Title	マルチプロセッサのためのリラックスメモリモデルに基づいたSMTソルバによるプログラム検証
Author(s)	Maleehuan, Pattaravut
Citation	
Issue Date	2018-09
Type	Thesis or Dissertation
Text version	ETD
URL	http://hdl.handle.net/10119/15532
Rights	
Description	Supervisor:青木　利晃，情報科学研究科，博士

Program Verification for Multiprocessors with Relaxed Memory Models using an SMT Solver

Pattaravut Maleehuan

Japan Advanced Institute of Science and Technology

Doctoral Dissertation

Program Verification for Multiprocessors with Relaxed Memory Models using an SMT Solver

Pattaravut Maleehuan

Supervisor : Toshiaki Aoki

School of Information Science
Japan Advanced Institute of Science and Technology

September, 2018

Abstract

In modern multiprocessors, the consistency of shared memory would be relaxed to increase the computing power; hence, the value of a memory location could be observed as different values at the same time on each execution unit. Note that, term memory model is usually used to determine the semantics of the memory system. In particular, the memory model that relaxes the consistency of the shared memory is usually called relaxed memory model. Consequently, an anomalous result of the concurrent programs could occur on relaxed memory models. Therefore, relaxed memory model is the primary concern to ensure the program correctness.

For ensuring program correctness, the program property is defined as the invariant of the concurrent programs. Due to the relaxed memory models, this research provides an abstraction, called operation structures, of the concurrent programs. The targets of this abstraction are (1) to be sufficient for program verification, and (2) can describe the essence of assembly programs to be verified. Consequently, the program verification approach should be introduced to prove the program property on target relaxed memory model. In particular, this research uses SMT-based program verification approach to ensure the program correctness automatically.

This thesis shows two program verification methods for relaxed memory models. Mainly, the methods rely on the SMT-based program verification approach. In both methods, the behavior of program execution and the program property are encoded into a verification condition represented by a first-order formula; the formula is then used to check every execution satisfies the program property. The primary difference between the proposed methods is the way to abstract the behavior of program executions into the verification condition.

In both methods, the program executions are abstracted symbolically. In particular, the computation of program execution is considered in SMT-based program verification. The first method uses the bounded loop unwinding technique to abstract the symbolic executions. In the bounded method, the loop iterations are unwound systematically within a bound. For the second method, the inductive invariant approach is used instead of loop unwinding. However, the proposed inductive invariant method has seemed to be sound for partial store ordering (PSO) and stronger memory models. For SMT-based program verification, the abstraction of program execution and the program property are encoded regarding the relaxed memory model into a first-order formula. Primarily, the encoded formula is a decidable formula to be solved by an SMT solver automatically. Consequently, the program correctness can be ensured automatically.

In the experiment, an experiment tool was developed, and the Z3 solver is adopted to solve the first-order formula. As a result, the tool can automatically verify the property of the abstraction of concurrent programs on a relaxed memory model. In particular, the abstraction of concurrent programs can represent some essential behaviors of assembly programs. Besides, the bounded method is an under-approximation approach, while the inductive invariant method is an over-approximation approach.

In summary, concurrent assembly programs can be abstracted for ensuring the correctness by our methods. For the bounded method, the program correctness on a relaxed memory model

can be ensured if there is no loop. Otherwise, the method can at least disprove the program property on a relaxed memory model. As for inductive invariant method, the correctness of concurrent program contains loop can be ensured on partial store ordering (PSO).

Keywords: Concurrent Program Verification, SMT-based Program Verification, Multiprocessors, Relaxed Memory Model, and Automated Program Verification.

Acknowledgment

This dissertation could not be completed without supports, comments, and suggestions from many people. Especially, I would like to express my sincere gratitude to my supervisor, Prof Toshiaki Aoki, who always give the valuable comments and suggestions on my research topic. Without his help, its quite hard to imagine how this research can reach this state. In addition to the contents of my research, its almost 5 years that I studied at JAIST and there are several advisories to help me to enjoy living here.

Besides, I would like to express mine sincerely to Dr. Yuki Chiba, former Assistance Professor in our lab, who always help me since I came here until the last day he works at JAIST. Once I was a new student here, my mathematical skill is not good enough for this research. Fortunately, Chiba-sensei always correct my mistakes in my research and advise the right way to express the mathematic expression.

In addition, I thank all members and former members of Aoki-lab who help me to refine my work and discuss with me to improve the contents of the research and the paper that I wrote. Although my research topic is quite different from them, their help also helps me to improve the work in a general way, in which non-specialize researchers can understand.

I also would like to give my special thank to my friends and seniors for their help, suggestions, and encouragements during living in Japan. I also need to give my thanks to Thais friends, whose usually help and make me have a great time in Japan.

Finally, I would like to thank my family for supporting and encouraging me all the time. With out their supports, It's quite hard to go through obstracle of my life.

Contents

List of Figures

List of Tables

List of Definitions

Chapter 1

Introduction

1.1 Background and Motivation

Nowadays, *multi-core processors*, or *multiprocessor units*, are usually adopted in various computer systems. Those processors allow us to execute concurrent programs and/or parallel programs in a system simultaneously to reduce the execution time. In addition to personal computers, multi-core systems are also adopted in various embedded systems, such as automotive systems, recently to serve the high-performance for the system. For parallel programs, each program does not communicate with each other and their tasks can be completed on their own. On the other hand, concurrent programs are not completely independent of each other, and they can access the same memory locations to exchange the program information. In particular, a flaw of concurrent programs executed on embedded systems is a critical issue to be aware of during the software development since the flaw might leads the failure of the whole systems and could risk our lives. Consequently, the correctness of concurrent programs on embedded systems must be ensured.

Besides, multiprocessors using shared memory is also our focus, which executes concurrent programs independently. Recently, most of the modern multiprocessors use various optimization techniques, such as using write buffer. Those techniques intend to reduce the memory latency of memory accesses to shared memory. Note that the effect of the techniques usually appears implicitly to programmers. In particular, modern multiprocessors aggressively use optimization techniques to improve the performance of program execution. As a result, the execution order of the program statements would be out-of-order. In multiprocessors using shared memory, even if the execution order is changed, *memory model* or *memory consistency model* is provided to confirm the consistency of shared memory among multiprocessors.

