DOI: 10.15514/ISPRAS-2021-33(3)-11

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

[1] V.P. Gladstein, ORCID: 0000-0001-9233-3133 <vovaglad00@gmail.com>
[1] D.V. Mikhailovskii, ORCID: 0000-0002-1026-1170 <mikhaylovskiy.dmitriy@gmail.com>
[1,2] E.A. Moiseenko, ORCID: 0000-0003-2715-1143 <e.moiseenko@2012.spbu.ru>
[3] A.A. Trunov, ORCID: 0000-0003-0719-4744 <anton@zilliqa.com>

[1] Saint Petersburg State University,
14 line of V.O., 29B, St. Petersburg, 199178, Russia
[2] JetBrains Research,
Kantemirovskaya st. 2, room 422, Saint Petersburg, 197342, Russia
[3] Zilliqa Research
12 Marina View, Asia Square Tower 2, #11-01, 018961, Singapore

Abstract. The true concurrency models, and in particular event structures, 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 mechanized 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.

Keywords: semantics; event structures; interactive theorem proving; Coq

For citation: Gladstein V.P., Mikhailovskii D.V., Moiseenko E.A., Trunov A.A. Mechanized Theory of Event Structures: A Case of Parallel Register Machine. Trudy ISP RAN/Proc. ISP RAS, vol. 33, issue 3, 2021, pp. 143-154. DOI: 10.15514/ISPRAS-2021-33(3)-11

Acknowledgements. Evgenii Moiseenko was supported by RFBR according to the research project № 20-31-90088

Механизированная теория структур событий: случай параллельной регистровой машины

[1] В.П. Гладштейн, ORCID: 0000-0001-9233-3133 <vovaglad00@gmail.com>
[1] Д.В. Михайловский, ORCID: 0000-0002-1026-1170 <mikhaylovskiy.dmitriy@gmail.com>
[1,2] Е.А. Моисеенко, ORCID: 0000-0003-2715-1143 <e.moiseenko@2012.spbu.ru>
[3] А.А. Трунов, ORCID: 0000-0003-0719-4744 <anton@zilliqa.com>

[1] Санкт-Петербургский государственный университет,
Россия, 199178, 14 линия В.О., Санкт-Петербург, 29б
[2] JetBrains Research,
Россия, 197342, Санкт-Петербург, Кантемировская ул. 2, каб. 422
[3] Zilliqa Research
Сингапур, 018961

Аннотация. Модели истинной конкурентности и, в частности, структуры событий были представлены в 1980-ых как альтернатива операционным семантикам с чередованием, и на сегодняшний день эти модели вновь обретают популярность. Структуры событий позволяют явно выразить отношения причинно-следственной связи и конфликта между атомарными событиями системы, что приводит к более компактному и лаконичному представлению семантики. В данной отчете о текущей работе мы представляем теорию структур событий, механизированную в системе интерактивного доказательства теорем COQ и демонстрируем пример применения этой теории к проблеме задания сертифицированной исполняемой семантики простой параллельной регистровой машины с разделяемой памятью.

Ключевые слова: семантика; структуры событий; интерактивное доказательства теорем; Coq

Для цитирования: Гладштейн В.П., Михайловский Д.В., Моисеенко Е.А., Трунов А.А. Механизированная теория структур событий: случай параллельной регистровой машины. Труды ИСП РАН, том 33, вып. 3, 2021 г., стр. 143-154 (на английском языке). DOI: 10.15514/ISPRAS–2021–33(3)–11.

Благодарности: Евгений Моисеенко выполнял данное исследование при финансовой поддержке РФФИ в рамках научного проекта № 20-31-90088.

1. 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' behaviors. For example, consider the following code snippet of a simple parallel program.

