Event Abstraction in Modeling Distributed Computations *

Twan Basten
Department of Mathematics and Computing Science
Eindhoven University of Technology, P.O. Box 513, 5600 MB Eindhoven, The Netherlands
E-mail: tbasten@win.tue.nl

Abstract

A well-known problem in event-based models of distributed computations is the amount and complexity of behavioral information. Event abstraction can be used to reduce the apparent complexity of a computation. Four aspects of event abstraction are studied: a model describing primitive behavior, a formalism for specifying abstract behavior, abstract descriptions of behavior, and verification of primitive or abstract descriptions against specified behavior.

This paper discusses the issues involved in each of these four aspects, presents basic results, and identifies remaining problems.

Key words: event abstraction – causality – partial order – (reversed) vector time – partially ordered multiset – pomset grammar – pomset LR parsing.

1 Introduction

The most important barriers to understanding distributed applications are complexity, concurrency, and inherent non-determinism. Many applications involve a large number of machines, processes, and events. The communication pattern is often irregular and complex. Concurrency causes the set of events in a distributed computation to be only partially ordered. And, due to unknown communication delays or unreliable communication channels, program behavior may vary for every execution.

To better understand the behavior of distributed applications, many tools and algorithms have been developed. To mention just a few of them, one could think of snapshot algorithms [10, 22, 25, 29], predicate-detection algorithms [14], and debugging tools including tools for visualizing behavior [35], breakpoint facilities [20, 28, 31], and tools for recognizing behavioral patterns [5, 23].

All these tools and algorithms have one thing in common: They use event-based models of distributed computations. More in particular, they use the *causal relationships* among events to extract information that is considered useful. Such causal relationships can be expressed in terms of *precedence*. An event might causally precede another, or two events might be unrelated or concurrent.

*The work presented in this paper was done while the author was staying at the University of Waterloo, Ontario, Canada, in 1992–93. At the time, he was doing research for his Master's thesis under the supervision of Jay Black and Mike Coffin.

Unfortunately, the number of events and the complexity of the causal relationships often make the abovementioned algorithms slow and ineffective. Consequently, obtaining detailed information about program behavior is a hard and tedious process. A powerful way to reduce the apparent complexity of a computation and, therefore, improve the process of obtaining behavioral information is *abstraction* [7].

This paper discusses one type of abstraction, namely *event abstraction*. Events are grouped together into higher-level abstract events, hiding their internal causal structure. Doing so, the programmer can obtain simpler, abstract views of program behavior, and focus only on information that is to his or her interest. Of course, the process can be repeated until a desired level of abstraction is reached. Ideally, a program can be executed or simulated at arbitrary levels of abstraction. Therefore, if tools for analyzing program behavior are adapted to use abstract descriptions, obtaining interesting information becomes faster and less tedious. The idea of using abstraction and refinement for analyzing and developing distributed applications, in particular real-time and embedded applications, is also expressed in [21].

This paper discusses four basic aspects of event abstraction. Section 2 describes primitive observed behavior in terms of events and causality. Section 3 introduces partially ordered multisets for specifying abstract behavior. Section 4 describes abstract behavior in terms of vector time and reversed vector time. Finally, Section 5 applies *LR*-parsing techniques for verifying primitive or abstract observed behavior against specified behavior. In Section 6, the results are summarized and some open problems are pointed out.

2 Primitive distributed computations

A *distributed system* is a collection of many loosely-coupled machines. These machines do not share any system resources and are only connected by a communication network. Communication channels may be lossy and delivery order may or may not be guaranteed.

A *distributed program* is a set of independent, cooperating program modules. Information is exchanged by message-passing only. Both synchronous and asynchronous communication are allowed. For the sake of simplicity, it is assumed that communication is point-to-point. However, it is straightforward to extend the results to multicast and broadcast schemes.

At runtime, program modules are instantiated as *processes* which do not share any memory. To avoid unnecessarily complex formulas, it is assumed that the number of processes is fixed and known in advance. Each process performs a *local computation*. A *distributed computation* is the collection of all local computations.

The model of computations presented below is based on the notions of actions and primitive events. Actions can occur more than once during a computation. A primitive event is a unique occurrence of an action. Primitive events are considered to be atomic. Therefore, a primitive event is modeled as if it occurred instantaneously.

A distributed computation is modeled by a *labeled partial order* $(E, \Lambda, \preceq, \lambda)$. The set of primitive events E is partially ordered by the ordering relation $\preceq$ that models causal precedence. The labeling function λ maps events to actions in the alphabet Λ. The model is based mainly on previous definitions of Mattern [29, 30], Charron-Bost [11, 12], and Fidge [15, 16]. Their definitions are, in turn, based on the "happens before" relation introduced by Lamport [26]. The algebraic structure underlying the model is Winskel's elementary event structure [36].

The set of primitive events E is the union of N mutually disjoint sets of events, $E_0, \ldots, E_{N-1}$, where N is the number of processes. Each of these sets represents a local computation. It is assumed that E is *finite*. Since this paper discusses timestamp schemes, this is not a real restriction: In practice, only finite (prefixes of) computations can be timestamped. The set of process identifiers, $\{0, \ldots, N-1\}$, is denoted $\mathcal{P}$.

As mentioned, both synchronous and asynchronous communication are allowed. Every communication is modeled by a send event and a corresponding receive event. The sets of send and receive events are denoted by $\mathcal{S}$ and $\mathcal{R}$ respectively. The two sets are disjoint subsets of the set of events E. A relation $\Gamma \subseteq \mathcal{S} \times \mathcal{R}$ relates send events to receive events. This relation is left- and right-unique. Furthermore, it is required that for every receive event in $\mathcal{R}$, there is a corresponding send event in $\mathcal{S}$. The absence of the converse condition means that messages might be lost or might still be in transit. A subset of Γ, Γ_s, denotes the set of synchronous message communications.

For every $i \in \mathcal{P}$, the set E_i is totally ordered by a relation $\prec_i$. This models the fact that processes are sequential. The relation $\prec_l$ is defined as the union of all $\prec_i$. It expresses the local ordering of events. The precedence relation $\preceq$ that models the causal ordering of events is defined as the smallest reflexive and transitive relation that satisfies the following two conditions.

C1 The relation $\prec_l \cup \Gamma$ is a subset of $\preceq$.

C2 For every $(s, r) \in \Gamma_s$ and $e \in E \setminus \{s, r\}$, $e \preceq s \Leftrightarrow e \preceq r$ and $s \preceq e \Leftrightarrow r \preceq e$.

Since the restrictions on Γ do not guarantee the absence of cycles in the precedence relation, the set of definitions given so far models a distributed computation if and only if the precedence relation is a partial order.

The precedence relation extends the "happens before" relation as defined by Lamport [26] to synchronous communication in a natural way. The condition **C2**, originally given by Fidge [15, 16], means that a synchronous communication can be interpreted as if it occurred instantaneously. Distinguishing a send and a receive event such that the send event precedes the corresponding receive event conforms to physical reality: a synchronous communication is initiated by one process and after a small but non-zero delay received by the other. A discussion of other models of synchronous communication can be found in [3, Section 2.1].

The relation $\preceq$ can be used to express concurrency in a natural way. Two events $e_0, e_1 \in E$ are concurrent if and only if $e_0 \not\preceq e_1$ and $e_1 \not\preceq e_0$. That is, two events are concurrent if and only if they are unrelated by the precedence relation.

For many applications in distributed computing, it is useful to have a characterization of causality. Since the causality structure of a computation is a partial order, it is not possible to use physical time or any other totally ordered set as a characterization. For this reason, Mattern [29, 30] and Fidge [16] independently introduced partially ordered vector time. Vector time extends the idea of logical clocks introduced by Lamport [26].

Definition 2.1. The function $T : E \longrightarrow \mathbb{N}^N$ defines a timestamp for every event as follows. For any event $e \in E$ and process $i \in \mathcal{P}$, $T.e.i = |\{e_i \in E_i \mid e_i \preceq e\}|$.

Note that the vector representation of timestamps is possible only because the number of processes is known. However, vector representation of timestamps is not essential. If the number of processes is not known, the timestamp of an event can be defined as a set of pairs, where each pair consists of a process identifier and the corresponding timestamp component [16].

The following theorems show that vector timestamps characterize causality. For two vectors $t_0, t_1 \in \mathbb{N}^N$, we define $t_0 \leq t_1$ to mean $t_0.i \leq t_1.i$ for all i, $0 \leq i < N$.