In practice, there is no standard to describe the memory model in a formal way; processors' vender also describes the memory model of the multiprocessors in their ways. This means the behavior of program execution would be different on each memory model. Consequently, the program correctness on each memory model could not be ensured in the same way. For example, if the concurrent programs are correct on a memory model, we cannot conclude the programs are correct on a different memory model. Thus, this

```
static inline void
    arch_spin_lock(
    arch_spinlock_t *lock){
  unsigned long tmp;
  __asm__ __volatile__(
  "1: ldstub [%1], %0\n"
  " brnz, pn %0, 2f\n"
  " nop\n"
  " .subsection 2\n"
  "2: ldub [%1], %0\n"
  " brnz, pt %0, 2b\n"
  " nop\n"
  " ba, a, pt %%xcc, 1b\n"
  " .previous"
  : "=&r" (tmp)
  : "r" (lock)
  : "memory");
}
```

(a) For SPARC processors

```
static inline void arch_spin_lock
    (arch_spinlock_t *lock){
  unsigned long tmp;
  __asm__ __volatile__(
  "1: ldrex %0, [%1] \n"
  " teq %0, #0\n"
  WFE("ne")
  " strexeq %0, %2, [%1]\n"
  " teqeq %0, #0\n"
  " bne 1b"
  : "=&r" (tmp)
  : "r" (&lock->lock), "r" (1)
  : "cc");
  smp_mb();
}
```

(b) For ARM processors

Figure 1-1: Spinlock implementation in Linux Kernel

research aims to provide a way to ensure the program correctness on various memory models.

Moreover, this research also considers the program behavior at the hardware-level. In software development, the concurrent program could be implemented in a high-level language. For example, Figure 1-1 shows two implementations of Spinlock mechanism[1] in Linux kernel; each program contains a fragment of assembly instructions to exploit the processor functionality. Instead of considering C program mixing with various assembly syntax, this research considers assembly programs as the target for program verification. In particular, the program behaviors that affected by memory models are considered.

In software development, there are various techniques to verify the correctness of programs, such as providing test cases and code reviewing. However, such methods are not suited for verifying concurrent programs because they are required to highly reliable, and concurrent behavior must be exhaustively checked. *Formal verification* is a rigorous approach that applies to verify the correctness of concurrent programs. Thus, our research adopts the formal verification techniques to ensure the correctness of concurrent programs.

To summarize, this research would like to use formal verification techniques to ensure the correctness of concurrent programs. Besides, *memory model* is the primary concern for ensuring the program correctness on modern multiprocessors. In particular, this research considers the behaviors of concurrent programs affected by memory models; however,

[1]Spinlock mechanism is a mutual exclusion algorithm usually used in kernel system, which is a busy-wait mechanism

```
mov r1, #1
str r1, [x]
str r1, [y]
```

R1

```
L:
  ldr r1, [y]
  cmp r1, #1
  bne L
ldr r2, [x]
assert(r2 = 1)
```

R2

Figure 1-2: Message passing

the considered behaviors are expected to represent the concurrent assembly programs. Consequently, the correctness of the concurrent programs written in an assembly language could be ensured on a memory model.

1.2 Memory Models of Multiprocessors

In modern multiprocessor systems using shared memory, there are various mechanisms that affect the behavior of program executions. Those mechanisms could be used to reduce the memory latency to share memory. However, the mechanisms appearing in the systems are too concrete and specific to each processor to be considered in program verification. In general, the multiprocessors usually specify their *memory model*, or *memory consistency model*, to guarantee the effect of memory accesses to shared memory.

Sequential Consistency *Sequential consistency model*, denoted as SC, is a standard memory model in which the effect of memory accesses is always as same as the programs executed in a sequential way regardless the mechanisms inside the processors. Figure 1-2 shows message passing programs written in ARM assembly language. The program `R1` writes the value 1 to the memory locations `[x]` and `[y]` in the order. On the other hand, program `R2` reads memory location `[Y]` until the read value becomes 1, then, reads memory location `[X]` and writes to register `r2`. The program property requires the value of register `r2` always equals 1 for any program execution. Although multiprocessors allow the latter write access issued by program `R1` is completed before the former write access, the multiprocessors using sequential consistency model guarantee the effect of executions is always as same as the behaviors are changed implicitly. This means the program property is always satisfied by any execution from programs $R1$ and R_2.

Relaxed Memory Models *Relaxed Memory Models*, or *weak memory models*, are memory models usually used by modern multiprocessors. Since sequential consistency guarantees the effect of the program executions to be as same as the program executed sequentially, in such model, providing efficient mechanisms to reduce memory latency of the systems using shared-memory would be difficult. Consequently, most of the modern

processors decide to relax the effect of memory accesses to shared memory so as not to restrict themselves to the sequential way of an execution.

Total Store Ordering (TSO) is the relaxed memory model that allows the effect of a read access to appear before a non-conflicting write access. For instance, the following ARM program writes a value to memory location `[X]` and reads a value from memory location `[Y]`.

```
1 str r1, [X]
2 ldr r2, [Y]
```

In TSO memory model, the read access is allowed to be completed before the prior write access, in particular, the effect of non-conflicting write accesses[2] can occur later than a read access, which would not affect the computation in a single processor. In a practical system, the write access can be stored in a write buffer before completing later, while the read access can be bypassed to read the memory location if there is no conflicting access to the location. However, in multiprocessor systems, the completion of write accesses can appear later to other processors' viewpoint and could cause anomalous effects in the system.

Partial Store Ordering (PSO) is another relaxed memory model which extends TSO memory model to allow the effect of non-conflicting writes to appear out-of-order. For instance, the following ARM program writes a value to memory locations `[X]` and `[Y]` in the program order.

```
1 str r1, [X]
2 str r1, [Y]
```

In PSO memory model, the latter write access is allowed to be completed before the prior write access, if the accesses are not in conflict. In a practical system, the write accesses can be stored in a write buffer before completing later, and the buffer could be a non-FIFO buffer, in which the order of issued writes can be out-of-order. By using PSO memory model, the effect can be observed by other processor and the program property could be violated.

Let's consider the programs in Figure 1-2 are executed on TSO and PSO memory models with the program property `r2 = 1` for any execution. For TSO memory models, the effect of every execution from the programs is as same as the programs executed on sequential consistency model. On the other hand, PSO memory model allows the write accesses in program `R1` to be completed out-of-order. This means if the latter write access in `R1` is completed the latter read access in `R2` can read the value of the initial value of location `[X]`, which violates the program property `r2 = 1`. This shows that the effect of each relaxed memory model is different from each other and could violate the program property because of the implicit behavior.

Note that total store ordering (TSO) and partial store ordering (PSO) memory models are conceptual models to illustrate the behavior of a relaxed memory model in a shallow way. In practice, there are various relaxed memory models provided for each processor such as x86-TSO, SPARC-PSO, POWER, and ARM. The practical memory models are

[2]Two memory accesses are considered as conflict if one of them is a write access and access to the same memory location.

usually described in the hardware manual and have no standard description. Besides, to exploit their hardware functionality, the behaviors of practical memory models are quite complicated than the conceptual memory models.

