a single γ form ($D(P1 \| P2) \xrightarrow{r*} \gamma$).

Proof. Lemma 3.1 yields that there is at least one γ expression in $D(P1 \| P2)$ which we can use in the roll-back process. Parentheses can be ignored while performing the roll-back process. Initially $D(R)$ is an expression with full associativity and commutativity (see Def. 2.5), hence parentheses can be placed in any desired order. Now every application of the roll-back process maintains an overall structure of $;E$ and hence parentheses can be further ignored. Note that every application of the roll-back step increases the overall length of the sum of gamma sub programs by at least one. Therefore this reproduction system terminates in a program of the form γ. □

So far we have found a derivation (the roll-back process) which changes $D(P1 \| P2)$ to a γ form program. It still remains to be shown that this γ form program consists of the time diagrams of $P1$ and $P2$ $(D(P1 \| P2) \xrightarrow{r*} \gamma = D(P1) \| D(P2))$.

Lemma 3.3. Let $\gamma i = (\alpha_1;...;\alpha_n) \| (\beta_1;...;\beta m)$ be the outcome of the roll-back process of $D(P1 \| P2)$ then $(\alpha_1;...;\alpha_n) \in T(P1)$ and the same for $(\beta_1;...;\beta_m)$.

Proof. The proof claim uses a graph representation of programs, where the nodes are the atomic instructions and an edge corresponds to the partial relation *before*. In the following, a program and its graph representation will have the same denotation. Let a "mixed-edge" in $D(P1 \| P2)$ be an edge between a vertex in $P1$ and a vertex in $P2$. In Lemma 3.4 it is shown that by removing all mixed-edges from $D(P1 \| P2)$ we obtain a separate time diagram for $P1$ and a separate time diagram for $P2$ $(D(P1 \| P2) \xmapsto{del.mix.adj.} D(P1) \| D(P2)$. Now Lemmas 3.5 and 3.6 show that the roll-back process does exactly that (i.e., removes *all* mixed-edges and *no* other edge). Hence the claim follows. □

Lemma 3.4. Let H be a projection graph of $D(P1 \| P2)$ onto $P1$ (wherein the vertices belong to $P1$ and an edge $(a, b) \in H$ iff $a, b \in P1$) then $H \in T(P1)$.

Proof. Let $S_1;...;S_k$ denote the states of $D(P1 \| P2)$. For each S_i let z_i denote the set of vertices in S_i that belongs to $P2$. Clearly H contains all atomic instructions of $P1$. If $s_i = S_i \backslash z_i$, then it is sufficient to show that the non empty s_i's are states of $P1$. $(s_1;...;s_k \in T(P1))$. Note that if $P1$ is not empty there is at least one s_i which is not empty. Let $a, b \in P1$ be such that a is before b in $D(P1 \| P2)$, then using the third condition of the time diagram there exists i, j where $i<j$ such that $a \in S_i \wedge b \in S_j$. Since the projection process does not remove a, b or the edge between them, $a \in s_i$ and $b \in s_j$. Hence, regarding $s_1;...;s_k$, the third condition of Def. 2.3 is met.

W.o.l.g. the second condition of Def. 2.3 yields that $\forall(E_a \| E_b) \in P1$ there exists $e_a \in E_a$, $e_b \in E_b$ such that $e_a, e_b \in S_i$. Thus $e_a, e_b \in s_i$ and the second condition is also valid for H. □

Lemma 3.5. Every step in the roll-back process removes mixed edges from $D(P1 \| P2)$ and *only* mixed edges.

Proof. Immediately follows from the observation that in every case of the roll-back step only mixed edges are removed. Consider for example:

$$\overbrace{(\alpha_x \| \beta_x)}^{\gamma};\ \overbrace{(\alpha_y \| \beta_y)}^{\gamma} \xrightarrow{r*} \overbrace{(\alpha_x; \alpha_y) \| (\beta x; \beta y)}^{\gamma}$$

Here only the mixed edges between a_x and β_y, β_x and α_y are removed. □

Lemma 3.6. At the end of the roll-back process all mixed edges are removed.

Proof. All mixed-edges result from programs of the form $E1; E2$ (see Def. 2.3). The final result of the roll-back process is a program in the form $\gamma = (\alpha_1;...;\alpha_n) \| (\beta_1;...;\beta_n)$. Therefore there are no mixed edges in the final result of the roll-back process. □

3.2. Proving that the Maximal $\xrightarrow{f*}$ Yields Only Time Diagrams

Theorem 3.3. Every member of R^* is in normal form, and represents a time diagram of R ($R^* \subset T(R)$).

The proof has two stages. First it is shown that every final derivation $R \xrightarrow{f*} R'$ must terminate in a normal form program (Lemma 3.7). The structure of the normal form program is then used to show that R' satisfies the three conditions of the time-diagram definition (Def. 2.3).

Definition 3.4. Let $m(R) = \sum_{(X \| Y) \in R} \#\{';' \text{ in } (X \| Y)\}$ be the sum of ';' in all the sub expressions of the form $(X \| Y)$.

Lemma 3.7. For the system defined by $\xrightarrow{f*}$, $m(R)$ is a descending function, i.e., $m(R) > m(P)$ where $R \xrightarrow{f*} P$. Moreover, $m(R) = 0$ iff R is in normal form. Therefore any derivation $R \xrightarrow{f*} P$ terminates in a normal form.

Proof. Let R' denote a program obtained from R by applying the time division axiom on R ($R \xrightarrow{t} R'$), and let $\#R$ denote the number of ';'s in R. The proof shows that $m(R') < m(R)$ for all cases of applying the axiom:

Case $(P1; P2) \| P3 \xrightarrow{t'} P1; (P2 \| P3)$: Evaluating $m(R)$ yields:

$$\begin{aligned} m((P1; P2) \| P3) &= \#(P1; P2) + \#P3 + m(P1; P2) + m(P3) \\ &= \#P1 + \#P2 + 1 + \#P3 + m(P1) + m(P2) + m(P3) \\ &> \#P2 + \#P3 + m(P1) + m(P2) + m(P3) \\ &= m(P1) + m(P2 \| P3) = m(P1 \colon (P2 \| P3)) \end{aligned}$$

Case $(P1; P2) \| P3 \xrightarrow{t'} (P1 \| P3); P2$: is symmetric to the previous case, hence follows in the same way.

Case $(P1; P2) \| (P3; P4) \xrightarrow{t'} (P1; P3) \| (P2; P4)$: Evaluating $m(R)$ for this case yields:

$$\begin{aligned} m((P1; P2) \| (P3; P4)) &= \#(P1; P2) + \#(P3; P4) \\ &\quad + m(P1; P2) + m(P3; P4) \\ &= \#P1 + \#P2 + \#P3 + \#P4 + 2 \\ &\quad + m(P1) + m(P2) + m(P3) + m(P4) \\ &> \#P1 + \#P2 + \#P3 + \#P4 \\ &\quad + m(P1) + m(P2) + m(P3) + m(P4) \\ &= m(P1 \| P3) + m(P2 \| P4) \\ &= m((P1 \| P3); (P2 \| P4)) \end{aligned}$$

Clearly $m(R) = 0$ iff R is in normal form, since all '$\|$' operate between atomic instruction only. In addition, if $m(R) > 0$ then there exists at least one sub expression $X \| Y$ for which either $\#X > 0$ or $\#Y > 0$, and $\xrightarrow{t'}$ can be applied once more. Hence, since when $m(R) > 0$, one can still apply $R \xrightarrow{t'} R'$ and $m(R')$ decreases, the derivation must terminate in $m(R') = 0$, i.e., with a normal form. □

Lemma 3.8. Let R' be a program in a normal form obtained by $R \xrightarrow{t^*} R'$, then R' is a time diagram of R.

