Weak memory models using event structures

Simon Castellan

Weak memory models using event structures

Simon Castellan †

†: *ENS de Lyon, LIP, CNRS, Inria, UCBL, Université de Lyon*
46 allée d'Italie, 69364 Lyon cedex 07
simon.castellan@ens-lyon.fr

Abstract

In this article, we investigate a denotational semantics based on event structures for a very simple imperative and concurrent programming language. The model incorporates behaviours of weak memory models such as reordering of instructions and non-locality. Our model can then be used to define a *function* from programs to their possible outcomes that can be used to give a formal semantics to a processor or a programming language.

Most of the semantic ideas come from game semantics and its recent development based on event structures, but taking advantage of the first-order setting, we present in this paper a self-contained simplification of these ideas.

1. Introduction

Compiler and hardware optimizations The first guideline to model concurrency was Lamport's sequential consistency (SC) [11] which states that the semantics of concurrent program is the set of its possible sequentializations, called *interleavings*.

This model although very simple to reason with, is nowadays disconnected from reality. To improve performance of sequential programs, compilers and processors aggressively reorder instructions concerning distinct parts of the memory. As a result, it is now common to observe executions of concurrent programs that break sequential consistency because of these optimizations. For instance, the parallel execution of the following threads on a modern processor can lead to the end result $r_1 = 0 \wedge r_2 = 0$ which no interleaving predicts:

$$\begin{array}{l||l} a := 1 & b := 1 \\ r_1 \leftarrow b & r_2 \leftarrow a \end{array}$$

In this snippet, a and b stand for shared variables (initialized to 0) accessible to both threads while r_1 and r_2 denote registers that are thread-local. Assignments of shared variables ($:=$) are called *stores* whereas assignments from shared variables to registers ($\leftarrow$) are called *loads*.

The processor (or the compiler) can decide to permute the memory operations within a thread of this program as they hit distinct variables. The sequential behaviour of each thread is not altered by this optimization but once the two threads are put in parallel this turns out to break SC.

Moreover, modern architectures have caches so that threads do not share the same view on the memory. For instance in the following example appearing in the Intel documentation:

$$\begin{array}{l||l} a := 1 & b := 1 \\ r_1 \leftarrow a & s_1 \leftarrow b \\ r_2 \leftarrow b & s_2 \leftarrow a \end{array}$$

it can be observed that $r_1 = s_1 = 1$ but $r_2 = s_2 = 0$ even if the platform is not allowed to permute the memory reads on a and b of each thread.

Hardware vs software specifications These optimizations create the need for a clear specification of which behaviours are allowed on a given platform. Indeed, on the one hand, processor manufacturers need to specify which behaviours assembly programs have, to provide the user with guarantees. Those specifications define a particular *execution model* that the processor exhibits and as a consequence we possess robust formalizations of them (see eg. [2]).

On the other hand, programming language designers need to explain the possible run-time behaviour of valid programs so users can reason (informally or formally) about it. Because a programming language can be implemented on several architectures, and compilers should be allowed to optimize the program, those specifications are much more relaxed. As a result, their formalization is often non-satisfactory: Java's specification is unsound [15, 9] and C11's specification allows undesired "out of thin air" behaviours [3].

In this article we will focus on modelling behaviours appearing in hardware specifications.

Mathematical description of executions Such specifications should be as formal as possible to avoid ambiguities. This requires a mathematical language in which to express the desired constraints. Two such languages are popular: an axiomatic approach is followed by [12, 2] which gives axioms satisfied by valid executions and an operational approach based on modeling the platform as an idealized machine [14, 4].

In this paper we follow a different approach: as [9] we would like to develop a denotational semantics based on causality between memory events to observe concurrency as a first-class notion and reduce the size of the model. Unlike this paper which works with an abstract structure called *configuration theories*, we work with *event structures* [17] which are a concrete representation for specific configuration theories. Event structures comprise a set of events equipped with a partial order representing causality and a binary relation denoting *conflict* between events that are mutually exclusive. They are much more concrete and compact.

An advantage of this approach is that the resulting semantics is *compositional*, derived by induction on the syntax and not extracted from an operational semantics. We believe that compositional semantics and modular reasoning are key to be able to scale to large programs. Moreover, using causality instead of traces (as in eg. [5]) allows us to escape the combinatorial explosion inherent to trace-based models of concurrency.

Game semantics based on event structures A recent line of work [13, 8, 6] has developed a game semantics for concurrent higher-order languages based on event structures. In this article, we want to take advantage of the model and the constructions to give a semantics to a simple first-order "toy" language, complex enough to illustrate the phenomenon at stake.

This paper assumes no prior knowledge of these models: the construction of the model is self-contained. Ideas and intuitions directly come from seminal papers of game semantics [1, 10] and have been simplified in this first-order setting. Using this technology, it is very easy to extend our model to fully-fledged programming languages with functions, control, recursion, and higher-order features.

Contribution of the paper This paper explores the usage of game semantics techniques to give an event structures-based denotational account of weak memory model. In this paper, a particular model is chosen although we believe this framework can be adapted to a great variety of models. Where the axiomatic approach specifies *which* executions are valid, our approach builds directly the correct executions. Moreover, event structures allow to represent several incompatible executions in a compact form by sharing the common parts.

A naive implementation of the model presented here as well as a few variants is available at http://iso.mor.phis.me/memory.

Outline of the paper In Section 2, we define event structures and the constructions on them that we need in order to build our model. In Section 3, we introduce the language we study and outline the construction of the model. In Section 4, we interpret each term as a labelled event structure where the behaviour of variables is left abstract. These event structures represent the dependency between the instructions enforced by the platform. In Section 5, we explain how to wire a chosen memory model in the interpretation of Section 4 to recover a complete description of executions.

2. Event structures

As mentioned in the introduction, we use event structures to analyze both the concurrency and the non-determinism (due to races) inside programs.

