Per-Location Simulation – Appendix

Liyi Li, Elsa L. Gunter {liyili2,egunter}@illinois.edu

Department of Computer Science,
University of Illinois at Urbana-Champaign

Abstract. Simulation/bisimulation is one of the most widely used frameworks for proving program equivalence/semantic preservation. In this paper, we propose a new per-location simulation (PLS) relation that is simple and suitable for proving that a compiled program semantically preserves its original program under a CFG-based language with a real-world, C/C++ like, weak memory model. To the best of our knowledge, PLS is the first simulation framework weaker than the CompCert [5]/CompCertTSO [9] one that is used for proving compiler correctness. With a combination of PLS, the compiler proof-framework Morpheus [7], and a language semantics with a weak memory model, we are able to prove that programs are semantically preserved through a transformation. All the definitions and proofs have been implemented in Isabelle/HOL.

1 Appendix

1.1 Morpheus Syntax

Here we introduce the syntax of Morpheus. More details can be found at the work of Mansky *et al.* [7]. The basic approach of the Morpheus specification language is modeled after the TRANS language of Kalvala *et al.* [2]. Optimizations are specified as conditional compositions of rewrites on a generalized control flow graph (GCFG) containing the program's code. The language is partitioned into three largely independent components: core graph transformations (H below), conditions given in a variant of Computation Tree Logic (CTL) (φ below), and a strategy language (T below) for building complex transformations out of component transformations and conditions.

Intuitively, the rewrite portion of an optimization expresses the local transformation to be made, the condition characterizes the situations in which the optimization should be applied, and the strategy language allows us to build whole-system transformations out of collections of local ones. Morpheus is a special-purpose language for the transformation of GCFGs, and as such is parametrized by aspects of GCFGs, namely node names, node labels (program instructions), and edge labels (marking control flow). Transformation specifications may mention aspects of GCFGs concretely, but more generally, they use pattern variables that will be instantiated with control flow graph components in each specific application. We will use the term "expressions" to refer to patterns built from both concrete entities and metavariables (which will be instantiated with concrete entities when the transformation is applied). We use the term metavariable (a) to refer to the variables in the patterns and expressions in Morpheus transformations, as opposed to the concrete programming variables that will be found in instructions.

$$
\begin{aligned}
H \triangleq\ & \texttt{add\_node(}\pi\texttt{,}B\texttt{,(}l_1\texttt{,}\pi_1\texttt{),}\ldots\texttt{,(}l_n\texttt{,}\pi_n\texttt{))} \mid \texttt{add\_node(}\pi\texttt{)} \\
& \mid \texttt{relabel\_node(}\pi\texttt{,}B\texttt{)} \mid \texttt{move\_edge((}\pi\texttt{,}l\texttt{,}\pi_1\texttt{),}\pi_2\texttt{)} \\
\varphi \triangleq\ & \texttt{true} \mid p(\overrightarrow{x}) \mid \varphi \wedge \varphi \mid \neg\varphi \mid \exists\, a.\ \varphi \mid \mathcal{AX}\ \varphi \mid \mathcal{AY}\ \varphi \mid \mathcal{EX}\ \varphi \mid \mathcal{EY}\ \varphi \\
& \mid \mathcal{A}\ \varphi\, \mathcal{U}\ \varphi \mid \mathcal{E}\ \varphi\, \mathcal{U}\ \varphi \mid \mathcal{A}\ \varphi\, \mathcal{S}\ \varphi \mid \mathcal{E}\ \varphi\, \mathcal{S}\ \varphi \\
T \triangleq\ & H \mid \texttt{SATISFIED\_AT}\ \pi\ \varphi \mid \texttt{NOT}\ T \mid T \backslash T \mid \texttt{EXISTS}\ a.\ T \mid T \texttt{ + } T \mid T \texttt{ ; } T \mid T \texttt{ *}
\end{aligned}
$$

