True Concurrency =
Interleaving Concurrency + Weak Conflict

Leila Ribeiro Korff

Department of Computer Science
Federal University of Rio Grande do Sul
Porto Alegre, Brazil

Martin Korff

Nutec Informática
Porto Alegre, Brazil

Abstract
In this paper we show that true concurrency is not the same as interleaving concurrency for systems that allow read-access. More precisely, whenever two actions are interleavingly concurrent they are also truly concurrent, but the converse is not always true due to a phenomena that we call weak conflict, that allows the possibility of one writer to act in parallel with one (or many) readers.

1 Introduction
The Turing Machine views a computation as set of discrete computational steps. In each of these steps a single data item is read from a distinguished place, a new value is written there, and control passed over to the next step. Here reading can be considered as a special case of writing, where the old and the new values are the same. However processing data needs not necessarily be sequential. If there are (virtually) more than one processing unit, processing is potentially parallel, i.e., concurrent. Today all modern operating systems support concurrent processes. Depending on the possibility to address concurrency on the syntactical level, concurrency control is provided by executional models like Semaphores, Monitors, and Path Expressions. Among the standard examples used to evaluate the expressive power of such executional models are reader/writer problems. Readers and Writers are processes acting on a shared data value. A writer may change its data whereas a reader

---

1 This work was partly supported by CNPq and by the project QaP-For (FAPERGS).

©1998 Published by Elsevier Science B. V.
may not. Two writers may never act in parallel (the result would be undetermined), but there may be an arbitrary number of readers with simultaneous data access. Even the concurrent access of one writer and \( n \) readers should intuitively be allowed. If however reading would be considered as a special case of writing, as in described for the sequential models above, concurrency on a shared data would be excluded.\(^2\) We claim that modeling true concurrency adequately must reflect this essential difference between reading and writing.

In the following we are going to present graph grammars as a syntactical means to describe read/write access systems, and compare true concurrency and interleaving semantic models for graph grammars, showing that the true concurrency model allows for more parallelism than the interleaving one, and discussing the reason for this difference, which we call weak conflict.

The aim of this paper is to be a position paper, rather than a technical paper containing all definitions and proofs. We will give references to papers where the formal theory can be found. The structure of the paper is as follows: Sect. 2 introduces graph grammars informally, Sect. 3 contains the descriptions of an interleaving and a true concurrency semantics for graph grammars, Sect. 4 compares the two semantic models and Sect. 5 discusses the results of the comparison and their implications.

2 Graph Grammars

Graphs are a very natural means to explain complex situations on an intuitive level. Graph rules may complementary be used to capture the dynamical aspects of systems. The resulting notion of graph grammars generalizes Chomsky grammars from strings to graphs. As graphs may naturally describe the distribution of a system and rules may be applied simultaneously at different parts (overlapping or not) of the same graph, graph grammars appear as a suitable formalism to describe concurrent and distributed systems [4,11,3,16]. The basic idea is that the states of a system are modeled by graphs and states changes by transformations of graphs described by rules. An example of a graph grammar \( GG \) is given in figure 1. The initial state of the system is the graph \( I \), and the rules are \( r_1, r_2 \) and \( r_3 \). Left and right hand sides are connected by a partial mapping (morphism) which was omitted in the pictures. In the graphical notation, everything that is on the left- and right-hand side of a rule is preserved, everything that is on the left- and not on the right-hand side is deleted and everything that is on the right- and not on the left-hand side is created. For example, in rule \( r_1 \), the vertex \( \bullet \) is preserved, the vertex \( ■ \) is deleted and the loop edge is created.

The operational semantics of a graph grammar is given through rule applications: first, a match of the left-hand side of the rule must be found in the actual graph \( I_N \), and then all items that shall be deleted by this rule are

\(^2\) It remains only concurrency on different (non-shared) shared data.
deleted from $IN$ (this may also require the deletion of dangling edges) and all items that shall be added by the rule are added. In the (SPO-) algebraic approach to graph grammars, the application of a rule to a match is modeled as the corresponding pushout in the category of graphs and partial graph morphisms. In Figure 2 we can see three subsequent applications of rules of $GG$: steps $s_1$, $s_2$ and $s_3$. This is called a sequential derivation of the grammar $GG$. The sequential semantics of a graph grammar is the class of all sequential derivations that can be obtained having its initial graph and using the rules of the grammar.

Graph grammars can be used to specify a variety of systems. In particular they are very suitable for object systems because of the ability of preserving items (that can be used to model persistent object identities). A modeling of actor systems using graph grammars can be found in [6,7]. The existence of operations among graph grammars that are compatible with their semantics makes it possible to specify complex systems in a modular way, as the telephone system specified using graph grammars in [14,15]. Although the research area of graph grammars and graph transformations is relatively young — its roots date back to the early seventies — methods, techniques, and results in this area have already been studied and applied in a variety of fields in computer science such as formal language theory, pattern recognition and generations, compiler construction, software engineering, concurrent and distributed system modeling, database design and theory, etc (see [13]).