1.3 Program Verification

Due to the fact that the effect of concurrent programs executed on relaxed memory models is not the same as the program executed on sequential consistency models, the program property could be violated by anomalous executions permitted by relaxed memory models. This means the property to be verified must consider the effect of program execution on target memory model.

Although, in program verification, there are various researches to verify concurrent programs, the verification techniques in most of these cannot be adapted to verify programs on multiprocessor system directly because they do not take relaxed memory models into account. There are several works that provide formal ways to verify program property on specific memory models, such as [Rid10, LV15]. By considering on specific models or specific architectures, the behavior to be verified could be more concrete for their target. Instead of considering on specific memory models, our research would like to provide a verification method for a variety of memory models. In program verification, the effect of program executions is of interested to verify if the program property is preserved.

To realize the effect of program executions for verification, there are frameworks to model the program behavior regarding the specification of memory models. In these frameworks, although there is no standard description of a memory model, they provide their specification styles to determine a valid behavior regarding their abstraction. Given instances of memory accesses occurring in the system, those instances are then considered based on the abstraction of a modeling framework, and the behavior on a memory model must satisfy the memory model specification provided by the framework. Intuitively, we can decide which behavior is valid regarding a memory model using a memory model specification provided by a modeling framework. Note that behavior considered in each framework can be different from each other, such as Gharachorloo [Gha95] represents the behavior in term of execution order, while Alglave [AMT14] represents the behavior in term of the communication of events.

Our program verification adopts the way to abstract the behavior and memory model specifications to realize the effect of program executions for a variety of memory models, in which herding cats framework [AMT14] and a framework provided by Adve and Gharachorloo [Gha95] are considered in our research. In particular, SMT-based program verification is proposed in this research so as to adopt a memory model specification to determine the valid effect on a program execution automatically based on axioms on the instances of memory accesses.

1.4 Objective

The objective of this research is to propose a formal program verification method of concurrent programs executed on a multiprocessor system. In particular, the relaxed memory models are also considered in program verification, in which the concurrent programs is considered at the hardware-level. Instead of verifying a high-level language, such as C language, assembly languages are 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, due to a variety of assembly languages for various multiprocessor architectures, an abstraction of assembly instructions is considered for program verification. The abstraction would be proposed to focus on the behavior that is necessary for realizing the effect of the programs executed on a relaxed memory model. This means the assembly instructions that are not related to the computation are not considered, such as interrupt instructions. In program verification, we focus on the properties of programs, in which the safety property on the computations on relaxed memory models is our concern to ensure the program correctness.

1.5 Thesis Outline

First of all, Chapter 2 explains the technical background of this research, which includes (1) the behavior of multiprocessors using shared memory, (2) the assembly instructions considered in this research, (3) frameworks to model the effect of programs executed on a relaxed memory model, and (4) program verification using SMT solver.

Then, Chapter 3 and Chapter 4 describe the methods proposed in this thesis. For Chapter 3, bounded loop unwinding method was described to verify assembly programs using SMT solvers. In particular, the method uses a bound to restrict the number of loop iterations to be unwound. Due to the restriction of the bounded method, Chapter 4 was described to abstract the loop behavior. In the latter method, the most definitions and approaches in the bounded method are adopted for program verification, while some mechanisms are changed to deal with the loop behavior which is a limitation of the bounded method.

After that, Chapter 5 shows the case studies and experiments on the proposed methods, then the evaluation of the methods is also shown. Then, Chapter 6 shows the related works to our research. Chapter 7 concludes the overall of this thesis and discuss the result of our method.

Chapter 2

Preliminaries

2.1 Multiprocessors using Shared Memory

Currently, there are various kinds of processors adopted in computer systems and embedded systems, such as automotive systems. Besides, multi-core processors become more popular to increase the performance of the systems. Although there are various kinds of processors, our research focuses on multiprocessor units that appear to execute program independently, called *multiprocessor systems*.

This research focuses on the concurrent programs that are executed on a multiprocessor system using shared memory. According to Figure 2-1, the system consists of 3 processing units to execute programs simultaneously. If there are concurrent programs, the read and write accesses are going to pass the caches `L1`, `L2` and the system bus to system memory. Note that there are various mechanisms to optimize and/or reduce the memory latency such as write buffer and speculative executions.

An assembly program is assumed to be executed on a processing unit, in which its instructions must be performed in the order defined by the program. In users' point-of-view, a performed instruction is deemed to be completed before the next instructions are performed. However, in practice, various mechanisms are adopted to improve the execution performance, such as using write buffer and read-forwarding mechanisms. Consequently, these mechanisms permit out-of-order executions of memory accesses in a system. Although out-of-order executions can occur, a processor usually provides mechanisms to correct the result of return values of the read accesses to be the same as the sequential execution of memory accesses. For concurrent programs, the order of the read accesses and the write accesses is permitted to be executed out-of-order without the mechanisms of a processor to control the behavior among processing units. Thus, a cache's protocol can be adopted to synchronize the values of a location that can be observed among processors.

In modern multiprocessors, such as ARM and POWER, various complicated mechanisms are used to maximize the performance of their processors. In practice, maintaining the order of memory accesses along with the optimizing mechanisms is quite difficult. Thus, most modern processors usually permit out-of-order executions of memory accesses to occur globally. Accordingly, anomalous return values of the read accesses could be pro-

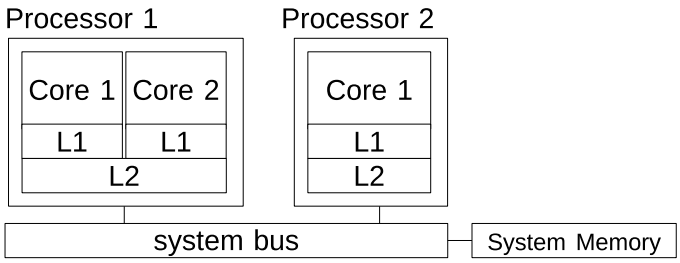


Figure 2-1: Overview of an multiprocessor systems

duced by concurrent programs. However, to correct the results of concurrent programs, synchronizing instructions are provided regarding a processor to ensure the executions of some fragments of programs.

The term *memory model*, or *memory consistency model* is used to describe the behavior of memory accesses to shared memory in multiprocessor systems. The behavior of programs occurring in a multiprocessor system can be determined by memory model, no matter what optimizing mechanisms are adopted in a practical processor. Note that the memory model of a multiprocessor is usually described in processor architecture's manual, which is described informally and differently to each processor's vendor.