The syntax of Morpheus consists of actions (H), conditions (φ), and transformations (T). The atomic actions H begin with `add_node` and `remove_node`, which add and remove nodes that have no incoming edges. In the case of `add_node`, the addition only takes place if the node description is well-formed as a CFG node (i.e., it has the right number and kind of outgoing edges for its instruction label). The `relabel_node` action relabels an existing node with a new instruction, as long as that new instruction is compatible with the existing edge structure. The only action that operates directly on edges (rather than nodes) is `move_edge`, which moves the destination of an edge from one node in the graph to another. The conditions φ of Morpheus are based on First-Order CTL (FOCTL). Starting from a set of atomic predicates p, they include all of the usual propositional and temporal operators. The $\mathcal{S}$ ("since") and $\mathcal{Y}$ ("yesterday") operators are the past-time counterparts to the $\mathcal{U}$ ("until") and $\mathcal{X}$ ("next") operators respectively; for instance, $\mathcal{E}\ \varphi_1\ \mathcal{S}\ \varphi_2$ holds when there exists some path backwards through the graph such that φ_1 holds until a previous point at which φ_2 holds. The existential quantifier $\exists$ is used to quantify over metavariables in a formula: these metavariables may then appear in the atomic predicates of a formula, enhancing the expressive power of the conditions. At the top level, a transformation T combines conditions and rewrites using strategies. Strategies are inherently non-deterministic, as is reflected by their returning a set of possible transformed graphs. The simplest strategy is just to perform an action H. The strategy `SATISFIED_AT` $\pi\ \varphi$ acts as the identity transformation if φ holds of the GCFG at the node π, and returns the empty set if φ fails to hold on π. Thus `SATISFIED_AT` $\pi\ \varphi$ acts as filter, allowing through only those GCFGs and nodes π that satisfy φ. On the other hand, `NOT` T and $T_1 \backslash T_2$ allow us to deselect graphs by the ability to perform a transformation. The transformation `NOT` T selects those graphs that T cannot transform, i.e., those not in the domain of T, and deselects those that it can transform. The transformation $T_1 \backslash T_2$ restricts the output of T_1 to those graphs that could not be produced by T_2, i.e., those not in the image of T_2. Note the difference between these two filters: `NOT` T filters based on the complement of the domain of a transformation, while $T_1 \backslash T_2$ filters based on the complement of the range of a transformation. The strategy `EXISTS` $a.\ T$ binds a in T, limiting its scope to the free occurrences of a in the conditions and actions of T. Finally, the constructs `+` and `;` allow for choice between and sequencing of two transformations respectively, and the iteration operator `*` allows for the repeated application of a transformation any number of times.

For a simple example of a Morpheus transformation, assume we have a language of instructions that supports assign- ments and binary arithmetic expressions. In this setting, if we have a variable assigned the result of applying an arithmetic operation to two constants, we might want to replace the operation with its result. This can be done by the following transformation:

$\texttt{simple\_constant\_folding}(\pi) \triangleq \texttt{EXISTS}\ x\ a\ b\ c\ oper.$
$\texttt{SATISFIED\_AT}\ \pi\ \texttt{stmt(}\ x = oper\texttt{(}\ a\texttt{,}b\texttt{))} \wedge \texttt{is\_const(}a\texttt{)} \wedge \texttt{is\_const(}b\texttt{)} \wedge \texttt{is\_const(}c\texttt{)}$
$\wedge\texttt{eval(}oper\texttt{(}a\texttt{,}b\texttt{),}c\texttt{)}\ \texttt{;}\ \texttt{relabel\_node(}\pi\texttt{,}x = c\texttt{)}$

This is an existentially quantified sequence of a condition and an action. (Note that *oper* is a metavariable that will be bound to an arithmetic operator appearing in the program syntax, and `eval(`e`,`c`)` is a predicate asserting that the expression e evaluates to the constant c.) We may apply `simple_constant_folding` to a program with a node labeled `diff = 10 - 2` , and the transformation will match π to (the name of) this node, x to `diff` , a to `10`, b to `2`, *oper* to `( op - )`, and c to `8` (because of the clause is `(`c`,`$oper$`(`a`,`b`))` , and relabel the node to `diff = 8`.

1.2 Memory Model

Here we introduce the memory model described in Sec. **??**. The model is supposed to be in the format of an axiomatic candidate execution model [1]. By defining a set of binary relations and predicates, a candidate execution model selects a valid set of memory executions from a set of candidate executions. Here, we use a subset of a model from the ATRCM model [6], which has been proved to be sound with respect to the C/C++ memory model defined by Lahav *et al.* [4] and the IMM model [8].

