A Verification Framework for Assembly Programs Under Relaxed Memory Model Using SMT Solver

Pattaravut MALEEHUAN\(^{(a)}\), Yuki CHIBA\(^{(b)}\), Nonmembers, and Toshiaki AOKI\(^{(c)}\), Member

SUMMARY In multiprocessors, memory models are introduced to describe the executions of programs among processors. Relaxed memory models, which relax the order of executions, are used in the most of the modern processors, such as ARM and POWER. Due to a relaxed memory model, the way to execute an instruction is described by an instruction semantics, which varies among processor architectures. Dealing with instruction semantics among a variety of assembly programs is a challenge for program verification. Thus, this paper proposes a way to verify a variety of assembly programs that are executed under a relaxed memory model. The variety of assembly programs can be abstracted as the way to execute the programs by introducing an operation structure. Besides, there are existing frameworks for modeling relaxed memory models, which can realize program executions to be verified with a program property. Our work adopts an SMT solver to automatically reveal the program executions under a memory model and verify whether the executions violate the program property or not. If there is any execution from the solver, the program correctness is not preserved under the relaxed memory model. To verify programs, an experimental tool was developed to encode the given programs for a memory model into a first-order formula that violates the program correctness. The tool adopts a modeling framework to encode the programs into a formula for the SMT solver. The solver then automatically finds a valuation that satisfies the formula. In our experiments, two encoding methods were implemented based on two modeling frameworks. The valuations resulted by the solver can be considered as the bugs occurring in the original programs.

key words: relaxed memory model, model checking, SMT solver, program verification, formalization

1. Introduction

Concurrent programs are usually expected to be executed in a sequential way\cite{1}. For multiprocessors, however, programs could be executed in a non-sequential way due to the need for performance improvement. Such improvements can permit out-of-order and speculative executions to reduce the memory latency of a multiprocessor system using shared memory. Nevertheless, such non-sequential execution caused by those improvements could change the behavior of programs in the multiprocessor system\cite{2}. Accordingly, non-sequential executions might lead programs to produce anomalous results due to the program semantics has been changed. This research focuses on whether the anomalous results do not violate the program correctness in a multiprocessor system.

In a practical multiprocessor system, the behavior of non-sequential execution could be controlled implicitly to restrict the allowance non-sequential executions, such as using cache protocol. Although the behavior of each processor could be varied, a multiprocessor processor architecture usually defines memory model which describes the abstract behavior of memory accesses in a multiprocessor using shared memory, such as total store ordering (TSO) and partial store ordering (PSO). Note that multiprocessor architectures also have their memory models to facilitate the hardware, such as x86-TSO\cite{3}, SPARC-PSO\cite{4}, POWER\cite{5}, and ARM\cite{6}. Our research, thus, considers memory models in program verification for multiprocessor systems instead of considering a practical system.

For instance, Fig. 1 shows message passing programs written in ARM instruction set\cite{6}, in which program R1 writes 1 to locations x and y in the order, while program R2 reads location y until it becomes 1 and then reads location x. This means the write issued by \texttt{str \ r1, [y]} is already completed. Assertion \texttt{assert(r1 = 1)} in program R2 is provided as the program property to checks every execution must satisfy its condition. If the programs are executed in a system using TSO, the write accesses to the shared memory always complete in the program order. On the other hand, for the system using PSO, the write issued by \texttt{str \ r1, [x]} can be complete after the read issued by \texttt{ldr \ r1, [x]} even if the latter write issued by \texttt{str \ r1, [y]} is already completed. This behavior occurs because PSO enforces the completing order of only conflicting write accesses. According to the behavior permitted by PSO, the read access issued by \texttt{ldr \ r1, [x]} is allowed to return the initial value 0, which violates the assertion. Thus, a viola-

---

\(1\) The authors are with the School of Information Science, Japan Advanced Institute of Science and Technology (JAIST), Nomi-shi, 923–1292 Japan.

\(2\) The author is with DENSO Corporation, Tokyo, 103–6015 Japan.

\(3\) Member

\(4\) Nonmembers

\(5\) Member

\(6\) Nonmembers
tion of the program property occurs in PSO memory model, but not occur in TSO memory model. This shows that an unexpected execution can occur under some memory models, and can violate the program property. Note that a memory model that relaxes the way to execute programs is called relaxed memory model.

Among various relaxed memory models, each model could affect the behavior of program execution in its way. Besides, the property to be verified does not rely on only the program property, such as satisfying values, anymore. According to the previous example, without the property of a memory model, we cannot know whether program property $\text{assert}(c1 = 1)$ is satisfied by a system or not. In the case of TSO, as the non-conflicting writes must be completed according to the program order, the program property cannot be violated. On the other hand, as PSO allows the non-conflicting writes to be completed out-of-order, the program property can be violated in some executions. Note that, as the behavior of each memory model is different from each other, the verification property could also be different even if the program property is the same. Thus, the verification property also depends on the behavior of programs executed on the target memory model.

This paper proposes a framework to verify concurrent assembly programs under a relaxed memory model, in which the concerned memory models are at the hardware level, such as TSO [3] and POWER [5]. Instead of verifying a high-level language, such as C, assembly language is our concern because an assembly instruction is a granule of a statement in a high-level language to interact with a practical processor. In addition, in the implementation of an operating system, there could be a fragment of assembly program to interact with a specific processor, which could be used to prevent the reordering of assembly instructions in a compiling process. Although the reordering by a compiler could optimize the execution of a program, the different program orders could affect the behavior of the program under relaxed memory models. Thus, using an assembly language for program verification, the behavior of a program is specific to a processor architecture, which is not varied by a compiler of a high-level language. Hence, an assembly program is our target which precisely considers the behavior at the hardware level.

As there is a variety of assembly instruction sets, an abstraction of concurrent assembly programs is proposed to capture the essences of programs to be verified. Then, our verification method symbolically derives program executions regardless of a memory model. In particular, the verification method can be changed by a memory model to produce the verification property of program execution. Then, an SMT solver [7] is used to verify the verification property that is encoded as a first-order formula automatically. Besides, in our proposed abstraction, the program verification can (1) express a safety property by injecting assertion conditions in the abstraction, and (2) realize the program executions in various ways regardless of instruction sets and verification methods. This means the proposed abstraction would be applicable for further verification methods.

2. Overview of Verification Approach

First of all, given a sequence of assembly programs, each program must be translated into a corresponding operation structure, defined in Sect. 3, which abstracts the way to execute a program in a general way among various instruction semantics. The idea of an operation structure is to capture the operations which are granules of assembly instructions to appear in multiprocessors, in which the operations that affected by relaxed memory models are considered, such as read and write accesses. However, the sequence of operation structures cannot solely express program semantics under a relaxed memory model. In program verification, the behavior of programs executed under a relaxed memory model must be considered.

As verification property depends on target memory model, the behavior of programs under a memory model is the key for program verification. There are related works that provide formal ways to verify program property on specific memory models, such as [8]–[10]. However, our research adopts existing frameworks that can model the program behavior on a memory model, such as herding cats framework [11] and the framework provided by Adve and Gharachorloo [12]. Given existing operation instances occurring in the system, the modeling frameworks can decide the valid execution under a memory model using the conditions on the instances. Thus, adopting the conditions regarding a memory model can realize the behavior of programs executed under the memory model, in which the behavior is the property to be considered in the program verification on target memory model.

To illustrate the way to realize an execution using a modeling framework, let’s consider a control flow in Fig. 2(a), in which a valid execution is not realized yet. For simplicity, this control flow considers only the memory accesses to shared-memory locations and the program order $\rightarrow^p$. The memory accesses in a system consists of 2 write accesses, $W_1$ and $W_2$, and 2 read accesses, $R_1$ and $R_2$, in which each memory access has a content in its parentheses that refers to either write value for a write access or return value for a read access. For instance, $[x]=1$ in $W_1$ means location $[x]$ is written by 1, while $[y]=1$ in $R_1$ means location $[y]$ returns 1 for read access $R_1$. Note that the control flow represents a situation that the read access to shared-memory location $[y]$ returned 1 and the program R2 already exited the loop. However, the return value of read

\[
\begin{align*}
W_1 \xrightarrow{(x)=1} & \quad R_1 \xrightarrow{(y)=1} \\
W_2 \xrightarrow{(y)=1} & \quad R_2 \xrightarrow{(x)=y} \quad W_2 \xrightarrow{(y)=1} \\
R_1 \xrightarrow{(y)=1} & \quad R_2 \xrightarrow{(x)=0} \quad R_1 \xrightarrow{(y)=1} \\
\end{align*}
\]

(a) A control flow (b) An anomalous result

Fig. 2 An abstraction of message passing
In high-level languages, such as C language, every state-

ize the valid executions of an execution path under the con-

ditions of a memory model. Our research encodes each ex-

ecution path with the condition of target memory model as a

first-order formula for an SMT solver, in which the for-

mula indicate the executions that violate the program prop-

erty. Note that an encoding method is implemented based on

a modeling framework. The solver, then, finds a val-

uation that satisfies the formula, which represents a valid

execution violating the program property. If there is a val-

uation produced by the solver, there is an execution under the

given relaxed memory model that violates the program prop-

erty. Thus, to ensure the program correctness under a

memory model, there must be no valuation provided by the

solver. In the experiment, the results were tested with en-

coding methods based on Gharachorloo framework [12] and


3. Formalization

This section provides an abstraction level of assembly pro-

grams in a formal way, which is also applicable for pro-

gram verification on relaxed memory models. In particu-

lar, our formalization is motivated by herding cats, a mod-

eling framework for relaxed memory models [11]. Among

various processor architectures, there is an assumption that

an operation is a granule of assembly instructions to be

performed on hardware. For instance, a swap instruction

can perform read operation and write operation to the same

memory location. Thus, the proposed abstraction intends to

be used for the representation of essential operations that are

sufficient for program verification on relaxed memory mod-

els.

Firstly, this section introduces an operation structure in

Sect. 3.1 as the abstraction of an assembly program. In par-

cular, the operation structure intends to represent the way

to perform operations. Besides, the relations on the opera-

tions to be performed are also represented in the operation

structure, such as program order and pairs of operations to

appear atomically. Thus, Sect. 3.1 shows the formalization

of operation structures to abstract an assembly program.

Given concurrent assembly programs, Sect. 3.2 intro-

duces an execution path as the abstraction of symbolic pro-

gram execution using the corresponding operation struc-

tures. In particular, the execution path represents the way

that operations are performed regarding the corresponding

operation structures of the programs. In an execution path, the

number of operation instances is finite, and the relations on

the instances appear explicitly. Hence, a framework to

model the program execution under a relaxed memory

model can determine the possible results based on the fi-
nite sets of instances and relations. Thus, the corresponding

execution paths can be used for program verification under

relaxed memory models.

3.1 Operation Structure

In high-level languages, such as C language, every state-
ment in a program is supposed to be performed in a sequential order. However, these statements are then translated to assembly instructions with respect to a target processor, in which the order of assembly statements might be changed due to the need for optimization. Note that a statement in a high-level language could be translated as multiple assembly instructions for a target processor. Although high-level programs are quite easy for implementation, the interpretation of the programs as assembly programs could be varied, such as the program order might be changed due to compiler optimization. In addition, the program order of memory accesses is also a concern in program verification under relaxed memory model. Hence, the proposed operation structure is an intermediate representation of an assembly program which is a low-level program with an unchanged program order.

First of all, Sect. 3.1.1 provides the primary definition as the underlying elements in an operation structure, such as variables and expressions. The definition of operation is then shown in Sect. 3.1.2, which uses the primary definitions to abstract granules of assembly instructions. Besides, to indicate the control flow of the program, a label annotation is defined in Sect. 3.1.3. Also, Sect. 3.1.3 defines the way to indicate the program property to be verified.