Theorem 2.2. For any events $e_0, e_1 \in E$, $e_0 \preceq e_1 \Leftrightarrow T.e_0 \leq T.e_1$.

Theorem 2.3. For any $i \in \mathcal{P}$ and events $e_0 \in E_i$ and $e_1 \in E$, $e_0 \preceq e_1 \Leftrightarrow T.e_0.i \leq T.e_1.i$.

A simple proof of these theorems and an algorithmic definition of the timestamp function can be found in [3, Section 2.2].

An example of the assignment of vector timestamps to events is given in Figure 1. For this example, the validity of the two precedence tests given above is easily verified. The vertical lines represent processes. Time increases from top to bottom. Events are depicted as dots. The arrows represent the communication relation Γ. A synchronous communication is represented by a horizontal arrow.

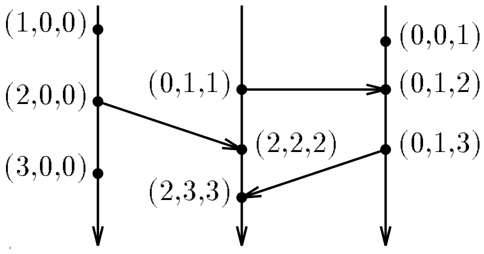


Figure 1: Timestamping events in a distributed computation.

3 Behavioral patterns

3.1 Introduction and related work

This section introduces a formalism for specifying abstract behavior of a distributed program. To be meaningful in an event-based computational model, the formalism must specify behavior in terms of activity and causality. Therefore, the notion of behavioral patterns is introduced. A behavioral pattern is defined as a multiset of actions and a set of causal relations among those actions. An occurrence of a behavioral pattern in a distributed computation, or an *abstract event*, is a set of events such that their labeling and causal relationships exactly match the pattern. A particular pattern can occur several times in a computation. Behavioral patterns can be used to construct a behavioral abstraction hierarchy. The programmer specifies one or more behavioral patterns. Sets of events matching these patterns are grouped together into abstract events, yielding an abstract view of the computation. The process is repeated until a desired level of abstraction is reached. In this process, behavioral patterns can be interpreted as high-level actions.

Since the precedence relation in a distributed computation is a partial order, it seems reasonable to require that the multiset of actions of a behavioral pattern also be partially ordered. In other words, a behavioral pattern is a *partially ordered multiset*, or pomset, of actions.

Pomsets [18, 33] are attracting more and more interest in distributed and parallel computing. The reason being that they model true concurrency, whereas another well-known model for concurrency, the interleaving model, does not. Interleaving is a convenient way to incorporate a notion of concurrency in the theory of string languages. It is based on the assumption that there exists an underlying real-time total ordering of events. However, for analyzing causality, the real-time ordering of events is usually not very meaningful. The pomset model does not make the assumption of a real-time ordering and is, therefore, more useful. Pratt discusses the advantages and disadvantages of both models extensively in [33].

One way of looking at a pomset is to consider it to be a generalization of a string. A string is nothing but a *totally* ordered multiset or *tomset* of symbols taken from some alphabet. This means that a pomset is just a string in which the total order is relaxed to a partial order. The theory of strings and their languages has been investigated in great detail (see [27] among others). Finite representations of several classes of languages have been developed, such as regular expressions and grammars. Regular expressions have been used to develop specification languages. In addition, algorithms based on regular expressions and grammars have been developed to decide whether a string belongs to a given language. Therefore, it seems worthwhile to investigate whether similar results can be derived for *pomset languages*, which are sets of pomsets. A finite representation of pomset languages, similar to regular expressions, can be the basis for a language to specify behavioral patterns. Algorithms that are able to decide whether a pomset belongs to a given pomset language can be used to develop a tool for recognizing occurrences of behavioral patterns in a distributed computation.

Many different specification languages based on regular expressions have been developed for many different purposes [6, 8, 9]. However, none of them include true-concurrency operators. In [23], Hseush and Kaiser propose *Data-Path Expressions*, that do include operators modeling true concurrency. Although the authors do not mention it explicitly, the model underlying Data-Path Expressions is the pomset model.

There are two more papers that are important in the context of specifying behavioral patterns. Both the papers of Miller and Choi [31] and Haban and Weigel [20] describe how behavioral abstraction can be used to incorporate breakpoints in a distributed debugging environment. However, the sets of operators they introduce are different from the set of operators used to build regular expressions. Consequently, there does not exist a straightforward interpretation of their specifications in terms of language theory. This complicates reasoning about the recognition of occurrences of patterns.

3.2 Pomset language theory

This subsection gives an introduction to the fundamentals of pomsets and their languages. The theory is developed analogously to the theory of string languages [27]. Based on the definition of pomsets, formal definitions of behavioral patterns and abstract events are given.

Definition 3.1. A 4-tuple $(V, \Sigma, \leq, \mu)$ is a labeled partial order, or lpo, if and only if $(V, \leq)$ is a partial order and μ, the labeling function, is a mapping from V to Σ.

Two lpos $M_0 = (V_0, \Sigma, \leq_0, \mu_0)$ and $M_1 = (V_1, \Sigma, \leq_1, \mu_1)$ are said to be *isomorphic* if and only if there is a bijective function $\tau : V_0 \longrightarrow V_1$ that preserves ordering and labeling. That is, for any $u, v \in V_0$, $u \leq_0 v \Leftrightarrow \tau.u \leq_1 \tau.v$ and $\mu_1.(\tau.u) = \mu_0.u$. A pomset is the isomorphism class of some lpo. Square brackets are used to denote pomsets. For example, $[V_0, \Sigma, \leq_0, \mu_0]$ is the pomset corresponding to M_0.

The set of all pomsets over alphabet Σ is denoted by $\Sigma\ddagger$. This corresponds to the notation used by Pratt [33]. Unless explicitly defined otherwise, the point set, alphabet, ordering relation, and labeling function of some lpo or pomset M are denoted by V_M, Σ_M, $\leq_M$, and μ_M respectively. The notation ε is used to denote the special pomset $[\emptyset, \Sigma, \emptyset, \emptyset]$, for any alphabet Σ. Note that this corresponds to the notation traditionally used for the empty string. The notation u_A is used to denote the *unit pomset* $(\{x\}, \Sigma, \{(x,x)\}, \{(x,A)\})$, for any A in some Σ.

At this point, it is possible to formally define the notions of behavioral patterns and abstract events, which are occurrences of behavioral patterns. Let $(E, \Lambda, \preceq, \lambda)$ be the labeled partial order representing a distributed computation. Using the definition above, a behavioral pattern is a pomset over Λ. An occurrence of a behavioral pattern M is a non-empty set of primitive events A such that $[A, \Lambda, \preceq\lceil(A \times A), \lambda\lceil A] = M$, where $f\lceil D$ denotes the restriction of function f to domain D.

In order to construct new pomsets from other pomsets and, subsequently, new pomset languages from other pomset languages, the following two operators on pomsets are introduced.

Definition 3.2. Let $M_0 = [V_0, \Sigma_0, \leq_0, \mu_0]$ and $M_1 = [V_1, \Sigma_1, \leq_1, \mu_1]$ be pomsets. Without loss of generality, it is assumed that V_0 and V_1 are disjoint. Pomset $[V_2, \Sigma_2, \leq_2, \mu_2]$ is the *sequential composition* of M_0 and M_1, denoted $M_0 M_1$, or sometimes $M_0; M_1$, if and only if $V_2 = V_0 \cup V_1$, $\Sigma_2 = \Sigma_0 \cup \Sigma_1$, $\leq_2 = \leq_0 \cup \leq_1 \cup V_0 \times V_1$, and $\mu_2 = \mu_0 \cup \mu_1$.

Pomset M_2 is the *concurrent composition* of M_0 and M_1, denoted $M_0 \& M_1$, if and only if $V_2 = V_0 \cup V_1$, $\Sigma_2 = \Sigma_0 \cup \Sigma_1$, $\leq_2 = \leq_0 \cup \leq_1$, and $\mu_2 = \mu_0 \cup \mu_1$.

An example of the use of these two operators is shown in Figure 2. In terms of distributed computing, sequential and concurrent composition model precedence and concurrency respectively.