2.1. Definitions and notations

In this article, we use event structures with binary conflict:

***Definition* 1** (Event structures)**.** — *An event structure is a tuple* $(S, \leq_S, \#_S)$ *where* S *is a set of events equipped with a partial order* $\leq_S$ *(called* causality of S*) and a binary symmetric irreflexive relation* $\#_S$ *(called* conflict relation of S*) satisfying the following two axioms:*

- (Conflict inheritance) *For all* $s, s', s'' \in S$ *such that* $s \#_S s'$ *and* $s' \leq s''$*, then* $s \#_S s''$ *(in this case, the conflict* $s \#_S s''$ *is said to be inherited from* $s \#_S s'$*)*
- (Finite causes) *For all events* $s \in S$*, the set* $[s] = \{s' \in S \mid s' \leq s\}$ *is finite.*

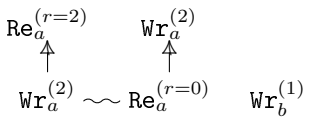
A labelled event structure is a pair (S, lbl) *where* S *is an event structure and* $\mathrm{lbl} : S \to L$ *is a function from the events of* S *to a set* L *of labels. We will often omit the labelling function when the context is clear.*

The causality of S has the following interpretation: if $s \leq s'$ then for the event s' to occur, the occurrence of s is necessary. In particular, causality is *conjunctive*: if $s_1 \leq s$ and $s_2 \leq s$ then the occurrences of s_1 *and* of s_2 are necessary to that of s. We write $s \rightarrowtriangle_S s'$ whenever $s < s'$ and there is no events in between s and s'. This relation is called the *immediate causality*.

The relation $\#_S$ represents the non-determinism of the system: two events in conflict cannot occur together in the same execution. By conflict inheritance, this relation is generated by a relation of minimal conflict: $s \ \sim_S s'$ whenever $s \#_S s'$ and for all $s_0 \leq s$ and $s'_0 \leq s'$ such that $s_0 \#_S s'_0$ then $s_0 = s$ and $s'_0 = s'$. Two events $s, s' \in S$ are said to be *concurrent* when they are incomparable for $\leq_S$ and they are not conflicting.

To give a graphical representation to a labelled event structure (S, lbl), we represent the labels, $\rightarrowtriangle_S$ and $\sim_S$ rather than $\leq_S$ and $\#_S$ as it is more compact.

Example 1. — The following event structure is the result of the interpretation in our model of the term $[\![r \leftarrow a \parallel a := 2 \parallel b := 1]\!]$ (variables are initialized to zero):



There is a race to determine in which order the load and the store on a are scheduled while the store on b is independent of the other two events.

The notion of *configuration* (states of the system) is central in the theory.

***Definition* 2** (Configuration)**.** — *A configuration of an event structure* $(S, \leq_S, \#_S)$ *is a* finite *subset* x *of* S *satisfying the two conditions:*

- (Down-closure) *If* $s \in x$ *then* $[s] \subseteq x$ *(ie.* x *is down-closed for* $\leq$*)*
- (Consistency) *If* $s, s' \in x$ *then* $\neg(s \#_S s')$.

The set of configurations of S is written $\mathscr{C}(S)$ and is naturally ordered by inclusion. A consequence of the axioms of event structures is that if $s \in S$, $[s] \subseteq S$ is a configuration of S.

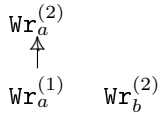
2.2. Simple parallel composition and sum of event structures

***Definition* 3** (Simple parallel composition)**.** — *Given two event structures* S *and* T *we form the event structure* $S \parallel T$ *called the simple parallel composition of* S *and* T*, defined by:*

- Events: $\{0\} \times S \cup \{1\} \times T$ *(Tagged disjoint union)*
- Causality: $\leq_{S\parallel T} = \{\big((0,s),(0,s')\big) \mid s \leq_S s'\} \cup \{((1,t),(1,t')) \mid t \leq_T t'\}$
- Conflict: $\#_{S\parallel T} = \{\big((0,s),(0,s')\big) \mid s \#_S s'\} \cup \{((1,t),(1,t')) \mid t \#_F t'\}$

The parallel composition $S \parallel T$ is the system obtained by letting the systems S and T evolve concurrently without interferences (conflict or causality). As a consequence, configurations of $S \parallel T$ are in one-to-one correspondence with pairs of configurations of S and T.

Example 2. — Parallel composition can be used to describe the semantics of threads that do not interfere. For instance, we have $[\![a := 1; a := 2 \parallel b := 2]\!] = [\![a := 1; a := 2]\!] \parallel [\![b := 2]\!]$ depicted as follows:

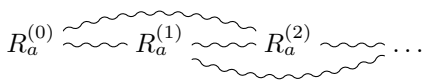


Non-deterministic sums of event structures are defined similarly as parallel composition except that S and T are set in conflict: executions (or configurations) are either contained in S or in T.

***Definition* 4** (Sum of event structures)**.** — *Given two event structures* S *and* T *we form the event structure* $S + T$ *defined by:*

- Events: *those of* $S \parallel T$
- Causality: *that of* $S \parallel T$
- Conflict: *that of* $S \parallel T$ *plus every pair* $((i,s),(j,t))$ *with* $i \neq j$.

Example 3. — Sums of event structures will be used to represent abstract load operations. For instance a load on a variable a whose context is not known can be represented as a sum of its possible outcomes (in any execution a given load reads only value):



These operations naturally extend to labelled event structures by letting $\mathrm{lbl}_{S_0\parallel S_1}(i,e) = \mathrm{lbl}_{S_0+S_1}(i,e) = \mathrm{lbl}_{S_i}(e)$.

2.3. Prefixing and concatenation

In the following, we will often want to "concatenate" event structures to represent sequential composition. To allow for the instructions to be reordered we need this composition not to be completely sequential in order to break some causalities. To that end, we introduce a relaxed concatenation operator.

***Definition* 5** (Concatenation)**.** — *Let $(S, \mathrm{lbl} : S \to L)$ and $(T, \mathrm{lbl} : T \to L)$ be labelled event structures and $R \subseteq L \times L$ be a relation satisfying:*

- *for any $t \in T$, there is a finite number of $s \in s$ such that $(\mathrm{lbl}(s), \mathrm{lbl}(t)) \in R$*
- *for any $t \in T$, there does not exist $s, s' \in S$ such that $s \#_S s'$ and both $s \leq_{S \otimes_R T} t$ and $s' \leq_{S \otimes_R T} t$.*

where $\leq_{S \otimes_R T}$ is defined as transitive closure of $\leq_{S \| T} \cup \{(0, s), (1, t) \mid (\mathrm{lbl}(s), \mathrm{lbl}(t)) \in R\}$. We form the labelled event structure $S \otimes_R T$ as follows:

- Events and labels*: Those of $S \parallel T$*
- Causality*: $\leq_{s \otimes_R T}$*
- Conflict*: $s \# s'$ iff there exists $s_0 \leq_{S \otimes_R T} s$ and $s'_0 \leq_{S \otimes_R T} s'$ such that $s_0 \#_{S \| T} s'_0$.*

In $S \otimes_R T$, S and T occur concurrently except for some causalities from S to T specified by R.