2.1.1 Hardware Optimization

Out-of-order issuing

In the fetching instruction, normally the instructions are fetched in the order decided by a program counter. Sometimes the next instruction should wait until the necessary registers are available. Instead of waiting, this technique stores the instruction in a reservation station or an instruction buffer. The reservation station will issue an instruction that is required registers are available. That means these behaviors will allow out-of-order issuing of instructions. In some cases that the programs are executed in multiprocessor systems, this behavior is known by only its processor. Therefore, the unexpected results may be produced.

Non-blocking read access

In the execution units, the instruction will be performed as micro-operations and memory accesses. As for the read accesses, there are some situations that the read access cannot be performed immediately. The causes maybe read miss in caches or the memory locations is not available yet. Hence, these mechanisms have been introduced to skip such read accesses to perform next micro-operation or memory access. However, this read access will be performed again once the value of the read is needed. This behavior can be realized using the read buffers.

```
A = 1
B = 1
a = A
```

Program A

```
x = B
y = A
```

Program B

Figure 2-2: Example for bypassing read access

Bypassing read access

The write accesses usually are put into write buffers. To issue the read access in the program order, the read operation usually has to wait until the previous write accesses already be issued from buffers. In this case, the processor should be stalled itself before performing the next operations. To reduce the stalls, bypassing read access have been introduced. The read access can be performed immediately if and only if there are no write accesses that access to the same address as the read access. Hence, this behavior will cause that a read access may be executed before write accesses specified as earlier operations. In the same processor, this behavior will not produce unexpected results. Nevertheless, in the multi-core systems, the order of some write accesses and read accesses maybe significant order to be considered. For example, let's consider Figure 2-2, we define 'A =1' and 'B=1' as write access, and the remaining are read accesses. Assume that the write access 'A=1' already executed in the shared-memory and 'B=1' is stored in the write buffers. In this case, the read access 'a = A'can read the value 'A = 1' from shared-memory immediately, even if the write access 'B = 1' is not executed yet. In this case, the result of (x, y) is (0, 1) can be happened in the multi-core systems.

Read Forwarding

This mechanism also reduces the stalls of that processor by immediately issuing the read access if and only if there is a write access stored in write buffers which access the same memory locations as the read access. However, such return value should be the value from the last write access that appeared in the buffer. Although this mechanism can reduce the stalls in processors, some hardware does not allow this mechanism to be implemented due to it may provide some unexpected results.

Non FIFO read/write buffers

Generally, buffers usually act like queues which issue an entity in order as First-In-First-Out(FIFO). In some cases, the earlier read or write accesses cannot be executed yet, because accessing memory location is not available. Moreover, In the case of out-of-order issuing, some accesses should be issued in the program order, but buffers might have previous accesses which should be issued later. Hence, such accesses will be selected to be issued before the previous accesses in the buffers.

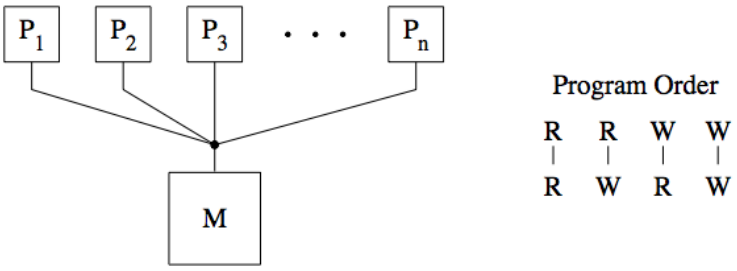


Figure 2-3: Conceptual model for sequential consistency model (SC) [Gha95]

2.1.2 Memory Models

Due to the fact that the behavior of a program execution is a significant issue to verify the correctness of concurrent programs, this section would like to explain about memory models, which describes the behaviors of memory accesses among multiprocessors. By considering on memory models, the program verification could be done in an abstraction of the system regardless of the mechanisms used in the practical system.

In practice, the term memory model is used either in a programming language such as C++11 and Java Memory Model (JVM) or at the hardware level, such as SPARC-TSO and x86-TSO. In this research, we focus on the memory models at the hardware level to ensure concurrent assembly programs are correct on a memory model. Thus, this section explains the behavior of memory models at the hardware level.

To explain the behavior of memory models, the conceptual model provided by [Gha95] is adopted as an abstraction of a multiprocessor system to explain the behavior regarding a memory model. Figure 2-3 shows a standard representation for sequential consistency model. There are n processing units, $P_1, \ldots, P_n$, sharing a single logical memory M in the conceptual model, which shows the concept for programmer's point-of-view. Read and write accesses are treated as **R** and **W** in a conceptual model. Read access **R** is considered as complete if the return value is determined. Write access **W** is considered as complete once the target location in logical memory M is updated. The program order in the right-side of the figure shows conditions on the read and write accesses issued by programs, in which the completing order must follow the program order if there is a line between them. To determine the return value of a read access, the value must be the last write to the same location that completed before the read completes.

Note that a conceptual model could not be used to determine the complex behavior of modern memory models, such as ARM and POWER. Nevertheless, a conceptual model is used in this section to introduce the simple memory models intuitively. This would help readers to understand some basic behaviors that could occur, which cause a violation in concurrent programs.

Sequential Consistency Model

Sequential consistency model, denoted by SC, is a standard memory model that requires the result of programs must be the same as the program executed in a sequential way even

if an out-of-order execution occurs. According to 2.1 provided by [Lam97], the constraint is given to implement a system that is a sequential consistency model.

Definition 2.1 (Sequential Consistency Model)**.** the result of any execution is the same as if operations of all the processors were executed in some sequential order, and the operations of each processor appear in this sequence in the order specified by its program.

This condition considers the side effects of executions that can occur by the behavior of program executions, in which no matter how the processor was implemented.

According to Figure 2-3, the completing order of all read and write accesses must follow the program order, by conditions appearing in the right-side of the figure. However, this conceptual model does not restrict the way to complete each of read and write accesses in a system. This means some processors might put a write access into a write buffer, and then a read access either enforces the conflicting writes to be completed in the order or read the writing value of the last conflicting write in a write buffer, which is the read-forwarding mechanism. However, both of implementations could provide the side effects as same as there are no optimizing mechanisms adopted in a processor.

This is a model which is usually easy for programmers to implement software on top of the model because the execution is always the same as they implement software. Besides, there are various verification techniques that can be used to verify this kind of multiprocessors using sequential consistency models, such as Hoare logic. As the side effect of executions always the same as the programs executed in an interleaving way, thus, program verification in an interleaving behavior is sufficient to ensure the correctness of the system.