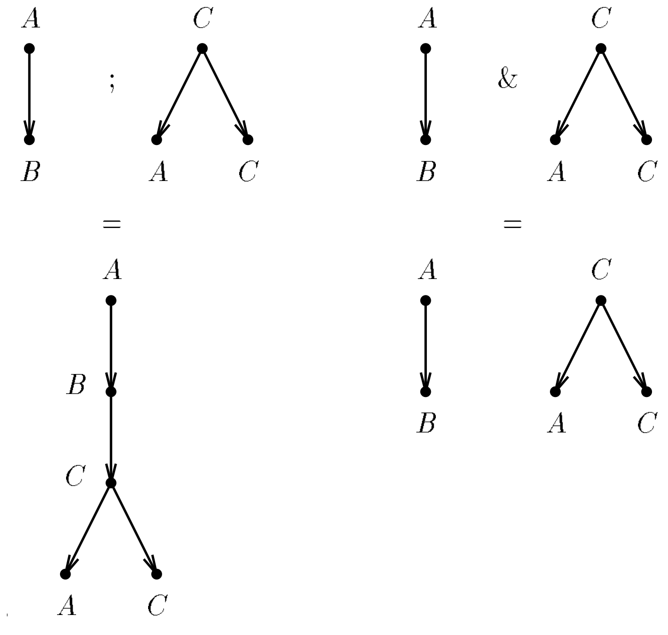


Figure 2: Sequential and concurrent composition of pomsets.

The two operators defined above yield an important class of pomsets, called *series-parallel* pomsets. Gischer [18] defines this class as the smallest set of pomsets that contains the empty pomset ε and all unit pomsets, and is closed under the two composition operators.

The composition operators can easily be translated to equivalent operators on pomset languages. In addition, three new operators on pomset languages are introduced.

Definition 3.3. Let L, L_0 and L_1 be pomset languages over alphabet Σ. Sequential composition $L_0; L_1 = \{M_0 M_1 \mid M_0 \in L_0 \wedge M_1 \in L_1\}$; concurrent composition $L_0 \& L_1 = \{M_0 \& M_1 \mid M_0 \in L_0 \wedge M_1 \in L_1\}$; choice $L_0 + L_1 = L_0 \cup L_1$; sequential closure $L^* = \{\varepsilon, L, LL, LLL, ...\}$; concurrent closure $L\dagger = \{\varepsilon, L, L\&L, L\&L\&L, ...\}$. The unary operators have the strongest binding, followed by ;, &, and +.

The set of operators on pomset languages defined so far is sufficient to generalize the notion of regular expressions to *concurrent regular expressions*. Each concurrent regular expression defines a unique pomset language. The mapping from concurrent regular expressions to pomset languages is straightforward and, therefore, omitted.

Definition 3.4. The set of concurrent regular expressions over an alphabet Σ is a subset of the strings over the alphabet $\Sigma \cup \{(,), \emptyset, ;, \&, +, ^*, \dagger\}$. Let β and γ be concurrent regular expressions. The string α is a concurrent regular expression if and only if $\alpha = \emptyset$, $\alpha \in \Sigma$, $\alpha = (\beta; \gamma)$, $\alpha = (\beta \& \gamma)$, $\alpha = (\beta + \gamma)$, $\alpha = \beta^*$, or $\alpha = \beta\dagger$.

The class of pomset languages defined by concurrent regular expressions is called the class of *regular pomset languages*. This class is a strict superset of the class of languages defined by regular expressions. Note that by definition, a regular pomset language only contains series-parallel pomsets.

The following pomset, defined by Grabowski [19] and Gischer [18], plays an important role in the theory of pomset languages. Define $\mathbf{N}$ as the irreflexive pomset $[V_{\mathbf{N}}, \{A\}, <_{\mathbf{N}}, \mu_{\mathbf{N}}]$, where $V_{\mathbf{N}} = \{0, 1, 2, 3\}$, $<_{\mathbf{N}} = \{(0, 2), (0, 3), (1, 3)\}$, and $\mu_{\mathbf{N}}.v = A$, for all $v \in V_{\mathbf{N}}$. This pomset gets its name from its shape that resembles an uppercase "N" (see Figure 3).

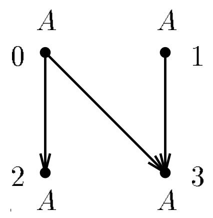


Figure 3: Graphical representation of the pomset $\mathbf{N}$.

A pomset is said to be $\mathbf{N}$-free if and only if it does not have $\mathbf{N}$ as a subpomset (possibly after relabeling). This yields the following theorem. The proof is fairly elaborate and can be found in [18]. A similar result is proven in [19].

Theorem 3.5. A finite pomset is series-parallel if and only if it is $\mathbf{N}$-free.

This theorem yields that a pomset language is regular if and only if all its elements are $\mathbf{N}$-free. The class of regular pomset languages is a strict subset of the class of all pomset languages.

It is not yet clear what the consequences of this theorem are for the expressiveness of specification languages based on concurrent regular expressions. In a distributed computation, an asynchronous message communication typically creates a causal pattern like the pomset $\mathbf{N}$. The literature describes only very little experience with such languages [32]. It might be necessary to extend the pomset formalism with a new operator that takes four pomsets as an argument and joins them into one pomset having the structure of the pomset $\mathbf{N}$. Pratt [33] defines such an operator.

The reason for not discussing this operator in this section is that it is not clear whether it is really necessary to extend the formalism for the purpose of event abstraction. There might be other useful extensions. Furthermore, adding an operator complicates the recognition of pomset languages as discussed in Section 5.

4 Abstract events and causality

In the previous section, a condition is given that a non-empty set of primitive events must satisfy to form a first-level abstract event. The set of events fully describes the abstract event. More in general, in a true hierarchy of descriptions of program behavior, an abstract event is described uniquely by its constituents in the previous level. However, to avoid recursive definitions, in the remainder, abstract events are usually represented by sets of primitive events. Only when necessary is explicitly referred to elements of an abstract event in the level below. For this purpose, the term "constituent" is used exclusively.

The set of abstract events is not the only part of an abstract description of program behavior. Another important part is a definition and representation of causality among abstract events.

4.1 A definition of causality

As for primitive events, the causality structure is defined by a precedence relation. For reasons of mathematical and practical convenience, this relation preferably has the same structural properties as the precedence relation on primitive events, i.e., it is a partial order. There are two obvious possible ways to define precedence.

The first possibility is to specify that an abstract event A precedes an abstract event B if and only if every event in A precedes every event in B. This definition has one advantage: It guarantees anti-symmetry and transitivity. Taking the reflexive closure yields a precedence relation on abstract events that is a partial order. However, it also has two important disadvantages. First, it does not conform to the intuitive meaning of concurrency. Concurrency between primitive events is defined as being unrelated by the precedence relation. If the same definition is used for abstract events, two abstract events can be concurrent while some primitive events in one abstract event precede some primitive events in the other. Second, this definition of precedence is too restrictive to be useful. Many abstract events will be unrelated. This means that in the first level of abstraction almost all structure will be lost. Higher levels of abstraction are likely to contain only unrelated events. This defeats the purpose of behavioral abstraction, which focuses on causal relations among events. Universal quantification seems to be too strong a requirement.

The second possibility is to specify that an abstract event A precedes an abstract event B if and only if there exists an event in A that precedes at least one event in B. Intuitively, this is also a meaningful definition, since at least part of A happens before B is completed. This definition has one important disadvantage: in general, the precedence relation on abstract events is no longer a partial order. The relation usually is neither anti-symmetric nor transitive. An advantage is that concurrency conforms to its intuitive meaning. Two abstract events A and B are causally unrelated if and only if every event in A is causally unrelated to every event in B. An additional advantage is that abstract behavior maintains a lot of the structure of underlying levels of description. Although this choice does, in general, not yield a precedence relation which is a partial order, it seems to be the only one that is practically both meaningful and useful. Therefore, precedence among abstract events is formally defined as follows.

Definition 4.1. For abstract events A and B, $A \preceq B \Leftrightarrow (\exists\, a, b:\ a \in A\ \wedge\ b \in B:\ a \preceq b)$

4.2 Timestamping abstract events

It seems a logical choice to investigate the use of vector timestamps to represent causality among abstract events. The goal is to derive precedence tests similar to those for primitive events. Since the precedence relation on abstract events is not necessarily a partial order, two approaches are possible.