An instance of this is *prefixing*: let $(S, \mathrm{lbl} : S \to L)$ be an event structure and $l \in L$ a label. We write $l \cdot S$ for $\{l\} \otimes_{\{l\} \times S} S$. The event structure $l \cdot S$ starts by doing l first and then proceeds to do S.

3. Setting up the stage

In this section, we introduce the syntax for a simple concurrent imperative programming language. We then discuss the general structure of our model.

3.1. Syntax of our language

For the purpose of this article we introduce a very simple language which features enough interesting behaviours to illustrate the model. We suppose given two disjoint sets of variable names $a, b, c, \ldots \in \mathcal{S}$ for shared variables between threads and $r, s, \ldots \in \mathcal{R}$ for register (thread-local) variables. The syntax is as follows:

$$
\begin{array}{rll}
e, e' ::= & \{\ \textit{Arithmetic expressions}\ \} & \\
& k \in \mathbb{N} \mid r \in \mathcal{R} \mid e + e' & \\
t ::= & \{\ \textit{Threads}\ \} & \\
& \mid a := e; t & \text{(Store to a shared variable)} \\
& \mid r \leftarrow a; t & \text{(Load from a shared variable)} \\
& \mid () & \text{(Empty thread)} \\
p ::= & \{\ \textit{Programs}\ \} & \\
& t_1 \parallel \ldots \parallel t_n &
\end{array}
$$

We use two different syntactic constructs for loads and stores to make the distinction more explicit. Moreover, the construct $r \leftarrow a; t$ binds r to a inside t. We use the shorthands $r \leftarrow a$ and $a := e$ for

$$\frac{\mathrm{fv}(e) \subseteq \Delta}{\Delta \vdash e} \qquad \frac{a, \Gamma; r, \Delta \vdash t}{a, \Gamma; \Delta \vdash r \leftarrow a; t} \qquad \frac{\Delta \vdash e \qquad a, \Gamma; \Delta \vdash t}{a, \Gamma; \Delta \vdash a := e; t} \qquad \frac{\Gamma; \vdash t_1 \qquad \dots \qquad \Gamma; \vdash t_n}{\Gamma \vdash t_1 \parallel \dots \parallel t_n}$$

Figure 1: Typing rules for the language

$r \leftarrow a; ()$ and $a := e; ()$ respectively. Write $\mathrm{fv}(e)$ for the set of register variables appearing in e. To ensure well-formedness we use a very simple type system whose judgments are of the form $\Gamma; \Delta \vdash t$ for threads and $\Gamma \vdash p$ for programs. The context Γ is a set of global variables and Δ is a set of thread-local (or register) variables. Rules are given in Figure 1.

Variables are initialized to zero at the beginning of an execution.

3.2. Open and closed semantics

In the rest of the paper, we show an example of a possible model based on event structures allowing for instructions reordering and non-locality.

The specification of such a model can be split into two parts:

- The processor part which explains which instructions may or may not be permuted inside a thread.
- The memory part which explains how memory operations in one thread are propagated to the others.

We believe these two parts should be cleanly separated in the model and should be independent from one another as it is implicitly the case in existing models (eg. in [2]).

The first step is what we call *open semantics*: it is about computing the exact control flow of the program leaving the side effects aside. It interprets a program $\Gamma \vdash p$ as if it was a pure functional program $\lambda\Gamma.p$ where the operations of read and write are kept abstract and not specified and ; is not interpreted as sequential composition but as something that allows concurrency between instructions that can be reordered. This construction is presented in Section 4.

The second step is what we call *closed semantics*: it applies the $\lambda\Gamma.p$ we get from the first step to implementations of memory cells (one for each variable in Γ). Giving the semantic behaviour of these cells is exactly as giving the formal semantics of the memory: how reads and write are propagated along threads. This construction is presented in Section 5.

For each step, different choices can be made, corresponding to different architectures. In Section 4 and 5, we pick a possible choice for each component.

3.3. An example of open semantics: a sequentially consistent semantics

In this subsection, we quickly illustrate what we mean by open semantics on a simple example by defining an open semantics that does not allow reordering any instructions within a thread. Coupled with a sequential memory model, this would only exhibit the sequentially consistent executions.

Our semantics will map programs over a context Γ to event structures with labels in the set $L_\Gamma = \{\mathtt{Wr}_a^{(k)}, \mathtt{Re}_a^{(r=k)} \mid a \in \Gamma, k \in \mathbb{N}, r \in \mathcal{R}\}$. There is an obvious map $\mathtt{v} : L_\Gamma \to \Gamma$ defined by $\mathtt{v}(\mathtt{Re}_a^{(r=k)}) = a$ and $\mathtt{v}(\mathtt{Wr}_a^{(k)}) = a$.

Environments To handle thread-local variables, we will use an environment to pass down their values as we compute the interpretation. An environment will be a map $\rho : \Delta \to \mathbb{N}$ assigning to each

thread-local variable a value. Such an environment extends naturally to arithmetic expressions with variables in Δ by the following equations:

$$\rho(k) = k \quad \rho(e + e') = \rho(e) + \rho(e').$$

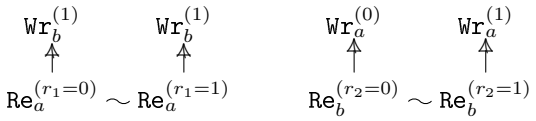
Interpreting threads Given a thread $\Gamma; \Delta \vdash t$ and an environment $\rho : \Delta \to \mathbb{N}$ we define a labeled event structure $[\![t]\!]_\rho$ as follows:

- $[\![a := e; t]\!]_\rho = \mathtt{Wr}_a^{(\rho(e))} \cdot [\![t]\!]_\rho$ – we perform the write on a and then continue to t.
- $[\![r \leftarrow a; t]\!]_\rho = \sum_{n \in \mathbb{N}} \mathtt{Re}_a^{(r=n)} \cdot [\![t]\!]_{\rho[r \leftarrow n]}$ – as the context is open, the model needs to account for all possible values, formalized as a non-deterministic sum of event structures.