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

$$r := x$$

W(x, 1)	W(x, 1)	W(x, 2)	W(x, 2)	W(x, 3)	W(x, 3)
↓	↓	↓	↓	↓	↓
W(x, 2)	W(x, 3)	W(x, 1)	W(x, 3)	W(x, 1)	W(x, 2)
↓	↓	↓	↓	↓	↓
W(x, 3)	W(x, 2)	W(x, 3)	W(x, 1)	W(x, 2)	W(x, 1)
↓	↓	↓	↓	↓	↓
R(x, 3)	R(x, 2)	R(x, 3)	R(x, 1)	R(x, 2)	R(x, 1)

Fig. 1. Example of program traces

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 `W(x,a)`, or a read of a value `a` from a shared variable `x` denoted as `R(x,a)`. The same information can be encoded in a single event structure containing 6 events in total (see fig.2). In the event structure, there are two types of edges between the events. The grey arrows represent the causality relation, a partial order reflecting the causal relationship between the atomic events of computation. The red edges 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 linearization 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 mechanization, as this process increases the reliability and reproducibility of scientific results. Yet, to the best of our knowledge, there is little work on mechanization 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.

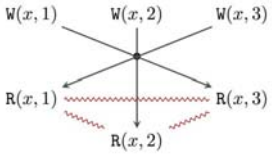

Fig. 2. Example of program event structure

Our end goal is to develop a COQ library containing a comprehensive set of common definitions, lemmas, and tactics that would allow researchers to utilize 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 mechanized 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.

2. 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 mechanization in a proof assistant. The authors formalized 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.

3. 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 $(E, \leq, \#)$ where

- E is a set of events
- $\leq$ is a causality relation on E such that
  - $(E, \leq)$ is a partial order
  - for every $e \in E$ its causality prefix $\lceil e \rceil \stackrel{\text{def}}{=} \{e' : e' \leq e\}$ 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 hereditary condition: $e_1 \# e_2$ and $e_2 \leq e_3$ implies $e_1 \# e_3$. 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 $(E, \leq, \#)$ is a set of events $X \subseteq E$ such that

- it is causally closed: $e_1 \leq e_2$ and $e_2 \in X$ then $e_1 \in X$
- and conflict-free: if $e_1, e_2 \in X$ then $e_1 \# e_2$ is false

4. Overview of Our Library

In this section, we sketch the design principles of our library. We build our mechanization 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 formalized algorithms and mathematical results: its core mod-ules 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., $\leq, \# : E \to E \to bool$, as opposed to propositional relations of type $\leq, \# : E \to E \to Prop$. 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 sought 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 [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 favor the computational encoding of semantics. Similar to the recent related works on mechanization of operational semantics [23–25], we encode the semantics 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 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.

5. Case Study

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

5.1 Register Machine

For our case study, we use a simple idealized 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. 3.

P	$\in$	Prog	::=	$i_1;\ldots;i_n$	program
I	$\in$	Instr	::=		instruction
			|	$r := v$	assign to register
			|	$r_1 := r_2 \otimes r_3$	apply binary operation
			|	`if` r `jump` i	conditional jump
			|	`exit`	exit
			|	$r := x$	read from memory
			|	$x := v$	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. 3. 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. 4). This encoding allows us to decouple the semantics of the register machine from a memory model.

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

Fig. 4. Syntax of labels

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

Fig. 5. Thread state of the register machine

The semantics is given in the form of a labelled transition system: $P \vdash s \to_\ell s'$, 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. 5. The rules of the semantics are standard (see fig. 6). 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.

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

$$\frac{P[i] = x := v}{P \vdash \langle i, \sigma \rangle \xrightarrow{\mathtt{W}(x,v)} \langle i+1, \sigma \rangle}\ \texttt{Store} \qquad \frac{P[i] = \texttt{exit} \quad \mathsf{len}(P) = n}{P \vdash \langle i, \sigma \rangle \xrightarrow{\epsilon} \langle n, \sigma \rangle}\ \texttt{Exit}$$

$$\frac{P[i] = r := x}{P \vdash \langle i, \sigma \rangle \xrightarrow{\mathtt{R}(x,v)} \langle i+1, \sigma[r \mapsto v] \rangle}\ \texttt{Load} \qquad \frac{P[i] = \texttt{if } r \texttt{ jump } j \quad \sigma(r) = 0}{P \vdash \langle i, \sigma \rangle \xrightarrow{\epsilon} \langle i+1, \sigma \rangle}\ \texttt{CJump}_z$$