The first approach is to restrict the structure of abstract events in such a way that the precedence relation is guaranteed to be a partial order, and then to try to timestamp these restricted abstract events in a meaningful way. This is the approach taken by Cheung [13], Summers [34], and Kunz [24]. Unfortunately, imposing structural requirements that guarantee transitivity and anti-symmetry either makes the resulting abstractions difficult to manage or severely limits their expressiveness. The second approach is to allow more flexible abstract events and, and then to try to derive general timestamp schemes. This approach is pursued in this subsection. Two timestamps and some precedence tests for arbitrary abstract events are given below.

However, before giving any results, it is important to have criteria to evaluate timestamps and precedence tests. First, timestamps and precedence tests must be reasonably efficient in storage and computation time. That is, storage and computation time must be similar to storage and computation time needed for determining precedence between primitive events. Second, in a hierarchy of true abstractions, a representation of causality should not depend on any other levels than the level immediately below. This means that it must be possible to calculate a timestamp for an abstract event from the timestamps of its constituents. It also means that precedence tests must be defined in terms of the timestamps in the level being described. If tests depend on lower levels, determining precedence between two abstract events becomes computationally expensive, because the abstraction hierarchy must be traversed to a level that contains the desired information.

In the following, the efficiency of precedence tests is measured by the number of integer comparisons. To determine whether an abstract event A precedes an abstract event B, one could simply check all the primitive events until a primitive event in A is found that precedes a primitive event in B. If the precedence test of Theorem 2.2 is used, this takes at most $|A| \cdot |B| \cdot N$ integer comparisons. Obviously, this is not very efficient. It is also clear that the aforementioned test depends on the primitive level of the computation. Therefore, it does not satisfy the second criterion for precedence tests either.

One would expect that it is possible to do better. If abstract events are timestamped in a meaningful way, it should be possible to reduce the number of integer comparisons and perhaps make it independent of the primitive level of the computation. However, before a timestamp for abstract events is given, first some auxiliary functions are defined.

Definition 4.2. The *location set* of an abstract event is defined by a function $\boldsymbol{l}. : 2^E \longrightarrow 2^{\mathcal{P}}$ as the set of processes in which the event occurs. For any $A \subseteq E$, $\boldsymbol{l}A = \{i \in \mathcal{P} \mid A \cap E_i \neq \emptyset\}$.

A property of primitive events that no longer holds for abstract events is atomicity. Consequently, abstract events have a duration. This is expressed by the following two functions.

Definition 4.3. The *beginning* of an abstract event is defined by a function $\lfloor . \rfloor : 2^E \longrightarrow 2^E$ as follows. For any $A \subseteq E$, $\lfloor A \rfloor = \{a_0 \in A \mid \neg(\exists\, a_1 :\ a_1 \in A :\ a_1 \prec a_0)\}$. The *end* is defined by a function $\lceil . \rceil : 2^E \longrightarrow 2^E$. For any $A \subseteq E$, $\lceil A \rceil = \{a_0 \in A \mid \neg(\exists\, a_1 :\ a_1 \in A :\ a_0 \prec a_1)\}$.

Note that to determine whether an abstract event A precedes another abstract event B, it is sufficient to consider the beginning of A and the end of B instead of A and B entirely.

Definition 4.4. The function $T : 2^E \longrightarrow \mathbb{N}^N$ defines the timestamp of an abstract event as follows. For any $A \subseteq E$, $T.A = (\mathrm{SUP}\ a :\ a \in A :\ T.a)$, where the quantifier SUP is defined as the componentwise maximum.

It can be shown that this timestamp is an encoding of the end of an abstract event [3]. This means that it is possible to improve the efficiency of determining precedence among abstract events. Furthermore, the associativity of the quantifier SUP implies that the timestamp of an abstract event is equal to the supremum of the timestamps of its constituents. The introduction of this timestamp yields the following precedence test that is a step towards the fulfillment of the two criteria mentioned earlier.

Theorem 4.5. For abstract events A and B, $A \preceq B \Leftrightarrow (\exists\, a :\ a \in \lfloor A \rfloor :\ T.a \leq T.B)$.

However, this test does still not satisfy the criteria. First, it is still not very efficient. In the worst case, the timestamp of every primitive event in $\lfloor A \rfloor$ must be compared to the timestamp of B, yielding $|\lfloor A \rfloor| \cdot N$ integer comparisons. Since the number of processes in the location set of A is an upper bound for the number of primitive events in the beginning, the maximum number of comparisons is $|\boldsymbol{l}A| \cdot N$. Second, precedence cannot be determined without using information from the primitive level of the computation.

The maximum number of integer comparisons can be reduced if the location set of events is known.

Theorem 4.6. For abstract events A and B, $A \preceq B \Leftrightarrow (\exists\, i, a : i \in \mathcal{P} \wedge\, a \in \lfloor A \rfloor \cap E_i : T.a.i \leq T.B.i)$.

This test is more efficient than the previous one. In the worst case, the number of comparisons is $|\lfloor A \rfloor|$, which in turn has an upper bound $|\boldsymbol{l}A|$. However, it still has the disadvantage that it depends on the primitive level of the computation. There seems to be no obvious way to overcome this problem within the current framework.

At this point, it is interesting to see that there is an asymmetry in the way the beginning and the end of the abstract events are used in the two tests above. The beginning of an abstract event is used explicitly. The end of an abstract event is encoded in its timestamp. In order to resolve this asymmetry, every event is timestamped a second time. The idea is to find a timestamp function that encodes the beginning of an abstract event. Therefore, the successor relation $\succeq$ is defined as the dual of the precedence relation $\preceq$. That is, for any $e_0, e_1 \in E$, $e_0 \succeq e_1$ if and only if $e_1 \preceq e_0$. This leads to the notion of *reversed vector time*.

Definition 4.7. The function $T^R : E \cup 2^E \longrightarrow \mathbb{N}^N$ defines a timestamp in *reversed vector time* as follows. For any $e \in E$ and $i \in \mathcal{P}$, $T^R.e.i = |\{e_i \in E_i \mid e_i \succeq e\}|$, and for any $A \subseteq E$, $T^R.A = (\mathrm{SUP}\ a :\ a \in A :\ T^R.a)$.

The function T^R represents for primitive events the set of timestamps that is obtained by applying a timestamp algorithm for ordinary vector timestamps while traversing event information backwards. For abstract events, the reversed timestamp can be calculated from the timestamps of its constituents. It encodes the beginning of an abstract event in *reversed* vector time. The beginning of an abstract event A in terms of ordinary vector time is given by the expression $\mathbf{E} - T^R.A$, where vector $\mathbf{E}$ is a constant such that for every $i \in \mathcal{P}$, $\mathbf{E}.i = |E_i|$.

In [3], it is shown that reversed vector time yields the following precedence test.

Theorem 4.8. For abstract events A and B, $A \preceq B \;\Leftrightarrow\; \mathbf{E} - T^R.A \not\geq T.B$.

The number of integer comparisons for this test is at most N. Without further knowledge it is not possible to derive a more efficient test. Therefore, this is the first test that satisfies the two criteria for precedence tests for abstract events. Figure 4 illustrates this result. Part (a) shows an abstract event as it is determined by the two times that mark its beginning and end. Part (b) shows two abstract events A and B, where $A \preceq B$.

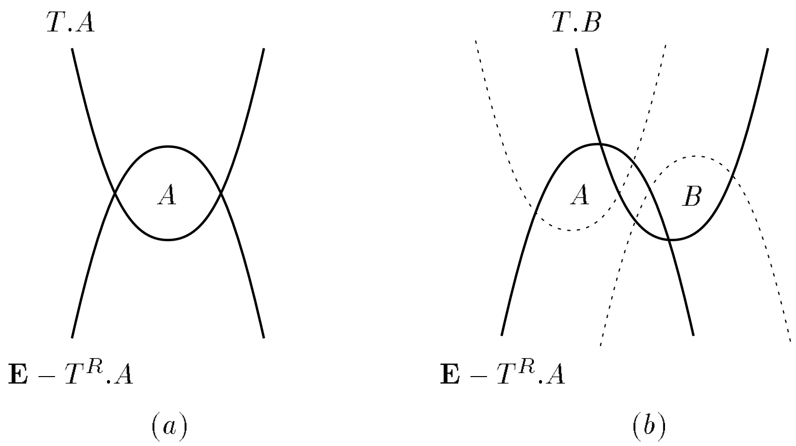


Figure 4: Precedence among abstract events in terms of vector time and reversed vector time.