Interpreting programs Given $\Gamma \vdash t_1 \parallel \ldots \parallel t_n$ a program, we define its open interpretation $[\![p]\!]$ as $[\![t_1]\!]_\emptyset \parallel \ldots \parallel [\![t_n]\!]_\emptyset$ where $\emptyset$ is the empty environment on the empty context.

This interpretation is very simple: threads are interpreted by *sequential* event structures: two events are either comparable or conflicting. There is no in-thread concurrency as no instructions can be permuted.

Example 4. — On the program $p = (r_1 \leftarrow a; b := 1) \parallel (r_2 \leftarrow b; a := r_2)$, this gives the following event structure ($\mathbb{N}$ is assumed to be $\{0, 1\}$ to simplify the picture):



4. Open semantics

In this section we develop an open interpretation that allows permuting operations that do not target the same variable. This behaviour can be found in the Alpha architecture [2]. In particular, we forbid permuting loads from the same variable even though this optimization is sequentially valid. This particular choice is simple to present as it is uniform but the theory supports other choices.

In the semantics, we represent instructions that can be reordered by *concurrent events*. The goal of this section is to build a semantics similar to the one of Section 3.3 which allows for concurrency inside a thread.

4.1. Operation dependency

The challenge of this interpretation is to determine what memory operations are causally related. Indeed, in the interpretation we want to replace the earlier definition:

$$[\![a := e; t]\!]_\rho = \mathtt{Wr}_a^{(\rho(e))} \cdot [\![t]\!]_\rho$$

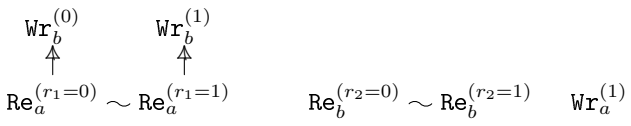
which is too sequential by something using the concatenation operator $\otimes_R$:

$$[\![a := e; t]\!]_\rho = \mathtt{Wr}_a^{(\rho(e))} \otimes_{\twoheadrightarrow_d} [\![t]\!]_\rho$$

for a well-chosen $\twoheadrightarrow_d$ which indicates when a label is a necessary cause to another one. We should have $\mathtt{Wr}_a^{(k)} \twoheadrightarrow_d s$ whenever $\mathtt{v}(s) = a$. But what about $\mathtt{Re}_a^{(r=k)}$? For instance, we do not know which stores depend on r: we can only observe the value written, not the arithmetic expression actually

computed. To that end, we need to enrich our labels to contain information about registers. But the problem is more subtle than that:

Example 5. — We want our semantics to compute the following event structure for the program $r_1 \leftarrow a; b := r_1 \parallel r_2 \leftarrow b; a := 1$:



We observe that $a := 1$ only generates one event where $b := r_1$ generates two. We have to track the dependency in r_1 of the store to b in the first thread and duplicate accordingly the write event, while the store of the second thread is independent of r_2 and is represented by a single event.

To solve this, we need to determine how many reads an operation depends on. Moreover, we need also to track the values read, not to put causal links between results of incompatible reads. This means that our labels should now carry a partial map $\mathcal{R} \rightharpoonup \mathbb{N}$ indicating which labels are used and what are their values. Defining $L'_\Gamma = [\mathcal{R} \rightharpoonup \mathbb{N}] \times L_\Gamma$, we can now define $\rightarrow_d \subseteq L'_\Gamma \times L'_\Gamma$ by the following equations:

$$(\rho, \mathtt{Wr}_a^{(k)}) \rightarrow_d (\rho', s) \equiv \rho \subseteq \rho' \wedge \mathtt{v}(s) = a \quad (\rho, \mathtt{Re}_a^{(r=k)}) \rightarrow_d (\rho', s) \equiv \rho \subseteq \rho' \wedge r \in \mathrm{dom}(\rho')$$

We do not need the environments to carry values of registers anymore: any operation that needs the value of a register will give rise to one event per value possibly read. However to compute which registers an operation depends on, we need to keep track of the register bindings. Indeed the thread $[\![a := e; t]\!]$ depends on any read on any register bound to a (ie. registers whose values are loaded from a). Therefore to define the semantics by induction, we will maintain an environment $\sigma : \Delta \to \Gamma$ containing exactly this information.

4.2. Semantics of threads

Assume we have a thread $\Gamma; \Delta \vdash t$.

Stores Suppose $t = (a := e; t')$, and an environment $\sigma : \Delta \to \Gamma$. The partial environment ρ for this thread is defined on $D = \mathrm{fv}(e) \cup \sigma^{-1}(a)$ where $\sigma^{-1}(a)$ is the set of registers bound to a in σ. We let:

$$[\![t]\!]_\sigma = \left(\sum_{\rho : D \to \mathbb{N}} \left(\rho, \mathtt{Wr}_a^{(\rho(e))} \right) \right) \otimes_{\rightarrow_d} [\![t']\!]_\sigma.$$

(Recall that $\rho(e)$ stands for the extension of ρ to arithmetic expressions with registers in D.) This picks a non-deterministic value for each read that has to occur before the write, performs the write and proceeds to t' letting $\otimes_{\rightarrow_d}$ insert the right causalities between the writes and t'.

Loads Suppose $t = (r \leftarrow a; t')$, and an environment $\sigma : \Delta \to \Gamma$. This time we let $D = \sigma^{-1}(a)$:

$$[\![t]\!]_\sigma = \left(\sum_{n \in \mathbb{N}} \sum_{\rho : D \to \mathbb{N}} \left(\rho[r \leftarrow n], \mathtt{Re}_a^{(r=n)} \right) \right) \otimes_{\rightarrow_d} [\![t']\!]_{\sigma[r \leftarrow a]}.$$

We proceed similarly to writes but we also pick non-deterministially the value that is read.

4.3. Semantics of programs

As before, given a program $\Gamma \vdash t_1 \parallel \ldots \parallel t_n$ we let $[\![t_1 \parallel \ldots \parallel t_n]\!] = [\![t_1]\!]_\emptyset \parallel \ldots \parallel [\![t_n]\!]_\emptyset$. This gives the desired event structure with enough concurrency to express possible reorderings of instructions. Having done the construction we project labels in L'_Γ to L_Γ by forgetting the register environment: it is not needed anymore.