Proof. Let: $;_j$ be a mark of the j'th ';' in R and $(X ;_j Y)$ denote its "surroundings." Lemma 3.7 yields that R' is in normal form, therefore $R' = S_1; \ldots; S_k$ where $S_i = \|E$. We will show that R' satisfies the time diagram conditions of Def. 2.3. Clearly each S_i is not empty. Now two properties are preserved in every application of the time division axiom:

Forward Preservation. If e_1 is *before* e_2 in R, then e_1 is *before* e_2 in P where $R \xrightarrow{t'} P$.

Parallel Preservation. Let $(X \| Y) \in P$ where $R \xrightarrow{t'} P$, then there is $(X_l \| Y_r) \in R$ such that $X_l \subset X$ and $Y_r \subset Y$ and both X_l, Y_r are not empty.

Let us assume that the forward preservation property holds for every application of $\xrightarrow{t'}$. Then if e_1 is *before* e_2 in R, then e_1 is *before* e_2 is R' where $R \xrightarrow{f^*} R'$. However, $R' = S_1;...;S_k$, $S_i = \|E$ thus, e_1 *before* $e_2 \in R$ implies that $e_2 \in S_j \wedge e_1 \in S_i$ where $j < i$, and the third condition of Def. 2.3 holds.

Let $(E_a \| E_b) \in R$ and assume that the parallel preservation property holds. Then by induction on the derivation $\xrightarrow{f^*}$, there exists $E'_a \in E_a$, $E'_b \in E_b$ such that $(E'_a \| E'_b) \in R'$. R' is in normal form and every $\|E \in R'$ fits to some S_i, hence $(E'_a \| E'_b) \in S_i$ and the second condition holds.

It remains to be shown that the forward and the parallel preservation properties are valid: For the case of $(A; B) \| (C; D) \xrightarrow{t'} (A \| C); (B \| D)$, the forward preservation holds since after the derivation still both A is *before* B and C is *before* D. For the parallel preservation, if $(X \| Y)$ belongs to either A, B, C, or D then the property trivially holds. Otherwise $X = (A; B)$ and $Y = (C; D)$ and the choice $X_l = A$ and $Y_r = C$ satisfies the parallel preservation property. In a similar way both properties are valid for the two other cases of applying the time division axiom, i.e., $A \| (B; C) \xrightarrow{t'} (A \| B); C$ and $A \| (B; C) \xrightarrow{t'} B;_j (A \| C)$ both satisfy the properties. □

4. SERIALIZATION AND SELF-SIMULATION IN PARALLEL MACHINES

The main result of the previous section is the notion of equivalence between an axiom based execution (see Def. 2.5) and execution based on the relation order induced by the conditions of Def. 2.3. This equivalence lies at the core of a formal understanding of parallel execution or time models. It actually defines a class of parallel machines (discrete machines) for which this equivalence is true. In this section we further pursue this equivalence by introducing new axioms, and verify their counterpart order relation conditions. Adding new axioms restricts the class of discrete parallel machines and forms new classes. Two additional classes are of interest and form a natural classification of parallel machines:

V1: Fully serializeable execution class- All discrete machines such that any parallel execution is equivalent to any sequential execution of the same program.

V2: Partially serializeable execution class- All discrete machines such that any parallel execution is equivalent to some order of sequential execution but not to all possible orders.

Note that both $V1$ and $V2$ are not equivalent to the interleaving time model. They still include a requirement for friction, which is expressed by adjacency of instructions in consecutive states rather than by true concurrent execution (see exact definitions next).

4.1. Axioms versus Time Condition of the V1 Class

In order to construct a twofold structure for $V1$, i.e., axioms versus time diagram conditions, we use the following definitions. The axioms for $V1$ are the three original ones plus a new axiom (called "the full serialization axiom" (fs)).

Definition 4.1. The fs-axiom is a *one direction axiom*, indicating that for any two atomic instructions a, b:

$$a \| b \Rightarrow a; b \wedge a \| b \Rightarrow b; a$$

The new conditions for time diagrams are as follow:

Definition 4.2. A time diagram $D^{V1}(R)$ for the execution of a program R by a machine in $V1$ is a division of the atomic instructions of R into steps $S_1, \ldots, S_k$ such that:

1. Each S_i contains one instruction.
2. If e_a is "before" e_b in R then there are $j < i$ such that $e_a \in S_j$ and $e_b \in S_i$.
3. $\forall (E_a \| E_b) \in R$ there exists $e_a \in E_a$, $e_b \in E_b$ such that $e_a; e_b \in D^{V1}(R)$ or $e_b; e_a \in D(R)$.

Intuitively, these conditions allow all possible arrangements of the instructions which are not in *before* relation to one another. Moreover the parallel friction condition is transformed to a sequential friction, thus maintaining the original interpretation of the '$\|$' operation.

Definition 4.3. Let

$R \xrightarrow{f*} R'$ be a nonempty derivation of R using only the $V1$ axiom until R' can not be reduced any further.

$R \xrightarrow{f*} R'$ be a nonempty derivation of R using the usual axioms until R' is in normal form, Theorem 3.3.

$R \xrightarrow{(f+s)*} R'$ be a final derivation of R using the fs-axiom plus the usual ones, until no further application is possible.

R^{V1*} be the set of all programs resulting from $\xrightarrow{(f+s)^*}$ which can not be further reduced.

$T^{V1}(R)$ the set of all legal time diagrams of R satisfying the time conditions of Def. 4.2.

In the following we will identify the program $R = e_1;...; e_n$ with the time diagram $S_i = e_i$.

Theorem 4.1. Under the above definitions $R^{V1*} = T^{V1}(R)$.

Proof. The first direction shows that $R^{V1*} \subset T^{V1}(R)$. Let $P \in R^{V1*}$, by Lemma 4.1 $P = e_1;...; e_n$. Now set $S_i = e_i$, it remains only to verify that P satisfies the conditions of Def. 4.2. Now, if e_i *before* e_j in R then e_i *before* e_j in P. This can be shown to be true, using a simple induction argument on the derivation $\xrightarrow{(f+s)^*}$ such that if e_a is before e_b then e_a is still before e_b after any application of the $V1$-axioms.

Let $R \xrightarrow{f} R'$ denote one application of the regular axioms (Def. 2.5). If $(E_a \| E_b) \in R$ then there are non empty $E_{a'} \in E_a$ and $E_{b'} \in E_b$ such that $(E_{a'} \| E_{b'}) \in R'$. This can be easily verified by checking all possible cases of $\xrightarrow{f}$. Clearly every '$\|$' which is transformed to ';' by the *fs*-axiom creates a sequential friction between some $(E_a \| E_b) \in R$. Since there are no '$\|$' in $P \in R^{V1*}$ all $(E_a \| E_b) \in R$ have a sequential friction in P, and the conditions of Def. 4.2 hold and $R^{V1*} \subset T^{V1}$.

