Mechanized Theory of Event Structures: A Case of Parallel Register Machine

Vladimir Gladstein
Saint Petersburg State University,
14 line of V.O., 29B,
St. Petersburg, 199178, Russia
vovaglad00@gmail.com

Dmitrii Mikhailovskii
Saint Petersburg State University,
14 line of V.O., 29B,
St. Petersburg, 199178, Russia
mikhaylovskiy.dmitriy@gmail.com

Evgenii Moiseenko
Saint Petersburg State University
JetBrains Research
Kantemirovskaya st. 2, room 422
Saint Petersburg, 197342, Russia
e.moiseenko@2012.spbu.ru

Anton Trunov
Zilliqa Research
anton@zilliqa.com

***Abstract*—The true concurrency models, and event structures, in particular, have been introduced in the 1980s as an alternative to operational interleaving semantics of concurrency, and nowadays they are regaining popularity. Event structures represent the causal dependency and conflict between the individual atomic actions of the system directly. This property leads to a more compact and concise representation of semantics.**

In this work-in-progress report, we present a theory of event structures mechanised in the Coq proof assistant and demonstrate how it can be applied to define certified executable semantics of a simple parallel register machine with shared memory.

I. Introduction

Event structures is a mathematical formalism introduced by Winskel [1] as a semantic domain of concurrent programs. In recent years there has been renewed interest in event structures, with the applications of the theory ranging from relaxed memory models [2]–[4] to model-based mutation testing [5].

The main advantage of event structures compared to traditional interleaving semantics is that they give a more compact and concise representation of programs' behaviours. For example, consider the following code snippet of a simple parallel program.

$$x := 1 \,||\, x := 2 \,||\, x := 3$$

$$r := x$$

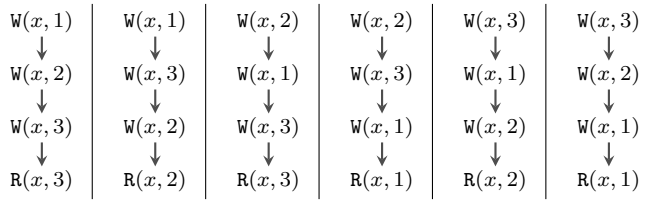
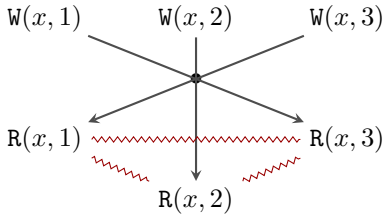


Fig. 1

Under the interleaving semantics, it has $3! = 6$ traces with each trace consisting of 4 events, as depicted in fig. 1. Events themselves represent atomic side-effects produced by instruction executions. In our case, an event is either a *write* of a value a to a shared variable x denoted as $\mathtt{W}(x, a)$, or a *read* of a value a from a shared variable x denoted as $\mathtt{R}(x, a)$.

The same information can be encoded in a single event structure containing 6 events in total (see section I). In the event structure, there are two types of edges between the events. The grey arrows $e_1 \rightarrow e_2$ represent the *causality relation*, a partial order reflecting the causal relationship between the atomic events of computation. The red edges $e_1 \leftrightsquigarrow e_2$ represent the *conflict relation* which is a symmetric and irreflexive relation encoding mutually exclusive events. Each particular trace can be extracted from the event structure as a linearisation of some *configuration*, that is a causally-closed and conflict-free subset of events, which additionally should satisfy the constraint that each read is preceded by a matching write.



The programming languages theory and formal semantics research communities are moving to increase the usage of *proof assistants* like Coq [6], Agda [7], Isabelle/HOL [8], Arend [9], and others, to complement theoretical studies with their mechanisation, as this process increases the reliability and reproducibility of scientific results. Yet, to the best of our knowledge, there is little work on mechanisation of the theory of event structures. The present report aims to close the gap.

We have chosen Coq as the proof assistant because it's a mature formal proof management tool with a rich ecosystem of libraries, plugins, documentation, and existing applications including the certification of properties of programming languages: the verified C compiler CompCert [10], the Verified Software Toolchain [11] for verification of C programs, and the Iris framework [12] for concurrent separation logic, to name a few.