Example 6 (Read/Write reordering). — Assume we have the thread $a, b \vdash r \leftarrow a; b := 1$ in the empty environment. Computing $[\![b := 1]\!]_{r \mapsto a}$ is easy as there is no dependency for this instruction in this environment: D will be empty. This yields the event structure with a single event labelled $(\emptyset, \mathtt{Wr}_b^{(1)})$. It follows that $[\![r \leftarrow a; b := 1]\!] = [\![r \leftarrow a]\!] \parallel [\![b := 1]\!]$ as those events are not in $\twoheadrightarrow_d$:

$$(\{r \mapsto 0\}, \mathtt{Re}_a^{(r=0)}) \rightsquigarrow\!\!\!\rightsquigarrow (\{r \mapsto 1\}, \mathtt{Re}_a^{(r=1)}) \qquad\qquad (\emptyset, \mathtt{Wr}_b^{(1)})$$

For the thread $a, b \vdash r \leftarrow b; a := r$ the situation is different. Computing $[\![a := r]\!]_{r \mapsto b}$ needs duplication as D will be $\{r\}$: the value written depends on r. This gives the following event structure $\sum_{n \in \mathbb{N}} (\{r \mapsto n\}, \mathtt{Wr}_a^{(n)})$. The whole thread gives then:

$$\begin{array}{ccc} (\{r \mapsto 0\}, \mathtt{Wr}_a^{(0)}) & & (\{r \mapsto 1\}, \mathtt{Wr}_a^{(1)}) \\ \Uparrow & & \Uparrow \\ (\{r \mapsto 0\}, \mathtt{Re}_b^{(r=0)}) & \sim & (\{r \mapsto 1\}, \mathtt{Re}_b^{(r=1)}) \end{array}$$

5. Memory models

Given the open interpretation of a program p, we would like to *close it* by restricting the scope of the shared variables to p – the environment cannot modify it anymore[1]. To do so, we need to define formally the behaviour of the memory: which sequences of memory operations are valid according to our model? Depending on the architecture different behaviours can be accepted. Throughout this section, for simplicity of presentation we make one assumption: the memory behaves as a server that receives commands (reads and write) in a *total order*, this assumption is however non-necessary. (See section 5.4).

***Definition* 6** (Linear memory model)**.** — *A* memory model *is a prefix-closed subset of* $(L_a)^*$*, the set of finite lists (or traces) of memory events concerning a single fixed variable* a*.*

Given a linear memory model T, we will write $T(b)$ for a given variable b to denote the set of memory events allowed on variable b simply obtained by substituting b for a in the traces of T. Note that in order to prove theorems quantified over all memory models, one would need more axioms in order to get only meaningful memory models, but the constructions presented here do not require them.

The goal of this section is to build the closed semantics $[\![\Gamma \vdash p]\!]_T$ of a program $\Gamma \vdash p$ with respect to a memory model T by eliminating behaviours that are not in T.

Example 7 (A sequential memory). — The first memory model that comes to mind is that of a sequential memory. Consider the grammar with non-terminal symbols $(T_k)_{k \in \mathbb{N}}$ defined as follows:

$$T_k ::= \mathtt{Re}_a^{(r=k)} \cdot T_k \mid \mathtt{Wr}_a^{(n)} \cdot T_n \mid \epsilon.$$

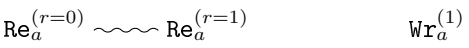
Then the language generated by $T_0 \subseteq (L_a)^*$ gives a memory model T representing a sequential memory cell initialized to zero: a load must read the last value written to it, or zero if there has been no write to it yet.

[1] This operation is similar to the restriction in π-calculus.

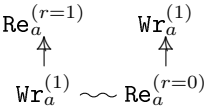
5.1. Closed semantics with respect to a memory model

In this section, we explain how to compute the closed semantics of a program $\Gamma \vdash p$ with respect to a specific memory model $T \subseteq (L_a)^*$. We start off by examining a simple example.

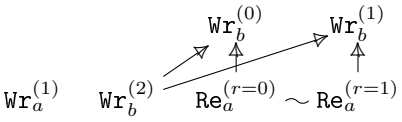
Example 8. — Consider the program $p = r \leftarrow a \parallel a := 1$. The open semantics of p is



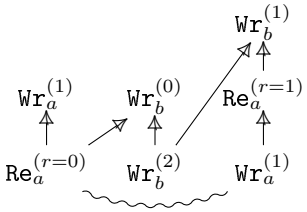
Against the sequential memory defined above, the desired result would be (the read and write must be sequentialized in certain order):



There are more events in this event structure than in the open semantics as the $\mathtt{Wr}_a^{(1)}$ event has been duplicated. Consider now $a, b \vdash a := 1 \parallel b := 2; r \leftarrow a; b := r$. We have the following open semantics:



Computing the closed semantics involves scheduling the operations on the two variables a and b in a consistent manner, which yields a more complicated picture:



As a consequence of the linearity assumption, two events concerning the same memory cell are either causally comparable or in conflict.

The configurations (or states) of the desired resulting event structures are easier to understand than the set of events: they always correspond to a configuration of the open semantics $[\![\Gamma \vdash p]\!]$ equipped with linearization information for each variable $a \in \Gamma$. Configurations of $[\![\Gamma \vdash p]\!]$ that have no possible linearizations disappear while configurations that have several per-variable linearizations yield several configurations of the result. We now make this intuition formal.

Projection It will be useful to project a configuration of an event structure to a particular variable. This is done through the notion of *projection*:

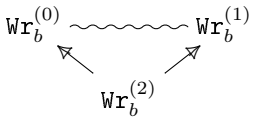
***Definition* 7** (Projection)**.** — *Given an event structure S and a set of events $V \subseteq S$, we form an event structure $S \downarrow V$ as follows:*

- *Events:* V
- *Causality:* $\leq_S \cap V^2$
- *Conflict:* $\#_S \cap V^2$

In $S \downarrow V$ only the events in V are *visible* while the others are thought to be occurring in background or *hidden*. Note that any configuration x of S induces a configuration $x \cap V$ of $S \downarrow V$.