Then, Sect. 3.1.4 introduces the way to perform operations and indicate the property regarding the semantics of practical assembly instructions. After that, an operation structure is then defined based on execution structure, defined in Sect. 3.1.4. Consequently, a sequence of operation structures is an abstraction of concurrent programs for program verification.

3.1.1 Preliminaries

In an assembly language, an instruction can access registers, memory locations, and temporal registers. Note that a temporal register is used to temporarily store a value for further processes. For instance, load instruction ld r1, [X] provides a read operation and a write operation to (1) get a value from memory location [X] and (2) write the value to register r1. To distinguish the operations, the read operation and write operation would use temporal registers to store the value before the next operations. Hence, these targets of accessing are defined as variable set \( V^\alpha \) with respect to processor \( \alpha \), such that

\[
V^\alpha = \text{Reg}^\alpha \cup \text{Loc}^\alpha \cup \text{Tmp}
\]

For simplicity, subscription \( \alpha \) is omitted in the rest of the paper and the sets are defined as follows.

\[
\text{Reg} = \{r0, r1, \ldots, r13\} \cup \{z, n, v, c\}
\]

\[
\text{Loc} = \{x\}, \{y\}, \{A\}, \{B\}
\]

\[
\text{Tmp} = \{\text{val}, v1, v2, \text{result}, \text{val}_z, \text{val}_n\}
\]

In addition, an expression and a Boolean expression are introduced as the following definitions to represent the computation during the program execution. In this formalization, the expression and Boolean expression are constructed based on variables, natural numbers and symbols \( \top \) and \( \bot \).

Definition 1 (Expression): Set \( \text{Exp} \) is the smallest set \( X \) with the properties

1. \( \mathbb{N} \subset X, \text{Tmp} \subset X \),
2. \( \varphi, \psi \in X \) implies \( (\varphi + \psi), (\varphi - \psi) \in X \),
3. \( \beta \in \text{Bexp} \) and \( \varphi, \psi \in X \) implies \( ((\beta)? \psi : \varphi) \in X \),

where \( + \) and \( - \) are connectives, \( (\, ) \), \( ; \), and ? are auxiliary symbols.

Definition 2 (Boolean Expression): Set \( \text{Bexp} \) is the smallest set \( X \) with the properties

1. \( \top, \bot \in X \),
2. \( e_1, e_2 \in \text{Exp} \) implies \( (e_1 = e_2), (e_1 < e_2), (e_1 > e_2) \in X \),
3. \( \varphi, \psi \in X \) implies \( (\varphi \land \psi), (\varphi \lor \psi) \in X \),
4. \( \varphi \in X \) implies \( \neg \varphi \in X \),

where \( =, <, >, \land, \lor \), and \( \neg \) are connectives, \( (\, ) \) and \( ; \) are auxiliary symbols.

According to expressions and Boolean expressions, the calculation is done on temporal registers and natural numbers. This would separate the calculation behavior from the read operation and write operations during the program execution.

3.1.2 Operation

Although an assembly instruction is a granule of high-level statements, in a multiprocessor system, an assembly instruction could produce multiple accesses to shared memory. Hence, let an operation be a granule of an assembly instruction that occurs in a multiprocessor system; An operation is either the followings.

- assignment \( v := e \), where \( v \in \mathcal{V} \) and \( e \in \text{Exp} \cup \mathcal{V} \). There is a restriction that one of \( v \) and \( e \) must be a temporal register, \( v \in \text{Tmp} \lor e \in \text{Tmp} \). Thus, an assignment can be classified as either:
  - read operation if expression \( e \in \mathcal{V} \setminus \text{Tmp} \),
  - write operation if variable \( v \in \mathcal{V} \setminus \text{Tmp} \), or
  - arithmetic assignment for otherwise.

According to this definition, the read and write operations do not take the calculation into account. The calculation is done by arithmetic assignment using temporal register Tmp, such as \( \text{val} := \text{val} + 1 \).

- branch \( \text{branch}(c, l) \), where \( c \in \text{Bexp} \) and \( l \) is a label annotation.

- fence \( f \in \text{Fence}_\alpha \), in which the set \( \text{Fence}_\alpha \) contains the fence instructions of processor \( \alpha \), such as \( \text{Fence}_{\text{ARM}} = \{\text{DSB}, \text{DMB}, \text{ISB}\} \).

- load-link \( \text{ll}(v, \text{loc}) \), where \( v \in \text{Tmp} \) and \( \text{loc} \in \text{Loc} \).

- store-condition \( \text{sc}(v_1, \text{loc}, v_2) \), where \( v_1, v_2 \in \text{Tmp} \) and \( \text{loc} \in \text{Loc} \).

For assignment \( v := e \), this represents the data flow of
the program explicitly. In particular, the assignments covers read operations, write operations and arithmetic assignment. Obviously, the assignments are the behavior which is our main concern because the computation is affected by relaxed memory models. In addition, fence operation also represents a significant behavior to control the computation on relaxed memory models. However, the behavior of a fence depends on processor architecture. Thus, set Fence presents a significant behavior to control the computation on relaxed memory models. In addition, fence operation also represents a flag to detect the end of the execution.

Branch operation $\text{branch}(c, l)$ represents the behavior of branch instruct that is usually used in an assembly program where $c$ is a Boolean expression and the label $l$, defined later. A label is used as the target of branch instructions if the branch condition is satisfied during program execution.

Load-link $\text{l1}(v, \text{loc})$ and store-condition $\text{sc}(v_1, \text{loc}, v_2)$ shown in the definition are the abstractions of synchronize instructions, introduced in modern processors. These operations are used as a pair in a program to access the same memory location. Note that these synchronize instructions are proposed in both programming language and hardware levels such as C++ and POWER [13], in which the semantics of these vary on a practical processor. Generally, load-link $\text{l1}(v, \text{loc})$ produces a read to location $\text{loc}$; Store-condition $\text{sc}(v_1, \text{loc}, v_2)$ writes the value of $v_2$ to location $\text{loc}$ and assigns a flag to $v_1$ if the write fails.

3.1.3 Annotation

In addition to the operations, we would like to use annotations for providing additional information, such as the program property. This information is used for the program execution information and is also used for the program verification. In the formalization, an annotation is either:

- label annotation $\text{label}(l)$ where $l$ is a label identifier,
- atomic annotation $\text{atom}(\text{opr})$ where $\text{opr}$ is either read operation or write operation,
- assumption annotation $\text{assume}($cond$)$ where $\text{cond} \in \text{BExp}$ is an assumption condition, or
- assertion annotation $\text{assert}($cond$)$ where $\text{cond} \in \text{BExp}$ is an assertion condition.

Label annotation is explicitly used for describing the target of a branch operation. In an assembly program, label and branch instruction are used to indicate the control flow of the program, in which label annotation and branch operation are used to abstract the practical branch instruction directly.

For atomic annotation, this annotation intends to indicate a pair of a read operation and a write operation to be executed atomically. In our formalization, operations are used as the granules of assembly instructions, while an assembly instruction could require issued memory accesses to be completed atomically. Thus, the atomic annotation is introduced to provide the requirement of a read operation and a write operation to be completed atomically. This represents the behavior of a read-modify-write instruction.

For program verification, assumption annotations and assertion annotations are used to indicate assumption conditions and assertion conditions, respectively. The conditions used in these annotations are Boolean expressions, which apply to determine the desired values of temporal registers. For assumption annotation, the program execution that satisfies the assumption condition is considered for program verification. As for assertion annotation, the program execution must satisfy the assertion condition. Consequently, the safety property can be expressed by these annotations.

3.1.4 Execution Structure

An execution structure captures the way to perform operations with respect to the instruction semantics of a processor architecture, in which the order to be performed can be in a partial order. This means an execution structure is constructed based on operations and annotations described in Sect. 3.1.2 and Sect. 3.1.3. Besides, an execution structure can be constructed recursively to define the partial order on the performing order of operations. Let $\gamma_1$ and $\gamma_2$ be either operations, annotations, or execution structures; an execution structure is either:

- Nil nil,
- Sequential execution $\gamma_1; \gamma_2$,
- Parallel execution $\gamma_1 \parallel \gamma_2$,
- Condition execution $\text{if}(c)[\gamma_1]$ where $c \in \text{BExp}$, or
- Instruction execution $\text{instr}[\gamma_1]$ where $\gamma_1$ does not contain instruction execution.

Nil nil is a basic execution structure that contains no operation to be performed. For sequential execution $\gamma_1; \gamma_2$, the operations in $\gamma_1$ must be performed before operations in $\gamma_2$. This represents the sequential behavior to perform operations in the order. However, this does not require the operations must be completed in the order. This means the completion of the operations is affected by relaxed memory models.

As for parallel execution $\gamma_1 \parallel \gamma_2$, operations in $\gamma_1$ and $\gamma_2$ can be performed in an interleaving way; this form is used to capture the behaviors of memory accesses in an instruction, in which the order to be performed is not strict. For instance, ARM instruction cmp r1, r2 could allow the read operations accessing registers r1 and r2 to be performed simultaneously.

Condition execution $\text{if}($cond$)[\gamma_1]$, in which $\gamma_1$ is executed if $\text{cond}$ is satisfied. This form is used to represent the behavior of a predicated instruction, in which the operations corresponding to an instruction is performed if the condition holds.

Consequently, the corresponding operations to be performed can be defined by execution structure $\gamma$, in which the order to be performed is defined with respect to the instruction semantics. The order to perform operations can be described in a partial order using defined execution structures, such as sequential execution and parallel execution. For instance, cmp r1, r2 that could read the value of r1 and r2 concurrently can be represented by parallel execution ($\forall v1$
To determine the candidate executions of $P$, the computation of branch operations and conditional executions should be evaluated to determine the set of issued events in the system. However, as the evaluation of relaxed memory models using a modeling framework requires all instances to determine the valid executions, we would like to determine the candidate execution using symbolic values. In particular, the number of events should be determined systematically without the evaluation of relaxed memory models. In our research, an execution path is introduced to represent the symbolic computation on the sequence of operation structures. In particular, the decision conditions, such as a branch condition, is evaluated by symbolic values to be always either satisfied or unsatisfied. Consequently, the number of events can be determined explicitly.

An execution path is a sequence of operation structures that satisfy the following conditions:

- All branch operations must be unique, and
- Each operation structure must be unique

where Definitions 3 and 4 are used to determine these conditions. In other words, each operation structure in an execution path always produces the finite set of events systematically, formalized by Definition 4. Besides, Definition 3 ensures a branch operation is the decided branch operations, in which the next control flow is already decided. For instance, branch operation $\text{branch}(\tau, l)$ in an operation structure is unique because the evaluation of branch condition $\tau$ is always satisfied for any execution. Consequently, the candidate executions can be considered from the corresponding execution paths of the target sequence of operation structures.

**Definition 3** (Unique branch): Branch operation $\text{branch}(c, l)$ in the operation structure is a unique branch if condition $c$ is always either satisfied or unsatisfied by every execution of the operation structure.

**Definition 4** (Unique operation structure): An operation structure is unique if there is no a condition structure and all branches are unique branches.

According to Definition 3, to make a branch operation becomes unique, we can add an assumption annotation to make the branch condition is always either satisfied or unsatisfied regarding the assumption condition. For instance, branch operation $\text{branch}(c, l)$ becomes unique if either $\text{assume}(c)$ or $\text{assume}(\neg c)$ is added before the branch operation such as $\text{assume}(c); \text{branch}(c, l)$ where the evaluation of $c$ is always satisfied. Note that this is a symbolic way to realize the computation for branch operations.