Our end goal is to develop a Coq library containing a comprehensive set of common definitions, lemmas, and tactics that would allow researchers to utilise the theory of event structures for the needs of their domain.

In this work-in-progress report, we sketch the common design principles behind our library and give a concrete example of its usage by developing a formal mechanised semantics of a simple register machine with shared memory.

Our library together with the examples of its usage is available online at https://github.com/event-structures/event-struct.

II. Related Work

Event structures were introduced by Winskel to study the semantics of the calculus of communicating systems [1], [13]. Several modifications of event structures [14], [15] were later proposed to tackle similar problems.

More recently, event structures were applied in the context of relaxed memory models [2]–[4], [16]. Among this line of work, we are aware of only one paper [16] that was accompanied by a mechanisation in a proof assistant. The authors formalised the WeakestMO [4] memory model in Coq. However, this memory model uses a custom variant of event structures, that does not obey the axioms of any conventional class of event structures [13]–[15]. This fact makes it harder to reuse and adapt it to other applications of the theory.

III. Background

There exist several modifications of event structures. Currently, we have implemented only the *prime event structures* [1] in our library. We give some background on this class of event structures below.

Definition 3.1: A prime event structure (PES) is a triple $(E, \leqslant, \#)$, where

- E is a set of events
- $\leqslant$ is a causality relation on E such that
  - $(E, \leqslant)$ is a partial order;
  - for every $e \in E$ its causality prefix $\lceil e \rceil := \{e' : e' \leqslant e\}$ is finite, *i.e.,* every event is caused by a finite set of events.
- $\#$ is a conflict relation on E such that:
  - $\#$ is irreflexive and symmetric;
  - it satisfies the *hereditary* condition:

$$\forall e_1, e_2, e_3 \in E.\ e_1 \# e_2 \wedge e_2 \leqslant e_3 \Rightarrow e_1 \# e_3$$

That is, if two events are in conflict, then all their causal successors are necessarily in conflict.

A single prime event structure can encode multiple runs of a program. Each individual run can be extracted as a *configuration*. In other words, configurations are used to model a history of computation up to a certain point.

Definition 3.2: A configuration of PES $(E, \leqslant, \#)$ is a set of events $X \subseteq E$ such that

- it is causally closed

$$\forall e_1, e_2 \in E.\, e_2 \in X \wedge e_1 \leqslant e_2 \Rightarrow e_1 \in X$$

- and conflict-free

$$\forall e_1, e_2 \in X.\, \neg(e_1 \# e_2)$$

IV. Overview of Our Library

In this section, we sketch the design principles of our library.

We build our mechanisation on top of the MathComp [17] library which is an extensive and coherent repository of formalized mathematical theories, whose implementation is based on the SSReflect [18] extension of the Coq system. By using MathComp, we draw on the large corpus of already formalised algorithms and mathematical results: its core modules feature support for a range of useful data structures, e.g. numbers, sequences, finite graphs, and also interfaces: types with decidable equality, subtypes, finite types, and so on.

We also use the *small-scale reflection* methodology [18], [19], a key ingredient of SSReflect.

The small-scale reflection approach is based on the pervasive use of symbolic representations *intermixed* with logical ones within the confines of the same proof goal, as opposed to large-scale reflection which does not allow such mixing. Symbolic representations are connected to the corresponding logical ones via user-defined *reflect predicates*. The symbolic representation can be manipulated by the computational engine of the language, allowing the user to automate low-level routine proof management by using various decision and simplification procedures. Whenever the user needs to guide the proof they can switch to the logical representation and perform some proof steps manually.

To achieve better automation and e.g. get *proof irrelevance* for free, one is encouraged to use *decision* procedures whenever possible. For example, in the context of our library, we encode the binary relations of the event structures as decidable `bool`-valued relations, *i.e.,* $\leqslant, \# : E \to E \to$ `bool`, as opposed to *propositional* relations of type $E \to E \to$ `Prop`.

Encoding computable relations in Coq, especially their (computable) transitive closures, can be quite challenging since Coq is a total language and its termination checker only understands termination patterns going slightly beyond simple structural recursion. To make it easier, we employ the Equations function definition plugin [20] which provides both notations for writing programs by dependent pattern-matching and good support for well-founded recursion.