The second direction uses the following argument: according to Lemma 4.2, for a given $D \in T^{V1}(R)$, one can find $P \in T(R)$ such that there is a derivation $P \xrightarrow{(f+s)^*} D$. According to Theorem 3.2 there is a $R \xrightarrow{f*} P$, hence there is a $R \xrightarrow{(f+s)^*} D$ or $D \in R^{V1*}$. □

The replacement process (to be described next) actually shows that a final derivation in $V1$ can be always divided into two phases: first a regular time diagram is obtained, then using the $V1$-axiom, it is transformed into the final form:

Corollary 4.1. Any mixed derivation which includes $\xrightarrow{f*}$ and the *fs*-axiom is equivalent to first applying $\xrightarrow{f*}$ and then applying $\xrightarrow{(f+s)^*}$ on the $\|E$ left in the result $\xrightarrow{(f+s)^*} = \xrightarrow{f*} \xrightarrow{(f+s)^*}$.

Lemma 4.1. If $P \in R^{V1*}$ then P is in the form $P = e_1;...; e_n$.

Proof. By negation, assume that P contains at least one sub expression of the form $(X \| Y)$. Three cases are possible:

- X, Y are atomic instructions, hence the fs-axiom can be applied.
- X or Y are non-atomic programs which do not contain a ';', then again the *fs-axiom* can be applied.

- Either X or Y contains a ';', hence the original axioms can be further applied.

If the axioms can be further applied, then $P \notin R^{V1*}$, which is a contradiction. □

Lemma 4.2. For any $D = e_1;...;e_n \in T^{V1}(R)$ there is a ';' to $\|$ replacement processes $P = \text{replacement}(D)$ such that $P \in T(R)$. Moreover, there is a derivation to the original time diagram $P \xrightarrow{(f+s)^*} D$.

Proof. Define a replacement process that yields $S_1;...;S_k$ which satisfies the conditions of Def. 2.3. The replacement process produces S_i by replacing adjacent ';'s to $\|s$. Hence by Theorem 3.2 this defines $P = S_1;...;S_k$ where $S_j = e_{i_1} \| ... \| e_{i_j}$ and $P \in T(R)$.

The replacement process is described in Fig. 3.

SS is a legal time diagram of R, since it preserves the conditions of Def. 2.3, as follows:

- Every $S_i \in SS$ is not empty because it contains at least one e_i.
- According to Def. 4.2 $\forall (E_a \| E_b) \in R$ there exists $e_a \in E_a$, $e_b \in E_b$ such that $e_a; e_b \in D(R)$ or $e_b; e_a \in D(R)$. Clearly e_a is not before e_b and the replacement process will replace $e_a; eb$ by $e_a \| e_b$ thus pursuing the friction condition of Def. 2.3.
- In the replacement process, a new state starts with an instruction e_i which has some instruction e_j *before* e_i where e_j is in the current state (see SS in Fig. 4). If e_j *before* e_i and e_j belongs to a previous state there is no need to start a new state since e_i is already separated from e_j by a ';'. Hence the third condition of Def. 2.3 is met.

```
INPUT D = e1;...;en ∈ T^{V1}(R) and R;
OUTPUT SS = S1;...;Sk ∈ T(R);
S' = e1;     /* current state */
SS = ∅;      /* list of all states */
FOR i = 2...n DO {
    IF(∃ ej ∈ S ∧ ej before ei) THEN {
        SS = SS + S';    /* add the new state */
        S' = {ei};    /* start a new state*/
    } ELSE S' = S' + ei;    /* replace ';' by a '||' */
}
SS = SS + S';    /* add the last state */
```

Fig. 3. Replacement process transforming $D^{V1}(R)$ to $D(R)$.

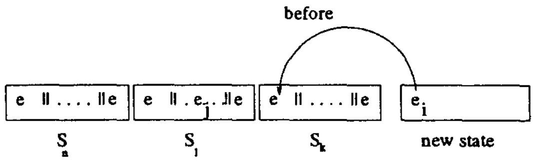


Fig. 4. Creating new states in the replacement process.

Lemma 4.3 completes the proof by showing that there is a derivation $P \xrightarrow{(f+s)^*} D$ which re-replaces every $\|$ back to a ';'. □

Lemma 4.3. For every $;E = e_1;...; e_k \in D^{V1}(R)$ there is a derivation $\|E \xrightarrow{(f+s)^*} ;E$ such that $\|E = e_1 \| ... \| e_k$ (in fact, to any permutation of $\langle e_1;...; e_k \rangle$).

Proof. By induction on the length of $\|E(k)$:

For $k = 1$, the claim trivially holds, also by using the fs-axiom it holds for $k = 2$.

For $k > 2$, assume that $e_1 \| ... \| e_k \xrightarrow{(f+s)^*} e_1;...; e_k$ hence:

$$\begin{aligned} e_1 \| ... \| e_{k-1} \| e_k \| e_{k+1} &\xrightarrow{(f+s)^*} ((e_1;...; e_{k-1}); e_k) \| e_{k+1} \\ &\xrightarrow{f*} ((e_1;...; e_{k-1}); (e_k \| e_{k+1})) \\ &\xrightarrow{fs-ax} ((e_1;...; e_{k-1}); e_k; e_{k+1})) \\ &= e_1;...; e_{k-1}; e_k; e_{k+1} \end{aligned}$$

□

Note that changing parentheses in $\|e_i$ and $;E$ expressions is a valid step according to Lemma 2.1.

4.2. Axioms versus Time Condition in the $V2$ Class

As in $V1$, the axioms for $V2$ are the original three plus a new axiom (called "the partial serialization axiom"):

Definition 4.4. The ps-axiom *differs* from one program to another, indicating that for any two atomic instructions $a, b \in R$ one of the following is true:

either $(a \| b \Rightarrow a; b \wedge \neg(a \| b \Rightarrow b; a))$ or $(a \| b \Rightarrow b; a \wedge \neg(a \| b \Rightarrow a; b))$

Note that the ps-axiom is a list which describes "directions" in transforming every expression of the form $e_i \| e_j$ to $e_i; e_j$ (justifying the notation ps-axiom(R)). This definition is reflected in the new conditions for time diagrams in $V2$:

Definition 4.5. A time diagram $D^{V2}(R)$ for the execution of a program R by a machine in $V2$ is a division of the atomic instructions of R into steps $S_1, ..., S_k$ such that:

1. Each S_i, contains one instruction.
2. $\forall (E_a \| E_b) \in R$ there exists $e_a \in E_a$, $e_b \in E_b$ such that $e_a; e_b \in D(R)$ or $e_b; e_a \in D^{V2}(R)$.
3. If e_a is "before" e_b in R then there are i and j such that $i < j$ and $e_a \in S_j$ and $e_b \in S_i$.
4. For all pairs of atomic instructions, there is a pre-defined list $LV2(R) = \langle ... \langle e_i; e_j \rangle ... \rangle$ which indicates a potential "before" relation. In addition, there should be a matching relation between some time diagram $D(R) \in V$ and a candidate P for a time diagram in $V2$. $P \in T^{V2}$ if the identity matching between instructions of $D(R)$ and the instructions of P preserves the order on the states of $D(R)$. I.e. P can be constructed by inducing order on the instructions of the states of $D(R)$. Hence, the third condition for $P \in T^{V2}(R)$ states that for all $e_i \in R$:
   - Let $S(e_i) = e_1 \ldots e_n$ be the state in a $D(R)$ that matches P, such that $e_i \in S(e_i)$.
   - There is at least one instruction $e_j \in S(e_i)$ such that either $S(e_i) = e_i$ or:

$$\langle e_i; e_j \rangle \in LV2(R) \Rightarrow e_i \text{ before } e_j \text{ in } P$$

$$\langle e_j; e_i \rangle \in LV2(R) \Rightarrow e_j \text{ before } e_i \text{ in } P$$

Intuitively, the last condition excludes all the time diagrams in $T^{V1}(R)$ which violate all the directions in $LV2(R)$. For instance, choose the ps-axiom to match $LV2$, .thus showing that there exist derivations which violate some, but not all of the directions. In particular, let $R = a \| b \| c$ and $LV2(R) = \langle \langle a; b \rangle \langle a; c \rangle \langle b; c \rangle \rangle$, then using the $V2$-axioms it is possible to derive:

$$a \| b \| c \rightarrow a \| (b; c) \rightarrow b; (a \| c) \rightarrow b; a; c \Rightarrow \text{contradiction to } \langle a; b \rangle \in LV2(R)$$

$$a \| b \| c \rightarrow c \| (a; b) \rightarrow (a \| c); b \rightarrow a; c; b \Rightarrow \text{acontradiction to } \langle b; c \rangle \in LV2(R)$$

However, it is not possible to obtain $a \| b \| c \rightarrow c; b; a$, a time diagram which violates all the directions in $LV2$.

The main equivalence theorem for $V2$ can be stated and proved. The following notations: $\xrightarrow{(f+ps)^*}$, R^{V2*}, $V^{V2}(R)$ and $T^{v2}(R)$ are used in the same way as in Def. 4.3.

Theorem 4.2. Under the above definitions if $LV2(R) = ps$-axiom(R) (i.e., the directions in $LV2(R)$ are the same as those given in the ps-axiom list) then $R^{V2*} = T^{V2}(R)$.

Proof. The proof follows the same path as that of Theorem 4.1. Lema 4.1 is also valid for $V2$ (just replace $V1$ by $V2$ and fs-axiom by ps-axiom). The first step is to prove that $R^{V2*} \subset T^{V2}(R)$. Let $P \in R^{V2*}$, according to Lemma 4.1 $P = e_1;...; e_n$. Set $S_i = e_i$, it remains to verify that P satisfies the conditions of Def. 4.5. As in the $V1$ case, an induction on the axioms can be used to show that the second and third conditions of Def. 4.5 are preserved by any axiom application.

The fourth condition is also preserved through the following argument: Any derivation in $V2$ is also a derivation in $V1$, and therefore (using cf. 4.1) can be broken into two stages $R \xrightarrow{(f+ps)^*} P \equiv R \xrightarrow{f^*} D(R) \xrightarrow{(f+ps)^*} P$. Clearly $D(R)$ matches P in the sense of condition 4.5, therefore requiring us to show that the third condition (of Def. 4.5) is preserved in every $e_i \in R$ and in every state $S(e_i) \in D(R)$.

If $S(e_i) = e_i$ the claim holds. Otherwise, every application of the ps-axiom $e_i \| e_b \xrightarrow{ps} e_i; e_b$ satisfies the third condition for the pair $\langle e_i; e_b \rangle \in LV2(R)$. This last claim can be proven by an induction argument showing that no further application of the axioms can reverse the order between e_i and e_b. Thus it suffices to prove that the ps-axiom was applied on every $e_k \in \|E \in D(R)$.

By negation, assume that in $\|E \in D(R)$ there is an instruction e_k on which the ps-axiom was not applied. Since e_k is separated by a $\|$ from the rest of the instructions in $\|E$ and no ps-axiom was applied on e_k, then this $\|$ should have "survived" and continued separating e_k in all applications of the regular axioms (Def. 2.5). This claim can be easily verified by induction on the axiom applications, e.g., $(A; B) \| e_k \rightarrow (A \| e_k); B$, showing that the $\|$ still separates between e_k and A. This contradicts the fact that R' should not contain any $\|$s according to Lemma 4.1. Therefore $P \in T^{V1}(R)$.

The other direction of the proof (to show that $T^{V2}(R) \subset R^{V2*}$) follows the same structure as that of the proof for $V1$. The replacement process of Lemma 4.2 remains the same as in $V2$, since it uses only the "*before*" relations in R. In order to prove Lemma 4.3 for $\xrightarrow{(f+ps)^*}$, it is sufficient to re-prove Lemma 4.3 as follows:

Lemma 4.4. For every $\|E = (e_1 \| ... \| e_k) \in D(R)$ such that $D(R)$ matches (in the sense of Def. 4.5) $D^{V2}(R)$ there is a derivation $\|E \xrightarrow{(f+ps)^*} ;E$ such that $;E = e_1;...; e_k \in D^{V2}(R)$.

Proof. By induction on k the length of $\|E$:

For $k=1$- the claim trivially holds.

For $k=2$- the ps-axiom hold as well.

For $k>2$- assume that $e_1 \| ... \| e_k \xrightarrow{(f+ps)^*} e_1 ;...; e_k$. Let $\|e = (e_1 \| ... \| e_j \| ... \| e_k) \| e_{k+1} \in D(R)$. The fourth condition of Def. 4.5 guarantees that there is some $e_j \in (e_1 \| ... \| e_k)$ such that e_j is before e_{k+1} in $(e_1 ;...; e_j ;...; e_k ; e_{k+1}) \in D^{V2}(R)$, hence:

$$(e_1 \| ... \| e_j \| ... \| e_k) \| e_{k+1} = (e_1 \| ... \| e_{j-1} \| e_{j+1} \| ... \| e_k) \| (e_j \| e_{k+1})$$

$$\xrightarrow{ps-ax: (e_j \| e_k)} (e_1 \| ... \| e_{j-1} \| e_{j+1} \| ... \| e_k) \| (e_j ; e_{k+1})$$

$$\xrightarrow{f*} ((e_1 \| ... \| e_{j-1} \| e_{j+1} \| ... \| e_k) \| e_j); e_{k+1}$$

$$\xrightarrow{\text{induction}} ((e_1 ;...; e_{j-1} ; e_j ; e_{j+1} ;...; e_k); e_{k+1}) \qquad \square$$

5. A CORRECTION NOTION FOR PARALLEL PROGRAMS

The proposed correction notion is built upon Th. 3.1, which states that the time diagrams of a given program $T(R)$ can be derived using the axioms based derivations. This defines a parallel execution model which actually ignores, semantic knowledge regarding the execution, such as infinite loops, dead-locks, and forced termination. Thus the set of all possible time diagrams of a specific program R is actually a subset of $T(R)$, containing all time diagrams which do not violate the semantics of R. For example, consider the following parallel program R in C style:

$$(x=0; (x=1; \|(\text{while}(x==0); \text{print}(x)))$$

(Note that as explained in the introduction, a complex code segment (such as *'while*$(x==0)$;') can be regarded as an atomic instruction.) Clearly, *'print*(x)' and '$x=1$ can not be executed at the same state, even though it is allowed by the V time model. Similarly, '$x=1$' and *'while*$(x==0)$' must be executed in the same state, forming a true concurrency condition, which is not required by V.