Figure 7 shows an execution path of message passing programs in Fig. 1. A unique operation structure $\psi_1^l$ is same to $\Gamma_1$ in Fig. 6, while operation structure $\psi_2^l$ is a unique operation structure of $\Gamma_1$. Assumption $\text{assume}(\neg(n = 1))$ appearing at line 25 in Fig. 7 make $\text{branch}(n = 1, \text{label}(L))$ at line 27 is always unsatisfied.

In verification, the set of execution paths is used to represent the behavior of the original operation structures. The

```c
1 instr{
2   if(z = 1){
3     val := [x];
4     r1 := val;
5   }
6   ldreq r1, [x]
7   ldstub [x], r2

Fig. 5 Examples of corresponding execution structures for instructions
```
behavior of each operation structure can be represented by
the set of unique operation structures, that should cover to
original behaviors. Thus, the set of execution paths is also
infinite. For automatically verifying, a bounded unwinding
approach is proposed in Sect. 4 to construct the finite set of
execution paths.

4. Path Exploring

First of all, we would like to describe the execution of a sequence of operation structures in a formal way using the
semantics. The semantics is defined in a general way to
target operations and perform them on the system, in which
the evaluation function, denoted by \( R \), of the return value
of a read access is provided for each memory model. Note
that the function \( R \) also relies on the issued operations in the
system, explained later.

As the events existing in the system could affect the
next computation, execution paths are proposed to restrict
the scope of execution to fix the number of events occurring
in the system for our verification purpose. Thus, the follow-
ing contents also propose a way to automatically explore the
execution paths in a systematic way.

4.1 Executions of Operation Structures

According to the execution of an assembly program, a pro-
cessor unit can fetch an assembly instruction to be per-
formed. Besides, due to the behavior of branch instructions,
program counter register is used to determine the next as-
sembly instruction to be fetched. Thus, given a sequence
of concurrent programs, each assembly instruction in the
programs is performed regarding the behavior of branch in-
structions.

Given an operations structure, its execution relies on
the fetch-cycle behavior of a processor unit. In addition, as
operations are granules of an assembly instruction, the op-
erations are stored in a temporal storage before performing
each of them regarding execution structure. For instance,
assembly instruction \( \text{str r1, [X]} \) represented by instruc-
tion execution \( \text{instr \{ v:=r1 ; [X]:=v \}} \) performs opera-
tion \( v:=r1 \) before operation \( [x]:=v \) regarding the descrip-
tion. Thus, we provide the semantics to capture such inter-
mediate state before the next instruction is performed com-
pletely. For the intermediate computation of operation struc-
tures, an execution state shown in Sect. 4.1.1 is introduced
represent the state of a processor and the issued events in
the system. Then, the execution behavior of the operation
structures shown in Sect. 4.1.2 is described based on the ex-
ecution state. Note that the execution behavior describes the
essential meanings of the definition used in the operation
structures, while the semantics is defined in Appendix A.

4.1.1 Execution State

An execution state \( \sigma \) is tuple \((\text{exec}, \text{reg}, \text{es})\) containing ex-
ecution units, register state, and event state. The ele-
ments can be indicated by the following abbreviations:
\( \sigma.\text{exec}, \sigma.\text{reg}, \) and \( \sigma.\text{es} \).

An execution unit appears in the system to perform

\[
\begin{align*}
13 & \text{label(L);} \\
14 & \text{instr}\{ \\
15 & \quad \text{val} := [y]; \\
16 & \quad \text{r1} := \text{val} \\
17 & \}; \\
18 & \text{instr}\{ \\
19 & \quad (\text{rd} := 1 || \text{rt} := \text{r2}); \\
20 & \quad \text{val}._z := (\text{rd} = \text{rt})?1:0; \\
21 & \text{val} := \text{val}._z; \\
22 & \text{val}._n := (\text{rd} = \text{rt})?0:1; \\
23 & \text{r1} := \text{val} \\
24 & \}; \\
5 & \text{instr}\{ \\
6 & \quad \text{val} := \text{r1} \\
7 & \quad \text{branch}(n = 1, \text{label(L)}) \\
8 & \}; \\
9 & \text{instr}\{ \\
10 & \quad \text{val} := \text{r1} \\
11 & \quad \text{br}(y) := \text{val} \\
12 & \}; \\
32 & \text{assert}(\text{r1} = 1)
\end{align*}
\]

- Operation structure \( \Gamma_1 \)
- Operation structure \( \Gamma_3 \)

\[
\begin{align*}
13 & \text{label(L);} \\
14 & \text{instr}\{ \\
15 & \quad \text{val} := [y]; \\
16 & \quad \text{r1} := \text{val} \\
17 & \}; \\
18 & \text{instr}\{ \\
19 & \quad (\text{rd} := 1 || \text{rt} := \text{r2}); \\
20 & \quad \text{val}._z := (\text{rd} = \text{rt})?1:0; \\
21 & \text{val} := \text{val}._z; \\
22 & \text{val}._n := (\text{rd} = \text{rt})?0:1; \\
23 & \text{r1} := \text{val} \\
24 & \}; \\
5 & \text{instr}\{ \\
6 & \quad \text{val} := \text{r1} \\
7 & \quad \text{branch}(n = 1, \text{label(L)}) \\
8 & \}; \\
9 & \text{instr}\{ \\
10 & \quad \text{val} := \text{r1} \\
11 & \quad \text{br}(y) := \text{val} \\
12 & \}; \\
32 & \text{assert}(\text{r1} = 1)
\end{align*}
\]

Fig. 7 Execution path \( \pi_1 = (\psi_1 \cdot \psi_2) \) of operation structures \( (\Gamma_1 \cdot \Gamma_2) \)
operations regarding the fetched instruction. Once an instruction is fetched, the corresponding operations must be performed regarding the semantics of the fetched instruction that is represented by an execution structure. Besides, in program execution, all operations in the execution unit must already be performed before the next instruction is fetched. Thus, the states of an execution unit must also be considered for defining the semantics of operation structures.

A **register state** represents the states of local variables used in each processor. In particular, the states of temporal registers are taken into account to consider the computation during the program execution. In addition, due to the behavior of branch instructions, program counter (pc) and next program counter (nPC) is also considered in the register to consider which are next program location to fetch an instruction.

An **event state** is a state to capture the issued events appearing in the multiprocessor system. In practice, the effect of issued operations could appear in the systems in various ways, such as caches and write buffers. In addition, the fence operations in each memory model could be used to prevent the anomalous effect in some cases. Thus, the evaluation function \( R_{es} \) of memory model \( M \) is used to evaluate the return value of read access \( r_{ev} \) based on the issued operations in \( es \) in an abstract way.

### 4.1.2 Semantics of Operation Structures

Let \( P \) be a sequence of operation structures and \( \sigma \) be an execution state, the semantics of operation structures captures the way in which an execution state is changed regarding the operation structures. Besides, due to the program property is defined using assertion annotations and assumption annotations, the computation during the program execution could violate the program property. This means the execution terminates instead of producing an execution state since the program property is violated.

To define the semantics of operation structures, we use an operational semantics to describe the behavior of the operation structures in a multiprocessor system. In the operational semantics, the behavior of fetching and the behavior of performing operations appear explicitly to manipulate the execution state. In particular, the semantics borrows the definitions of a structural operational semantics [14], which describes the individual steps of the execution. Let \( P \) be a sequence of operation structures and be an execution state. The transition has the form

\[
\langle P, \sigma \rangle \rightarrow \theta
\]

where \( \langle P, \sigma \rangle \) is called a configuration and \( \theta \) is either \( I \) (violation), of the form \( \langle P', \sigma' \rangle \), or of the form \( \sigma' \). A transition rule is used to describe the step of an execution of \( P \) from state \( \sigma \). The possible outcomes of \( \theta \) are:

- \( \theta \) is of the form \( \langle P', \sigma' \rangle \): This means the execution is not finished.
- \( \theta \) is of the form \( \sigma' \): This means the execution from state

\( \sigma \) is finished.

- \( \theta \) is \( f \): This means the execution violates the program property.

To define transitions for the semantics, the transition rules similar to the followings are used.

**[rule-1]** \( \langle \text{nil}; \gamma_2, \sigma \rangle \rightarrow \langle \gamma_2, \sigma \rangle \)

**[rule-2]** \( \langle \text{if}(c); \gamma, \sigma \rangle \rightarrow \sigma \) if \( \mathcal{B}[c] \parallel \sigma = \bot \)

For transition rule [rule-1], configurarion \( \langle \text{nil}; \gamma_2, \sigma \rangle \) can be transformed into \( \langle \gamma_2, \sigma \rangle \) if the program is of the form \( \text{nil}; \gamma_2 \). As for transformation rule [rule-2], there is condition \( \mathcal{B}[c] \parallel \rho = \bot \) to be satisfied before the configuration can be transformed regarding the rule. Note that the semantics of \( \mathcal{B}[c] \parallel \rho \) is defined in Fig. A.2. However, in this section, the desired behavior is described briefly to define the semantics of operation structures. For the detail, the transition rules are shown in Appendix A.3.

In the semantics, \( n \) operation structures are assumed to be performed on \( n \) processors independently. For each processor, an instruction execution \( \text{instr}(y) \) representing an assembly instruction is fetched into an execution unit if the unit is ready to fetch the next instruction. Especially, the instruction execution to be fetched from an operation structure is indicated by the program counter stored in the register state of the current execution state. Then, the processor waits for the execution unit is ready for fetching the next instruction again. Note that this behavior is similar to the fetch-cycle behavior of any processor, however, the instruction is not completed immediately as there are the operations as granules of assembly instructions to be performed.

After an execution unit received the execution structure from the fetched behavior, an operation of the execution structure can be performed as a single transition in the semantics. In particular, the performing order also depends on the description in the execution structure. Besides, the most of operations issue events into the system to be done later. Especially, those issued events are then added in the event state of the current execution state in the program order. Consequently, the computation of a read event can determine the return value from the event state, in which function \( R \) is provided regarding the target memory model.

In addition to the transitions for operations, the transitions for assertion annotations and assumption annotations are defined for program verification. For assumption annotations, the program execution continues if the assumption condition is satisfied. This means the execution states that do not satisfy the condition are not taken into account. As for assertion annotations, the program execution must satisfy the assertion condition. If the condition is not satisfied, this means the considering execution state violates the program property, in which the violation state is denoted by \( I \).

A **derivation sequence** of a sequence of operation structures \( P \) from execution state \( \sigma \) could be written as either:
• a finite sequence:
  \[ \theta_1 \to \theta_2 \to \ldots \to \theta_k \]
  such that \( \theta_i = (P, \sigma) \), \( \theta_i \to \theta_{i+1} \) for \( 1 \leq i \leq k \), and \( k > 1 \).
• an infinite sequence:
  \[ \theta_1 \to \theta_2 \to \ldots \]
  such that \( \theta_i = (P, \sigma) \) and \( \theta_i \to \theta_{i+1} \) for \( i \geq 1 \).