We first show a set of useful conventions. $R^?$, R^+ and R^* represent the reflexive, transitive, and reflexive-transitive closures of relation R. A memory execution (memory trace) in this paper is viewed as a sequence of **memory events** ($\mathscr{Ev}$ in Fig. **??**) representing interactions between program executions and the main memory. The structure of a memory event is a triple of a thread-ID (type Tid), a unique memory **action-ID** (type $\mathscr{Aid}$ described in Fig. **??**), and a memory action (type l). Each non-τ memory event (`W` and `R`) comes from a memory instruction (store and load instructions) in a program execution produced by the program semantics. A memory execution is defined as a triple of $(T, \rho, \texttt{rf})$, where T is a downward closed natural number set excluding `0` representing time points, ρ is a partial function from natural numbers to memory events and a reads from relation (`rf`, type $T \times T$) determining the write-read pairs in the execution. To determine if a memory execution is valid, we need other entities: a set of threads (Tid, type TID `set`), a family (`sbs`, type $TID \rightarrow T \times T$) of sequenced-before relations `sb`, each of which is based on the information from a CFG, and a family (`dds`, type $TID \rightarrow T \times T$) of data dependency relations `dd`. For a CFG

Domain

Time Points: $T \subseteq \mathbb{N}$ Action-IDs: $\mathcal{Aid} \triangleq \mathcal{Name}$ Thread-IDs: $tid \in Tid \subseteq \mathcal{Tid}$

Memory Concurrency Model

Execution Map: $\rho \subseteq T \to (Tid \times \mathcal{Aid} \times \mathcal{Act})$ Data Dependency: $\texttt{dd} \subseteq T \times T$

Data Dependency Family: $\texttt{dds} \subseteq Tid \to (T \times T)$ Sequenced-Before Relation: $\texttt{sb} \subseteq T \times T$

Sequenced-Before Family: $\texttt{sbs} \subseteq Tid \to (T \times T)$

$\texttt{acq}(T, \rho, \texttt{sb}) \triangleq \{(s,t) \in T^2 | (s,t) \in \texttt{sb} \wedge \texttt{is\_read}(\rho(s)) \wedge \texttt{is\_acq}\ (\rho(s))\}$

$\texttt{rel}(T, \rho, \texttt{sb}) \triangleq \{(s,t) \in T^2 | (s,t) \in \texttt{sb} \wedge \texttt{is\_rel}\ (\rho(t)) \wedge \texttt{is\_write}(\rho(t))\}$

$\texttt{po}(T, \rho, \texttt{sb}, \texttt{dd}) \equiv \texttt{dd} \cup \texttt{acq}(T, \rho, \texttt{sb}) \cup \texttt{rel}(T, \rho, \texttt{sb})$

$\texttt{pos}(Tid, T, \rho, \texttt{sbs}, \texttt{dds}) \equiv \bigcup_{tid \in Tid} \texttt{po}(T, \rho, \texttt{sbs}(tid), \texttt{dds}(tid))$

$\texttt{co\_cw}(T, \rho, \texttt{rf}) \triangleq$
$\quad \{(s,t) | (s,t) \in \texttt{rf} \wedge (\exists (r, r') \in \texttt{rf}. r < s \wedge t < r' \wedge \texttt{same\_loc}(\rho(s), \rho(r)) \wedge \texttt{same\_thread}(\rho(r'), \rho(t)))\}$
$\quad \cup \{(s,t) | \exists r\ r'. (s, r) \in \texttt{rf} \wedge (t, r') \in \texttt{rf} \wedge \texttt{same\_thread}(\rho(s), \rho(t))$
$\qquad \wedge \texttt{same\_thread}(\rho(r), \rho(r')) \wedge ((s < t \wedge r > r') \vee (s > t \wedge r < r'))\}$

$\texttt{single\_prop}(Tid, T, \rho, \texttt{sbs}, \texttt{dds}, \texttt{rf}) \triangleq \forall (s,t) \in (\texttt{rf} \cup \texttt{pos}(Tid, T, \rho, \texttt{sbs}, \texttt{dds})). s < t$

$\texttt{at\_co}(T, \rho, \texttt{rf}) \triangleq \texttt{co\_cw}(T, \rho, \texttt{rf}) = \emptyset$

$\texttt{sat}(Tid, T, \rho, \texttt{sbs}, \texttt{dds}, \texttt{rf}) \triangleq \texttt{single\_prop}(Tid, T, \rho, \texttt{sbs}, \texttt{dds}, \texttt{rf}) \wedge \texttt{at\_co}(T, \rho, \texttt{rf})$

$\texttt{mo}_x(T, \rho) \triangleq \{(s,t) \in T^2 | a < b \wedge \texttt{same\_loc}(\rho(s), \rho(t)) \wedge \texttt{is\_write}(\rho(s)) \wedge \texttt{is\_write}(\rho(s))\}$

$\texttt{mo} \triangleq \bigcup_{x \in Loc} \texttt{mo}_x$

Fig. 1: Parts of the Definition of the Memory Model

G, sb describes the relations on memory instructions in the CFG that simulate the instruction evaluation order relations from an in-order execution machine to execute G, while dd describes the data, control, and alias dependencies that can be generated by traditional data flow, control flow, and alias analysis algorithms.