If the location set of abstract events is known, one only has to consider components of the time vectors within the location set of one of the abstract events.

Theorem 4.9. For abstract events A and B, $A \preceq B \Leftrightarrow (\exists\, i :\; i \in \boldsymbol{l}A :\; \mathbf{E}.i - T^R.A.i < T.B.i) \Leftrightarrow (\exists\, i :\; i \in \boldsymbol{l}B :\; \mathbf{E}.i - T^R.A.i < T.B.i)$

For these tests, the maximum number of integer comparisons is $|\boldsymbol{l}A|$ and $|\boldsymbol{l}B|$ respectively. This is similar to the efficiency of the test given in Theorem 4.6. However, unlike this test, the tests in Theorem 4.9 are independent of the primitive level of the abstraction hierarchy.

Summarizing, this section presents two timestamps and some precedence tests for arbitrary abstract events in terms of these timestamps. In [3], these results are further improved and applied to a subclass of abstract events, so-called *convex* abstract events. It is beyond the scope of this paper to discuss these additional results in detail.

5 Recognizing abstract events

This section introduces a formalism to decide whether a pomset belongs to some pomset language. Such a formalism can be the basis for a tool to recognize abstract events in a distributed computation. Observed behavior is matched against specified behavior constructing an abstract view of the computation. However, there is one important difference between recognizing pomsets and recognizing abstract events. Recognizing abstract events means recognizing *sub*pomsets embedded in a larger pomset. Although this is a very important issue, the complexity of the problem is a reason to separate the two problems of recognizing pomsets and subpomsets, and so this section discusses the recognition of pomsets. The analogy with language theory suggests the following two approaches: the automata-theoretic approach and the parsing approach.

Bates [4] describes an automata-theoretic model for recognizing abstract events, called *shuffle automata*. However, shuffle automata are not useful for our purposes. First, they cannot be used to recognize true concurrency. Only the interleaving, or shuffle, of events can be recognized. Second, the ordering imposed on events is *temporal* instead of causal and, therefore, not very meaningful for analyzing causality. Third, like ordinary finite-state automata, shuffle automata can be non-deterministic. Obviously, a deterministic recognition algorithm is preferred.

Hseush and Kaiser [23] introduce another class of automata, called *predecessor automata*. For specifying behavioral patterns, they use Data-Path Expressions which are essentially the same as concurrent regular expressions. Therefore, predecessor automata seem useful for recognizing abstract events. However, the formalism of predecessor automata is very complicated, particularly the construction of compact, deterministic automata. Furthermore, the class of languages recognized by predecessor automata is fairly limited. The main reason for this being that the input to predecessor automata can be ambiguous. A more detailed discussion of predecessor automata including references to relevant literature can be found in [2, pages 64–67].

Since the results of the automata-theoretic approach are rather disappointing, this section investigates the parsing approach. In the next two subsections, so-called pomset grammars are introduced and based on those grammars, *pomset LR* parsing, or *PLR* parsing, is described informally.

5.1 Pomset grammars

In this subsection, some relevant definitions concerning grammars are generalized to pomset languages. Only context-free grammars are discussed, because the purpose of this subsection is to derive a simple representation of concurrent regular expressions that can be used with parsing algorithms.

The following definition defines point substitution on pomsets. Point substitution generalizes the process of replacing a non-terminal by a string according to a production rule in a derivation based on an ordinary grammar.

Definition 5.1. Let $M = [V, \Sigma, \leq, \mu]$ be a pomset in $\Sigma\ddagger$. A *point substitution* is a function f that maps points of M to pomsets. Without loss of generality, it is assumed that for every distinct v_0, v_1 in V, the sets $V_{f.v_0}$ and $V_{f.v_1}$ are mutually disjoint. Pomset $M[f]$ is defined as the pomset $[V_f, \Sigma_f, \leq_f, \mu_f]$, where $V_f = (\cup\ v :\ v \in V :\ V_{f.v})$, $\Sigma_f = (\cup\ v :\ v \in V :\ \Sigma_{f.v})$, $\leq_f = (\cup\ v :\ v \in V : \leq_{f.v}) \cup (\cup\ v_0, v_1 : v_0, v_1 \in V \wedge v_0 < v_1 : V_{f.v_0} \times V_{f.v_1})$, and $\mu_f = (\cup\ v :\ v \in V : \mu_{f.v})$.

An example of a point substitution is shown in Figure 5. For any pomset $[V, \Sigma, \leq, \mu]$, point $v \in V$, and pomset M, the point substitution $v := M$ is the substitution that assigns M to v and leaves all other points in V unchanged.

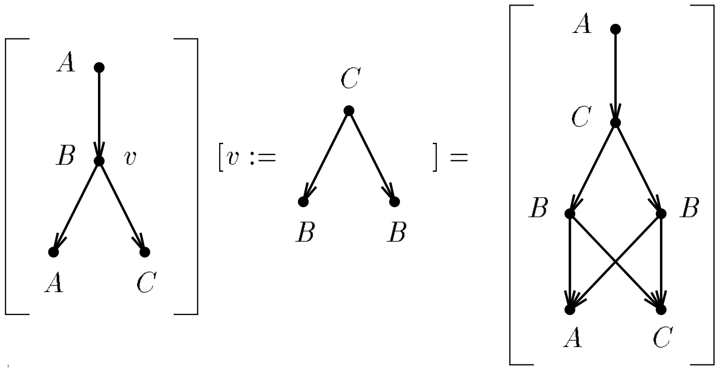


Figure 5: An example of point substitution.

Definition 5.2. A *context-free pomset grammar*, or CFPG, is a 4-tuple $\mathcal{G} = (\mathcal{N}, \mathcal{T}, \mathcal{R}, \mathcal{S})$, where $\mathcal{N}$ is a finite, non-empty set of *non-terminals*, $\mathcal{T}$ is a finite, non-empty set of *terminals*, $\mathcal{S} \in \mathcal{N}$ is the start symbol, and $\mathcal{R} \subseteq \mathcal{N} \times \mathcal{V}\ddagger$, where $\mathcal{V}$ denotes the union of $\mathcal{N}$ and $\mathcal{T}$, is the set of *production rules*.

The only difference between CFPGs and CFGs is that a production rule of a CFPG produces pomsets and a production rule of a CFG produces strings. Since a string is just a tomset, CFPGs are a true generalization of ordinary CFGs. A production rule (T, M) of a pomset grammar is often written $T \rightarrow M$.

For the purpose of parsing, CFPGs are *augmented*. An augmented grammar has the advantage that there is a unique production rule with the start symbol as the left-hand side.

Definition 5.3. If $\mathcal{G} = (\mathcal{N}, \mathcal{T}, \mathcal{R}, \mathcal{S})$ is a pomset grammar, then the corresponding *augmented* pomset grammar $\mathcal{G}' = (\mathcal{N}', \mathcal{T}', \mathcal{R}', \mathcal{S}')$ is the pomset grammar $(\mathcal{N} \cup \{\mathcal{S}'\}, \mathcal{T} \cup \{\#\}, \mathcal{R} \cup \{\mathcal{S}' \rightarrow \mathcal{S}\#\}, \mathcal{S}')$, where $\#$ is a special endmarker.

Every CFPG generates a pomset language.

Definition 5.4. For CFPG $\mathcal{G} = (\mathcal{N}, \mathcal{T}, \mathcal{R}, \mathcal{S})$ the binary relation on $\mathcal{V}\ddagger^2$ *produces in one step*, denoted by $\Longrightarrow$, is defined as follows. Let $M_0 = [V_0, \mathcal{V}, \leq_0, \mu_0]$ and M_1 be pomsets in $\mathcal{V}\ddagger$.

$M_0 \Longrightarrow M_1 \;\Leftrightarrow\; (\exists\, v, M :\; v \in V_0 \;\wedge\; \mu_0.v \rightarrow M \in \mathcal{R} :\; M_1 = M_0[v := M])$.