We could write \( \theta_1 \overset{i}{\to} \theta_{i+1} \) to indicate \( i \)th steps to reach \( \theta_{i+1} \) from \( \theta_1 \). In addition, \( \theta_1 \overset{\ast}{\to} \theta_{i+1} \) is written to indicate that there is a finite sequence from \( \theta_1 \) to \( \theta_{i+1} \). Thus, we could write a semantics function to define the semantics of operation structure as the following equation.

\[
S[P]R^R[\theta] = \begin{cases} 
\sigma' & \text{If } (P, \sigma) \overset{\ast}{\to} \sigma' \\
\varnothing & \text{If } (P, \sigma) \overset{\ast}{\to} \varnothing \\
\text{undef} & \text{Otherwise}
\end{cases}
\]  \hspace{1cm} (3)

where \( R \) is a function to realize the return value form existing events regarding memory model \( M \). Note that result \( \text{undef} \) means the execution does not terminate properly, which can be either invalid execution under the memory model or there is an infinite derivation sequence. In program verification, every derivation from the initial state must not reach violation state \( \varnothing \) to ensure the program correctness.

In our approach, execution paths are explored in a systematic way, in which an infinite derivation sequence cannot occur for each execution path. Especially, every derivation of an execution path provides a fixed number of events to be issued. Note that every evaluation from \( R[P]Es \) is represented in a symbolic way. Then, the realization of those values is done by using existing modeling frameworks. Thus, the way to explore execution paths is proposed in Sect. 4.2.

4.2 Bounded Loop Unwinding

An execution path is an essential component in this framework, in which the set of execution paths should cover every execution of the given operation structures. A way to explore execution paths adopts a bounded model checking approach [15] to unwind a loop under bound \( k \). First, this section introduces control flow graph definition for our path exploring approach. Then, an approach to unwinding loops under bound \( k \) is proposed.

Given an operation structure, a corresponding control flow graph (CFG) describes the way in which the structure can execute. A CFG is defined as tuple \( \langle V, E, n \rangle \) where \( V \) is a set of nodes, \( E \) is a set of directed edges, and \( n \in V \) is an initial node. A node is either: nil, instruction execution, or annotation. Note that the initial node \( n \) is the first element in the operation structure. An edge is a directed edge defined for indicating the possible ways to perform operations from the current node, in which the target can be either the consequence instruction in a program order or the target label annotation. In addition, Definition 6 and 7 show operations on nodes in a control flow graph for loop detecting in path exploring algorithm.

**Definition 6 (Path):** Given \( \langle V, E, n \rangle \), a path is a sequence of node \( u_1, \ldots, u_n \) such that \( u_1, \ldots, u_n \in V \) and \( (u_k, u_{k+1}) \in E \) where \( 1 \leq k < n \).

**Definition 7 (Dominate):** Given \( \langle V, E, n \rangle \) and \( u, v \in V \), \( u \) dominates \( v \) if \( u = v \) or every path from \( n \) to \( v \) must have \( u \) in the path.

In practice, the way to instantiate a program depends on fetch-cycle behavior in a processor, in which instructions are expected to be fetched follow the control flow graph of the program. Nevertheless, SPARC architecture [4] allows the following instruction after a branch to be fetched before the branch decides the next instruction to be fetched. Thus, our research suspects the generation of control flow graphs of some processors might be different for program verification. However, due to the most processors could fetch instructions in a sequential way, Algorithm 1 is proposed for generating a CFG from an operation structure. This algorithm considers the behavior of a branch operation contained in an execution structure, which corresponds to transformation rules \([\text{br-}T]\) and \([\text{br-}L]\) proposed in Sect. A.3. Instead of realizing the behavior of fetching execution in a detail as same as the semantics, edges in control flow graph can represent the possible directions of the node containing a branch operation. For instance, an operation structure \( \Gamma_2 \) in Fig. 6 has a sequence of 6 elements. Figure 8 is the output of CFG(\( \Gamma_2 \)), in which the nodes corresponding to elements in \( \Gamma_2 \), and the edges represent the flows to fetch the next element into the system. Note that the detail of instruction \( \text{cmp} \) \( r1, r2 \) is omitted in this control flow graph.

![Algorithm 1](image)

An approach to loop unwinding under a bound is a way to explore program behaviors for program verification using an SAT/SMT solver [15]–[17]. Although there are transformations for while-loop and reducible CFG shown in [15], [17], the transformations cannot be used directly to an unstructured program, such as assembly program, in which the control flow graph of a program could be irreducible. Thus, our approach extended the transformations for loop unwinding under a bound \( k \) in an assembly program. In addition, the behavior of predicated instruction described as a
condition execution is also extracted as two control flows by this approach. Then, a set of execution paths can be constructed by combining the possible control flows under a giving bound in our loop unwinding approach. However, this is an under-approximation approach to program verification.

As operation structures can contain loops and/or condition executions such as Fig. 6, the number of events cannot be determined systematically for program verification using existing modeling frameworks. Thus, procedure PathExploring shown in Algorithm 2 is supposed to explore execution paths, such that there is no condition to decide the number of instances to appear in a system; the program verification using existing modeling framework can use each execution path directly. For exploring execution paths, the set of control flow graphs corresponding to target operation structures are explored by procedure PathExploring shown in Algorithm 2. Each control flow graph of an operation structure is expected to be a unique operation structure. These preparations are done by PrepareCond and PrepareBranch appearing in Algorithms 4 and 5.

For the behavior of condition execution if(c)[γ], rules [if-∨] and [if-∧] in Sect. A.3 show that there are two ways to execute γ, in which the number of events could be affected by this behavior. Thus, procedure PrepareCond shown in...
Algorithm 4 replaces every condition structure in the graph by two additional control flows, in which one for taking the condition structure and another one for taking nothing. In each additional flow, a proper assumption annotation is added. Figure 9(a) shows a node containing execution condition if (z = 1) { val := [A]; r1 := val } . This node preserves the execution of instruction ldreq r1, [A] of ARM architecture. Figure 9(b) shows the result of replacing node by PrepareCond.

For the behavior of branch operation branch(c, l), rules [br-T] and [br-⊥] show two ways for executing programs, which affect the number of events to be considered in program verification. In path exploring process, every branch is expected to be uniquely determined by the semantics. This means among rules [br-T] and [br-⊥], the execution state can be determined in a deterministic way. Thus, procedure PrepareBranch shown in Algorithm 5 adds a proper assumption for each path and branch operation also exists in the path; This means the assumption enforce the execution state to satisfy the condition before considering the condition in a branch, which is the same. Thus, in each control flow, the semantics can determine the way to decide the branch in a deterministic way. Note that this procedure also collects the edges that causes a backward branch in the set loopEdges. Figure 10 illustrates the translation of the given graph. Figure 10 (a) shows the original graph that contains a branch node; Figure 10 (b) shows that each control flow can be considered as an execution in which branches are unique.

For procedure Explore in Algorithm 3, inputs are a control flow graph, a considering node, and a bound for loop unwinding. The input graph (V, E, n) is assumed to be prepared by procedures PrepareCond and PrepareBranch.

This procedure firstly checks the number of the next edges from node v. If there is no consequence edge, the procedure returns the content of a current node. If the node v is a backward branch, which can be checked by set loopEdges, the procedure checks the current bound for exploring the unique operation structures. If it exceeds the bound, the procedure does not consider the path that causes the backward branch from node v. Otherwise, the procedure explores every consequence nodes. In case of a forward branch, the procedure explores every possible operation structures by considering every consequence paths. Finally, this procedure can produce a set of unique operation structures under a bound.

Giving the set of control flow graphs G = {G1, ..., Gn} corresponding to operation structures for program verification, PathExploring(G, k) in Algorithm 2 explores a set of execution paths of G under bound k. For each graph G ∈ G, procedures PrepareCond and PrepareBranch are applied to the graph G. For instance, Fig. 11 shows the result after applying PrepareCond and PrepareBranch to the graph in Fig. 8. Then, the possible unique operation structures are explored by procedure Explore shown in Algorithm 3. The unique operation structures are combined together as the Cartesian product of the sets produced by procedure Explore. Note that a unique operation structure is considered as a way to perform operations of one program, and the result of Explore are all ways to perform operations of a CFG that can be explored under bound k. Finally, those results, sets of unique operation structures provided by Explore, are matched each other as sequences of unique operation structures for representing execution paths. Then, the set of execution paths must be verified to ensure there is no assertion violation occur in every execution path of G. However, one would propose an alternative approach to exploring execution paths that cover every execution of the programs.

5. Verification

This section firstly analyzes the way to execute an assem-
ably program for program verification under a relaxed memory model. Then, the corresponding operation structures are verified with its program property by an approach to using an SMT solver [7].

5.1 Assembly Programs

An assembly language has its instruction semantics and has its way to fetch an instruction for a processor. A variation of assembly language for an architecture is used to translate an assembly program into an operation structure. This step is a way to represent a program as the way to execute the program in a general abstraction.

**Fetch-cycle behavior.** Usual execution of an assembly program is to perform each of instruction line by line by fetching and executing. In SPARC architecture, a delayed branch instruction can fetch the followed instruction in the program to be executed before go to its destination [4]. This means an individual architecture can have the executing order of instructions on their own. In the prior work [18], this behavior is captured by a control flow graph for instructions and duplicate the following instruction of each branch to each control flow after the branch. Thus, the way to deal with fetch-cycle behavior should be analyzed before translating a program into an operation structure.

**Instruction Semantics.** After considering fetch-cycle behavior, the semantics of an instruction must be considered to construct the way to execute the program. In this paper, each assembly instruction performed by a program is assumed to have operation granules, in which every assembly instruction is not executed atomically. The instruction semantics can be represented by an instruction execution \( \text{instr}(...) \), which is introduced in Sect. 3. To translate an assembly program into an operation structure, the individual set of variables, which an instruction can access, must be realized. For instance, the registers for ARM architecture [6] can be defined as \( \text{Reg}_{\text{ARM}} = \{r0, r1, \ldots, r13\} \cup \{z, n, v, c\} \). Figure 6 shows the operation structure of message passing programs in Fig. 1, which expressed in ARM syntax.

5.2 Verification Using SMT Solver

Given a sequence of operation structure \( S = S_1, \ldots, S_n \), let \( C_S \) be the set of formulas corresponding to the behavior of \( S \) and \( P_S \) be a formula representing the program property. The behavior of \( S \) means the way that the data can be flow in a concurrent system without any constraints of a memory model. Hence, the formulas in \( C_S \) keeps the information such as program order and the writing value of a write operation. Note that, to represent the behavior as formulas, uninterpreted functions are used to abstract the valuation of a program execution. For example, an event of read operation \( \text{val} := [x] \) could be abstracted as a formula \( \text{return_val}(ev) = \text{val} \land \text{target}(ev) = [x] \). where \( ev \) is a fresh event constant, return_val and target are uninterpreted functions, in which the value of \( \text{val} \) is not realized yet.

For program property of a sequence of operation structures \( S \), let \( P_S \) be a formula that gathers assertion conditions in \( S \), such as \( \text{val} = 1 \). In program verification, every valuation from \( C_S \) under a target memory model must satisfy \( P_S \) to ensure the program correctness. By adopting a modeling framework, a valuation of \( C_S \) can be realized by introducing the set of axioms \( A_M^S \) to deduce the behavior of \( S \) under the constraints of memory model \( M \). Note that the way to represent formulas depends on the abstraction and constraints of the modeling framework. Therefore, to ensure the program correctness, every valuation of \( S \) under a memory model \( M \) must satisfy property \( P_S \), denoted by:

\[
C_S, A_M^S \models P_S \text{ for every execution}
\]

(4)

In verification, instead of checking every execution must satisfy \( P_S \), an SMT solver is adopted to check the unsatisfactory of the unsatisfied property \( \neg P_S \), which is expressed by the following formula.

\[
C_S, A_M^S \not\models \neg P_S
\]

(5)

Our work adopts the Z3 solver [19], which supports uninterpreted functions and first-order formulas with quantifiers for defining axioms, to verify the given operation structures \( S \). To use the solver, the corresponding formulas in sets \( C_S \) and \( A_M^S \) are gathered with the negation of program property \( \neg P_S \) using conjunctions \(^1\) as the following formula.

\[
\bigwedge (C_S \cup A_M^S) \land \neg P_S
\]

(6)

The SMT solver, then, finds a valuation that satisfies the formula. If there is a valuation provided by the solver, this means the property \( P_S \) does not hold on \( S \) under memory model \( M \). Otherwise, property \( P_S \) holds on \( S \) under memory model \( M \).