The notion for correctness is based on the ability to express the semantics of a parallel execution of a program R in a time model V, as a restriction of V, to match R's semantics. Only the part of $T(R)$ which matches the semantics of R (left unspecified) should be allowed by the chosen Ct conditions. The choice of Ct is left undefined, making Ct an open slot for adding semantic knowledge about R. Once Ct is chosen, a suitable Cx must de devised such that $\langle R, Cx, Ct \rangle$ is a time model for R.

Such a notion of correctness avoids the need for specifying a precise semantics for R, using formalisms like petri-nets, ccs and logic formulas.[16] This notion is therefore correct up to the ability to express semantics by Ct conditions. If the user has failed to do so (i.e., his or hers choice of Ct contradicts a possible formal semantics), a correctness proof might be false. The advantage of such a notion stems from its ability to supply proofs against all possible schedulings or executions, without computing all of them.

A time diagram of a program represents a history of some parallel execution. Thus all its states (except the last) should terminate, making the time diagram consistent with the semantics (what ever it may be). The last state need not halt, since the execution may end in an infinite loop.

This intuition leads to the following definition:

Definition 5.1. For a given program R executed by a parallel machine M, which belongs to one of the classes V, $V1$, or $V2$, let:

Ct- be a finite set of conditions which restricts all time diagrams of R, such that:

1. $T/Ct(R) \subseteq T(R)$.
2. Let $D(R) = S_1;...;S'_n \in T/Ct(R)$ then $S_1;...;S_{n-1}$ *halts* when executed by M.
3. Let $D(R) = S'_1;...;S'_n \in T(R) - T/Ct(R)$, then $S'_1;...;S'_{n-1}$ when executed by M does not halt. All time diagrams which violate $D(R) \in T(R) - T/Ct(R)$ "contradict any possible" execution of R.

Where $T/Ct(R)$ denote the set of all time diagrams of R which preserve Ct.

Cx- be a set of derivation rules, such that $R/Cx^* \subseteq R^*$, where R/Cx^* denotes the set of programs obtained by a final derivation which preserves Cx.

Induction system The triple $\langle R, Cx, Ct \rangle$ is an induction system if $T/Ct(R) = R/Cx^*$.

Correctness R is correct in respect to a condition C, if there is an μinduction system $\langle R, Cx, Ct \rangle$, such that every $D(R) \in T/Ct$ halts and satisfies C.

Once an induction system has been devised for R, it can be used to prove desired properties of R, e.g.:

- Assume that the initial program R satisfies some property C, and $\xrightarrow{Cx}$ preserves this property, then so does $T/Ct(R)$.

- Assume that some time diagram $D/Ct(R)$ satisfies some property, and $\xrightarrow{Cx}$ preserves this property both backwards and forwards, then so does $T/Ct(R)$.

Note that finding Cx which generates all $T/Ct(R)$ might be difficult, since Cx reflects "on-line" conditions while Ct characterizes the final result. However, if the program halts, then clearly there are some scheduling rules by which it was executed. Thus if the program is correct, then Cx exists.

The choice of Ct as the *open slot* rather than Cx is natural. Clearly it is easier to characterize a subset $(T/Ct(R) \in T(R))$ by conditions than to devise a general rule for constructing this subset.

5.1. An Induction System for Parallel Loop Programs

This section presents an induction system for a special category of programs, namely *parallel loop programs* executed in the V time model. We are interested in a synchronous execution of these programs, i.e., the ith state in every time diagram contains all the ith instructions from every loop, thus all loops are actually executed synchronously. Note that the V time model is an asynchronous model and allows all possible interleavings of instructions from different loops. Thus we seek to find an induction system for parallel loop programs characterizing (via Ct and Cx) the desired synchronous execution.

Definition 5.2. Let a loop of length l be a sequential sequence of l instructions. A parallel loop program R_k^l consists of k parallel loops:

$$R_k^l = (x_1^1; ...; x_1^l) \parallel ... \parallel (x_k^1; ...; x_k^l)$$

Definition 5.3. The length of a program $[R]$ is recursively defined as follows:

$$[R] = \begin{cases} 1 & \text{if } R \text{ is an atom} \\ [A] + [B] & \text{if } R = (A; B) \\ \max([A], [B]) & \text{if } R = (A \parallel B) \end{cases}$$

Definition 5.4. An $S-J$ derivation $\xrightarrow{s-j}$ is a "symmetric" application of the time derivation axiom such that:

$$((A; B) \parallel (C; D)) \xrightarrow{s-j} ((A \parallel C); (B \parallel D)) \qquad \text{if } [A] = [C], \quad [B] = [D]$$

The proposed induction system uses $S-J$ as Cx, and its Ct will simply restrict all time diagrams to having l states, and containing the regular conditions of Def. 2.3. In addition to the proof showing that $\langle R_k^l, S-J, [D(R_k^l)]=l\rangle$ is an induction system, we show that there is only one possible time diagram in T/Ct satisfying synchronous execution of R_k^l. Intuitively, the induction system proves that if a parallel execution deviates from $S-J$ the result (any time diagram) will no longer be synchronous and will have more than l states. The difficulty, or the non-triviality of such a proof stems from the need to prove that any deviation from $S-J$ will end by a non synchronous execution (i.e., $S-J$ is not only sufficient but necessary).

Lemma 5.1. If $R_k^l \xrightarrow{s-j*} R'$ then $[R']=l$ and for every $(A \parallel B) \in R'$ the following condition holds $[(A \parallel B)]=[A]=[B]$.

Proof. By definition $[R_k^l]=l$, assume by induction that $[R'']=l$, where $R'' \xrightarrow{s-j} R'$. Hence there exists $E''=(A; B) \parallel (C; D) \in R''$ such that $[A]=[C]$, $[B]=[D]$ and $E''=((A; B) \parallel (C; D)) \xrightarrow{s-j} E'=((A \parallel C); (B \parallel D))$. Using the assumption $[A]=[C]$ and $[B]=[D]$ yields that

$$\begin{aligned}[E'] &= [((A \parallel C); (B \parallel D))] = \max([A], [C]) + \max([B], [D]) \\ &= [A]+[B]=[C]+[D]=\max([A]+[B], [C]+[D])=[E'']\end{aligned}$$

Hence, by the induction hypothesis $[R'']=[R']=l$. □

Lemma 5.2. Let $G(R)$ denote the graph representation of a program R, such that there is a direct edge from a to b if a and b are atomic instructions of R and a is in "before" relation to b. Clearly $G(R)$ is a directed acyclic graph. Let $\mathscr{L}(\mathscr{R})$ be the set of all maximal paths in $G(R)$, i.e., there is a "last" node u in every path, such that there is no other node v satisfying u before v. Let L_0 be the maximal length in $\mathscr{L}$, then $[R]=L_0$ (i.e., the length of a program R is the length of the maximal path in $G(R)$).