In Fig. 1, we describe how we judge if a memory execution is valid. We first define the program order relation po for a thread as the union of all single-threaded instruction dependency relations in the thread. It is a function taking in a time point set T, a ρ function and sb for the thread, and it outputs a relation set defining the instruction dependency. For simplicity, throughout this paper, we use refer to po as the relation set generated by the $\texttt{po}(T, \rho, \texttt{sb}, \texttt{dd})$. The po relation for a set of memory executions represents the exact order of evaluation based on the program meaning. The sequenced-before relation (sb) represents an approximation of the order of evaluation in a memory execution, but it does not mean that an instruction must happen before another instruction only because the programmer writes them in that order. For example, program (a) in Fig. **??** has the read from y sequenced-before the write to x in the left thread, but the program allows the write to happen before the read. In this paper, po is defined as a union of the dd, acq, and rel relations. dd is the data dependence relation for a memory execution including all dependency generated by data flow analysis, control flow analysis, alias analysis and dynamic or static methods that determine the data dependency of instructions in a CFG. acq and rel relations are defined in Fig. 1. Here, is_something is a predicate checks if a memory event has the property described as something. acq defines the binary relations between all acq reads and all memory operations (reads and writes) after it in a thread; rel defines the relations between a rel write in a thread and all operations prior to it. We also define pos as the union of all po in each thread. A valid C/C++ memory model has the property that there is a global total order (modification order) on observations of writes for each location. We define modification order for each location ($\texttt{mo}_x$) in Fig. 1, and $\texttt{mo}$ is the union of all per-location modification orders. To guarantee the valid multi-threaded behaviors, the predicate `at_co` (Fig. 1) ensures that bad behaviors defined by the `co_cw` relation do not happen. The `co_cw` relation collects bad relations in an execution that violates two execution orders: first, every two read instructions in the same thread (by the `same_thread` function) at the same location (by the `same_loc` function) read from writes in a timely order. Second, the sends (writes) and receives (reads) between two threads are in FIFO order. Finally, the predicate $\texttt{sat}(Tid, T, \rho, \texttt{sbs}, \texttt{rf})$ checks if an execution is valid by checking an execution satisfy the `at_co` predicate, as well as checking if all edges in the union of program order relations and the reads-from relation on a program are from an early time to a later time.

We have introduced the memory model briefly. The model is a C/C++ like weak memory model because we have proved that the model is sound with respect to the RC11 model [3]. More specifically, the model is SRA-consistent as the proof below. The details of the proof are in the technical report [6].

Lemma 1. For any valid execution $\texttt{sat}(T, Tid, \rho, \texttt{sbs}, \texttt{dds}, \texttt{rf})$ in ATRCM, and a location set (Loc) given as collecting all locations used in ρ, then $\texttt{pos} \cup \texttt{mo} \cup \texttt{rf}$ is acyclic.

1.3 Program Transition Semantics

Here, we define the semantics for a program based on the program syntax in Fig. **??** and instruction semantics in Sec. **??**. The program semantics is building an abstract machine executing single threaded instructions through a family of conceptual CPUs (one for each thread), and multi-threaded instructions through a conceptual memory machine. We assume that a CPU execute a basic block of instructions at a time, and any one of executing basic blocks in an program execution, named **dynamic basic block**, can be identified as a unique number $m \in \mathbb{N}$ in a program execution. We use a pair of a unique number and a basic block number of a block $(m, \pi) \in \mathbb{N} \times \mathbb{N}$ to refer to the **dynamic basic block number** (as $\overline{\pi}$) for a dynamic block whose content is the basic block whose node is π in a CFG. Then, a pair of dynamic block number and an instruction number ($\mathbb{N} \times \mathbb{N} \times \mathbb{N}$), named action-ID (as d having type $A = \mathbb{N} \times \mathbb{N} \times \mathbb{N}$), can uniquely identify an executing instruction in a thread in a program execution. In Sec. 1.2, we have described a $\texttt{po}$ relation (or a family $\texttt{pos}$), we extend the idea to a relation $\overline{\texttt{po}}$ (or a family $\overline{\texttt{pos}}$, one for each thread) describing similar program order relations as $\texttt{po}$. $\overline{\texttt{po}}$ for a thread is a relation on $A \times A$, and it describes the program order relations on different instructions instead of memory events in $\texttt{po}$. Every $\overline{\texttt{po}}$ in $\overline{\texttt{pos}}$ is generated along with program semantics transitions.