Using a modeling framework, a valuation can be found by giving events in a concurrent system. However, as operation structures can have loops and condition executions, an execution path represents a control flow to execute program to extract explicit events in a system. In addition, the operation structures of an execution path are then encoded with respect to the formalization of the modeling framework to be constrained.

**Execution Path.** As the set of operation structures \( S \) could have loops and condition executions, the set of execution paths are explored for verifying instead of the original behavior. Let \( \Pi_k^S \) be the set of execution paths explored by \( \text{PartExploring}(S, k) \). The verification can be done by ensuring there is no valuation that satisfy formula \( \bigwedge (C_S \cup A_M^S) \land \neg P_S \) for all \( \pi \in \Pi_k^S \) such that:

\[
\forall \pi \in \Pi_k^S, C_S, A_M^S \not\models \neg P_S
\]

(7)

Obviously, the behavior of set \( \Pi_k^S \) does not cover the behavior of \( S \), in which the correctness of original programs implies the correctness of execution paths such that:

\(^1\) A \( P \) is a conjunction of formulas in set \( P \). If set \( P \) is empty, it results true.
\begin{verbatim}
1 label(L);
2 instr{
3    val_3 := [y];
4    r1_1 := val_3
5};
6 instr{
7    (rd_0 := 1 || rt_0 := r1_1);
8    val_z_0 := (rd_0 = rt_0)
9    z_0 := val_z_0;
10    val_n_0 := (rd_0 = rt_0)
11    n_0 := val_n_0
12};
13 assume(¬(n_0 = 1));
14 instr{
15    branch(n_0 = 1, label(L
16    val_1 := r1_0
17    [x] := val_1
18    val_2 := r1_0 || r1_2 := val_4
19    [y] := val_2
20    assert(r1_2 = 1)
21};
22 return
\end{verbatim}

Fig. 12 The SSA form of execution path \(\pi_1\)

\[C_S, A^S_M \not\models \neg P_S \implies \forall \pi \in I^S_3. C_\pi, A^\pi_M \not\models \neg P_\pi \]  
(8)

However, if a valuation is found in any \(\pi \in I^S_3\) by the solver, the valuation can also occur in \(S\).

**Encoding.** The way to encode operation structures as a first-order formula varies on the formalization of a modeling framework, such as herding cats [11] uses single-event for memory operation, while Gharachorloo framework [12] uses multiple-events for a write operation. In our prior work [18], the encoding method for Gharachorloo framework [12] is proposed for encoding assembly programs. Currently, the method is extended for encoding our operation structures, which supports parallelism among operations. In addition, the encoding method for herding cats framework [11] is also proposed.

Due to a memory model focuses on the execution of access to shared-memory location, the accesses of registers and temporal registers are assumed to be executed sequentially with respect to its own processor. Hence, in our encoding methods, the input is required to be prepared in static single assignment form (SSA). Every register and temporal register is assigned only one time, which can be done by renaming each variable. For instance, Fig. 12 shows the corresponding SSA form of an execution path from Fig. 7. In encoding method, statements \(\text{val}_0 := 1\) and \(\text{val}_1 := \text{r1}_0\) are translated into \(\text{val}_0 := 1\) and \(\text{val}_1 := \text{r1}_0\), respectively, in which \(\text{val}_0\) and \(\text{val}_1\) indicate a different assignment of variable \(\text{val}\) that are assigned in the sequential order.

To encode the operation structures of an execution path as formulas, the abstraction of a modeling framework must be a concern. In general, the operations in operation structures \(S\) are instantiated as events in the system. The behavior of each event can be encoded as formulas using uninterpreted functions, such as write_val(ev) = 1 means the write value of event \(ev\) is 1, in which these formulas are contained in set \(C_S\). In addition, relations in the operation structures, such as program order (po) and intra-instruction causal order (iico), are also defined by uninterpreted functions follow the execution structures in \(S\). Note that, in each modeling framework, the way to represent corresponding events might be different, such as Gharachorloo framework that uses multiple events to represent a write operation. Moreover, assumption conditions appearing in \(S\) are constraints to restrict the behavior to be considered, such as \(¬(n_0 = 1)\) appearing in \(\psi_1\) in Fig. 12. Therefore, the formulas in \(C_S\) contains the information of the events, relations among them, and assumption conditions in \(S\).

To provide the information of relations among events, the execution structure is considered. As the abstraction of a modeling framework is different, the following explanation is described in an intuitive way to show some usages of execution structures for program verification. For program order (po), the order appearing in the path already represent the order of the operation to perform those operations. Thus, the events performed by a prior operation must be ordered before the events of consequence operations. For instance, if operation \(A\) appears before operation \(B\) in sequential executions, such as \(A \parallel B\), or nested execution structures, such as \(\text{instr}[A]; \text{instr}[B] \parallel \ldots\), the events of \(A\) must be ordered before the events of \(B\) in the program order (po). As the parallel execution, such as \(A \parallel B\), do not restrict the program order among its components, a system can freely change the execution order of events. For intra-instruction casual order (iico), this relation is used to represent the order of events performed by the same instruction. For example, if there is an instruction structure \(\text{instr}[A; B]\), the events of \(A\) must be ordered before the events of \(B\) in the intra-instruction casual order (iico).

In addition to the behavior of \(S\), assertion conditions in \(S\) represent the program property to be verified for every possible valuation of the behavior of \(S\) under memory model \(M\). Thus, set \(P_S\) is a formula that is a conjunction of assertion conditions in \(S\).

Moreover, to realize a valuation under memory model \(M\), axioms to deduce the uninterpreted functions in \(C_S\) must be defined in \(A^S_M\). These axioms are defined with respect to the constraints of memory model \(M\) in the modeling framework. To illustrate axioms in \(A^S_M\) let’s consider the sequential consistency constraints shown in Fig. 3 used by herding cats framework. Relation \(\rightarrow\) can be defined by considering the conflicting write operations in the system, such as axiom \(\forall ev_1, ev_2 \in \text{Ev}_S, \text{rf}(ev_1, ev_2)\) where \(ev_1\) is a read event instantiated by \(S\), \(W_S(ev)\) is the set of write events including an initial write that conflict to \(ev\), and \(\text{rf}\) is an uninterpreted function representing relation \(\rightarrow\). Return value return_val(ev) of read event \(ev\) can be deduced by axiom \(\forall ev \in \text{Ev}_S, \text{rf}(ev, ev) \implies \text{return}_\text{val}(ev) = \text{write}_\text{val}(ev)\). Note
that additional relations might be needed, in which a relation can be described using an uninterpreted function. In addition, in the constraints, relation requirements empty($r_l$) and acyclic($r_l$) are used to ensure there is no relation $r_l$ and there is no cycle relation $r_2$ in the system. For instance, axioms $\forall ev_1, ev_2, ev_3. r_2(ev_1, ev_2) \land r_2(ev_2, ev_3) \Rightarrow r_2(ev_1, ev_3)$ and $\forall ev. \neg (r_2(ev, ev))$ can represent requirement acyclic($r_2$).

6. Experiment

An experimental tool was developed to realize the proposed method to verify the program property of operation structures on a memory model. In particular, Z3 solver[19] is used as an SMT solver for program verification. In the experiment, the concurrent programs are considered as the case study for the proposed method. In particular, each case study is considered among the memory model specifications based on Gharachorloo and herding cats frameworks[11], [12]. Note that, in the experiment, a number for bounding the loop unwinding is also given for the program containing loops.

For Gharachorloo framework[12], the memory models to be considered are: sequential consistency (SC), total store ordering (TSO), and partial store ordering (PSO). As for herding cats, the memory models to be considered are: sequential consistency (SC), total store ordering for x86 architecture (x86-TSO), and ARM. In the tool, the encoding method relies on the framework and the memory model specification to translate an execution path as a first-order formula for program verification. Consequently, we check the program correctness of message passing programs and mutual exclusion algorithms on those memory model specifications.

6.1 Case Study

Message passing. The message passing is a famous program that could raise a bug in relaxed memory models. The programs are shown in Fig. 1 and their corresponding operation structures are also shown in Fig. 6. In verification, annotation assert($r_1 = 1$) is added at line 6 in program R2 in Fig. 1. Then, the operation structures are unwound with bound 1, two execution paths are produced for program verification; One of them is shown in Fig. 7. After that, each execution path is encoded as the first-order formula for an SMT solver. According to the program property, the completing order of write operations must not be out-of-order to ensure the assertion condition. As TSO preserves the completing order of write operations, the program property is expected to be ensured on TSO. On the other hand, as ARM and PSO memory models allow the completing order of write operations, the program property would be violated by some program execution.

Message passing with fence. To preserve the correctness of message passing under PSO model, a proper fence operation is added between writes for preventing write reordering. For instance, operation structure $\Gamma_1$ in Fig. 6 adds the following operation structure between writes to maintain the writing order for SPARC architecture.

```
1 instr{
  2 STBar
  3 }
```

STBar refers to fence instruction “stbar” in SPARC architecture that restricts the order of writes. As the result, our tool cannot find any violation under PSO model by Gharachorloo framework. In the case of ARM model, a proper fence operation is added between writes and between reads. Then, a violation is not expected to be found in the modified programs.

Mutual exclusion algorithms. In practice, there are various mutual exclusion algorithms such as Spinlock, Dekker and Peterson. In general, the program property of a mutual exclusion algorithm requires more than one programs cannot enter to their critical section simultaneously. In our experiment, assertion assert($\bot$) is added in their critical sections to ensure that they must not enter the critical section at the same time. By our path exploring approach, the assumption of an execution path that cannot occur under a memory model is then evaluated to be false. Hence, if there is a path that evaluated to be true and meet with assertion assert($\bot$), that execution path is an assertion violation path.

Spinlock for SPARC. In prior work[18], the spinlock program that is implemented as mutual exclusion algorithm in Linux Kernel for SPARC was verified under SC, TSO, and PSO models by Gharachorloo framework[12]. In this paper, the program is represented in an operation structure as an intermediate representation. For the behavior of delayed branch instructions, the following instruction of a branch instruction is added to the branch instruction to preserve the execution semantics of SPARC architecture. For example, the assembly code

```
1 brnz r5, L2
2 ldub [L], r5
```

can be represented as the following operation structure.

```
1 instr{
  2 branch($\neg(r5 = 0)$, label(0));
  3 val := [L];
  4 r5 := val
}
```

In program verification, two spinlock programs are considered for ensuring the program correctness as not to enter to their critical section simultaneously. As the algorithm uses an assembly instruction that provides read-modify-write semantics to ensure the locking mechanism is always successful.

In addition, the program was checked with ARM model, which is a weaker memory model, provided by herding cats framework[11], due to the structure is generalized enough for both frameworks. One would notice that
a program for the specific architecture is not needed to be checked under another memory model such as ARM model. However, this case study also shows that our operation structure can capture the way to execute the program which could appear on an another architecture.

**Spinlock for TOPPERS.** TOPPERS is a platform for embedded real-time systems [20], such as automotive systems. A Spinlock program in TOPPERS/FMP kernel is implemented for ARM processors, which is shown in Fig. 13. The key of this program is the usage of synchronize instructions, `ldrex` and `strex`, represented by execution structures in Fig. 14. However, the program omits interrupt instructions, `wfe`, because the correctness focuses on the return value of a read operation.