Proof. By induction of the structure of R:

$[R]=1$, then $G(R)$ contains one node, and a maximal path of length one.

$R=(A; B)$, then $G(R)$ is formed by placing an edge between every node in A and B. Hence the longest path in $\mathscr{L}(R)$ is the longest path in $\mathscr{L}(A)$ joined by an edge to the longest path in $\mathscr{L}(B)$, and the claim follows.

$R=(A \parallel B)$, then $\mathscr{L}(R)=\mathscr{L}(A) \cup \mathscr{L}(B)$ and the claim follows. □

Lemma 5.3. If $R \xrightarrow{f} R'$ then $[R'] \geq [R]$.

Proof. By verifying cases of possible derivations.

Associativity or commutativity- By definition does not change the length. $E=((A;B)\|(C;D))\xrightarrow{f}E'=((A\|C);(B\|D))$, then

$$[E]=\max([A]+[B],[C]+[D])$$
$$\leqslant \max([A],[C])+\max([B],[D])=[E']$$

W.l.o.g. $E=((A;B)\|C)\xrightarrow{f}E'=(A;(B\|C))$, then

$$[E]=\max([A]+[B],[C])\leqslant [A]+\max([B],[C])=[E'] \qquad \square$$

Lemma 5.4. If $R_k^l\xrightarrow{s-j+}R''\xrightarrow{f}R'$ and $R''\xrightarrow{f}R'$ not an $S-J$ derivation, then $[R']>l$.

Proof. By verifying cases of possible derivations.

$E''=((A;B)\|(C;D))\xrightarrow{f}E'=((A\|C);(B\|D))$, then since $\xrightarrow{f}$ violates the $S-J$ condition, w.l.o.g. $[A]>[C]$, by Lemma 5.1 $[A]+[B]=[C]+[D]$, hence $[D]>[B]$. Now

$$[E']=\max([A],[C])+\max([B],[D])$$
$$>\max(([A]+[B]),([C]+[D]))=[E'']$$

W.l.o.g. $E''=((A;B)\|C)\xrightarrow{f}E'=(A;(B\|C))$, then since by Lemma 5.1 $[A]+[B]=[C]$

$$[E']=[A]+\max([B],[C])>[A]+[B]=[E''] \qquad \square$$

Definition 5.5. Let the "synchronous" normal form of R_k^l, denoted by $\Delta(R_k^l)$ be the program:

$$\Delta(R_k^l)=(x_1^1\|\dots\|x_k^1);\dots;(x_1^l\|\dots\|x_k^l)$$

Lemma 5.5. All normal forms of $D(R_k^l)$ different from $\Delta(R_k^l)$ have lengths greater than $[\Delta(R_k^l)]=l$.

Proof. By definition $[\Delta(R_k^l)]=l$, by Lemma 5.3 $[D(R_k^l)]\geqslant l$, hence by Lemma 5.2 (length of the maximal path in $G(R_k^l)$) the number of states in $D(R_k^l)\geqslant l$. Assume that $D(R_k^l)\neq\Delta(R_k^l)$; yet, $[D(R_k^l)]=[\Delta(R_k^l)]$. $G(R_k^l)$ contains k distinct paths of length l. The time diagram conditions of Def. 2.3 employs that the jth node of a path should be placed in the jth state of $D(R_k^l)$. Since there are exactly l states $D(R_k^l)=\Delta(R_k^l)$, a contradiction. $\square$

Lemma 5.6. There is an $S-J$ derivation such that $R_k^l\xrightarrow{s-j*}\Delta(R_k^l)$.

Proof. $\Delta(R_k^l)$ satisfies the conditions of Def. 2.3, hence $\Delta(R_k^l) \in T(R_k^l)$. Let $R_k^l \xrightarrow{s-j+} R' \xrightarrow{f} R'' \xrightarrow{f*} \Delta(R_k^l)$ where $R' \xrightarrow{f} R''$ is the first derivation different from $S-J$. By Lemma 5.1 $[R'] = l$ and by Lemma 5.4 $[R''] > l$, then by Lemma 5.3 $[\Delta(R_k^l)] > l$, which contradicts Lemma 5.5. □

Theorem 5.1. $T/ct(R_k^l) = R^{(S-J)*}(R_k^l) = \Delta(R_k^l)$.

Proof. By Lemma 5.5 $T/ct(R_k^l) = \Delta(R_k^l)$. By Lemma 5.6 $\Delta(R_k^l) \in R^{(s-j)*}$. Using Lemma 5.5 yields that every $D(R_k^l) \neq \Delta(R_k^l)$ has length $[D(R_k^l)] > l$. Finally, if $R_k^l \xrightarrow{s-j*} D(R_k^l)$ then by Lemma 5.1 $[D(R_k^l)] = l$, resulting in a contradiction. Therefore $D(R_k^l) \notin R^{(s-j)*}(R_k^l)$ and $R^{(s-j)*}(R_k^l) = \Delta(R_k^l)$ as well. □

5.2. Verification of a Specific Program

The induction system for R_k^l can be used to prove correctness (as defined in Def. 5.1) of a specific parallel program, such as the program in Fig. 5. A common structure in parallel programming with shared memory is *flag* synchronization. In this type of programming several processes "wait" for a flag to be changed. The program spawns $k+1$ processes all waiting for one activity to reset a flag, and all this is repeated $2l$ times in a loop. Assume that the user wants to avoid re-spawning $k+1$ activities. Thus, he needs to "nest" the outer loop inside the *spawn* statement. This problem is interesting in its own right; however, its solution leads to the complicated program of Fig. 5. This solution overcomes the problem of resetting the flag ($flag = 0$) by implementing two counters (*counta*, *countb*) through which the 13 process can determine when it is safe to set and reset the flag. The *faa* instructions (*fetch and add*[17,18]) are used to decrement the counter in parallel. Note that a "naive" nesting of the outer loop will cause infinite loops, as it might be that the 13 processes will terminate long before the rest.

There are only a few methods for verifying asynchronous programs which use *f&aa*.[19] A proof based on constructing an induction system is naturally considered, since *flag*() is "correct" only when all possible orders of execution (schedulings) of *flag*() halt.

Let $R_{flag} = (z; w)^l \| (x_1; y_1)^l \| \dots \| (x_k, y_k)^l$ be an abstraction of *flag*() such that z, w, x, y are mapped to statements as follows: $z = 7; 8; 9$, $w = 10; 11; 12$, $x_i = 14; 15$ and $y_i = 16; 17$ (i is the process id). We will also use the notation x_i^j to describe the jth iteration of x_i (sometimes referred to by x).