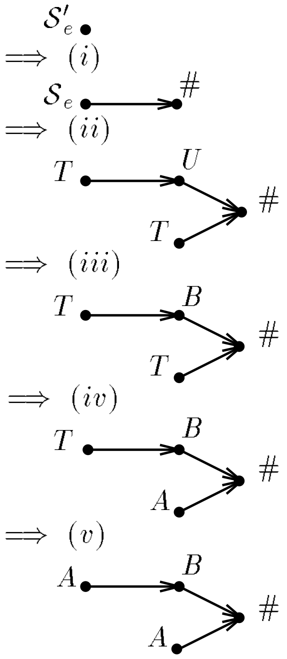
The reflexive, transitive closure of $\Longrightarrow$ is denoted by $\stackrel{*}{\Longrightarrow}$. The pomset language generated by a pomset M_0 in $\mathcal{V}\ddagger$, denoted $\mathcal{L}.M_0$, is defined as follows:

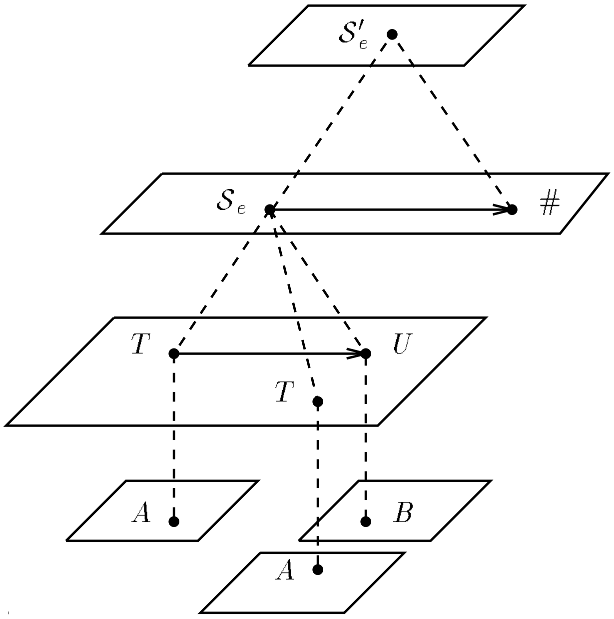
$\mathcal{L}.M_0 = \{M_1 \in \mathcal{T}\ddagger \mid M_0 \stackrel{*}{\Longrightarrow} M_1\}$.

The pomset language generated by grammar $\mathcal{G}$, denoted $\mathcal{L}.\mathcal{G}$, is defined as the pomset language generated by the start symbol $\mathcal{S}$, i.e. $\mathcal{L}.\mathcal{G} = \mathcal{L}.\mathcal{S}$.

A sequence of steps resulting in some pomset M is called a *derivation* for M. In general, there are many possible, not essentially different derivations for a pomset. Such derivations can be depicted in a so-called *pomset derivation tree* or *pomset parse tree*. Pomset derivation trees generalize derivation trees known from string-language theory.

Example 5.5. Consider the CFPG $\mathcal{G}_e = (\mathcal{N}_e, \mathcal{T}_e, \mathcal{R}_e, \mathcal{S}_e)$ where $\mathcal{N}_e = \{\mathcal{S}_e, T, U\}$, $\mathcal{T}_e = \{A, B\}$, and $\mathcal{R}_e = \{\mathcal{S}_e \to TU \,\&\, T, T \to A, U \to B\}$. The augmented SCFPG $\mathcal{G}'_e$ generates the pomset $(AB\&A)\#$. One of its derivations and its pomset parse tree are as follows. The steps in the derivation are numbered for future reference. Note that the last three steps can be done in any order, which yields six different derivations for the same pomset.





The next definition defines two types of derivations that are of particular interest in the context of parsing.

Definition 5.6. Let $\mathcal{G} = (\mathcal{N}, \mathcal{T}, \mathcal{R}, \mathcal{S})$ be a CFPG. Consider a step $M_0 \Longrightarrow M_1$ of some derivation. Assume that $M_0 = [V_0, \mathcal{V}, \leq_0, \mu_0]$ and that $M_1 = M_0[v := M]$, for some $v \in V_0$ and some $M \in \mathcal{V}\ddagger$. The derivation step is *leftmost* if and only if $(\forall\, v_0 :\ v_0 \in V_0 \wedge v_0 < v :\ \mu_0.v_0 \in \mathcal{T})$. It is *rightmost* if and only if $(\forall\, v_0 :\ v_0 \in V_0 \wedge v < v_0 :\ \mu_0.v_0 \in \mathcal{T})$. A derivation is called leftmost (rightmost) if and only if every step is leftmost (rightmost).

A derivation is leftmost (rightmost) if and only if in each step a point labeled with a non-terminal is substituted such that all predecessors (successors) of this point are labeled with terminals. This is the natural generalization of leftmost and rightmost derivations known from CFGs. The derivation in Example 5.5 is a rightmost derivation. Leftmost and rightmost derivations are the basis for two well-known parsing techniques called *LL* parsing and *LR* parsing respectively.

Since the production rules of CFPGs are allowed to have arbitrary pomsets as their right-hand side, CFPGs are more general than is necessary to define regular pomset languages. Furthermore, there is no obvious way to adapt existing parsing algorithms for CFPGs. Therefore, the class of CFPGs is restricted to the subclass of *simple* CFPGs, or SCFPGs, that is sufficient to describe regular pomset languages. The right-hand side of a production rule of an SCFPG consists of two concurrent strings. Since the right-hand side of a production rule of an ordinary context-free grammar consists of a single string, it seems possible to adapt existing parsing algorithms.

Definition 5.7. A CFPG $\mathcal{G} = (\mathcal{N}, \mathcal{T}, \mathcal{R}, \mathcal{S})$ is *simple* if and only if $\mathcal{R} \subseteq \mathcal{N} \times (\mathcal{V}^* \& \mathcal{V}^*)$.

The next theorem shows that SCFPGs are powerful enough to generate the entire class of regular pomset languages. The proof can be found in [2, page 72]. The proof is also of practical use because it describes transformations from concurrent regular expressions to SCFPGs.

Theorem 5.8. The class of pomset languages generated by SCFPGs coincides with the class of regular pomset languages.

5.2 An introduction to *PLR* parsing

In the area of compiler design, many parsing algorithms have been developed. Well-known algorithms are those based on the *LL*- and *LR*-parsing techniques [1, 17]. Theoretically, *LR* parsers are the most powerful and most general class of deterministic parsing algorithms. In practice, *LR* parsers have one disadvantage. For large grammars, the amount of storage required for an implementation is very large. A generalization of *LR*-parsing techniques to pomset languages will probably suffer from the same problem. However, the concurrent regular expressions that are of practical interest are not likely to yield large pomset grammars compared to the grammars used to describe programming languages. Therefore, since *LR* parsing is the most powerful parsing technique, it seems worthwhile investigating whether this technique can be generalized to pomset languages.

LR parsing was first introduced by Knuth in 1965. The name is derived from the fact that *LR* parsers scan the input from *left* to right and in doing so, try to reconstruct a *rightmost* derivation in reversed order. This subsection gives an introduction to *PLR* parsing. The basic operation is similar to the working of *LR* algorithms.

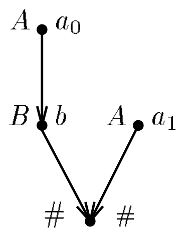
The first step is to define a suitable format of the input to *PLR*-parsing algorithms. Essentially, a pomset is a partial order plus a labeling function. A partial order can be represented by a set of pairs, consisting of a point and its immediate predecessors. Therefore, a pomset can be represented by a set of *triples* consisting of a point, its label, and its immediate predecessors. A list of such triples can be read easily by *PLR*-parsing algorithms. However, two more conditions must be satisfied. First, it must be guaranteed that before reading a triple, the algorithm has already read all the points in the set of predecessors. Second, the algorithm must be able to detect the end of the input.

To fulfill the first condition, the order in which the triples are read must be such that the total ordering imposed on the set of points is a linearization of the pomset.

Definition 5.9. Let $(V, \leq)$ be a poset. The pair $(V, \leq_t)$ is a linearization of $(V, \leq)$ if and only if, for any $u, v \in V$, $u \leq v \;\Rightarrow\; u \leq_t v$ and $\leq_t$ is a total ordering of V. Let $M = (V, \Sigma, \leq, \mu)$ be an lpo. The lpo $(V, \Sigma, \leq_t, \mu)$ is a linearization of M if and only if $(V, \leq_t)$ is a linearization of $(V, \leq)$. The definition for pomsets is analogous to the one for lpos.