Given $a \in \Gamma$ and (S, lbl) an event structure labelled in L_Γ, we write $S \downarrow a$ as a short-hand for $S \downarrow \{s \in S \mid \mathtt{v}(\mathrm{lbl}\, s) = a\}$ – the projection of S to the memory events concerning a. Any configuration $x \in \mathscr{C}(S)$ induces a configuration $x \downarrow a$ of $\mathscr{C}(S \downarrow a)$.

Example 9. — The projection of the open semantics of Example 9 gives:



This is a partial-order that can be linearized in several ways, each one representing a *trace*:

***Definition* 8** (Trace of a configuration)**.** — *Let (S, lbl) be a labelled event structure. A covering chain of $x \in \mathscr{C}(S)$ is a sequence $s_0, \ldots, s_n \in S$ such that $\{s_0, \ldots, s_n\} = x$ and for all $i \leq n$, $\{s_0, \ldots, s_i\}$ is a configuration of S. A trace of x is a sequence of the form $\mathrm{lbl}(s_0), \ldots, \mathrm{lbl}(s_n)$ where $s_0, \ldots, s_n$ is a covering chain of x.*

We write $\mathrm{tr}(x)$ for the set of traces of x.

We can define the desired configurations of our event structure as follows. Let $\Gamma \vdash p$ be a program and write $[\![\Gamma \vdash p]\!]$ its open interpretation. We define

$$\mathscr{S}_p = \{(x, (t_a)_{a \in \Gamma}) \mid x \in \mathscr{C}([\![\Gamma \vdash t]\!]) \text{ and } \forall a \in \Gamma, t_a \in \mathrm{tr}(x \downarrow a)\}.$$

The set $\mathscr{S}_p$ is naturally ordered: $(x, (t_a)) \sqsubseteq (y, (t'_a))$ if $x \subseteq y$ and for all $a \in \Gamma$, t_a is a prefix of t'_a.

Such pairs are the configurations of our program. To get an event structure S such that $(\mathscr{C}(S), \subseteq)$ is order-isomorphic to $(\mathscr{S}_p, \sqsubseteq)$, we use a construction similar to synchronized product of event structures [16]. First, for each $(x, (t_a)) \in \mathscr{S}_p$, we equip x with a partial order written $\leq_{(x,(t_a))}$:

$$s \leq_{(x,(t_a))} s' \text{ iff } \forall (x', (t'_a)) \sqsubseteq (x, (t_a)), s' \in x' \Rightarrow s \in x$$

This reads: s is below s' within $(x, (t_a))$ when for every sub-configuration of $(x, (t_a))$ where s' occurs, then s must occur as well.

***Proposition* 1** (Prime construction)**.** — *For a given memory model T, the following form an event structure written $[\![\Gamma \vdash p]\!]_T$:*

- Events*: $(x, (t_a)) \in \mathscr{S}_p$ such that $(x, \leq_{(x,(t_a))})$ has a greatest element (such a configuration is called* prime*).*
- Causality*: given by restricting $\sqsubseteq$ to the set of events.*
- Conflict*: $(x, (t_a)) \# (x', (t'_a))$ when*
  - *either there exist $s \in x$ and $s' \in x'$ such that $s \# s'$*
  - *or there exists $a \in \Gamma$ such that t_a and t'_a are* not *comparable for the prefix order.*

Proof. Since configurations are finite sets, any prime configuration $(x, (t_a))$ has a finite number of elements below it for $\sqsubseteq$, so in particular a finite number of prime configurations.

Moreover, assume that $(x, (t_a)) \# (x', (t'_a))$ and $(x', (t'_a)) \sqsubseteq (x'', (t''_a))$. If we have $s \in x$ and $s' \in x'$ such that $s \# s'$, the conflict is indeed inherited as $x' \subseteq x''$. Otherwise, write b for a variable such that t_b and t'_b are incomparable. It is easy to see that t_b and t''_b have also to be incomparable since the prefix order is a tree. □

We also have that any partial order of the form $(x, \leq_{(x,(t_a))})$ is order-isomorphic to a unique configuration of $[\![\Gamma \vdash p]\!]_T$, hence we have that $(\mathscr{C}([\![\Gamma \vdash p]\!]_T, \subseteq))$ is order-isomorphic to $(\mathscr{S}_p, \sqsubseteq)$.

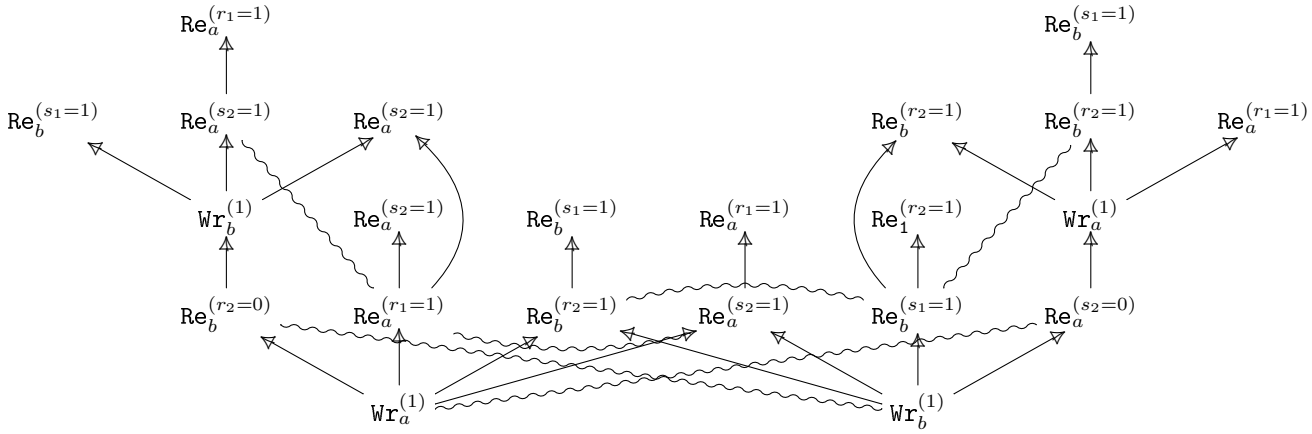


Figure 2: Closed semantics for a strict memory model of the store buffering example

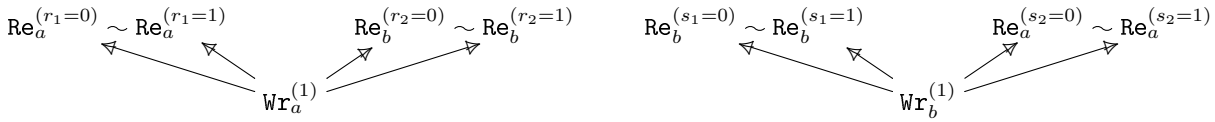
5.2. A relaxed memory model

In this section, we show an instance of a relaxed memory model that exhibits behaviours forbidden by the sequential model of Section 5.