Graph Grammars can also be considered as a generalization of Petri nets [2,9], allowing dynamical topology changes and references among tokens. These
properties are quite useful in the modeling of growing and shrinking communities of objects that make references to each other.

3 Concurrency Semantics

Sequential derivations are concurrently equivalent whenever they only differ w.r.t. the sequential order of ‘their’ rule applications. Figure 3 shows two equivalent sequential derivations, derivation $\sigma_1 = s_1; s_3'$ and derivation $\sigma_2 = s_3; s_1'$. To find out whether steps $s_1$ and $s_1'$ represent “the same” action we must verify that they use the same rule ($r_1$) and $m_1 = d_3^{-1} \circ m_1'$ (that is, the match is “the same”).

These equivalence classes of sequential derivations may explicitly be represented. In [10] a canonical derivation sequence was used in which exchangeable (independent) steps were successively replaced by corresponding applications of parallel rules. This construction, however, is not so easy to perform because the independent steps may not be subsequent in a derivation sequence (as in figure 2), and it did not give raise to a partial order among sequential steps. Here, as in [7,8], an equivalence class of sequential derivations is represented by a concurrent derivation. It is obtained from gluing all input/output graphs of the sequential derivation: edges/vertices in the resulting core (glued) graph are canonical representations of ‘their’ sequential instances. Figure 4 shows a concurrent derivation $\kappa$ obtained from the sequential derivation $\sigma$, written $\sigma \leadsto \kappa$.

Each rule application in a concurrent derivation is called an action. The actions in figure 4 are $a_1$, $a_2$ and $a_3$. Dependencies among actions can directly be obtained by considering the core graph items being accessed. These dependencies give raise to relations that can be used to verify whether or not actions may be executed concurrently.
One of the relations between actions that can be described is the sequentialization relation:

**Sequentialization** (\(<<\)): One action must occur before the other in some sequential derivation. This relation is based on two situations: i) if one action \(a\) creates something that is needed by another action \(b\), then \(a << b\); ii) if some action \(c\) deletes something that is preserved by some action \(d\) then we have that \(d << c\) (because in any sequence that contains these two actions, \(d\) must occur first). This relation is a partial order. In the example (see figure 4) we have that \(a_1 << a_2, a_2 << a_3, a_1 << a_3\) (the last pair is obtained by transitivity).

This relation can then be used to describe the interleaving concurrency of a graph grammar:

Two actions are **interleavingly concurrent** if and only if they are not related by the sequentialization relation.

Note that this definition is equivalent to the usual one that says that two actions are interleavingly concurrent if and only if they can be observed in both orders (in different sequential derivations).

Using the sequentialization relation we can find out which sequential derivations give raise to a particular concurrent derivation: we just have to take any total order that is compatible with the sequentialization and construct the corresponding sequence of derivation steps. This construction is called a sequentialization of a concurrent derivation.

As a formal result that have been obtained is a one-to-one relation between all sequentializations of a concurrent derivation and the corresponding set of concurrently equivalent sequential derivations, that is, all the sequential derivations from which this concurrent can be obtained [8].

### 3.2 Causality Relation

True concurrency semantical models typically rely on the causality relation between actions (and not on a sequentialization relation). This relation can be described as:
Causality ($\leq$): One action creates an item that is used (deleted/preserved) by another action. The causality relation a partial order. In the example we have that $a_1 \leq a_2$ because the $\square$ needed by $a_2$ was created by $a_1$.

Concurrency here is defined via absence of causal relationships: Two actions are truly concurrent if and only if they are not related by the causal relation.

4 True Concurrency and Interleaving Semantics

There are different ways to define relationships among concurrent derivations. We will show two that are particularly interesting from the computational point of view: the sequential and the concurrent prefix relationships. These two kinds of relationships give raise to different semantics to graph grammars: the interleaving and the true concurrency semantics.

Intuitively, a sequential action prefix relates two derivations $\kappa_1$ and $\kappa_2$ when for any sequentialization $\sigma_1$ of $\kappa_1$, there is a sequentialization $\sigma_2$ of $\kappa_2$ that has $\sigma_1$ as its “beginning”. Figure 5 shows three concurrent derivations, where $\kappa_1$ is a sequential action prefix of $\kappa$, but $\kappa_2$ is not a sequential action prefix of $\kappa$ (because the only sequentialization of $\kappa_2$ would be $a_1; a_3$, and the only sequentialization of $\kappa$ is $a_1; a_2; a_3$).