The second condition can be fulfilled by adding a special triple to the input. Let M be the pomset to be recognized and let $\#$ be the special marker introduced earlier. The input to the algorithm is a sequence representing the pomset $M\#$ that satisfies the first condition. The end of such a sequence is marked by the special triple $(X_M, \#, \#)$, where X_M is the set containing all points in V_M without any successors and where $\#$, labeled with $\#$, is a new point not in V_M.

Example 5.10. Assume we wish to determine whether the pomset $AB\&A$ belongs to the pomset language generated by $\mathcal{G}_e$ (Example 5.5). The input to the recognition algorithm is the representation of the pomset $(AB\&A)\#$. The order in which this representation is read can be any of three possibilities:



Possible inputs:

$(\emptyset, A, a_1)(\emptyset, A, a_0)(\{a_0\}, B, b)(\{a_1, b\}, \#, \#)$
$(\emptyset, A, a_0)(\emptyset, A, a_1)(\{a_0\}, B, b)(\{a_1, b\}, \#, \#)$
$(\emptyset, A, a_0)(\{a_0\}, B, b)(\emptyset, A, a_1)(\{a_1, b\}, \#, \#)$

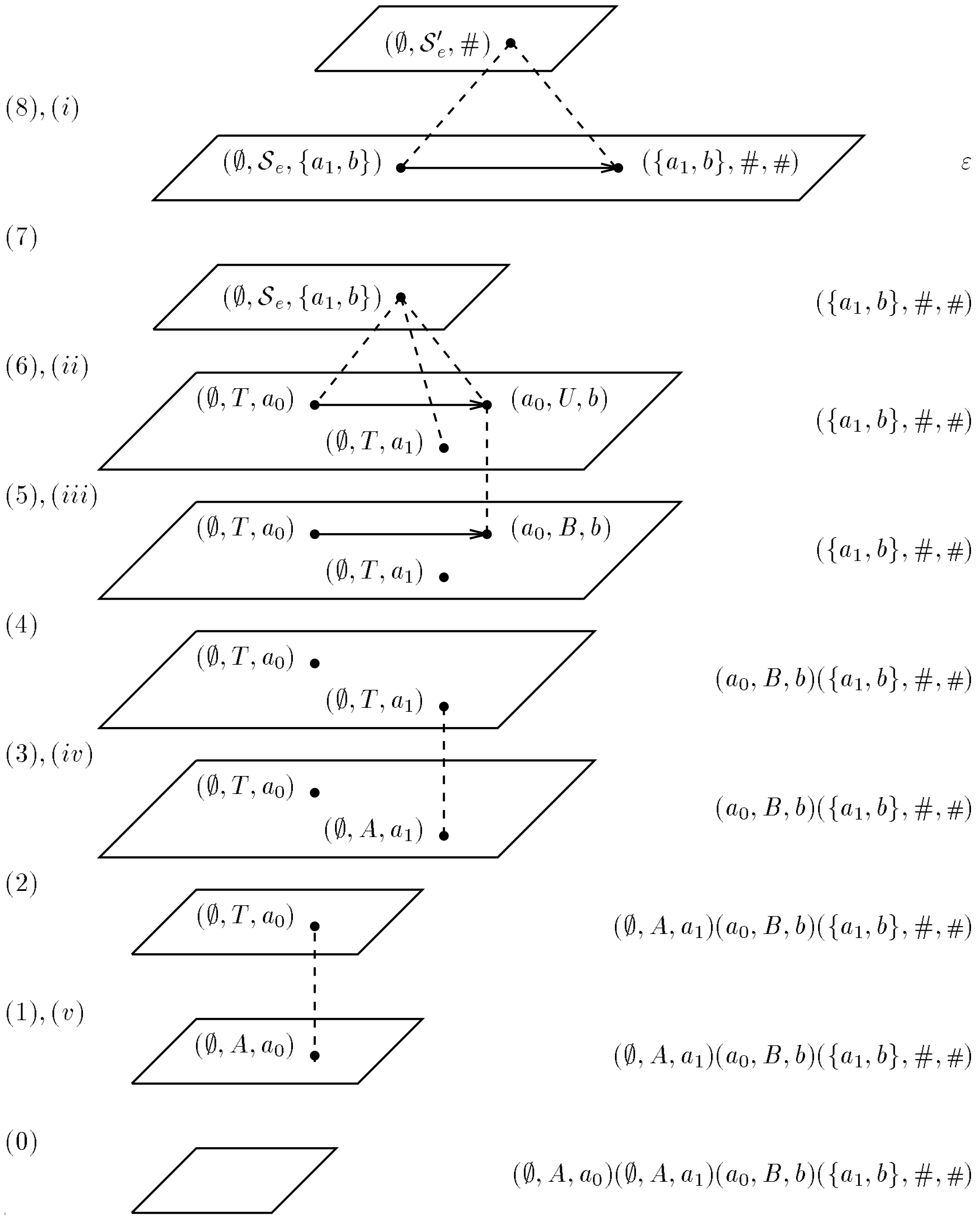
In order to improve understanding of the *PLR*-parsing process, it is useful to make a comparison with *LR*-parsing algorithms. *LR*-parsing algorithms are so-called bottom-up parsers. That is, an *LR*-parsing algorithm tries to reconstruct a rightmost derivation in reversed order and, doing so, it essentially builds a parse tree starting from the leaves and working its way up to the root. If the algorithm succeeds, the input is recognized. Otherwise, the algorithm either could not make a decision or recognized a syntax error.

A *PLR* algorithm is also a bottom-up parsers. It also tries to reconstruct a rightmost derivation. During the process, a *pomset parse tree* is constructed starting from the leaves. Again, only if the tree can be constructed all the way up to the root, is an input recognized successfully.

LR parsers are members of the family of *Shift-Reduce* parsers. *Shift-Reduce* parsers essentially work as follows. A *stack* contains (terminal and non-terminal) symbols that have already been parsed. Input symbols are read and *shifted* onto the stack until the right-hand side of some production rule is matched. Then, the parser executes a *reduce* action. The symbols that match the right-hand side are removed from the stack and the left-hand side of the production rule, which is the parent node in the parse tree, is pushed on top of the stack so that it can be used for future reduce actions. Every reduce action corresponds to a step in a rightmost derivation for the input. If the input is read completely and the stack contains only the start symbol, then the input is recognized successfully. This essentially describes the operation of any *Shift-Reduce* parser. The details vary for every member of the *Shift-Reduce* family. The stack usually contains some encoding of the symbols already parsed instead of the symbols themselves. Furthermore, every *Shift-Reduce* parser has a different way of determining what the next parse action should be.

PLR-parsing algorithms are also *Shift-Reduce* parsers. The following example illustrates the *PLR*-parsing process.

Example 5.11. Consider again the grammar $\mathcal{G}_e$. A *PLR*-parsing algorithm that gets the second input string of the previous example reconstructs the derivation in Example 5.5 in reversed order as depicted below. The process consists of nine steps, numbered (0) to (8). They should be read from bottom to top. The state of the parsing process before and after every step is described by the contents of the stack and the remainder of the input. The pictures in the middle represent the stack contents. The sequences on the right are the remainder of the input. Every time an input is read, it is shifted onto the stack. That is, it is added to the figure in the middle. If the right-hand side of a production rule is matched, a reduce action is executed. The matching symbols are removed from the picture and replaced by a triple with the left-hand side of the production rule as its second element. This non-terminal can be interpreted as an identifier of a pomset whose immediate predecessors are defined by the first element of the triple and whose set of maximal points is the third element. The latter specifies the required predecessors for any possible succeeding pomset. Replacing a set of matched triples by a new triple depicts a reduce action. Every reduce action corresponds to a step in the derivation of Example 5.5. The corresponding number is added in the leftmost column. If the whole figure is compressed in the vertical direction, it strongly resembles the pomset parse tree.



This concludes the introduction to *PLR* parsing. In [2, Sections 4.3 and 4.4], the process is formalized yielding a *PLR*(0) algorithm and a proposal for a *PLR*(1) algorithm. As in other parsing formalisms, the numbers denote the number of lookahead symbols used to determine parsing actions. The practical use of these algorithms for the purpose of event abstraction is discussed briefly in the next section.

6 Conclusions and open problems

This paper discussed four basic aspects of event abstraction. First, a computational model was defined that accurately describes the primitive level of a computation. The model includes primitive events and their causal relationships. Vector time was introduced as a representation of causality. Second, pomset language theory was proposed as a simple, powerful formalism to specify behavior in terms of activity and causality. A condition was given that observed behavior must satisfy to match a specification in the pomset model. Third, the computational model was extended to abstract events. Causality among abstract events was defined and its representation in vector time was investigated. Finally, the recognition of abstract events was studied. The theory of *PLR* parsing was introduced as a basis for recognition algorithms.