Example 10 (Store buffering). — Consider the following program:

$$\begin{array}{l|l} a := 1 & b := 1 \\ r_1 \leftarrow a & s_1 \leftarrow b \\ r_2 \leftarrow b & s_2 \leftarrow a \end{array}$$

Assume an open semantics where Store/Load reorderings are not allowed so that for this program it would look like:



(We keep events $\mathtt{Re}_a^{(r_1=0)}$ and $\mathtt{Re}_b^{(s_1=0)}$ because the environment can modify it.)

Closing it with the sequential memory model defined above gives the large event structure of Figure 5.2. The key point is that there are unique events corresponding to reading $r_2 = 0$ and $s_2 = 0$ and they are in (inherited from the writes) conflict so the outcome $r_2 = s_2 = 0$ is not possible.

However, in some situations, $r_2 = s_2 = 0$ can be observed. This is an example of store buffering exhibited by Intel processors for instance: there might be delays between one a write is available locally and when a write is available globally.

To deal with that we need to make our model thread-aware in order to distinguish events from different threads. We assume now our labels carry an extra integer called the *thread-id* indicating the origin of this event: we now let $L_\Gamma^t = \mathbb{N} \times L'_\Gamma$. It is straightforward to extend our open interpretation of threads to depend on a integer representing the thread's id and then define

$$[\![\Gamma \vdash t_1 \parallel \ldots \parallel t_n]\!] = [\![\Gamma \vdash t_1]\!](1) \parallel \ldots \parallel [\![\Gamma \vdash t_n]\!](n).$$

With this, we can make a linear memory model that takes into account store buffering using a grammar similarly as at the beginning of Section 5. Now our symbol is indexed over pairs $(\rho, k) \in (\mathbb{N} \to \mathbb{N}) \times \mathbb{N}$ where k is the global value of the cell and $\rho(\iota)$ is the local value in thread ι. Writing $\rho[\iota \leftarrow x]$ to update environments, we define:

$$\begin{aligned} T'_{\rho,k} ::= {} & \mid \epsilon \mid \mathtt{Re}_{\iota,a}^{(r=k)} \cdot T'_{\rho[\iota\leftarrow k],k} && \text{(Read from the global memory)} \\ & \mid \mathtt{Re}_{\iota,a}^{(r=\rho(\iota))} \cdot T'_{\rho,k} && \text{(Read from the local cache)} \\ & \mid \mathtt{Wr}_{\iota,a}^{(n)} \cdot T'_{\rho[\iota\leftarrow n],n} && \text{(Write)} \end{aligned}$$

Then $T' \equiv T'_{(n\mapsto 0),0}$ gives a list of traces representing a memory model allowing for store buffering. The first axiom says that when a read fetches the value of the global memory, then this value is available to all threads. This is similar to the Value axiom of Sparc [2].

Example 11. — Considering again the program of Example 10, we can now compute the closed interpretation with the memory model T'. Since now we have that $\mathtt{Wr}_{1,a}^{(1)} \cdot \mathtt{Re}_{1,a}^{(r=1)} \cdot \mathtt{Re}_{2,a}^{(r=0)} \in T'(a)$, more behaviours are allowed in particular the one where $r_2 = s_2 = 0$. The event structure is too big to be drawn here (but can be visualized via the implementation).

5.3. Executions in our model

In our model, complete executions correspond to maximal configurations. In more detail, given a program $\Gamma \vdash p$, a possible execution of p in a memory model T is given by a configuration x of $[\![\Gamma \vdash p]\!]_T$ maximal with respect to inclusion: no more events can be added to it. From such data, we discuss here how one could recover an *execution witness* in the sense of [2]. In the open semantics, the program order of p is already lost because of the concurrency between events that correspond to instructions that can be executed in any order. What the open semantics gives us is the *preserved program order*: causalities from the program that the architecture cannot break.

We show here how to recover the *read-from* and *write-serialization* maps. We have seen in Section 5.1 that configurations of $[\![\Gamma \vdash p]\!]$ correspond to pairs $(x, (t_a)_{a\in\Gamma})$ in $\mathscr{S}_p$, where $x \in \mathscr{C}([\![\Gamma \vdash p]\!])$ is an "event structure" in the sense of [2] (up to the fact that the partial order represents the preserved program order not the actual program order, as pointed out above). Because operations on a variable a are scheduled to be operated in a total order, it follows that any non-zero read has to be preceded by a write in t_a. To define the read-from map, we need that T satisfies the property: any read in a trace is either with value zero or preceded by a write of this value. The T presented above do satisfy it. The write-serialization map comes for free as traces are linearly ordered.

5.4. Non-linear memory models

The reader might have noticed that the event structures drawn in the last examples are rather large and sequential. The linearity assumption on the memory model forces the behaviour on each variable to be sequential: two events are either in conflict or comparable. In this section we investigate the possibility of non-linear models to keep more concurrency in the generated event structure.

***Definition* 9** (Non-linear memory model)**.** — *A non-linear memory model is a collection of partial orders labelled on L_a closed under rigid inclusion, defined as follows: $q \hookrightarrow q'$ when the support of q is included in that of q' and the identity map is order-preserving and preserves down-closed sets*[2].