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

Fig. 6. Thread semantics of the register machine

5.2 Event Structure of Register Machine

In this section we present operational semantics which constructs a prime event structure encoding a set of possible behaviors 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, such that $e \prec fresh(e)$

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

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 $lab : E \to L$, 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.

$$\leq \stackrel{\text{def}}{=} (po \cup rf)^*$$

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_{po} and f_{rf}.

Definition 5.2 (Covering): Let $\leq$ be a partial order. Then $\lessdot$ is covering relation w.r.t. $\leq$ 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., $f^{\uparrow} \stackrel{\text{def}}{=} \{ \langle x, y \rangle \mid y \in f(x) \}$, 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.

$$\lessdot_{po} \stackrel{\text{def}}{=} f_{po}^{\uparrow\,-1} \qquad po \stackrel{\text{def}}{=} \lessdot_{po}^{+}$$

$$\leqslant_{rf} \stackrel{\text{def}}{=} f_{rf}^{\uparrow\,-1} \qquad rf \stackrel{\text{def}}{=} \leqslant_{rf}^{+}$$

We define the conflict relation in two steps. First, we define the primitive conflict relation $\sim_{\#}$ which is generated by the f_{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`.

$$e_1 \sim_{\#} e_2 \stackrel{\text{def}}{=} e_1 \neq e_2 \wedge \; f_{po}(e_1) = \; f_{po}(e_2)$$

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

$$e_1 \# \; e_2 \stackrel{\text{def}}{=} \exists e'_1, e'_2 \in E. e'_1 \sim_{\#} e'_2 \wedge e'_1 \leq e_1 \wedge e'_2 \leq e_2$$

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 $\Sigma : E \to ThrdState$ 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. 7.

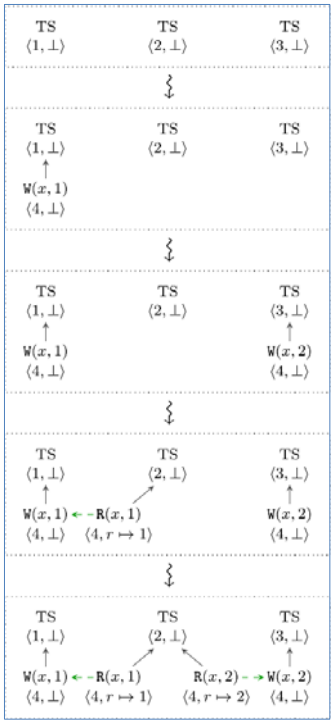

Fig. 7. Example of the event structure construction

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 $\perp$. 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 `W(x,2)` before `W(x,1)`, or we could have added the read with label `R(x,2)` before another read `R(x,1)`.

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

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

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

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

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

Fig. 8. Semantics of register machine's event structure construction

The rules of operational semantics constructing the event structure are presented in fig. 8. 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 anyside 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_{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_{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 $(E, \leq, \#)$, where $\leq$ and $\#$ are defined as described above, forms prime event structure.

We sketch the proof below (one can also find mechanized proof in our COQ development).

First, we need to show that $\leq \stackrel{\text{def}}{=} (po \cup rf)^*$ is a partial order. Reflexivity and transitivity follows immediately from the definition of the reflexive-transitive closure. To show antisymmetry note that $\leqslant_{po} \subseteq \prec$ and $\leqslant_{rf} \subseteq \prec$ by construction. Therefore $\leq$ is a subset of the reflexive closure of $\prec$. Since $\prec$ is a partial order, it is antisymmetric, and thus $\leq$ 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 $\neg\,(e \,\#\, e_{rf})$ of the rule (Load) ensures that the conflict relation is irreflexive.

In fig. 9 one can see the prime event structure obtained as a result of the incremental construction depicted in fig. 7.