In fact, binary relations are omnipresent in our formalization. This quickly manifested in a substantial amount of proof overhead and we seeked for tools to automate our proofs. Since binary relations form a *Kleene Algebra with Tests* (KAT) [21], we have chosen to use the relation-algebra[1] [22] package which provides a number of tactics to solve goals using decision procedures for a number of theories, such as partially ordered monoids, lattices, residuated Kleene allegories and KATs.

We also favour the computational encoding of semantics. Similar to the recent related works on mechanisation of operational semantics [23]–[25], we encode the semantics

[1] https://github.com/damien-pous/relation-algebra

as monadic interpreters. This allows us to extract [26] the semantics as a functional program and run it. We believe that the possibility to run the semantics is a very useful feature, as it allows to debug the formal semantics and helps to develop better intuition about it.

To facilitate computable semantics, we define a subclass of finitely supported event structures as a finite sequence of events combined with a finitely supported function which enhances events with additional information, such as their labels, causality predecessors, *etc.* Encoding finitely supported functions is not a trivial endeavor in a proof assistant and for this task we use the FINMAP [2] library which is an extension of MATHCOMP providing finite sets and finite maps on types with a choice operator (rather than on finite types).

Finally, to encode the algebraic hierarchy of various classes of event structures we use yet another feature of MATHCOMP — *packed classes* [27], which is a design pattern providing multiple inheritance, maximal sharing of notations and theories, and automated structure inference.

V. CASE STUDY

In this section, we provide a case study demonstrating an application of our mechanised theory of event structures. We show how it can be used to encode the semantics of a parallel register machine equipped with shared memory.

A. Register Machine

For our case study, we use a simple idealised model of a register machine, which consists of a finite sequence of instructions, an instruction pointer, and an infinite set of registers. The syntax of the machine's language is shown in fig. 2.

P	$\in$	Prog	$::=$	$i_1; \ldots; i_n$	program
I	$\in$	Instr	$::=$		instruction
			$\mid$	$r := v$	assign to register
			$\mid$	$r_1 := r_2 \otimes r_3$	apply binary operation
			$\mid$	**if** r **jump** i	conditional jump
			$\mid$	**exit**	exit
			$\mid$	$r := x$	read from memory
			$\mid$	$x := v$	write to memory
r	$\in$	Reg			thread-local register
x	$\in$	Loc			shared memory location
v	$\in$	$\mathbb{Z}$			value
$\otimes$	$\in$	BinOp			binary operation
i	$\in$	$\mathbb{N}$			instruction label

Fig. 2: Syntax of the register machine

We first present the semantics of a single-threaded program. Under this semantics, memory access instructions do not operate on shared memory but rather produce a *label* denoting the side-effect of the operation (see fig. 3). This encoding allows us to decouple the semantics of the register machine from a *memory model*.

ℓ	$\in$	Lab	$::=$	
			$\mid$	$\mathtt{R}(x, v)$ read of value v from location x
			$\mid$	$\mathtt{W}(x, v)$ write of value v to location x

Fig. 3: Syntax of Labels

The semantics is given in the form of a *labelled transition system*: [3] $P \vdash s \xrightarrow{l} s'$, where P is a program, l is a label, s and s' are states of the machine. The state of the machine itself consists of an instruction pointer i and a map from registers to their values σ, as shown in fig. 4. The rules of the semantics are standard (see fig. 5).

s	$\in$	ThrdState	$::=$	$\langle i, \sigma \rangle$
i	$\in$	$\mathbb{N}$		instruction pointer
σ	$\in$	Reg $\to \mathbb{Z}$		register mapping

Fig. 4: Thread state of register machine

B. Event Structure of Register Machine

In this section we present operational semantics which constructs a prime event structure encoding a set of possible behaviours of the register machine.

The event structure is constructed incrementally in a step-by-step fashion by adding a single event on each step. In order to generate a new event on each step, we require that events behave as *identifiers*.

Definition 5.1: We say that a set E together with strict partial order $\prec$ form an *identifier set* if:

- there exists a distinguished initial identifier $e_0 \in E$;
- there exists a function fresh : $E \to E$ which generates a new fresh identifier, *s.t.*

$$\forall e \in E.\, e \prec \mathsf{fresh}(e)$$

We will encode the event structure as a tuple $\langle \mathcal{E}, \mathtt{lab}, f_{\mathtt{po}}, f_{\mathtt{rf}} \rangle$ and explain below the meaning of each component, and how they form a prime event structure together.