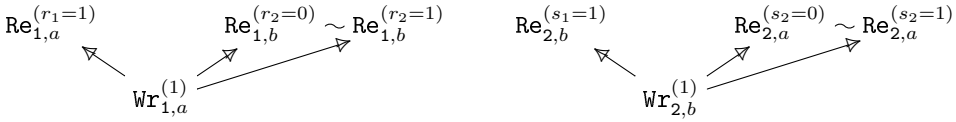
Given a non-linear memory model T, computing the closed interpretation with respect to that model is more technical. The definition of Section 5.1 relies heavily on the fact that the models are linear. If they are not, one needs to use a synchronized product of event structures [16], that will pair

[2] Also called a rigid map of event structure.

up a configuration x of $[\![t]\!]$ and a list of partial-orders ($x_a \in T(a)$) satisfying a condition saying that the orders of x and the x_a are compatible: their union does not have a cycle. For lack of space, we do not detail this construction and content ourselves with an example.

Example 12 (A simple example of non-linear memory model). — Let A be the collection of partial-orders labelled in L_a satisfying the following two axioms: *(1)* the projection to any thread-id gives a linear order (operations in the same thread cannot be concurrent) and *(2)* every chain of the form $s_1 \rightarrowtriangle \ldots \rightarrowtriangle s_n$ where s_1 is minimal is in T: every read event reads the last value written or zero if there is not any.

Looking back at the example 10, we see that apart from the two events $\mathtt{Re}_a^{(r_1=0)}$ and $\mathtt{Re}_b^{(s_1=0)}$ that cannot appear in a partial order of $A(a)$, all other configurations are actually partial-orders in $A(a) \parallel A(b)$: every configuration of the open semantics has a unique minimal synchronization with $A(a) \parallel A(b)$. Hence the synchronized product does not change anything, yielding the result:



6. Conclusion

In this article, we have built a translation from a simple imperative concurrent programming language to event structures representing the possible executions. This model takes advantage of concurrency to encode the possibility of instruction reordering in order to cut down the size of the representation. We believe this representation is a short and faithful representation of all the possible behaviours of a program that could be used to give a semantics to programming languages or processors.

This translation is heavily inspired by game semantics techniques that can be simplified in this first-order setting. This simplified approach can be easily extended to support conditionals and while loops. The original game semantics model [7, 6] however already supports a higher-order concurrent programming language at a price of increased mathematical complexity.

In the future, we would like to use this approach to handle all the existing architectures using non-linear memory models to describe compactly behaviours of modern shared memories.

Acknowledgements I would like to thank Pierre Clairambault and Olivier Laurent for helpful comments, Jade Alglave and Jean Pichon for introducing me to the world of memory models.

References

[1] S. Abramsky and G. McCusker. Full abstraction for Idealized Algol with passive expressions. volume 227, pages 3–42. 1999.

[2] J. Alglave. A formal hierarchy of weak memory models. *Formal Methods in System Design*, 41(2):178–210, 2012.

[3] M. Batty, K. Memarian, K. Nienhuis, J. Pichon-Pharabod, and P. Sewell. The problem of programming language concurrency semantics. In J. Vitek, editor, *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, volume 9032 of *Lecture Notes in Computer Science*, pages 283–307. Springer, 2015.

[4] M. Batty, K. Memarian, S. Owens, S. Sarkar, and P. Sewell. Clarifying and compiling C/C++ concurrency: from C++11 to POWER. In J. Field and M. Hicks, editors, *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, pages 509–520. ACM, 2012.

[5] S. Burckhardt, M. Musuvathi, and V. Singh. Verifying local transformations on relaxed memory models. In R. Gupta, editor, *Compiler Construction, 19th International Conference, CC 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, volume 6011 of *Lecture Notes in Computer Science*, pages 104–123. Springer, 2010.

[6] S. Castellan. La stratégie de la fourchette. In D. Baelde and J. Alglave, editors, *Vingt-sixièmes Journées Francophones des Langages Applicatifs (JFLA 2015)*, Le Val d'Ajol, France, Jan. 2015.

[7] S. Castellan, P. Clairambault, and G. Winskel. Concurrent Hyland-Ong games, 2014.

[8] S. Castellan, P. Clairambault, and G. Winskel. The parallel intensionally fully abstract games model of PCF. In *LICS 2015*. IEEE Computer Society, 2015.

[9] P. Cenciarelli, A. Knapp, and E. Sibilio. The java memory model: Operationally, denotationally, axiomatically. In R. D. Nicola, editor, *Programming Languages and Systems, 16th European Symposium on Programming, ESOP 2007, Held as Part of the Joint European Conferences on Theory and Practics of Software, ETAPS 2007, Braga, Portugal, March 24 - April 1, 2007, Proceedings*, volume 4421 of *Lecture Notes in Computer Science*, pages 331–346. Springer, 2007.

[10] D. R. Ghica and A. S. Murawski. Angelic semantics of fine-grained concurrency, 2007.

[11] L. Lamport. *IEEE Transactions on Computers*, 46, 1997.

[12] S. Mador-Haim, L. Maranget, S. Sarkar, K. Memarian, J. Alglave, S. Owens, R. Alur, M. M. K. Martin, P. Sewell, and D. Williams. An axiomatic memory model for POWER multiprocessors. In P. Madhusudan and S. A. Seshia, editors, *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, volume 7358 of *Lecture Notes in Computer Science*, pages 495–512. Springer, 2012.

[13] S. Rideau and G. Winskel. Concurrent strategies. In *Logic in Computer Science (LICS), 2011 26th Annual IEEE Symposium on*, pages 409–418. IEEE, 2011.

[14] S. Sarkar, K. Memarian, S. Owens, M. Batty, P. Sewell, L. Maranget, J. Alglave, and D. Williams. Synchronising C/C++ and POWER. In J. Vitek, H. Lin, and F. Tip, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, pages 311–322. ACM, 2012.

[15] J. Sevcík and D. Aspinall. On validity of program transformations in the Java memory model. In J. Vitek, editor, *ECOOP 2008 - Object-Oriented Programming, 22nd European Conference, Paphos, Cyprus, July 7-11, 2008, Proceedings*, volume 5142 of *Lecture Notes in Computer Science*, pages 27–51. Springer, 2008.

[16] G. Winskel. Event structure semantics for CCS and related languages. In M. Nielsen and E. M. Schmidt, editors, *Automata, Languages and Programming, 9th Colloquium, Aarhus, Denmark, July 12-16, 1982, Proceedings*, volume 140 of *Lecture Notes in Computer Science*, pages 561–576. Springer, 1982.

[17] G. Winskel. Event structures. In *Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986, Part II, Proceedings of an Advanced Course, Bad Honnef, 8.-19. September 1986*, pages 325–392, 1986.