However, in practice, this model could provide the low performance of a multiprocessor system. Due to the need for constraints to restrict the executions, the optimizing mechanisms could not be used. Therefore, most of the modern multiprocessors provide a *relaxed memory model*, or *weak memory model*, for its processor, which permits anomalous results to occur, but they are acceptable in some programs. By permitting anomalous results, considered as relaxing the execution, the processor can use more optimizing mechanisms to improve the performance of multiprocessors.

Relaxed Memory Models

Relaxed memory models are memory models permitting the execution orders not to follow the program order. Although the execution order is permitted to be changed implicitly, there are some restrictions of a relaxed memory model to control the behaviors as an acceptable relaxation permitted by the memory model. In general, each memory model usually provides different constraints for its conceptual system. Primarily, each multiprocessor's provider usually describes the behavior in an informal way and different from each other. However, using a conceptual model, readers could intuitively see how the execution order of programs is changed at an abstract level. Note that relaxed memory models explained in this section are abstracted models, in which the practical systems could provide more details for their memory model to facilitate the processors, such as fence instruction for a pair of write and read accesses.

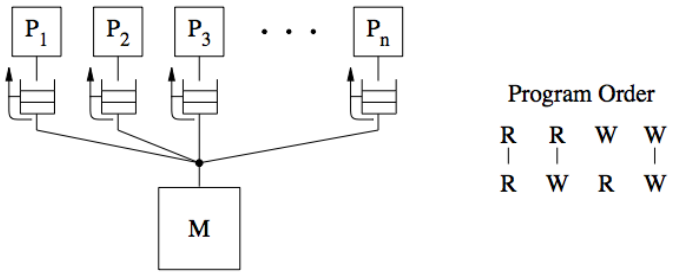


Figure 2-4: Conceptual model for total store ordering (TSO)[Gha95]

```
1 [X]  := 1          1 [Y]  := 1
2 v  :=  [Y]         2 u  :=  [X]
```

P_1 P_2

Figure 2-5: Store Buffer (SB)

Total Store Ordering (TSO): Figure 2-4 shows the conceptual model for total store ordering (TSO), which relaxed the order of reads following a write access to be completed out-of-order. This conceptual model is deemed to be same as the model for sequential consistency model. In contrast to the sequential consistency model, this model provides a write buffer between a processor and the logical memory. These buffers store the issued write accesses to reduce the memory latency of the system. Besides, read accesses can read the write value from the last conflicting write access in the buffer *if they are issued by the same processor*. This means the conflicting reads can be complete before the former write accesses in a processor. In addition, there is no line condition for the pair of write and read accesses in the conceptual model for total store ordering. This means any read access can be completed before a write accesses is completed among processors.

Figure 2-5 shows concurrent programs that the processors write value 1 to the share-memory location `[X]` and `[Y]` for processing unit P_1 and P_2, respectively. These write accesses are supposed to be stored in the write buffers of the processors. Consequently, each read access of each processor, which is non-conflicting accesses, could read the value from the shared memory directly, without any update from a write access. Consequently, the local variables u and v can be 0, which is the initial value of a memory location, in some executions on total store ordering (TSO). In contrast, if the programs are performed on sequential consistency model, the variables v and u are not permitted to be 0 at the same time. Note that total store ordering model adopted in practical processors usually provides fence operations to prevent the changing order between write and read accesses.

Partial Store Ordering (PSO): Figure 2-6 shows the conceptual model for partial store ordering (PSO), which is an extension of TSO memory model. PSO model is quite same as TSO model, excepts the conditions between write accesses. The dotted line appearing in the figure means the conflicting write accesses must be completed in the program order. The line with annotation **F** means non-conflicting write accesses are permitted to be completed out-of-order if there is no fence between them. In an imple-

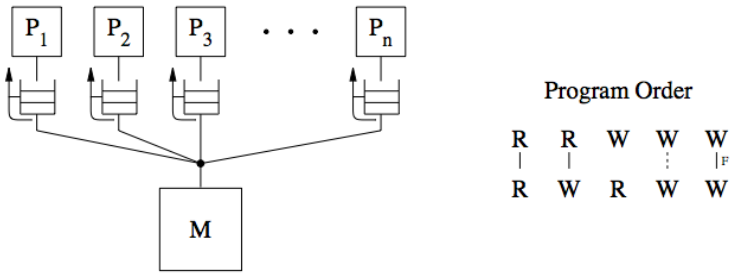


Figure 2-6: Conceptual model for partial store ordering (PSO) [Gha95]

```
1 [X] := 1          1 u := [Y]
2 [Y] := 1          2 v := [X]
```

P_1 P_2

Figure 2-7: Non-FIFO Buffer

mentation, Non-FIFO write buffers could be used to issue the latter writes that can be completed faster than the former write to be completed in logical memory M.

Figure 2-7 shows concurrent programs that processor P_1 writes value 1 to the location `[X]` and `[Y]` in the order, while processor P_2 reads location `[Y]` and then location `[X]`. The write accesses are supposed to be stored in a Non-FIFO write buffer, in which the write accesses can be issued from the buffer in either order. Consequently, the first read to location `[Y]` can read the value 1, while the latter read to location `[X]` got value 0 in which the write access to `[X]` is not completed yet. In contrast, if the programs are performed on TSO memory model, the write accesses must be completed following the program order even if they do not conflict with each other. To prevent the out-of-order execution in PSO model, a fence must be added between write accesses to prevent such anomalous behaviors. Obviously, the multiprocessors using PSO model seems to be more efficient than the multiprocessors using TSO model, while there is a trade-off between ease of implementation and efficiency of program execution.

POWER and ARM: In modern processors such as ARM and POWER architectures, they provide memory models weaker than TSO and PSO. Intuitively, the completion order of the read accesses is allowed to be out-of-order if there is no dependency between the read accesses. However, as there is no standard description for memory models, it would be difficult to realize the memory behavior on POWER and ARM architectures. There are researches such as [AFI+09, ISS12, MHMS+12, AFI+09, SSA+11] to analyze the behavior of POWER processors. For ARM multiprocessors, the behavior is quite similar to POWER multiprocessors; thus, the intuitive way to consider the behavior on POWER and ARM multiprocessors would be similar.

In POWER and ARM multiprocessors, most of the memory accesses are allowed to be reordered if there is no dependency on them. In POWER multiprocessors, the sequential consistency of memory accesses can be achieved by using `sync` instruction. However, if `sync` instruction is added between any instruction, there might be time-consuming which

causes the inefficient of program execution.

According to [SSA+11], there are dependencies detected by POWER multiprocessors to maintain the completing order on the memory accesses that have such dependencies. There are various kinds of dependencies:

- **Address dependency (addr)**: if the read value of a read access is used as the memory address of the following read access or write access in the program order.
- **Data dependency (data)**: if the following write access uses the read value of a read access in the program order.
- **Control dependency (ctrl)**: if the read value of a read access is used as a branch condition, there is a control dependency from the read access to the following memory access after a branch.
- **Control+isync dependency (crtlisync)**: if there is a control dependency from a read access to a read access after a branch and there is isync instruction after branch and before the second read access in the program order.