The first component $\mathcal{E}$ is a sequence of events $e_1 \succ \ldots \succ e_n$ in reverse order w.r.t the order in which events get added to the structure. The second component is a labelling function $\mathtt{lab} : E \to \mathsf{Lab}$, assigning a label to each event.

Next, following the theory of axiomatic weak memory models [28], we define the causality relation of the register machine's event structure as the reflexive transitive closure of the union of two relations — *program order* and *reads-from*, denoted as po and rf correspondingly.

[2] https://github.com/math-comp/finmap

[3] As we have mentioned, in our COQ development we actually use the monadic encoding of the operational semantics. The labelled transition system can be derived from this encoding.

$$\frac{P[i] = \ r := v}{P \vdash \langle i, \sigma\rangle \xrightarrow{\epsilon} \langle i+1, \sigma[r \mapsto v]\rangle}\ \texttt{Assign}$$

$$\frac{P[i] = \ x := v}{P \vdash \langle i, \sigma\rangle \xrightarrow{\texttt{W}(x,v)} \langle i+1, \sigma\rangle}\ \texttt{Store}$$

$$\frac{P[i] = \ r := x}{P \vdash \langle i, \sigma\rangle \xrightarrow{\texttt{R}(x,v)} \langle i+1, \sigma[r \mapsto v]\rangle}\ \texttt{Load}$$

$$\frac{P[i] = \ r_1 := r_2 \otimes r_3 \qquad v = \sigma(r_2) \otimes \sigma(r_3)}{P \vdash \langle i, \sigma\rangle \xrightarrow{\epsilon} \langle i+1, \sigma[r_1 \mapsto v]\rangle}\ \texttt{Binop}$$

$$\frac{P[i] = \ \textbf{exit} \qquad \mathsf{len}(P) = n}{P \vdash \langle i, \sigma\rangle \xrightarrow{\epsilon} \langle n, \sigma\rangle}\ \texttt{Exit}$$

$$\frac{P[i] = \ \textbf{if}\ r\ \textbf{jump}\ j \qquad \sigma(r) = 0}{P \vdash \langle i, \sigma\rangle \xrightarrow{\epsilon} \langle i+1, \sigma\rangle}\ \texttt{CJump}_z$$

$$\frac{P[i] = \ \textbf{if}\ r\ \textbf{jump}\ j \qquad \sigma(r) \neq 0}{P \vdash \langle i, \sigma\rangle \xrightarrow{\epsilon} \langle j, \sigma\rangle}\ \texttt{CJump}_{nz}$$

Fig. 5: Semantics of register machine

$$\leqslant \ \triangleq \ (\texttt{po} \cup \texttt{rf})^*$$

The program order relation tracks precedence of events within a single thread. The reads-from relation captures the flow of values from write events to read events, and ensures that values do not appear out of thin air [28], [29].

In order to construct po and rf incrementally we represent them via their inverse covering functions $f_{\texttt{po}}$ and $f_{\texttt{rf}}$.

Definition 5.2 (Covering): Let $\leqslant$ be a partial order. Then $\lessdot$ is covering relation *w.r.t.* $\leqslant$ whenever $x \lessdot y$ is true if and only if $x < y$ and there is no z *s.t.* $x < z$ and $z < y$. A (non-deterministic) function f from A to the set of finite subsets of A is a covering function if its corresponding relation, *i.e.,* $f^{\uparrow} \triangleq \{\langle x, y\rangle \mid y \in f(x)\}$, is a covering relation.