The proof for R^{flag} correctness (according to Def. 5.1) includes: determining Ct^{flag}, fixing Cx, proving that $\langle R^{flag}, Ct^{flag}, Cx \rangle$ is an induction

```
flag(k,l){
     INT flag,counta,countb;
     flag=0;
     counta=k;
     FOR ALL i = 0...k SPAWN PROCESS:
     { INT n,j;
         FOR(n=0;n<l;n++){
             IF(i==13){
                 flag = 1;
                 countb = k;
                 WHILE(counta > 0);
                 counta = k;
                 flag = 0;
                 WHILE(countb > 0);
             }ELSE {
                 WHILE(flag == 0);
                 faa(&counta,-1);
                 WHILE(flag == 1);
                 faa(&countb,-1);
             }
         }
     } EPAR
}
```

Fig. 5. "flag( )" a program demonstrating flag synchronization between processes.

system, and showing that any time diagram that violates Ct will not halt (in a state different from the last one).

For every $D/Ct^{flag}(R^{flag}) = S_1;...;S_n$, Ct^{flag} includes the following conditions:

Ct^V- this condition determines the time model V, $V1$, $V2$ to be used. Note that after the abstraction, x, y, w, z contains *busy waiting loops* with interdependencies, such that a true parallel execution model is needed. V, being the least restrictive of all three, is naturally considered. For example, the first state $S_1 \in D/Ct(R_{flag})$ will not terminate if it contains a single x, y, z or w instruction (i.e., it will be "stacked" in its busy-waiting loop).

Ct^z- for every $z \in S_j$, $j < n$ there is a sequence of kx instructions in states $S_{j-m} \ldots S_j$, $m < k$, which contains no y, w instructions.
Clearly each z can not terminate before its counter has been decreased to zero, which can happen only by executing kx instructions. A w executed between these k instructions will reset the counter back to k. A y executed between these 20 instructions will iterate forever since the flag must be set to 1 if the kx-instructions are to terminate. Let S^A, $A = x, y, z, w$ denote a state containing an x, y, z or w instruction, respectively. Hence the set of states between S^z and S^x contains only x instructions. The dual condition for w, Ct^w is obtained by replacing z by w and x by y respectively.

Ct^x indicates that every $x \in S_j$, $j < n$ has some $z \in S_q$, $q \leqslant j$ such that there is no y, w in these states $(S_q \ldots S_j)$.
Each x can not terminate before *flag* has been set to 1, which can happen only by executing z instructions. A w executed between S_q and S_j will reset the *flag* back to 0. A y executed between S_q and S_j, will iterate forever since the flag must be set to 1 if the x instructions ought to terminate. The dual condition for y, Ct^y is obtained by replacing x by y and z by w respectively.

Note that these conditions are valid for all states except the last one. This follows from the interpretation of a time diagram as a history of parallel execution. Thus all states, except the last one S_n, should have terminated, and therefore can not contain infinite loops. The termination of the last state of every time diagram of R_{flag} is equivalent to proving that R_{flag} halts no matter what scheduling took place.

The proof is based on the following claim:

Lemma 5.7. $T/Ct^{flag}(R^{flag}) = \Delta(R^{flag})$ and any time diagram of R_{flag} with more than $2l$ states violates Ct, before the last state.

Proof. Let $S^z \in D/Ct^{flag}$ denote a state containing z (same for S^x, S^y, S^w), so that any time diagram of R^{flag} has the following form:

$$D/Ct^{flag} = \ldots; S_1^z; \ldots; S_1^w; \ldots; S_2^z; \ldots; S_2^w; \ldots \ldots \ldots; S_l^z; \ldots; S_l^w; \ldots$$

Let $pre(S_i^z)$ denote all states between S_i^z and S_{i-1}^w including S_i^z (and $pre(S_j^w)$ respectively). Then according to Ct^z, for all S^z, $pre(S^z)$ contains k instructions of type x, and $pre(S^w)$ contains k instructions of type y except for the last S^w state. Since only $k \cdot l$ instructions of type x are distributed evenly among all $pre(S^z)$, then all $pre(S^w)$ do not contain x instructions. Thus, all $pre(S^z)$ contain no y instructions either. Hence, $pre(S_i^z) = x^j$; $(\overbrace{z \| x \| \ldots \| x}^{k-j})$; however, Ct^x requires all $x \in x^j$ to be executed after or in parallel to some z, so $pre(S_i^z) = z \| \overbrace{x \| \ldots \| x}^{k}$. Similarly $pre(S_i^w) = w \| \overbrace{y \| \ldots \| y}^{k}$, $i < l$ except the last state S_l^w which might violate this structure. Clearly, if S_l^w is not the last state, it has already terminated, and so $pre(S_l^w)$ contains k instructions of type y. This plus the previous observation contradicts the assumption that S_l^w is not the last state, yielding that $D/Ct^{flag}(R^{flag})$ can only be of the following form:

$$D/Ct^{flag}(R^{flag}) = \begin{array}{l} S_1^z = z \| \overbrace{x \| \ldots \| x}^{k};\ S_1^w = w \| \overbrace{y \| \ldots \| y}^{k} \| \overbrace{y \| \ldots \| y}^{f_1}; \\ \ldots\ldots\ldots\ldots;\ \ldots\ldots\ldots\ldots\ldots\ldots; \\ \ldots\ldots\ldots\ldots;\ \ldots\ldots\ldots\ldots\ldots\ldots; \\ S_{l-1}^z = z \| \overbrace{x \| \ldots \| x}^{k};\ S_{l-1}^w = w \| \overbrace{y \| \ldots \| y}^{k} \| \overbrace{y \| \ldots \| y}^{f_{l-1}}; \\ S_l^z = z \| \overbrace{x \| \ldots \| x}^{k};\ S_l^w = w \| \overbrace{y \| \ldots \| y}^{j} \| \overbrace{y \| \ldots \| y}^{f_l} \\ \text{such that } \sum_{i=1}^{l} f_i = k - j \end{array}$$

However, if $f_i > 0$ for $i < l$ then there are at least two y instructions parallel to one another which originally belonged to the same loop in $(x, y)^l \in R^{flag}$. This contradicts Ct^V, which requires all time diagrams to preserve the *before* relations. Finally we obtain that T/Ct^{flag} contains one time diagram of the form:

$$D/Ct^{flag}(R^{flag}) = z_1 \| \overbrace{x \| \ldots \| x}^{k};\ w_1 \| \overbrace{y \| \ldots \| y}^{k}; \ldots;\ z_l \| \overbrace{x \| \ldots \| x}^{k};\ w_l \| \overbrace{y \| \ldots \| y}^{k}$$

with exactly $2l$ states. If there are more than $2l$ states and no state can contain more than k instructions of type x (or y), then there are at least two states of type S^z, S^w that violate Ct^z or Ct^w. □

Theorem 5.2. R^{flag} is correct, in the sense of Def. 5.1.