The operational transition semantics in Fig. 2 is combination of the instruction level semantics and memory concurrency model. It is represented as a labeled transition system whose states are pairs of programs (μ) and the state environment (ω), and whose labels are memory events. A state environment is

$\texttt{form\_D}(\overline{\pi},\beta)\equiv\{d|\exists e\ i.d=(\overline{\pi},i)\wedge e=\texttt{ins}(\beta,d)\}$ $\texttt{gen}(\overline{\texttt{po}},\texttt{D},W)\equiv\{d\in\texttt{D}|(\neg\exists d'\in W.(d',d)\in\overline{\texttt{po}})\}$

$$\text{(a)}\quad \frac{\begin{array}{l}(N,\pi_0,\lambda,E)=C\wedge\pi=\texttt{snd}(\overline{\pi})\wedge\texttt{ins}(\beta,d)=e\wedge\texttt{is\_CInst}(\beta,d)\wedge\lambda(\pi)=\beta\\ \wedge l=\eta(e,\phi)\wedge(\pi,l,\pi')\in E\wedge\lambda(\pi')=\beta'\wedge\overline{\pi}'=(\texttt{fst}(\overline{\pi})+1,\pi')\wedge D=\texttt{form\_D}(\overline{\pi}',\beta')\\ \wedge\rho'=\rho[\texttt{max}(T)+1\mapsto(tid,d,\tau)]\wedge\overline{\texttt{po}}'=\texttt{gen\_}\overline{\texttt{po}}(G,\overline{\pi}',\overline{\texttt{po}},\rho',D,\beta',\varphi)\wedge W=\texttt{gen}(\overline{\texttt{po}},D,D)\end{array}}{\begin{array}{l}\texttt{trans}\big(C,tid,\overline{\pi},\overline{\texttt{po}},\texttt{sb},\texttt{dd},T,\rho,\varphi,\Gamma,(\{d\},S)\big)\\ \quad=\big(\overline{\pi}',\overline{\texttt{po}}',\texttt{sb},\texttt{dd},T\cup\{\texttt{max}(T)+1\},\rho',\varphi,\Gamma,(W,\emptyset),\emptyset\big)\end{array}}$$

$$\text{(b)}\quad \frac{\begin{array}{l}(N,\pi_0,\lambda,E)=C\wedge\lambda(\texttt{snd}(\overline{\pi}))=\beta\wedge\texttt{ins}(\beta,d)=e\wedge\neg\texttt{is\_CInst}(\beta,d)\wedge\Gamma(tid)=\gamma\\ \wedge\Gamma(tid')=\gamma'\wedge\gamma'(x)=(t,v)\wedge(\varphi',\texttt{R}^{x}_{v,o})=\psi(e,\varphi,\gamma[x\mapsto(t,v)])\\ \wedge W'=\texttt{gen}(\overline{\texttt{po}},D-(S\cup\{d\}),W)\wedge\texttt{rf}=\{\texttt{fst}(\gamma'(x)),\texttt{max}(T)+1)\}\\ \wedge\texttt{sb}'=\texttt{gen\_sb}(\texttt{sb},\overline{\texttt{po}},d)\wedge\texttt{dd}'=\texttt{gen\_dd}(\texttt{dd},\overline{\texttt{po}},d)\end{array}}{\begin{array}{l}\texttt{trans}\big(C,tid,\overline{\pi},\overline{\texttt{po}},\texttt{sb},\texttt{dd},T,\rho,\varphi,\Gamma,(W\cup\{d\},S)\big)\\ \quad=\big(\overline{\pi},\overline{\texttt{po}},\texttt{sb}',\texttt{dd}',T\cup\{\texttt{max}(T)+1\},\rho[\texttt{max}(T)+1\mapsto(tid,d,\texttt{R}^{x}_{v,o})],\varphi',\Gamma,(W',S\cup\{d\}),\texttt{rf}\big)\end{array}}$$