We use the inverse covering function because it is more convenient in our setting. Indeed, the semantics adds a new event at each step. Then it is convenient to require that, in addition, the small-step relation is provided with the po and rf predecessors of a new event.

$$\lessdot_{\texttt{po}} \ \triangleq \ f_{\texttt{po}}^{\uparrow\,-1} \qquad \lessdot_{\texttt{rf}} \ \triangleq \ f_{\texttt{rf}}^{\uparrow\,-1}$$
$$\texttt{po} \ \triangleq \ \lessdot_{\texttt{po}}^{+} \qquad \texttt{rf} \ \triangleq \ \lessdot_{\texttt{rf}}$$

We define the conflict relation in two steps. First, we define the primitive conflict relation $\sim_{\#}$ which is generated by the $f_{\texttt{po}}$ function. The two events are considered to be in primitive conflict if they are not equal and have a common po predecessor. For this definition to work properly, we also need to assume that each thread has a special initial event labelled by a distinguished *thread start* label TS.

$$e_1 \sim_{\#} e_2 \iff e_1 \neq e_2 \wedge f_{\texttt{po}}(e_1) = f_{\texttt{po}}(e2)$$

Second, we extend the primitive conflict along the causality relation:

$$e_1 \# e_2 \iff \exists e_1', e_2' \in E.\, e_1' \sim_{\#} e_2' \wedge e_1' \leqslant e_1 \wedge e_2' \leqslant e_2$$

We also need a way to reconcile the event structure with the states of the machine's threads. To do so, we use a function $\Sigma : E \to \mathsf{ThrdState}$ which maps an event to a thread state obtained as the result of the execution of the event's side-effect.

Let us consider an example. Given the program below, our semantics builds the corresponding event structure as shown in fig. 6.

$$1:\ x := 1 \ \big|\big|\ 2:\ r := x \ \big|\big|\ 3:\ x := 2$$

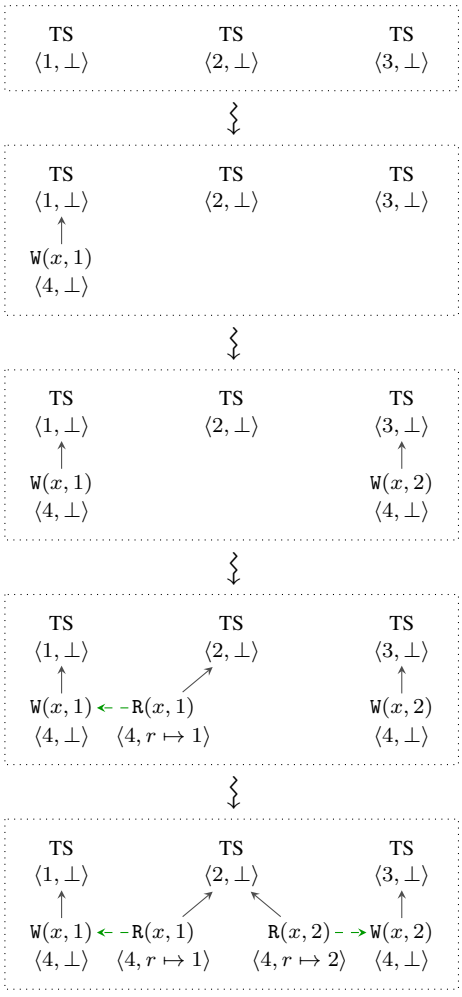


Fig. 6: Example of the event structure construction

$$\frac{e = \mathsf{fresh}(\mathsf{first}(\mathcal{E})) \qquad e_{\mathtt{po}} \in \mathcal{E} \qquad e_{\mathtt{rf}} \in \mathcal{E}}{\langle \mathcal{E}, \mathtt{lab}, f_{\mathtt{po}}, f_{\mathtt{rf}} \rangle \xhookrightarrow{\langle e, \ell, e_{\mathtt{po}}, e_{\mathtt{rf}} \rangle} \langle e :: \mathcal{E}, \mathtt{lab}[e \mapsto \ell], f_{\mathtt{po}}[e \mapsto e_{\mathtt{po}}], f_{\mathtt{rf}}[e \mapsto e_{\mathtt{rf}}] \rangle} \text{ (Add Event)}$$

$$\frac{e \in S.\mathcal{E} \qquad s = \Sigma(e) \qquad P \vdash s \xrightarrow{\epsilon} s'}{P \vdash \langle S, \Sigma \rangle \overset{\epsilon}{\Rightarrow} \langle S, \Sigma[e \mapsto s'] \rangle} \text{ (Idle)}$$