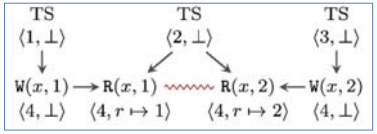

Fig. 9. Example of prime event structure

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.

Our construction of event structures allows to encode a wide class of so-called $po \cup rf$ 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.

6. 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 mechanized 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 mechanized denotational semantics of concurrent languages and relaxed shared memory models [35].

Finally, we plan to mechanize 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. Lecture Notes in Computer Science, vol. 255, 1986, pp. 325-392.
[2]. A. Jeffrey and J. Riely. On thin air reads: Towards an event structures model of relaxed memory. In Proc. of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science, 2016, pp. 759-767.
[3]. J. Pichon-Pharabod and P. Sewell. A concurrency semantics for relaxed atomics that permits optimisation and avoids thin-air executions. ACM SIGPLAN Notices, vol. 51, issue 1, 2016, pp. 622-633.
[4]. S. Chakraborty and V. Vafeiadis. Grounding thin-air reads with event structures. Proceedings of the ACM on Programming Languages, vol. 3, issue POPL, 2019, pp. 1-28.
[5]. A. Fellner, T. Tarrach, and G. Weissenbacher. Language inclusion for finite prime event structures. Lecture Notes in Computer Science, vol. 11990, 2020, pp. 314-336.
[6]. The Coq Development Team. The Coq Proof Assistant, 2021. Available at https://coq.inria.fr/, accessed 7-May-2021.
[7]. Agda language reference. Available at https://agda.readthedocs.io/, accessed 7-May-2021.
[8]. T. Nipkow, L. C. Paulson, and M. Wenzel. Isabelle/HOL: a proof assistant for higher-order logic. Lecture Notes in Computer Science, vol. 2283, 2002, 240 p.
[9]. Arend theorem prover. Available at https://arend-lang.github.io/, accessed 7-May-2021.