For *address dependency*, the dependency is from a read access to any memory access if the read value is used as the memory address. For example, the following program shows the dependency between two instructions such that register `r1` is used as the memory address for the second instruction. Thus, there is an address dependency from the load instruction to store instruction.

```
1 li r2, 2
2 lwz r1,0(r2)
3 stw r1,0(r1)
```

For *data dependency*, the dependency is from a read access to a write access if the following write access uses the read value. This dependency also includes the address dependency; however, the behavior that the value to be stored is also considered. For example, the following programs used the value of register `r3` as the write value of the write access. Hence, there is a data dependency from the load instruction to the store instruction.

```
1 li r1, 1
2 li r2, 2
3 lwz r3, 0(r1)
4 stw r3, 0(r2)
```

For *control dependency*, the dependency is from a read access to any memory access if there the read value of the read access affecting the branch condition and the memory access appear after the branch instruction. For example, the following program show the condition of branch instruction `bt eq L1` relies on the read value of load instruction `lwz r3, 0(r1)`. Thus, there is a control dependency from load instruction to store instruction `stw r5, 0(r2)` appear after the branch instruction.

```
li r1, 1
li r2, 2
lwz r3, 0(r1)
cmpwi r3, 1
bt eq L1
stw r5, 0(r2)
```

For *control+isync dependency*, the dependency is between two read access which similar to the control dependency. In this dependency, the isync should appear before the second read access and after the branch instruction. For example, the following program shows the control+isync dependency between two load instructions.

```
li r1, 1
li r2, 2
lwz r3, 0(r1)
cmpwi r3, 1
bt eq L1
isync
stw r5, 0(r2)
```

2.2 Assembly Program

In program verification on systems using relaxed memory models, the behavior of read accesses and write accesses to shared memory is the concern. In particular, the target memory models in this research are at the hardware-level, in which *assembly programs* are considered to ensure the program correctness on relaxed memory model.

An assembly language is a low-level programing language to interact with specific processor architecture. Primarily, a compiler is usually provided for a specific processor to translate the language for facilitating the processor. In the kernel development, there could be various assembly languages used for implementing the same functionality in a kernel, such as Spinlock. Although each assembly language provides different mechanisms to facilitate target processor, there could be similar behaviors that are necessary for program verification on relaxed memory models.

As there are various assembly languages, Section 2.2.1 shows some assembly instructions based on ARM instruction set. This would shows the practical behaviors of assembly program. Then, Section 2.2.2 shows our assumption that an assembly instruction is not performed atomically in the hardware's point-of-view. Consequently, the precise behavior of assembly programs can be considered on relaxed memory models.

2.2.1 Assembly Instructions

In practice, there are various assembly instructions to be used for specific processor architectures. In this explanation, ARM instruction sets described in ARMv7 manual [ARM07] are used to show the practical instructions provided for program implementation. First,

the preliminary behavior for assembly instructions is introduced. Then, the explanations of some practical instructions are shown.

Preliminary behavior of Assembly Instructions

Conditional execution is introduced for the most of ARM instructions. In this research, the instructions that contain the condition to be considered are called predicated instructions. Intuitively, a condition can be put on an instruction for performing the instruction. In ARM processors, a condition flag appearing in application status program register (ASPR) on the executing processor are used to determine the execution of the instruction. For example, the following program uses compare instruction `cmp r1, #1` and save the result to the condition flags in the application state program register (ASPR).

```
cmp r1, #0x01
streq r1, [0x01]
```

Note that ASPR is used to save the state of the executing program. Then, the store instruction `streq r1, [0x01]` checks the condition flags whether the condition is satisfied before performing the store instruction in a usual way. Intuitively, the condition denoted by '`eq`' intents to check the value of register `r1` equals 1. Note that, for other processors, the condition to be checked could be the register directly.

Writing to program counter (PC) is allowed in many data processing ARMv7 instructions. In general, program counter is a register in a processor to determine the instruction to be performed. To manipulate the value of the counter, branch instructions are the basic instruction to change the control flows of the program. Besides, ARMv7 allows other instructions, such as load instructions and data-processing instructions, to change the value of the program counter.

Label in Unified Assembler Language (UAL) syntax can be used in an ARM assembly program. In actual processing, the information to be used in the program must be indicated by the memory address. For instance, the program counter is used to indicate the instruction to be fetched. For the ease of implementation, UAL syntax allows users to use labels for indicating the program information at the specific program locations. Then, such labels would be translated to be the address of the program location using the value relating to the value of program counter. For example, the label would be replaced by `[PC-32]` which refers to the address before the current instruction.

Branch Instructions

In practice, there are various branch instructions to be used. The *basic branch instruction* (`b`) is to jump the program execution into the target address, which can be described by label in UAL syntax. Normally, this basic branch is used for defining the control flow of a program. The syntax of basic branch instruction in ARMv7 is of the form `b`$\langle c \rangle$ $\langle label \rangle$, where $\langle c \rangle$ is the condition to be checked and $\langle label \rangle$ is to indicate the address to be indicated in the program.

In addition to checking the condition flag, *compare and branch instruction* (`cb`) can be

used to check the value of the target register before branch to the target address. The syntax of compare and branch instruction is of the form `cb`$\{n\}z$ $\langle Rn\rangle$ $\langle label\rangle$ where $\langle Rn\rangle$ is the target register to be checked and $\{n\}$ can be specified if we need to check the target register not equal to zero. Otherwise, the register equals zero is checked as the condition if there is no $\{n\}$ in the instruction.

Moreover, ARM assembly language allows users to define subroutines for program implementation. Normally, subroutines would process on the given parameters stored in some registers and return the result in some registers. For the branch instruction to subroutines, branch link instruction (`lb`) is used to jump to the target label or pc-relative address. In addition to the basic branch instruction (`b`), the current address before changing is saved in the link register (`LR`). After the routine is completed, the branch and exchange instruction (`bx`) is used to jump to the address specified by the target register. For instance, the following program shows the usage of branch to subroutine.

```
__main:
  ldr r1, [0x01]
  mov r0, #0
  bl AFUNC    ; Call subroutine AFUNC
  str r0, [r3]
  b STOP
AFUNC:
  add r0, r0, r1
  subs r1, r1, #1
  bx LR     ; restore the address to PC
STOP:
```

Data-Processing Instructions

This group of instructions is used to manipulate the data in a program. These instructions provide the ways to compute on registers for each processor, such as arithmetic calculation. The following contents show some instructions that are provided for ARMv7 processor architecture.