Some of the abovementioned aspects need further study. Since primitive behavior is accurately described by a partial order, Section 3 proposes the pomset model as a useful formalism for specifying abstract behavior. However, the precedence relation on *abstract* events is not necessarily a partial order. This means that the pomset model might not be meaningful for specifying multiple levels of abstraction. A better integration of the computational model and the pomset model is needed.

The discrepancy between the two models also means that *PLR* parsing cannot be used unchanged to build multiple levels of abstraction. In addition, the *PLR*-parsing formalism needs to be adapted to recognize *sub*pomsets. Besides adapting *PLR* parsing to event abstraction, the formalism itself needs to be further developed. The *PLR*(0) algorithm given in [2, Section 4.3] only recognizes a very limited class of pomset languages. The *PLR*(1) algorithm [2, Section 4.4] is only a proposal. A formal correctness proof is needed and the class of languages recognized by this algorithm needs to be studied. Other *PLR*-parsing algorithms might be needed.

Summarizing, although considerable progress is made in each of the four aspects of event abstraction, the integration of the computational model and the pomset model needs further study, as does the theory of *PLR* parsing and its application to event abstraction.

References

1. A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison–Wesley, Reading, Massachusetts, USA, 1988.

2. A.A. Basten. *Hierarchical Event-Based Behavioral Abstraction in Interactive Distributed Debugging: A Theoretical Approach.* Master's thesis, Eindhoven University of Technology, Department of Mathematics and Computing Science, The Netherlands, August 1993.

3. A.A. Basten, T. Kunz, J.P. Black, M.H. Coffin, and D.J. Taylor. Time and the Order of Abstract Events in Distributed Computations. To appear.

4. P.C. Bates. Shuffle Automata: A Formal model for Behavior Recognition in Distributed Systems. COINS Technical Report 87-27, University of Massachusetts, Computer and Information Science Department, Amherst, Massachusetts, USA, January 1987.

5. P.C. Bates. Debugging Heterogeneous Distributed Systems using Event-Based Models of Behavior. *ACM SIGPLAN Notices*, 24(1):11–22, January 1989.

6. P.C. Bates and J.C. Wileden. EDL: A Basis for Distributed System Debugging Tools. In *Proceedings of the 15th Hawaii International Conference on System Sciences*, volume I, pages 86–93, Honolulu, Hawaii, USA, January 1982.

7. J.P. Black, M.H. Coffin, T. Kunz, D.J. Taylor, and A.A. Basten. Linking Specification, Abstraction, and Debugging. CCNG Technical Report E–232, Computer Communications and Networks Group, University of Waterloo, Ontario, Canada, November 1993. Submitted 1993.11.04 to *IBM Systems Journal*, 26pp in ms.

8. B. Bruegge and P. Hibbard. Generalized Path Expressions: A High-Level Debugging Mechanism. *The Journal of Systems and Software*, 3(4):265–276, December 1983.

9. R.H. Campbell and A.N. Habermann. The Specification of Process Synchronization by Path Expressions. In E. Gelenbe and C. Kaiser, editors, *Operating Systems*, volume 16 of *Lecture Notes in Computer Science*, pages 89–102. Springer Verlag, Berlin, Germany, 1974.

10. K.M. Chandy and L. Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.

11. B. Charron-Bost. Combinatorics and Geometry of Consistent Cuts: Application to Concurrency Theory. In J.-C. Bermond and M. Raynal, editors, *Distributed Algorithms*, volume 392 of *Lecture Notes in Computer Science*, pages 45–56. Springer Verlag, Berlin, Germany, 1989.

12. B. Charron-Bost, F. Mattern, and G. Tel. Synchronous and Asynchronous Communication in Distributed Computations. Technical Report LITP 91.55, Institut Blaise Pascal, Université Paris 7, Paris, France, September 1991.

13. W.-H. Cheung. *Process and Event Abstraction for Debugging Distributed Programs*. Ph.D. Thesis, CCNG Technical Report T-189, University of Waterloo, Department of Computer Science, Waterloo, Ontario, Canada, 1989.

14. R. Cooper and K. Marzullo. Consistent Detection of Global Predicates. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 163–173, Santa Cruz, California, US, USA, May 1991.

15. C.J. Fidge. *Dynamic Analysis of Event Orderings in Message-Passing Systems*. PhD thesis, Australian National University, Department of Computer Science, Canberra, Australia, 1989.

16. C.J. Fidge. Logical Time in Distributed Computing Systems. *IEEE Computer*, 24(8):28–33, August 1991.

17. C.N. Fischer and R.J. LeBlanc, Jr. *Crafting a Compiler*. The Benjamin/Cummings Publishing Company, Menlo Park, California, USA, 1988.

18. J.L. Gischer. The Equational Theory of Pomsets. *Theoretical Computer Science*, 61(2,3):199–224, November 1988.

19. J. Grabowski. On Partial Languages. *Fundamenta Informaticae*, 4(2):427–498, 1981.

20. D. Haban and W. Weigel. Global Events and Global Breakpoints in Distributed Systems. In *Proceedings of the 21st Annual Hawaii International Conference on System Sciences*, volume II, pages 166–175, Kailua-Kona, Hawaii, USA, January 1988.

21. D.K. Hammer, J.P.C. Verhoosel, and K.H. Ecker. Refinement of Resource Constrained Executions for Modelling Dynamic System Aspects. This volume.

22. J.-M. Helary, N. Plouzeau, and M. Raynal. Computing Particular Snapshots in Distributed Systems. In *Proceedings of the 9th Annual International Phoenix Conference on Computers and Communications*, pages 116–123, Scottsdale, Arizona, USA, March 1990.

23. W. Hseush and G.E. Kaiser. Moddeling Concurrency in Parallel Debugging. *ACM SIGPLAN Notices*, 25(3):11–20, March 1990.

24. T. Kunz. Event Abstraction: Some Definitions and Theorems. Technical Report TI-1/93, Technische Hochschule Darmstadt, Fachbereich Informatik, Darmstadt, Germany, February 1993.

25. T.H. Lai and T.H. Yang. On Distributed Snapshots. *Information Processing Letters*, 25(3):153–158, May 1987.

26. L. Lamport. Time, Clocks and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.

27. H.R. Lewis and C.H. Papadimitriou. *Elements of the Theory of Computation.* Prentice–Hall, Englewood Cliffs, New Jersey, USA, 1981.

28. Y. Manabe and M. Imase. Global Conditions in Debugging Distributed Programs. *Journal of Parallel and Distributed Computing*, 15(1):62–69, May 1992.

29. F. Mattern. Virtual Time and Global States of Distributed Systems. In M. Cosnard et al., editor, *Parallel and Distributed Algorithms*, pages 215–226. Elsevier Science Publishers B.V., Amsterdam, North-Holland, The Netherlands, 1989.

30. F. Mattern. On the Relativistic Structure of Logical Time in Distributed Systems. *Bigre*, 78:3–20, March 1992.

31. B.P. Miller and J.-D. Choi. Breakpoints and Halting in Distributed Programs. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, pages 316–323, San Jose, California, USA, June 1988.

32. M.K. Ponamgi, W. Hseush, and G.E. Kaiser. Debugging Multithreaded Programs with MPD. *IEEE Software*, 8(3):37–43, May 1991.

33. V.R. Pratt. Modeling Concurrency with Partial Orders. *International Journal of Parallel Programming*, 15(1):33–71, February 1986.

34. J.A. Summers. *Precedence Preserving Abstraction for Distributed Debugging.* Master's thesis, University of Waterloo, Dept. of Computer Science, Waterloo, Ontario, Canada, 1992.

35. D.J. Taylor. A Prototype Debugger for Hermes. In *Proceedings of the 1992 CAS Conference, Volume I*, pages 29–42, Toronto, Ont., Canada, November 1992. IBM Canada Ltd. Laboratory, Centre for Advanced Studies.

36. G. Winskel. An Introduction to Event Structures. In J.W. de Bakker, W.P. de Roever, and G. Rozenberg, editors, *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, volume 354 of *Lecture Notes in Computer Science*, pages 364–397. Springer Verlag, Berlin, Germany, 1989.