[10]. X. Leroy. Formal verification of a realistic compiler. Communications of the ACM, vol. 52, no. 7, 2009, pp. 107–115.
[11]. A.W. Appel. Verified software toolchain. Lecture Notes in Computer Science, vol. 6602, 2011, pp. 1-17.
[12]. R. Jung, R. Krebbers et al. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. Journal of Functional Programming, vol. 28, 2018, article e 20.
[13]. G. Winskel. Event structure semantics for CCS and related languages. Lecture Notes in Computer Science, vol. 140, 1982, pp. 561-576.
[14]. R. Langerak. Bundle event structures: a non-interleaving semantics for LOTOS. In Proc. of the 5th International Conference on Formal Description Techniques for Distributed Systems and Communications Protocols, 1992, pp. 331-346.
[15]. G. Boudol and I. Castellani. Flow models of distributed computations: event structures and nets. Research Report RR-1482, INRIA, 1991, 40 p.
[16]. E. Moiseenko, A. Podkopaev et al. Reconciling event structures with modern multiprocessors. In Proc. of the 34th European Conference on Object-Oriented Programming, 2020, 26 p.
[17]. A. Mahboubi and E. Tassi. Mathematical components, 2017. Available at http://doi.org/10.5281/zenodo.4457887, accessed 7-May-2021.
[18]. G. Gonthier, A. Mahboubi, and E. Tassi. A small scale reflection extension for the Coq system. Research Report RR-6455, Inria Saclay Ile de France, 2016, 69 p.
[19]. G. Gonthier and A. Mahboubi. An introduction to small scale reflection in Coq. Journal of formalized reasoning, vol. 3, no. 2, 2010, pp. 95-152.
[20]. M. Sozeauand, C. Mangin. Equations reloaded: High-level dependently-typed functional programming and proving in Coq. Proceedings of the ACM on Programming Languages, vol. 3, 2019, article no. 86.
[21]. D. Kozen. Kleene algebra with tests. ACM Transactions on Programming Languages and Systems (TOPLAS), vol. 19, no. 3,1997, pp. 427-443.
[22]. D. Pous. Kleene algebra with tests and Coq tools for while programs. Lecture Notes in Computer Science, vol. 7998, 2013, pp. 180-196.
[23]. L.-y. Xia, Y. Zakowski et al. Interaction trees: representing recursive and impureprograms in Coq. Proceedings of the ACM on Programming Languages, vol. 4, issue POPL, 2019, pp. 1-32.
[24]. T. Letan and Y. R´egis Gianas. Freespec: specifying, verifying, and executing impure computations in Coq. In Proc. of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, 2020, pp. 32-46.
[25]. R. Affeldt, D. Nowak, and T. Saikawa. A hierarchy of monadic effects for program verification using equational reasoning. Lecture Notes in Computer Science, vol. 11825, 2019, pp. 226-254.
[26]. P. Letouzey. Extraction in Coq: An overview. Lecture Notes in Computer Science, vol. 5028, 2008, pp. 359-369.
[27]. F. Garillot, G. Gonthier et al. Packaging mathematical structures. Lecture Notes in Computer Science, vol. 5674, 2009, pp. 327-342.
[28]. O. Lahav, V. Vafeiadis et al. Repairing sequential consistency in C/C++11. In Proc. of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2017, pp. 618-632.
[29]. H.-J. Boehm and B. Demsky. Outlawing ghosts: Avoiding out-of-thin-air results. In Proc. of the Workshop on Memory Systems Performance and Correctness, 2014, article no. 7.
[30]. L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Transactions on Computers, vol. 28, no. 9, 1979, pp. 690-691.
[31]. R. Milner. A calculus of communicating systems. Springer-Verlag, 1980, 260 p.
[32]. R. Milner. Communicating and mobile systems: the pi calculus. Cambridge university press, 1999, 176 p.
[33]. O. Lahav, N. Giannarakis, and V. Vafeiadis. Taming release-acquire consistency. ACM SIGPLAN Notices, vol. 51, no. 1, 2016, pp. 649-662
[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, 2019, pp. 1-31.
[35]. M. Dodds, M. Batty, and A. Gotsman. Compositional verification of compiler optimizations on relaxed memory. Lecture Notes in Computer Science, vol. 10801, 2018, pp. 1027-1055.
[36]. M. Nielsen, G. Plotkin, and G. Winskel. Petri nets, event structures and domains, part I. Theoretical Computer Science, vol. 13, no. 1, 1981, pp. 85-108.

Информация об авторах / Information about authors

Владимир Петрович ГЛАДШТЕЙН – студент бакалавриата Санкт-Петербургского Государственного Университета. Сфера научных интересов: методы формальной верификации программ, системы интерактивного доказательства теорем, теория зависимых типов.

Vladimir GLADSTEIN – bachelor student at Saint Petersburg State University. Research interests: formal verification of programs, interactive theorem proving, dependent types theory.

Дмитрий Владимирович МИХАЙЛОВСКИЙ – студент бакалавриата Санкт-Петербургского Государственного Университета. Сфера научных интересов: методы формальной верификации программ, системы интерактивного доказательства теорем, теория зависимых типов.

Dmitrii MIKHAILOVSKII – bachelor student at Saint Petersburg State University. Research interests: formal verification of programs, interactive theorem proving, dependent types theory.

Евгений Александрович МОИСЕЕНКО – аспирант Санкт-Петербургского Государственного Университета, исследователь в JetBrains Research. Сфера научных интересов: методы формальной верификации программ, семантика конкурентных программ.

Evgenii MOISEENKO – PhD student at Saint Petersburg State University, Researcher at JetBrains Research. Research interests: formal verification of programs, semantics of concurrency.

Антон Александрович ТРУНОВ – инженер-исследователь в Zilliqa Research. Сфера научных интересов: методы формальной верификации программ, системы интерактивного доказательства теорем.

Anton TRUNOV – research engineer at Zilliqa Research. Research interests: formal verification of programs, interactive theorem proving.