Formally, a sequential prefix relation is a relation between actions of two
concurrent derivations that preserves and reflects the sequential and causality relations. Using sequential prefixes we can define an interleaving semantics for graph grammars:

The **interleaving semantics** of a graph grammar is a category containing all concurrent derivations (the computations) of this grammar as objects and all sequential prefix relations as morphisms.

A **concurrent action prefix** is a relation that preserves the sequential and the causality relations, but only reflects the latter one. In the example of figure 5, we have that κ1 as well as κ2 are concurrent prefixes of κ. The intuitive idea behind the concurrent prefix is that the computation κ2 can evolve to the computation κ by executing the action a2 concurrently with the others involved in κ2 (in fact, action a2 can only occur after a1, but it can occur in parallel with a3 because they overlap in an item—the triangle—that is read-only accessed by a2).

The **true concurrency semantics** of a graph grammar is a category containing all concurrent derivations (the computations) of this grammar as objects and all concurrent prefix relations as morphisms.

The categories of concurrent derivations described here can be seen as partial orders of computations, where the ordering describes possible evolutions of the system (in fact, these categories are complete partial orders). In these categorical semantics, we can find out whether actions a1 and a2 can be executed concurrently by finding concurrent derivations κ1 and κ2 containing these two actions respectively, and an upper-bound for these derivations.

Now if we observe the two kinds of concurrency semantics we have defined, we can see that they are different: some actions that are truly concurrent may not be interleavingly concurrent. This can also be seen if we compare the categories: the true concurrency semantics has more morphisms than the interleaving semantics, and thus allows for more upper-bounds. The source of this difference are cases in which there is one writer acting in parallel with one (or many) reader(s). This kind of concurrency is allowed by our truly concurrent model because the readers do not causally depend on the writer (this is what one intuitively would expect, although some true concurrency models do not allow this kind of parallelism). The interleaving semantics can not allow this kind of parallelism because we are forced to sequentialize these actions, and reading before differs from reading after the writing.

### 4.1 Weak Conflict

The **weak conflict** relation captures exactly the special cases described above:

**Weak conflict (≤#):** One action deletes something that is used by another action. For example, in Figure 4 we have that a2 ≤# a3 because the ▲ preserved by a2 is deleted by a3. This conflict is called weak because it is asymmetric: the occurrence of a3 excludes the possibility of a2 to occur,
but not vice versa. Note that a (symmetric) conflict situation \((a \leq \# b \text{ and } b \leq \# a)\) can never happen in a concurrent derivation because this would mean that two mutually exclusive actions occurred in a sequential derivation, and this is impossible.

4.2 Comparison between True Concurrency and Interleaving Semantics

By considering our definitions above, we arrive in the following formula:

\[
\text{True Concurrency} = \text{Interleaving Concurrency} + \text{Weak Conflict}
\]

In models that do not allow read-access, true concurrency coincides with interleaving because the weak conflict relation is empty in this case. This is also the case if we allow read-access, but not read/write-access (in this case, usually the causality relation if defined in such a way that actions in weak conflict become causally related). But if we allow read- and read/write-access, a truly concurrent model allows for more parallelism than an interleaving model. This kind of concurrency semantics can be used for other specification formalisms, for example, to give a semantics for contextual nets [1], which are Petri nets that allow read-access to tokens, or for actor systems [1].

5 Conclusion

In this paper we have shown that graph grammars can be used to specify concurrent systems in which not only read-only items may be accessed in parallel, but also one writer and many reader processes may share data. We have provided two kinds of concurrency semantics for these systems, an interleaving and a true concurrent models, and have reached the conclusion that if a system is supposed to allow simultaneous read and write access to some data, the true concurrency semantics is more adequate (the interleaving semantics forbids this kind of parallelism).

The true concurrency semantics described here was shown to be equivalent to a prime algebraic domain [8]. This implies that we can transform the category that represents the true concurrency semantics into a prime event structure [17]. However, in doing so we have to choose which relation (the sequentialization or the causality) shall be the causal relationship of the event structure. Choosing the causality relation would lead to an event structure in which the configurations would not correspond to actual sequential derivations of the grammar (because in an event structure, if two events can occur concurrently they can occur in either order, too). But if we choose the sequentialization relation, the corresponding event structure allows for less parallelism than the concurrent derivation semantics (read/write parallelism is ruled out). Thus, event structures seem to be inadequate as a model of concurrency for read/write systems as described here. The problem is that one vital information is missing: the weak conflict relation.

In the future, one of the topics of our research is to study better the
structure of concurrency, trying to extend the event structure model with a further relation. A first approach can be found in [5], but the axioms defining this relation there were not suited to model the weak conflict relation.

Another research topic is to verify the suitability of the semantics based on categories of concurrent computations (derivations) to other formalisms than graph grammars, such as models of actor systems and process algebras.

Verification techniques based on these semantics models are also a research topic that can be developed within the cooperation.

References