Standard data-processing instructions provide basic calculations on the registers, such as adding and subtraction. These would compute the program using the basic operator for the data stored in the registers. Besides, there are some compare instructions and test instruction. These instructions provide some decision on the registers and store the result on the condition flag register, which is stored in application program status register (APSR).

Multiplication instructions and **divide instructions** provide more complicated calculation on the registers. For instance, instruction `MUL r2, r1, r0` can be interpreted as the multiplication of `r1` and `r0` is stored in register `r2`.

Parallel addition and subtracting instructions provide the functionality to perform addition and subtractions on the values of two registers and write the result to a target register. These instructions are single instruction multiple data (SIMD) instruc-

tions that provide various actions within an instruction.

Status Register Access Instruction

In program execution, the application program status register (APSR) is used to indicate the state of programs during program execution, such as the state of condition flag registers. Besides, there are reservation bits in the register that can be used for a special purpose such as disable the interruption behavior of the program. To manipulate the behavior, ones can use MRS and MSR instructions to move the contents between APSR to or from an ARM core register.

Load/Store Instructions

Load/store instructions provide the functionality to communicate with a memory location. There is a variety of instructions to load or store the memory locations, such as the data type and the number of accesses to be performed. In the implementation, the data type is the matter in the implementation of an assembly program; thus, the variety of instructions regarding the data type is provided. Besides, ARMv7 architecture also provides the load/store multiple instructions to communicate with multiple memory locations.

Push and pop instructions are provided in ARMv7 multiprocessors to realize the stack behavior in the system. In particular, stack pointer register (SP) is used to refer the last register that is put in the stack. Moreover, in ARM multiprocessors, load-exclusive and store-exclusive instructions provides the functionality to facilitate synchronization behavior among multiprocessors. The behavior of load-exclusive and store-exclusive is considered as the behavior of load-link and store-condition in this research.

Load-link is a group of special instructions that are introduced in modern processors, such as ARM and POWER. This instruction is supposed to be used as a pair with store-condition instruction. Intuitively, load-link instruction issued a read access to the memory location and then trigger the processor unit to link the exclusive address, implemented in modern processors, to the target memory location. The exclusive address in a processor unit is used to check whether is there other load-link instructions or a write access to that target address or not.

Store-condition is a group of instructions that is also introduced in modern processors, such as ARM and POWER. The behavior of this kind of instructions is to check the exclusive address still be the same as the address of the instruction or not, and there is no write access that access the target memory location after the load-link instruction, which is used as a pair. If the condition is satisfied, the store-condition instruction issues a write access to the memory location and writes 0 to the result register to let users know that the write access is completed. Otherwise, there is no write access to the system and the instruction writes 1 to the result register to let users know the write access cannot be completed. Note that, in practical processors, the mechanism to link the address could use a cache-line in the processor, which relates more than one memory addresses. This means if there is a non-conflicting write access on the same cache-line, the store-condition

Instruction	Summarize
Clear Exclusive (clrex)	Clears the local record of the executing processor that an address has had a request for an exclusive access.
Debug Hint	Provides a hint to debug and related systems.
Data Memory Barrier (dmb)	Ensures the ordering of observations of memory accesses
Data Synchronization Barrier (dsb)	Ensures the completion of memory accesses
Instruction Synchronization Barrier (isb)	Flushes the pipeline in the processor
No Operation (nop)	does nothing
Send Event (sev) and Wait For Event (wfe)	`wfe` instruction permits the executing processor to enter the low-power state until received an event.
Swap (swp)	Swaps a word between registers and memory
Wait For Interrupt (wfi)	`wfi` permits the executing processor to enter the low-power the low-power state until one of a number of asynchronous events occurs.

Table 2.1: Miscellaneous Instructions

might fail. Obviously, the semantics of these instructions depends on the implementation of a processor.

As the load/store instructions are used to communicate with memory locations, the memory address is used to indicate the target accessing of the instruction. Normally, the address is formed from a based register and an offset. The based register can be any ARM core register, while the offset is of either immediate, register, or scaled register. There are three different ways to indicate the memory address: *offset*, *pre-indexed*, and *post-indexed*.

Miscellaneous Instructions

In ARM multiprocessors, there are additional instructions such as barrier instructions and clear exclusive instructions. Table 2.1 shows the summarization of instructions in this category. Note that this table shows some of the instructions that can occur in actual implementation.

2.2.2 Granulity of Assembly Instruction

An assembly instruction is deemed to be completed atomically, however, the actual computation might not be complete yet. For instance, the write accesses of a write instruction are added to write buffers and the next instruction can be fetched instantly even if the write is not completed. In addition, a single assembly instruction can produce multiple micro-operations to the hardware, such as multiple writes. Thus, this research considers

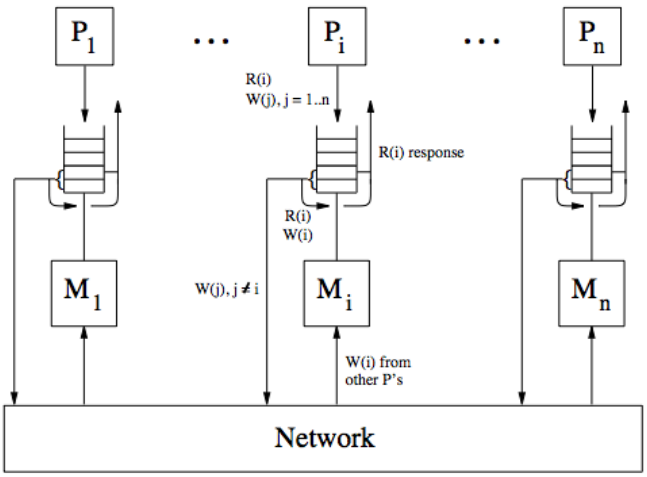


Figure 2-8: General model for shared-memory [Gha95]

an **operation** as a granule of an assembly instruction and not consider the instruction is completed once the instruction is performed. In particular, the side effects of operations performed on a relaxed memory model are considered in program verification.

Given an assembly instruction, the corresponding operations must cover the behavior of the concerned instruction. In particular, the behavior must be sufficient for program verification on relaxed memory models. For example, read operation would be necessary to represent the behavior of load instruction. Consequently, each of the corresponding operations is then performed by a processor in a defined order regarding the behavior of the instruction.

2.3 Modeling Framework

Due to our research focuses on program verification for various memory models, proposing a semantics to each memory model is seemed to be not efficient for the program verification. In particular, the verifying property that we concern focuses on the return values of read accesses which could be produced differently on each memory model. Besides, the way to consider the value relies on the behavior of a memory model, which is described differently for each processor and there is no standard description.