To ensure the program correctness, the corresponding operation structure is used to preserve the program execution and verify it by our approach. However, the semantics of synchronize instruction, `ldrex` and `strex`, is not formally defined explicitly in neither herding cats [11] nor Gharachorloo framework [12]. For the verification purpose, semantics of synchronize instructions is defined with respect to semantics described in [13]. As the implementation of synchronize instructions varies on processors, the semantics includes the arbitrary failure of a store condition instruction even if the condition is satisfied.

**Dekker and Peterson.** In addition to Spinlock algorithms, mutual exclusion algorithms Dekker and Peterson are also considered. Originally, these algorithms are proposed for sequential consistency model. In particular, these algorithm manipulates the global variables using read operations and write operations that can be affected by relaxed memory models. Although these algorithms seems to be correct for the most of concurrent programs, the algorithms would not be correct if the programs are executed on relaxed memory models.

### 6.2 Experiment Result

According to Table 1, the tested results of various memory models and programs, described in operation structures, are done with the encoding methods of Gharachorloo and herding cats frameworks [11, 12]. The bounded value of each test program is also shown in its parentheses. In addition to programs explained in Sect. 6.1, program ‘Known PSO bug’ is the program described in [21], in which a bug can occur under partial store ordering (PSO). The results in ‘Violation’ column show if there is an assertion violation denoted by ‘y’ and ‘n’ for otherwise. From the experiment, all bugs are raised under expected memory models. Note that result ‘-’ appearing in TOPPERS Spinlock under Gharachorloo framework means the program has not tested yet because the semantics of load-link and store-condition is not realized in the encoding method for Gharachorloo framework.

For run time, the time includes exploring path, encoding each path, and solving the encoding formula by the Z3 solver [19]. However, if there is any violating assertion, the program is then stopped immediately. All tested results were produced on a Macbook Air with a 1.4 GHz Intel Core i5 processor with memory 4 GB.

Tables 2 and 3 show the solving time and encoding time of three execution paths of Dekker programs executed under sequential consistency model; According to Table 2, the solving time of both frameworks is not too different, however, the encoding time shown in Table 3 is quite different. The number of events, written as #Events, is affected by the formalization of a framework. The encoding method for Gharachorloo [12] considers only shared-memory accesses as events, while the encoding method for herding cats [11] also consider register accesses as events. Besides, the con-

### Table 1 Evaluation results

<table>
<thead>
<tr>
<th>Program (bound)</th>
<th>SC Run time (s)</th>
<th>TSO Violation</th>
<th>PSO Violation</th>
<th>SC Run time (s)</th>
<th>TSO Violation</th>
<th>PSO Violation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message passing (1)</td>
<td>0.683</td>
<td>5.614</td>
<td>3.718</td>
<td>5.724</td>
<td>6.721</td>
<td>36.595</td>
</tr>
<tr>
<td>Message passing with fence (1)</td>
<td>0.761</td>
<td>5.724</td>
<td>3.718</td>
<td>5.724</td>
<td>6.721</td>
<td>36.595</td>
</tr>
<tr>
<td>TTOPPERS Spinlock (1)</td>
<td>-</td>
<td>461.081</td>
<td>-</td>
<td>3385.248</td>
<td>n</td>
<td>y</td>
</tr>
<tr>
<td>SPARC Spinlock (1)</td>
<td>7.441</td>
<td>435.416</td>
<td>561.755</td>
<td>523.214</td>
<td>y</td>
<td>y</td>
</tr>
<tr>
<td>Dekker (1)</td>
<td>32.499</td>
<td>267.575</td>
<td>74.058</td>
<td>523.214</td>
<td>y</td>
<td>y</td>
</tr>
<tr>
<td>Peterson (1)</td>
<td>37.621</td>
<td>267.575</td>
<td>74.058</td>
<td>523.214</td>
<td>y</td>
<td>y</td>
</tr>
<tr>
<td>Known PSO bug (0)</td>
<td>2.896</td>
<td>4.338</td>
<td>5.559</td>
<td>3385.248</td>
<td>n</td>
<td>n</td>
</tr>
</tbody>
</table>

### Table 2 Solving time of 3 execution paths of Dekker under SC

<table>
<thead>
<tr>
<th># Events</th>
<th>Solving (s)</th>
<th># Events</th>
<th>Solving (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>0.007699</td>
<td>67</td>
<td>0.007831</td>
</tr>
<tr>
<td>14</td>
<td>0.003791</td>
<td>82</td>
<td>0.002714</td>
</tr>
<tr>
<td>10</td>
<td>0.003179</td>
<td>62</td>
<td>0.008685</td>
</tr>
</tbody>
</table>

Fig. 13 Spinlock for TOPPERS/FMP kernel

```
1 l:mov r2, #1
2 ldrex r1, [lock]
3 cmp r1, #0
4 strexeq r1, r2, [lock]
5 cmp eq r1, #0
6 bne L
7 dmb
8 CS:

1 instr{
2   if (z = 1) {
3     val := r2;
4     ll(val, lock);4
5     sc(result, lock, val);
6     r1 := val;
7     r1 := result
8   }
}
ldrex r1, [lock] strex eq r1, r2, [lock]
```

Fig. 14 Execution structures for synchronize instructions
strains that decide valid executions are also proposed in different way; The ways to encode were implemented in a different style, in which the encoding method for herding cats analyzes the behaviors of programs in detail than the method for Gharachorloo framework [12]. Because of the difference of formalization, the encoding time, appearing in column ‘Encoding (s)’, in Table 3 is quite different.

7. Discussion

7.1 Assertion Language

Assertion language is expressed as a Boolean expression on registers and temporal registers. This assertion language appears in annotations of assertion and assumption, which are attached to a program and an operation structure for program verifying at the specific points. Note that, as our formalization does not have the abstraction of memory locations, the assertion language excludes shared-memory locations in which their current value can be known by a read operation to that location.

Proposed assertion language can basically express a safety property that requires the value at the specific points of programs. In addition, assert(⊥) can be used to ensure that no execution can reach that point. This assertion is adopted to realize the property that prevents execution access to an inappropriate point of the program, such as two programs are entering its same critical section at the same time.

7.2 Algorithm for Path Exploring

In sequential consistency model, the memory accesses from the former iteration always be executed before the following iterations. Due to a relaxed memory model could change the order of program executions, the execution order of two memory accesses from different iterations might be changed. In addition, an assembly program is an unstructured program, in which a loop is not defined explicitly in the program.

Our proposed algorithm is a candidate for exploring the finite set of execution paths under a bound k. The loops in an unstructured program are unwound under a bound to produce the finite executions of the program. Besides, the changing order of program executions between loop iterations can be captured by an execution path because all memory accesses appear explicitly in the execution path. Using the finite set of execution paths, the programs can be verified automatically using an SMT solver. Nevertheless, the approach is an under-approximation approach to reveal the behaviors of the programs.

8. Related Work

Program verification under a relaxed memory model becomes a concern for embedded and real-time systems, which require high-reliability. Due to a variety of memory models, there are works that propose a way for program verifying to specific memory models, such as [8]–[10]. CheckFence [8], however, proposes a relaxed memory model that is weaker than TSO and PSO models. The model is an over-approximated model to capture program semantics under TSO and PSO models. In contrast, our work adopts modeling frameworks, Gharachorloo [12] and herding cats [11], to realize program semantics under a specific memory model.

Besides, using those modeling frameworks, our verification framework is parameterized by the memory model specifications provided by those frameworks.

There are works of program verification that focus on a variety of memory models, such as [21]–[24]. Stateless Model Checking with Maximal Causality Reduction [22] is proposed for program verification under sequential consistency model and is extended for relaxed memory models, TSO and PSO, in the later work [21]. Stateless model checking for TSO, PSO, and POWER proposed by [23], [24] has adopted herding cat framework [11] to design how to realize memory model. Although our work also adopts herding cat frameworks instead of modeling behavior by ourselves, in contrast, our framework also adopts a different modeling framework which has different formalization to herding cat framework. In addition, by introducing an operation structure, our framework would be able to adopt a further framework in the future.

For verifying method, one would provide program logic for relaxed memory models, such as [9], [10], [25]. [9], [10] provide the specific program logics for x86- TSO and C++11 models respectively, while [25] introduces a program logic under a relaxed memory model that uses observation variables to indicate the possible next value of a variable with respect to the relaxed memory model. However, the restrictions on observation variables for a program execution are not derived from a memory model specifications systematically.

A recent work on program verification under a memory model is Ogre and Pythia proof method [26], which proposes a new style of the invariant proof method for a variety of weak memory models. This work proposes anarchic semantics to LISA language [27] for describing program semantics, that contains communication semantics parameterized by a weak memory model described in cat specifications [28]. The purpose of our operation structure is similar to LISA language, that generalizes program for program verification under a memory model. In contrast, the operation structure captures the parallelism on operations on the same program. For verifying with loop iterations, Ogre and Pythia method uses an indicator to a memory operation and a condition to restrict the program executions must be invented to

<table>
<thead>
<tr>
<th>Table 3</th>
<th>Encoding time of 3 execution paths of Dekker under SC</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Gharachorloo</strong></td>
<td><strong>Herding cats</strong></td>
</tr>
<tr>
<td># Events</td>
<td>Encoding (s)</td>
</tr>
<tr>
<td>12</td>
<td>2.344894</td>
</tr>
<tr>
<td>14</td>
<td>3.91485</td>
</tr>
<tr>
<td>10</td>
<td>1.298048</td>
</tr>
</tbody>
</table>

Program verification under a relaxed memory model becomes a concern for embedded and real-time systems, which require high-reliability. Due to a variety of memory models, there are works that propose a way for program verifying to specific memory models, such as [8]–[10]. CheckFence [8], however, proposes a relaxed memory model that is weaker than TSO and PSO models. The model is an over-approximated model to capture program semantics under TSO and PSO models. In contrast, our work adopts modeling frameworks, Gharachorloo [12] and herding cats [11], to realize program semantics under a specific memory model. Besides, using those modeling frameworks, our verification framework is parameterized by the memory model specifications provided by those frameworks.

Program verification under a relaxed memory model becomes a concern for embedded and real-time systems, which require high-reliability. Due to a variety of memory models, there are works that propose a way for program verifying to specific memory models, such as [8]–[10]. CheckFence [8], however, proposes a relaxed memory model that is weaker than TSO and PSO models. The model is an over-approximated model to capture program semantics under TSO and PSO models. In contrast, our work adopts modeling frameworks, Gharachorloo [12] and herding cats [11], to realize program semantics under a specific memory model. Besides, using those modeling frameworks, our verification framework is parameterized by the memory model specifications provided by those frameworks.

There are works of program verification that focus on a variety of memory models, such as [21]–[24]. Stateless Model Checking with Maximal Causality Reduction [22] is proposed for program verification under sequential consistency model and is extended for relaxed memory models, TSO and PSO, in the later work [21]. Stateless model checking for TSO, PSO, and POWER proposed by [23], [24] has adopted herding cat framework [11] to design how to realize memory model. Although our work also adopts herding cat frameworks instead of modeling behavior by ourselves, in contrast, our framework also adopts a different modeling framework which has different formalization to herding cat framework. In addition, by introducing an operation structure, our framework would be able to adopt a further framework in the future.

For verifying method, one would provide program logic for relaxed memory models, such as [9], [10], [25]. [9], [10] provide the specific program logics for x86-TSO and C++11 models respectively, while [25] introduces a program logic under a relaxed memory model that uses observation variables to indicate the possible next value of a variable with respect to the relaxed memory model. However, the restrictions on observation variables for a program execution are not derived from a memory model specifications systematically.