Proof. Clearly R^{flag} is a parallel loop program such that $R_{flag} = R_{k+1}^{2l}$. Theorem 5.1 combined with Lemma 5.7 yield that $T/Ct^{flag}(R^{flag}) = \Delta(R^{flag}) = R^{(S-J)*}(R^{flag})$. Hence Ct^{flag} is equivalent to the general condition for synchronous execution such that $[D(R^{flag})] = 2l$ and $\langle R_{flag}, S-J, Ct^{flag} \rangle$ is an induction system. Moreover, by Lemma 5.5, any violation of $S-J$ yields a time diagram with more than $2l$ states, which by Lemma 5.7 violates Ct^{flag} is an inner state (a state different from the last one). Clearly, using Ct^{flag} definition, any violation of Ct^{flag} in an inner state will cause this state to be in an infinite loop. However, if every state satisfies Ct^{flag} then every state halts and so does $\Delta(R^{flag})$. Thus all conditions of Def. 5.1 are fulfilled. □

6. CONCLUSIONS

In this paper we study a model for parallel execution of parallel programs. Parallel programs have been defined to be expressions for expressing partial order relations of atomic instructions. It contains explicit $(X \| Y)$ or "incomparable" relation (indicating parallelism), and the usual $(X ; Y)$ relation (indicating sequentiality or $<$). The $\|$ relation has been interpreted as a weak requirement for true concurrency, namely that at least two instructions (one from X and one from Y) will be executed simultaneously.

Our goal is to develop a framework in which verification of a parallel program against all possible orders of execution (schedulings) can be realized. The proposed framework is based upon two observations:

- Sometimes, a compact representation for all possible execution orders can be devised.
- True concurrency[1] must be used when compound instructions are used instead of the original ones.

Three novel classes of parallel execution models have been defined, such that different degrees of the intended parallelism in $(X \| Y)$ must be preserved in every execution. It is assumed that verification of parallel programs is simplified when it is performed using these classes. In particular, two dual compact representations are used to characterize all execution orders of a parallel program in every class:

Ct- A set of conditions or relations between the program and all its execution orders.

Cx- A set of derivation rules from which one can construct all possible execution orders of a program.

A proof that shows equivalence between Ct and Cx is devised for every class of parallel execution. This equivalence is referred to as an induction system $(\langle R, Ct, Cx \rangle)$.

The execution of a specific program R is viewed as a sub-class with a specific induction system of its own. This induction system generates exactly all possible executions which agree with the semantic of R. Recall that an induction system contains two redundant ways to represent all possible executions, namely Ct and Cx. This is used to determine a novel verification method for parallel programs with three phases:

1. The semantics of R is expressed as a set of conditions and added to the Ct of the general class in which R is executed.
2. A set of derivation rules Cx is devised such that $\langle R, Cx, Ct \rangle$ is an induction system.
3. Since Cx is a *rewriting system* which generates all possible executions of R, it can be used to show that all executions of R halt or preserve some desired property.

We use this method to show that the set of all possible executions of a specific parallel program, consists of a single synchronous execution (out of a large set of possible asynchronous executions). This program realizes a complex pattern of synchronization between 21 processes, each setting and resetting common flag 10 times. The fact that only one synchronous execution is possible is used to show that the program halts and terminates.

Further research is needed in order to give this method a more "solid" base. In particular more types of parallel programs must be studied using the proposed framework. Future research efforts may focus on the following set of problems:

1. Which restrictions of Ct (such as restriction to first order logic) can guarantee suitable Cx such that $\langle R, Ct, Cx \rangle$ is an induction system?
2. For a given program and a set of Ct conditions, is there a systematic way (an algorithm) to find suitable Cx?
3. Devise a notion of *execution time* and *efficiency* which exploits the induction system to predict performances.
4. Study different types of classes, for parallel execution, which might be useful for verification of all possible execution orders.
5. Determine specific types of programs (such as parallel loop programs) for which an induction system can be determined.
6. Operators like the ones suggested by Pratt[1] and Gaifman[13] can be embedded into the framework, such that the syntax of a parallel program will include *choice*, *recursion*, *loops*, and *communication.*

REFERENCES

1. V. Pratt, Modeling Concurrency with Partial Orders, *International Journal of Parallel Programming* **15**(1):33–71 (1986).
2. INMOS Ltd. *Occam Programming Manual*, Prentice Hall (1984).
3. United Stated Department of Defense, Reference Manual for the Ada Programming Language. ANSI MIL-STD-1815 (1983).

4. S. Ahuja, N. Carriero, and D. Gelernter, Linda and Friends, *Computer* **19**(8):26–34 (1986).
5. A. H. Karp and R. G. Babb II, A Comparison of 12 Parallel Fortran Dialects, *IEEE Software* **5**(5):52–67 (1988).
6. J. T. Kuehn and H. J. Siegel, Extensions to the C Programming Language for SIMD/MIMD Parallelism, *Intl. Conf. Parallel Processing*, pp. 232–235 (August 1985).
7. L. Lamport, How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs, *IEEE Trans. Computers* **C-28**(9):690–691 (1979).
8. C. A. R. Hoar, *Communicating Sequential Process*, Prentice Hall (1985).
9. C. H. Papadimitriou, The Serializability of Concurrent Database Updates, *Journal of the ACM* **26**(4):631–653 (1979).
10. W. Reisig, Concurrency Is More Fundamental Than Interleavings, *EATCS Bull*, Vol. 36 (1988).
11. V. S. Adve and M. D. Hill, Weak Ordering—A New Definition, *Ann. Intl. Symp. Computer Architecture* **17**:2–14 (1990).
12. A. Pnueli, The Temporal Logic of Programs, *Proc. of the 18th Symp. on the Foundations of Computer Science*, ACM (November 1977).
13. H. Gaifamn, Modeling Concurrency by Partial Orders and Nonlinear Transition Systems, Springer Verlag, *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, Lecture Notes in Computer Science* **354**:467–488 (1988).
14. R. Janicki and M. Kountny, Structure of Concurrency, *Theoretical Computer Science* **112**:5–52 (1993).
15. A. K. Deshpande and K. M. Kavi, A Review of Specification and Verification Methods for Parallel Programs, including the Dataflow Approach, *Proc. IEEE* **77**(12):1816–1828 (December 1989).
16. E. R. Olderog, Nets, Terms and Formulas, Cambridge Tracts in Theoretical Computer Sciences, Vol. 23 (1991).
17. L. Rudolph, Software Structures for Ultraparallel Computing, Ph.D. Thesis, Courant Inst., NYU (1982).
18. E. Freudenthal and A. Gottlieb, Process Coordination with Fetch-and-Increment, *Ann. Intl. Symp. Computer Architecture* **17**:2–14 (1990). *Intl. Symp. Architect. Support for Prog. Lang. & Operating Syst.* **4**:260–268 (1991).
19. B. D. Lubachevsky, An Approach to Automating the Verification of Compact Parallel Coordination Programs, *Acta Informatica* **21**:125–169 (1984).
20. H. E. Bal, J. G. Steiner, and A. S. Tanenbaum, Programming Languages for Distributed Computing Systems, *ACM Comput. Surv.* **21**(3):261–322 (1989).
21. H. Gaifamn and V. Pratt, Partial Order Models of Concurrency and the Computation of Function, *Symp. on Logic in Computer Science* (1987).
22. M. P. Herlihy and J. M. Wing, Axioms for Concurrent Objects, *JPDC* **14**:13–26 (1987).
23. U. Montanari, *True Concurrency: Theory and Practice*, Springer-Verlag, Proc. of Mathematics of Program Construction, Oxford (1992).
24. W. E. Weihl, Commutativity-based Concurrency Control for Abstract Data Types, *IEEE Trans. on Computers* **37**(12):1488–1505 (December 1988).