$$\frac{s = \Sigma(e_{\mathtt{po}}) \qquad P \vdash s \xrightarrow{\ell} s' \qquad S \xhookrightarrow{\langle e, \ell, e_{\mathtt{po}}, \bot \rangle} S' \qquad l = \mathtt{W}(x, v)}{P \vdash \langle S, \Sigma \rangle \xRightarrow{\langle e, \ell, e_{\mathtt{po}}, \bot \rangle} \langle S', \Sigma[e \mapsto s'] \rangle} \text{ (Store)}$$

$$\frac{s = \Sigma(e_{\mathtt{po}}) \qquad P \vdash s \xrightarrow{\ell} s' \qquad S \xhookrightarrow{\langle e, \ell, e_{\mathtt{po}}, e_{\mathtt{rf}} \rangle} S' \qquad \neg(e \# e_{\mathtt{rf}}) \qquad \begin{array}{c} l = \mathtt{R}(x, v) \\ \mathtt{lab}(e_{\mathtt{rf}}) = \mathtt{W}(x, v) \end{array}}{P \vdash \langle S, \Sigma \rangle \xRightarrow{\langle e, \ell, e_{\mathtt{po}}, e_{\mathtt{rf}} \rangle} \langle S', \Sigma[e \mapsto s'] \rangle} \text{ (Load)}$$

$$\frac{s = \Sigma(e_{\mathtt{po}}) \qquad P \vdash s \xrightarrow{\ell} s' \qquad S \xhookrightarrow{\langle e, \ell, e_{\mathtt{po}}, \bot \rangle} S' \qquad l = \mathtt{R}(x, \bot)}{P \vdash \langle S, \Sigma \rangle \xRightarrow{\langle e, \ell, e_{\mathtt{po}}, \bot \rangle} \langle S', \Sigma[e \mapsto s'] \rangle} \text{ (Load-Bottom)}$$

Fig. 7: Semantics of register machine event structure

The construction starts from an initial event structure containing, for each thread, an event labelled by TS. We depict the corresponding thread state below each label. Initially, each event is mapped to an initial thread state consisting of an instruction pointer pointing to the first instruction to be executed and an initial mapping of registers denoted as $\bot$. The first step executes the store instruction from the leftmost thread and exits the program, since the execution of this thread terminates (we omit the **exit** instructions at the end of each thread for brevity). Next, the store from the rightmost thread is executed and the corresponding write event gets added to the structure. After that, the load instruction from the middle thread is executed. Since there are two matching write events in the event structure, two conflicting reads are conjoined to the event structure. Note that the events can be added non-deterministically in any order respecting causality. We could have first executed the rightmost thread and added write $\mathtt{W}(x, 2)$ before $\mathtt{W}(x, 1)$, or we could have added the read with label $\mathtt{R}(x, 2)$ before another read $\mathtt{R}(x, 1)$.

The rules of operational semantics constructing the event structure are presented in fig. 7. The first auxiliary rule (Add Event) adds a new event, sets its label, po and rf predecessors. The (Idle) handles the case when a thread of the register machine performs an internal step without any side effect. It chooses an event e together with the thread state s corresponding to it and performs one step reduction to a new state s'. It then updates the mapping of events to thread states. The last three rules (Store), (Load), and (Load-Bottom) correspond to store and load performed by some thread. Similarly to (Idle), an event $e_{\mathtt{po}}$ is selected and one reduction is performed from the corresponding thread state s. Unlike the (Idle) case, however, a new event e is also generated. In the case of (Load), additionally, an event $e_{\mathtt{rf}}$ is selected, such that it has a write label matching the read label of the new event. The rule (Load-Bottom) corresponds to a case when load is performed "too early", before any write to the given location is available.

The following theorem asserts that the event structure built this way indeed satisfies the axioms of the prime event structure.

Theorem 1: The tuple $\langle E, \leqslant, \# \rangle$, where $\leqslant$ and $\#$ are defined as described above, forms prime event structure.

We sketch the proof below[4].

First, we need to show that $\leqslant \triangleq (\mathtt{po} \cup \mathtt{rf})^*$ is a partial order. Reflexivity and transitivity follows immediately from the definition of the reflexive-transitive closure. To show antisymmetry note that $\lessdot_{\mathtt{po}} \subseteq \prec$ and $\lessdot_{\mathtt{rf}} \subseteq \prec$ by construction. Therefore $\leqslant$ is a subset of the reflexive closure of $\prec$. Since $\prec$ is a partial order, it is antisymmetric, and thus $\leqslant$ should also be antisymmetric. The axiom of finite cause, *i.e.,* $\lceil e \rceil$ is finite for every event e, follows from the fact that at each step of the construction the set of possible predecessors of the new event can be over-approximated by the finite sequence $\mathcal{E}$.