A recent work on program verification under a memory model is Ogre and Pythia proof method [26], which proposes a new style of the invariant proof method for a variety of weak memory models. This work proposes anarchic semantics to LISA language [27] for describing program semantics, that contains communication semantics parameterized by a weak memory model described in cat specifications [28]. The purpose of our operation structure is similar to LISA language, that generalizes program for program verification under a memory model. In contrast, the operation structure captures the parallelism on operations on the same program. For verifying with loop iterations, Ogre and Pythia method uses an indicator to a memory operation and a condition to restrict the program executions must be invented to
show a proof under a memory model. Contrary to Ogre and Pythia method, our work automatically finds the finite set of execution paths to extract loop iterations for verification.

9. Conclusion

This paper introduces a framework for verifying assembly programs under a memory model. In particular, our framework focuses on a safety property for program verification. The corresponding operation structure of an assembly program captures essential behaviors of assembly syntax, which depends on its architecture. For verification, the approach to exploring the finite set of execution paths using a bound captures the subset program executions of the original programs. Each execution paths are then encoded by an encoding method, which is realized by a modeling framework, and use the Z3 solver to find an execution of the given program that violates the program property under a memory model.

The abstract way to execute programs in operation structures can be used to realize program executions with respect to various modeling frameworks. Currently, the Gharachorloo [12] and herding cats frameworks [11] are used to realize program executions for encoding and verification. In the future, there might be a framework that provides different style to model memory models. Thus, our abstraction would be able to realize program executions with an another framework for encoding.

By proposing path exploring algorithm in Sect. 4, an unstructured program, such as assembly program, can be used to explore the execution behaviors under a bound. In software verification, there are approaches to dealing with loops for SAT/SMT solvers, such as inductive invariant [29], k-induction methods [30], [31], and combine-case k-induction [17]. Those approaches would be applicable to explore the finite set of execution paths that cover more behaviors of the original programs.

In verification, proposed assertion language can ensure the values of registers at each program point and can prohibit inappropriate control flows of the programs, in which the assertion conditions are encoded in the first-order formula directly in verification. To extend the limitations of an assertion language, one would provide a property to indicate the status of other programs, such as a processor is accessing its critical section. However, the extension must be encoded in a proper first-order formula for program verification.

Acknowledgments

This work was supported by JSPS KAKENHI Grant Number 18H03220.

References


Appendix A: Semantics of Operation Structures

According to Sect. 4.1, Given a sequence of operation structures $P$ and execution state $\sigma$, the configuration is of the form $(P, \sigma)$ and the transition rules are defined to describe the operational semantics is of the form $(P, \sigma) \rightarrow \theta$ where $\theta$ is either $(P, \sigma')$, $\sigma'$, or $\hat{\sigma}$. To defined an operational semantics, first of all, the formalization of an execution state is shown in Sect. A.1. Then, the convention to define a transition rule is shown in Sect. A.2. Consequently, the transition rules are shown in Sect. A.3 for the semantics of operation structures.

A.1 Formalization of Execution State

Execution state. An execution state is tuple $(exec, reg, es)$ where $exec$ is an execution unit, $reg$ is a register state, and $es$ is an event state. Let $\sigma$ be an execution state, the elements of the execution state can be referred as:

- $\sigma$.exec corresponds to the execution unit,
- $\sigma$.reg corresponds to the register state, and

$$N[\sigma] = n$$
$$N[\sigma] = \rho(\hat{\sigma})$$
$$N[\sigma] \cdot N[\sigma] = \{\sigma_1 \text{ if } B[\sigma_1] = \top \}
\{\sigma_2 \text{ if } B[\sigma_2] = \bot \}$$

Fig. A-1 The semantics of an expression

$$B[\top] = \top$$
$$B[\bot] = \bot$$
$$B[e_1 = e_2] = \{\top \text{ if } N[\sigma_1] = N[\sigma_2] \}
\{\bot \text{ otherwise } \}$$
$$B[e_1 > e_2] = \{\top \text{ if } N[\sigma_1] > N[\sigma_2] \}
\{\bot \text{ otherwise } \}$$
$$B[e_1 < e_2] = \{\top \text{ if } N[\sigma_1] < N[\sigma_2] \}
\{\bot \text{ otherwise } \}$$
$$B[e_1 \land e_2] = \{\top \text{ if } B[\sigma_1] = \top \text{ and } B[\sigma_2] = \top \}
\{\bot \text{ otherwise } \}$$
$$B[e_1 \lor e_2] = \{\top \text{ if } B[\sigma_1] = \top \text{ or } B[\sigma_2] = \top \}
\{\bot \text{ otherwise } \}$$
$$B[\neg e_1] = \{\top \text{ if } B[\sigma_1] = \bot \}
\{\bot \text{ otherwise } \}$$

Fig. A-2 The semantics of a boolean expression

- $\sigma$.es corresponds to the event state.

Processor identifiers. Given a sequence of operation structures $P = P_1 \ldots P_n$, structures $P_i$ is supposed to be executed by processor $i$, where $1 \leq i \leq n$. From this assumption, let Pid be the set of processors identifiers such that $\text{Pid} = \{1, \ldots, n\}$.

Execution unit. Let the states of an execution unit are ready, performing, and complete. Ready state means the unit is ready to fetch the next assembly instruction into the unit. For performing state, the operations corresponding to the fetched instruction are expected to be performed in the order regarding the execution structure of the fetched instruction. After all operations in the execution unit are performed, the state of the execution unit becomes complete state.

Given execution state $(exec, reg, es)$, execution unit $exec$ is a mapping defined as the following.

$$exec : Pid \rightarrow \text{ExecStructure} \cup \{\text{ready, complete}\}$$

where ExecStructure is the set of execution structures. If the execution unit represents an execution structure, the operations in the structure are supposed to be performed on the system.

Register State. Given execution state $(exec, reg, es)$, register state $reg$ is a mapping of each program executed on processor $i \in \text{Pid}$ such that

$$reg : Pid \rightarrow (\text{Tmp} \cup \{\text{nPC} \cup \text{pc} \rightarrow \mathbb{N})$$

This refers to the state of registers for each processor $i \in \text{Pid}$. In addition, for the execution purpose, additional registers pc and nPC are considered for capturing the branch behavior in an assembly program. Moreover, there are the semantics of an expression $N$ and the semantics of a Boolean
expression $B$, shown in Fig. A-1 and Fig. A-2, defining for each processor, in which $n \in \mathbb{N}, x \in \text{Tmp}, e_1, e_2 \in \text{Exp}, c_1, c_2 \in \text{BExp}$, and $\rho$ is the register state for processor $i$, e.g., $\rho = \sigma \cdot \text{reg}(i)$.

**Event State.** In this paper, an event state is considered in an abstraction that contains the issued events and program orders on those events. Given operation $op$, the event issued by processor $i$ is denoted as $ev^i(op)$ at an abstract level. Operator $\oplus$ is a binary operator to add an event to an even state. For instance, $(ev_0 \oplus ev_1^i) \oplus ev_2^i$ means the $ev_1^i$ is issued before $ev_2^i$ in the program order of processor 1 if events $ev_1^i, ev_2^i$ are issued by processor 1 and event $ev_2^i$ is issued by processor 2. This information is then used to consider the possible return value regarding a memory model.

A.2 Conventions

As the formalization of the execution state shown in Sect. A.1 is quite detailed, a transition rule would be complicated to understand. Thus, to simplify the formalization, we would like to use the following conventions to reduce the complexity of the description.

**Generic Substitution.** Let $X, Y$ be any arbitrary sets, and $\varrho$ is any mapping $X \rightarrow Y$, we could define a generic substitution as following

$$\varrho[y \mapsto v](x) = \begin{cases} v & \text{if } x = y \\ \varrho(x) & \text{if } x \neq y \end{cases}$$

where $v \in Y$ and $x, y \in X$. Thus, this generic substitution can be used to update state in the semantics definition.

**Substitution of Operation Structures.** Given a sequence of operation structures $P = \varphi_1 \ldots \varphi_n$, and each operation structure is in a form of $\varphi_i = \gamma_1; \ldots; \gamma_{m_i}$, where $1 \leq i \leq n$ and $n, m_i \in \mathbb{N}^+$. To simplify the formalization of the semantics, the substitutions of operation structure and a sequence of operation structures are defined as followings.

$$(\gamma_1; \gamma_2; \ldots; \gamma_{m_i})^{j}[\gamma'](k) = \begin{cases} \gamma' & \text{if } j = k \text{ and } 1 \leq k \leq m_i \\ \gamma_1 & \text{if } k = 1 \text{ and } j \neq k \\ (\gamma_2; \ldots; \gamma_{m_i})^{j-1}[\gamma'](k-1) & \text{if } j \neq k \text{ and } 2 \leq k \leq m_i \\ \bot & \text{otherwise} \end{cases}$$

$$(\varphi_1 \cdot \varphi_2 \cdot \ldots \varphi_n)^{j}[\gamma'](k) = \begin{cases} (\varphi_1)^{j}[\gamma'] & \text{if } k = 1 \\ (\varphi_2 \cdot \ldots \cdot \varphi_n)^{j-1}[\gamma'](k-1) & \text{if } 2 \leq k \leq n \\ \bot & \text{otherwise} \end{cases}$$

The intention of the former substitution is to replace element $\gamma'$ into an operation structure $\gamma_1; \gamma_2; \ldots; \gamma_{m_i}$ at index $j^\text{th}$, and then access the element at index $k^\text{th}$. For the latter substitution, the element $\gamma'$ replaces the element at index $j^\text{th}$ of structure $\varphi_i$. Thus, the semantics can replace or refer to the specific element using these substitutions. For example, let $P = P_1 \cdot P_2$ and $P_2 = \gamma_1; \gamma_2; \gamma_3$, $P^3[\text{assume}(z = 1)](k)$ replaces the element $\gamma_3$ with $\text{assume}(z = 1)$ for any $k$ such that $1 \leq k \leq 3$. This means $P^2, 3[\text{assume}(z = 1)](3)$ is $\text{assume}(z = 1)$.

**Abbreviations of Execution State.** The substitutions of execution units and register states to processor $i \in \text{Pid}$ are abbreviated as followings to simplify the expression in semantics.

$$(\text{exec, reg, es})^{r(i)}[\text{reg}'] = (\text{exec, reg}[i \mapsto \text{reg'}], \text{es})$$

$$(\text{exec, reg, es})^{e(i)}[\gamma'] = (\text{exec}[i \mapsto \gamma], \text{reg}, \text{es})$$

Superscripts $r(i)$ and $e(i)$ are used to indicate the register state and the execution unit of processor $i$ to be substituted by $\text{reg'}$ and $\gamma'$, respectively.

**Evaluation Context of Execution Unit.** Given execution structure $\gamma$ to be issued, the order to perform operations could be defined using sequential execution and/or parallel operation. We would like to define an evaluation context, shown in Eq. (A-1), to indicate the fragment of structure to be evaluated in the system.

$$E ::= \square \mid E \cdot \gamma \mid E \parallel \gamma \parallel E \quad (A-1)$$

where $\square$ is a placeholder to place an execution structure into the context. For example, given evaluation context $(\gamma_1 \parallel (\square \mid \gamma_3))$ and execution structure $\gamma_2$, thus, $(\gamma_1 \parallel (\square \mid \gamma_3))\gamma_2$ = $(\gamma_1 \parallel (\gamma_2; \gamma_3))$. Besides, the execution structure $(\gamma_1 \parallel (\gamma_2; \gamma_3))$ can be separated to be context $(\square \parallel (\gamma_2; \gamma_3))$ and $\gamma_1$, such that $(\gamma_1 \parallel (\gamma_2; \gamma_3)) = (\square \parallel (\gamma_2; \gamma_3))(\gamma_1)$. Thus, the context can simplify the representation of the structure in instruction structure, such as $\text{instr}[E[\text{val}, z := 1]]$ where $E = (\square \parallel \text{val}, n := 0)$.