$$\text{(c)}\quad \frac{\begin{array}{l}tid\in Tid\wedge\overline{\texttt{pos}}'=\overline{\texttt{pos}}[tid\mapsto\overline{\texttt{po}}']\wedge\texttt{sbs}'=\texttt{sbs}[tid\mapsto\texttt{sb}']\wedge\texttt{dds}'=\texttt{dds}[tid\mapsto\texttt{dd}']\\ \wedge\overline{\Pi}'=\overline{\Pi}[tid\mapsto\overline{\pi}]\wedge\Phi'=\Phi[tid\mapsto\varphi']\wedge\Theta'=\Theta[tid\mapsto\theta']\wedge\texttt{sat}(Tid,T',\rho,\texttt{sbs}',\texttt{dds}',\texttt{rf}\cup\texttt{rf'})\\ \wedge\texttt{trans}(\mu(tid),tid,\overline{\Pi}(tid),\overline{\texttt{pos}}(tid),\texttt{sbs}(tid),\texttt{dds}(tid),T,\rho,\Phi(tid),\Gamma,\Theta(tid))\\ \quad=(\overline{\pi}',\overline{\texttt{po}}',\texttt{sb}',\texttt{dd}',T',\rho',\varphi',\Gamma',\theta',\texttt{rf'})\end{array}}{\begin{array}{l}\big(\mu,Tid,\overline{\texttt{pos}},\texttt{sbs},\texttt{dds},\overline{\Pi},\Phi,\Theta,\Gamma,T,\rho,\texttt{rf}\big)\\ \quad\xrightarrow{\rho'(\texttt{max}(T'))}\big(\mu,Tid,\overline{\texttt{pos}}',\texttt{sbs}',\texttt{dds}',\overline{\Pi}',\Phi',\Theta',\Gamma',T',\rho',\texttt{rf}\cup\texttt{rf'}\big)\end{array}}$$

Fig. 2: The Program Semantics

a long tuple of a set of thread-IDs (Tid), a program order family (pos, one for each thread), a sequenced-before relation family (sbs), a data dependency family (dds), a current dynamic block number family ($\overline{\Pi}$), a registers family (Φ), a heap snapshot family (Γ) representing different views of the threads of the main memory, a program pointer family (Θ) representing the current executing instruction of each thread, a time point set (T), a ρ mapping, and a reads-from relation (rf). We show the rule (c) as the top-most rule of the transition system in Fig. 2. This rule selects a thread tid, applies the one-step transition function trans to the state environment of tid, checks the result of the one-step transition to see if the accumulated result satisfies the predicate of the memory model (sat), and then moves forward to a new step via the memory event label $\rho'(\texttt{max}(T'))$. The function max produces the maximum number in T'. We can retrieve the memory event by the max function because the trans function always creates a map entry in ρ from the maximum time point plus 1 to the current memory event.

We show two rules ((a) and (b)) of trans in Fig. 2. It needs to finish several tasks as a one step evaluation for a thread tid with a CFG C. First, if its program pointer $\Theta(tid)$ points to the end of a basic block (no instructions left for execution), it selects a new basic block according to the edge information in C (applying function η to it with registers ($\Phi(tid)$) to get the edge label), and assigns a new dynamic basic block number with a new program pointer pointing to the top of the new block. In this case, trans also adds new relations of program order, sequenced-before, and data dependency to the existing relation sets inside the new basic block. Second, if $\Theta(tid)$ indicates that there are instructions in the basic block waiting for execution, an instruction is randomly selected for execution (applying function ψ to it with registers ($\Phi(tid)$) and heap snapshot

($\Gamma(tid)$)) if the instruction satisfies the program order relation on the basic block. Third, for a step, `trans` also picks a new time point (the maximum number of the time point set T plus `1`) to add to the set T, and assigns the new time point to a new memory event. The creation of the event is to combine the thread-ID tid, a newly generated action-ID (the action-ID is calculated by combining the dynamic block number with the instruction number), and a memory action calculated from the function ψ (if the instruction is a termination, we assume that the action is τ). Fourth, `trans` also generates a new `rf` pair if the action is a read, and modifies the memory snapshot by inserting the current time point and write value if the action is a write.