Second, we need to show that the conflict relation $\#$ defined as described above obeys the laws of the conflict relation. Trivially, this relation is symmetric, and obeys the hereditary property. The side condition $\neg(e \# e_{\mathtt{rf}})$ of the rule (Load) ensures that the conflict relation is irreflexive.

In fig. 8 one can see the prime event structure obtained as a result of the incremental construction depicted in fig. 6.

Once the event structure is constructed, one can extract the configurations corresponding to the particular runs of the parallel register machine, and further filter them via the *consistency predicate* defining the *memory consistency model*.

[4]One can also find its mechanized proof in our Coq development.

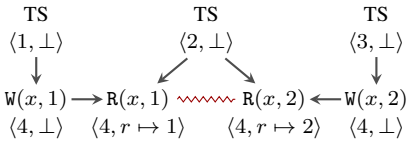


Fig. 8: Example of the prime event structure

Our construction of event structures allows to encode a wide class of so-called po∪rf acyclic relaxed memory models [28].

For example, a predicate corresponding to *sequential consistency* [30] requires that the causality order can be extended to a total order on all events of the configuration, such that for each read event the last preceding write event to the same location has the same value as the read.

VI. Future Work

There are several directions for future work.

First, we plan to apply our library to a wider range of problems. We are going to develop a mechanised semantics of some long-established languages used to model concurrency, in particular, the calculus of communicating systems (CCS) [31] and π-calculus [32]. We also plan to continue our work on expressing various relaxed models of shared memory [28], [33], [34] in terms of event structures.

Second, we want to cover other classes of event structures in our library, in particular, bundle [14], flow [15], and stable [1], [13] event structures. We plan to use them to develop mechanised *denotational* semantics of concurrent languages and relaxed shared memory models [35].

Finally, we plan to mechanise in Coq classical results that connect various classes of event structures [15], [36]. It would allow us to easily establish the connection between operational and denotational semantics of concurrent languages.

References

[1] G. Winskel, "Event structures," in *Advanced Course on Petri Nets*, pp. 325–392, Springer, 1986.

[2] A. Jeffrey and J. Riely, "On thin air reads: Towards an event structures model of relaxed memory," in *LICS 2016*, IEEE, 2016.

[3] J. Pichon-Pharabod and P. Sewell, "A concurrency semantics for relaxed atomics that permits optimisation and avoids thin-air executions," in *POPL 2016*, pp. 622–633, ACM, 2016.

[4] S. Chakraborty and V. Vafeiadis, "Grounding thin-air reads with event structures," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–28, 2019.

[5] A. Fellner, T. Tarrach, and G. Weissenbacher, "Language inclusion for finite prime event structures," in *International Conference on Verification, Model Checking, and Abstract Interpretation*, pp. 314–336, Springer, 2020.

[6] The Coq Development Team, "The Coq Proof Assistant," Jan. 2021.

[7] "Agda language reference." Available at https://agda.readthedocs.io/ [Online; accessed 7-May-2021].

[8] T. Nipkow, L. C. Paulson, and M. Wenzel, *Isabelle/HOL: a proof assistant for higher-order logic*, vol. 2283. Springer Science & Business Media, 2002.

[9] "Arend theorem prover." Available at https://arend-lang.github.io/ [Online; accessed 7-May-2021].

[10] X. Leroy, "Formal verification of a realistic compiler," *Communications of the ACM*, vol. 52, no. 7, pp. 107–115, 2009.

[11] A. W. Appel, "Verified software toolchain," in *Proceedings of the 20th European Conference on Programming Languages and Systems*, ESOP'11/ETAPS'11, (Berlin, Heidelberg), p. 1–17, Springer-Verlag, 2011.

[12] R. Jung, R. Krebbers, J.-H. Jourdan, A. Bizjak, L. Birkedal, and D. Dreyer, "Iris from the ground up: A modular foundation for higher-order concurrent separation logic," *Journal of Functional Programming*, vol. 28, p. e20, 2018.