A.3 Transition Rules

According to the formalization in Sect. A.1 and the conventions shown in Sect. A.2, the transition rules can be described in this section.

First of all, the behavior of instruction fetching and the way to operate execution structures is defined as following rules.

\begin{itemize}
  \item $[\text{nil}] \cdot (P, \sigma)^{e(\text{nil})}[\text{nil}] \rightarrow (P, \sigma)^{e(\text{null})}[\text{complete}]$
  \item $[\text{seq}] \cdot (P, \sigma)^{e(\text{seq})}[E[\text{nil}, \gamma]] \rightarrow (P, \sigma)^{e(\text{seq})}[E[y]]$
  \item $[\text{par-l}] \cdot (P, \sigma)^{e(\text{par-l})}[E[\text{nil} \parallel \gamma]] \rightarrow (P, \sigma)^{e(\text{par-l})}[E[y]]$
  \item $[\text{par-r}] \cdot (P, \sigma)^{e(\text{par-r})}[E[\gamma \parallel \text{nil}]] \rightarrow (P, \sigma)^{e(\text{par-r})}[E[y]]$
  \item $[\text{fetch}] \cdot (P, \sigma)^{e(\text{fetch})}[	ext{instr}'[\gamma], \sigma^{e(\text{fetch})}[\text{ready}]) \rightarrow (P, \sigma)^{e(\text{fetch})}[	ext{instr}'[\gamma], \sigma^{e(\text{fetch})}[\text{ready}]]$
  \item $[\text{terminate}] \cdot (P, \sigma) \rightarrow \sigma \quad \text{if } \forall i \in \text{Pid}.$
\end{itemize}

\begin{itemize}
  \item $[\text{next}] \cdot (P, \sigma)^{e(\text{next})}[\text{complete}] \rightarrow (P, (\sigma\cdot\text{reg}(i(\text{pc})) = \bot))$
\end{itemize}

where $\sigma^{e(\text{next})}$ and $\sigma^{e(\text{fetch})}$ are defined as followings.

---

\(1\) This term is adopted from reduction semantics with evaluation contexts, which is an alternative representation of operational semantics.
According to rule \([\text{read]}\), \(\sigma^v_r\ll\langle\langle \sigma, \text{reg}(i)[\text{pc} \mapsto j+1]\rangle\)\]
\(\langle \langle \sigma, \text{reg}(i)[\text{nPc} \mapsto nPC]\rangle\)\]
\(\langle \langle \sigma, \text{reg}(i)[\text{pc} \mapsto nPC]\rangle\)\]

The following rules describe the behavior of operations that are performed by an execution unit on processor \(i\).

\[\langle P, \sigma^v_r[E[v := e]] \rangle \rightarrow \langle P, \sigma^{v+r}\rangle\] if \(e \in V \setminus \text{Tmp}\;
\]
where result \(\sigma^{v+r}\) is defined as follows.

\(R_v = e\ll(v := e)\)\]
\(\sigma_{es} = \sigma_{es} \oplus e\ll(v := e)\)\]
\(\sigma^{v+w} = \langle \langle \sigma, \text{exec}, \sigma, \text{reg}, \sigma_{es}\rangle \ll(\sigma_{es}) \ll(E[\text{nil}]\rangle\)\]

According to rule \([\text{read]}\), \(\mathcal{R}[r]es\) is a return value of read event \(r\) considered on event state \(es\). The function depends on each memory model, which could be a non-deterministic value. For instance, if there is only a write event to location \(X\) with value 1, the read event to location \(X\) can return either value 1 or an initial value 0. However, the decision could be more complicated on the relaxed memory models if there are various events in the system.

\[\langle P, \sigma^v_r[E[v := e]] \rangle \rightarrow \langle P, \sigma^{v+w}\rangle\] if \(v \in V \setminus \text{Tmp}\;
\]
where result \(\sigma^{v+w}\) is defined as the followings.

\(\sigma_{es} = \sigma_{es} \oplus e\ll(v := e)\)\]
\(\sigma^{v+w} = \langle \langle \sigma, \text{exec}, \sigma, \text{reg}, \sigma_{es}\rangle \ll(\sigma_{es}) \ll(E[\text{nil}]\rangle\)\]

\[\langle P, \sigma^v_r[E[v := e]] \rangle \rightarrow \langle P, \sigma^{v+w}\rangle\] if \(e, v \in \text{Tmp}\;
\]
where result \(\sigma^{v+w}\) is defined as follows.

\(\sigma_{es} = \sigma_{es} \oplus e\ll(v := e)\)\]
\(\sigma^{v+w} = \langle \langle \sigma, \text{exec}, \sigma, \text{reg}, \sigma_{es}\rangle \ll(\sigma_{es}) \ll(E[\text{nil}]\rangle\)\]

These rules, \([\text{if-T]}\) and \([\text{if-L]}\), represents the behavior of predicated instruction, which is abstracted by our operation structure.

\[\langle P, \sigma^v_r[E[\text{if}(c)(y')]] \rangle \rightarrow \theta_T\] if \(\mathcal{B}[c]\rho = \top\)
\[\langle P, \sigma^v_r[E[\text{if}(c)(y')]] \rangle \rightarrow \theta_L\] if \(\mathcal{B}[c]\rho = \bot\)

where \(\theta_T\) and \(\theta_L\) are defined as followings.

\(\rho = \sigma, \text{reg}(i)\)\]
\(\theta_T = \langle P, \sigma^{v}[E[y']\rangle\)\]
\(\theta_L = \langle P, \sigma^{v}[E[\text{nil}]\rangle\)\]

These rules, \([\text{if-T]}\) and \([\text{if-L]}\), represents the behavior of predicated instruction, which is abstracted by our operation structure.

\[\langle P, \sigma^v_r[E[\text{branch}(c, l)]] \rangle \rightarrow \theta_T\] if \(\mathcal{B}[c]\rho = \top\)
\[\langle P, \sigma^v_r[E[\text{branch}(c, l)]] \rangle \rightarrow \theta_L\] if \(\mathcal{B}[c]\rho = \bot\)

where \(\theta_T\) and \(\theta_L\) are defined as followings.

\(\rho = \sigma, \text{reg}(i)\)\]
\(\sigma_{abr} = \langle \langle \sigma, \text{exec}, \sigma, \text{reg}, \sigma_{es} \oplus e\ll(\text{branch}(c, l))\ll(E[\text{nil}]\rangle\)\]
\(P = Q^k[l]\)\]
\(\theta_T = \langle P, ((\sigma_{abr})\ll(\sigma_{es} \oplus e\ll(\text{branch}(c, l)))) \ll(E[\text{nil}]\rangle\)\]
\(\theta_L = \langle P, (\sigma_{abr}) \ll(E[\text{nil}]\rangle\)\]

Rules \([\text{br-T]}\) and \([\text{br-L]}\) represent the behavior of a branch operation that can change the next fetching instruction of the processor. To simulate such behavior, the register nPC is used to indicate next program counter for fetching the next instruction. In this semantics, label \(l\) is located in line number \(k\) in structure \(P(i)\). Thus, register nPC is substituted by \(k\) if condition \(c\) is satisfied.

\[\langle P, \sigma^v_r[E[\text{atom}(v_1 := e_1)]] \rangle \rightarrow \theta_{RK}\] if \(e_1 \in V \setminus \text{Tmp}\)
\[\langle P, \sigma^v_r[E[\text{atom}(v_2 := e_2)]] \rangle \rightarrow \theta_{RW}\] if \(v_2 \in V \setminus \text{Tmp}\)

where \(\theta_{RK}\) and \(\theta_{RW}\) are defined as followings.

\(\sigma_{es} = \sigma_{es} \oplus e\ll(\text{atom}(v_1 := e_1))\)
\[ \text{reg}_{i,r} = \sigma.\text{reg}(i)[v_1 \mapsto \mathcal{R}[\text{R}_{v_1}]\sigma_{\text{ex+r}}] \]
\[ \sigma_{+R} = \langle \sigma.\text{exec}, \sigma.\text{reg}, \sigma_{\text{es+r}} \rangle^{\sigma_{\text{reg}(i)}}[\text{reg}_{i,r}] \]
\[ \theta_{+R} = \langle P, (\sigma_{+R})^{\sigma_{\text{reg}(i)}}[\text{E}[\text{nil}]] \rangle \]
\[ \sigma_{\text{es+w}} = \sigma.\text{es} \oplus \text{ev}(\text{atom}(v_2 := e_2)) \]
\[ \sigma_{+W} = \langle \sigma.\text{exec}, \sigma.\text{reg}, \sigma_{\text{es+w}} \rangle \]
\[ \theta_{+W} = \langle P, (\sigma_{+W})^{\sigma_{\text{reg}(i)}}[\text{E}[\text{nil}]] \rangle \]

For the remaining, the behavior of annotations is defined as followings.

\begin{align*}
\text{[label]} & \quad \langle P^{i,j}[\text{label}(l)], \sigma \rangle \rightarrow \langle P^{i,j}[\text{label}(l)], \sigma' \rangle \\
\text{[assume]} & \quad \langle P^{i,j}[\text{assume}(c)], \sigma \rangle \rightarrow \langle P^{i,j}[\text{assume}(c)], \sigma' \rangle \\
& \quad \quad \text{if } B[c] \sigma.\text{reg}(i) = \top \\
\text{[assert-\top]} & \quad \langle P^{i,j}[\text{assert}(c)], \sigma \rangle \rightarrow \langle P^{i,j}[\text{assert}(c)], \sigma' \rangle \\
& \quad \quad \text{if } B[c] \sigma.\text{reg}(i) = \top \\
\text{[assert-\bot]} & \quad \langle P^{i,j}[\text{assert}(c)], \sigma \rangle \rightarrow f \quad \text{if } B[c] \sigma.\text{reg}(i) = \bot
\end{align*}

where
\[ j = \sigma.\text{reg}(i)(\text{pc}) \]
\[ \text{reg}_{\text{new}} = \sigma.\text{reg}(i)[\text{pc} \mapsto j + 1][\text{nPC} \mapsto j + 1] \]
\[ \sigma' = \sigma^{\sigma_{\text{reg}(i)}}[\text{reg}_{\text{new}}] \]

Pattaravut Maleehuan received the B.S. degree in Computer Engineering from Chulalongkorn University, Thailand (2013) and M.S. degree from Japan Advanced Institute of Science and Technology (2015). He is currently a doctor student at Japan Advanced Institute of Science and Technology. His research interests include formal methods, formal verification, model checking, SMT solver, and relaxed memory models.

Yuki Chiba received his B.S, M.S., and Ph.D. degrees from Tohoku University (2003, 2005, 2008). He is currently a researcher at DENSO Corporation, Tokyo. His research interests include program transformation, term rewriting system, automated theorem proving, and model checking.

Toshiaki Aoki is a professor in School of Information Science, JAIST (Japan Advanced Institute of Science and Technology). He received B.S. degree from Science University of Tokyo (1994), M.S. and Ph.D. from Japan Advanced Institute of Science and Technology (1996, 1999). His research interests include formal methods, formal verification, theorem proving, model checking, object-oriented design/analysis, and embedded software.