Rules (a) and (b) (Fig. 2) are sample rules of the `trans` function that connects between the transition function ($\rightarrow$) and the single-instruction semantics defined in Sec. **??**. `trans` transitions from a tuple of a CFG C, a thread-ID tid, a current dynamic block number $\overline{\pi}$, a program order $\overline{\texttt{po}}$, a sequenced-before relation `sb`, a data dependency relation `dd`, a time point n, a mapping ρ, a register ϕ, a family of memory snapshots Γ, and a program counter θ; to another tuple of a possible new dynamic block number $\overline{\pi}'$, an updated program order $\overline{\texttt{po}}'$, an updated sequenced-before relation $\texttt{sb}'$, an updated data dependency relation $\texttt{dd}'$, a new time point n', a new mapping ρ', a possible new register ϕ', an updated family of memory snapshots Γ', an updated program counter θ, and a set of reads-from relations `rf` containing a possible write-read pair generated by a load instruction. In a thread tid, `trans` selects one of the possible next instructions in θ to execute. The (a) rule deals with the transition after executing a branching state at the end of a basic block, while rule (b) deals with the case of a load instruction. In these rules, `ins` is a function producing the instruction expression from a basic block and an action-ID. $\gamma|_v$ means to form a new mapping by getting rid of the time point in the value pair $T \times val$. The function $\texttt{gen\_}\overline{\texttt{po}}$ takes the existing $\overline{\texttt{po}}$ and a basic block and generates a new $\overline{\texttt{po}}'$ containing all relations in the `po` relation, all program order relations between instructions in the basic block, and the program order relations between the old-instructions in $\overline{\texttt{po}}$ and the instruction in the new basic block. $\texttt{gen\_}\overline{\texttt{po}}$ happens once when a new dynamic basic block is generated. The function `gen_sb` generates an updated `sb` relation based on the information in the updated $\overline{\texttt{po}}$ relation, while `gen_dd` generates an updated `dd` relation based on the information in the updated $\overline{\texttt{po}}$ relation.

References

1. Alglave, J., Maranget, L., Tautschnig, M.: Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory. ACM Trans. Program. Lang. Syst. **36**(2), 7:1–7:74 (Jul 2014). https://doi.org/10.1145/2627752, `http://doi.acm.org/10.1145/2627752`
2. Kalvala, S., Warburton, R., Lacey, D.: Program transformations using temporal logic side conditions. ACM Trans. Program. Lang. Syst. **31** (05 2009). https://doi.org/10.1145/1516507.1516509
3. Lahav, O., Giannarakis, N., Vafeiadis, V.: Taming release-acquire consistency. SIGPLAN Not. **51**(1), 649–662 (Jan 2016). https://doi.org/10.1145/2914770.2837643, `http://doi.acm.org/10.1145/2914770.2837643`

4. Lahav, O., Vafeiadis, V., Kang, J., Hur, C.K., Dreyer, D.: Repairing sequential consistency in c/c++11. SIGPLAN Not. **52**(6), 618–632 (Jun 2017). https://doi.org/10.1145/3140587.3062352, `http://doi.acm.org/10.1145/3140587.3062352`
5. Leroy, X.: A Formally Verified Compiler Back-end. J. Autom. Reason. **43**(4), 363–446 (Dec 2009). https://doi.org/10.1007/s10817-009-9155-4, `http://dx.doi.org/10.1007/s10817-009-9155-4`
6. Li, L., Gunter, E.: The axiomatic timed relaxed memory model (2019), `https://github.com/liyili2/timed-relaxed-memory-model`
7. Mansky, W., Gunter, E.L., Griffith, D., Adams, M.D.: Specifying and executing optimizations for generalized control flow graphs. Science of Computer Programming **130**, 2–23 (Nov 2016). https://doi.org/10.1016/j.scico.2016.06.003
8. Podkopaev, A., Lahav, O., Vafeiadis, V.: Bridging the Gap Between Programming Languages and Hardware Weak Memory Models. Proc. ACM Program. Lang. **3**(POPL), 69:1–69:31 (Jan 2019). https://doi.org/10.1145/3290382, `http://doi.acm.org/10.1145/3290382`
9. Ševčík, J., Vafeiadis, V., Zappa Nardelli, F., Jagannathan, S., Sewell, P.: CompCertTSO: A Verified Compiler for Relaxed-Memory Concurrency. J. ACM **60**(3), 22:1–22:50 (Jun 2013). https://doi.org/10.1145/2487241.2487248, `http://doi.acm.org/10.1145/2487241.2487248`