[13] G. Winskel, "Event structure semantics for CCS and related languages," in *International Colloquium on Automata, Languages, and Programming*, pp. 561–576, Springer, 1982.

[14] R. Langerak, "Bundle event structures: a non-interleaving semantics for LOTOS," in *5th International Conference on Formal Description Techniques for Distributed Systems and Communications Protocols, FORTE 1992*, pp. 331–346, North-Holland Publishing Company, 1991.

[15] G. Boudol and I. Castellani, *Flow models of distributed computations: event structures and nets*. PhD thesis, INRIA, 1991.

[16] E. Moiseenko, A. Podkopaev, O. Lahav, O. Melkonian, and V. Vafeiadis, "Reconciling event structures with modern multiprocessors," in *34th European Conference on Object-Oriented Programming*, 2020.

[17] A. Mahboubi and E. Tassi, "Mathematical components," 2017.

[18] G. Gonthier, A. Mahboubi, and E. Tassi, "A small scale reflection extension for the Coq system," 2016.

[19] G. Gonthier and A. Mahboubi, "An introduction to small scale reflection in Coq," *Journal of formalized reasoning*, vol. 3, no. 2, pp. 95–152, 2010.

[20] M. Sozeau and C. Mangin, "Equations reloaded: High-level dependently-typed functional programming and proving in Coq," *Proc. ACM Program. Lang.*, vol. 3, July 2019.

[21] D. Kozen, "Kleene algebra with tests," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 19, no. 3, pp. 427–443, 1997.

[22] D. Pous, "Kleene algebra with tests and Coq tools for while programs," in *International Conference on Interactive Theorem Proving*, pp. 180–196, Springer, 2013.

[23] L.-y. Xia, Y. Zakowski, P. He, C.-K. Hur, G. Malecha, B. C. Pierce, and S. Zdancewic, "Interaction trees: representing recursive and impure programs in Coq," *Proceedings of the ACM on Programming Languages*, vol. 4, no. POPL, pp. 1–32, 2019.

[24] T. Letan and Y. Régis-Gianas, "Freespec: specifying, verifying, and executing impure computations in Coq," in *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pp. 32–46, 2020.

[25] R. Affeldt, D. Nowak, and T. Saikawa, "A hierarchy of monadic effects for program verification using equational reasoning," in *International Conference on Mathematics of Program Construction*, pp. 226–254, Springer, 2019.

[26] P. Letouzey, "Extraction in Coq: An overview," in *Conference on Computability in Europe*, pp. 359–369, Springer, 2008.

[27] F. Garillot, G. Gonthier, A. Mahboubi, and L. Rideau, "Packaging mathematical structures," in *International Conference on Theorem Proving in Higher Order Logics*, pp. 327–342, Springer, 2009.

[28] O. Lahav, V. Vafeiadis, J. Kang, C.-K. Hur, and D. Dreyer, "Repairing sequential consistency in C/C++11," in *PLDI 2017*, ACM, 2017.

[29] H.-J. Boehm and B. Demsky, "Outlawing ghosts: Avoiding out-of-thin-air results," in *MSPC 2014*, pp. 7:1–7:6, ACM, 2014.

[30] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Trans. Computers*, vol. 28, no. 9, pp. 690–691, 1979.

[31] R. Milner, "A calculus of communicating systems," 1980.

[32] R. Milner, *Communicating and mobile systems: the pi calculus*. Cambridge university press, 1999.

[33] O. Lahav, N. Giannarakis, and V. Vafeiadis, "Taming release-acquire consistency," *ACM SIGPLAN Notices*, vol. 51, no. 1, pp. 649–662, 2016.

[34] A. Podkopaev, O. Lahav, and V. Vafeiadis, "Bridging the gap between programming languages and hardware weak memory models," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–31, 2019.

[35] M. Dodds, M. Batty, and A. Gotsman, "Compositional verification of compiler optimisations on relaxed memory," in *European Symposium on Programming*, pp. 1027–1055, Springer, 2018.

[36] M. Nielsen, G. Plotkin, and G. Winskel, "Petri nets, event structures and domains, part I," *Theoretical Computer Science*, vol. 13, no. 1, pp. 85–108, 1981.