Fortunately, currently, there are frameworks to model the behavior of memory accesses occurring on a relaxed memory model, called *modeling framework*. Most of the frameworks provide abstractions of the memory accesses and axioms to consider the valid executions or data flow that permitted on a memory model. Thus, our research would like to adopt these modeling frameworks for realizing the values of read accesses for our program verification.

2.3.1 Gharachorloo Framework

Abstract Model

Figure 2-8 shows the abstract model proposed by [Gha95] representing a multiprocessor system using shared memory. The model consists of n processing units $P_1, \ldots, P_n$, in which the operations are performed in each processor would issue memory accesses to the *buffer* of a processor. Each processor node also has its own memory, M_i, which belong to processor P_i where $1 \leq i \leq n$. Memory M_i is a complete copy of a shared memory in the system. Each node is connected by the network to distribute the write accesses across the nodes to update the memory location.

Read and Write Accesses

For memory accesses issued by processor P_i, each access is supposed to consist of granule operations, called *sub-operations*. Read access r issued by processor P_I consists of a sub-operation, denoted by $r(i)$, while write access w issued by processor P_i is supposed to consist of n sub-operations $w(1), \ldots, w(n)$, where n is the number of processors.

Once processor P_i issues a write access, the corresponding sub-operations are added into the buffer of the processor. For a read access, its sub-operation is also stored in the buffer of the processor. Note that the buffer is a Non-FIFO buffer, which performs a sub-operation regardless the issued order by the processor. If read sub-operation $r(i)$ is performed, the return value is come from either memory M_i or the last conflicting write sub-operation $w(i)$ issued before the read access r. If write access $w(j)$ is performed, it updates the value of memory location in M_j appearing in the system.

To control the behavior to perform memory accesses, the conditions are given by specification regarding a memory model. The conditions are provided on the performed instructions and the issuing order on them. Then, the behavior is represented in the term of *execution order* $\xrightarrow{xo}$ on sub-operations.

Execution Order

An execution order $\xrightarrow{xo}$ is a total order on sub-operations performed by the buffers. This order relies on the program order $\xrightarrow{po}$ of the issued operations and the conflicting information among the accesses. An execution order $\xrightarrow{xo}$ is considered as a *valid execution* if the conditions of a specification regarding the target memory model are satisfied. The conditions of a specification in the framework consists of *underlying condition* and conditions for a specific memory model.

Termination of writes Every write access issued by a processor must eventually complete. Intuitively, the write value of a write access must eventually appear to every processor. The condition 1 shows the definition of termination condition for write accesses. This ensures corresponding write sub-operations appear in the execution order $\xrightarrow{xo}$.

Condition 1 (Termination Condition for write operations). *Suppose write access W issued by P_i, the termination condition requires the n corresponding sub-operations $W(1)$, ..., $W(n)$ appear in the execution order $\xrightarrow{xo}$.*

Return value of reads A read access to a memory location is abstracted as a corresponding read sub-operation appeared in the buffer. Once the sub-operation is performed to execution order, the read value of a memory location is considered from the performed conflicting write sub-operations in execution order $\xrightarrow{xo}$. Note that two memory operations *conflict* if one of them is a write operation and access to the same memory location. Condition 2 shows the definition of the return value of read sub-operations.

Condition 2 (Return value for read sub-operations). *A read sub-operation $R(i)$ performed by P_i returns a value that satisfies the following conditions: (1) If there is a conflicting write access W issued by P_i such that $W \xrightarrow{po} R$ and $R(i) \xrightarrow{xo} W(i)$, then $R(i)$ return the last value of the last such W in $\xrightarrow{po}$ which issued by the same processor. (2) Otherwise, $R(i)$ returns the writing value of W' (from any processor) such that $W'(i)$ is the last conflicting write sub-operation that is ordered before $R(i)$ by $\xrightarrow{xo}$. (3) If there are no conflicting write access that satisfy either of above two categories, then $R(i)$ returns the initial value of the location.*

Atomicity of read-modify-write Processor architecture usually provides primitive instructions to read and write a shared-memory location atomically. The atomic behavior of a read access and a write access, which access to the same location, is generalized as *read-modify-write*, denoted as RMW. Condition 3 is provided to ensure that no conflicting write operation from another processor interrupts the execution of read-modify-write behavior.

Condition 3 (Atomicity of Read-Modify-Write). *If R and W are the read and write accesses behaves as read-modify-write on P_i, for every conflicting write operation W' from a different processor P_k, either $W'(i) \xrightarrow{xo} R(i)$ and $W'(i) \xrightarrow{xo} W(i)$ for all i or $R(i) \xrightarrow{xo} W'(i)$ and $W(i) \xrightarrow{xo} W'(i)$ for all i.*

Specification for a memory model To provide a specification for a memory model, the following notations are often used in a specification: (a) R and W denote any read and write accesses, respectively, (b) RW denotes either a read access or a write access, (c) $X(i)$ and $Y(i)$ denote the sub-operations of memory accesses X and Y appearing to processor P_i, and (d) X(in RMW) means memory access X, either read or write, that behaves as read-modify-write. Besides, there are other underlying orders $\xrightarrow{co}$ and $\xrightarrow{co'}$ stand for conflicting order and inter-conflicting order, respectively. Conflicting order $X \xrightarrow{co} Y$ is defined if X and Y are conflict and there is an execution order such that $X \xrightarrow{xo} Y$. For inter-conflicting order $X \xrightarrow{co'} Y$ is defined if $X \xrightarrow{co} Y$ and accesses X and Y are issued by different processors.

Moreover, a specification might define its own orders, such as significant program order $\xrightarrow{spo}$, and operations, such as `MEMBAR(WR)` for fence instruction in SPARC processors. In

define $\xrightarrow{spo}$: $X \xrightarrow{spo} Y$ if X and Y are to *different* locations and $X \xrightarrow{po} Y$
define $\xrightarrow{sco}$: $X \xrightarrow{sco} Y$ if X and Y are the first and last operations on one of:
$X \xrightarrow{co'} Y$
$R \xrightarrow{co'} W \xrightarrow{co'} R$
Conditions on $\xrightarrow{xo}$:
(a) the following conditions must be obeyed:
Condition 1: termination condition for writes.
Condition 2: return value for read sub-operations.
Condition 3: atomicity of read-modify-write instructions.
(b) given memory operation X and Y, if X and Y conflict and X, Y are the first and last operations in one of:
uniprocessor dependence: $RW \xrightarrow{po} W$
coherence: $W \xrightarrow{co'} W$
multiprocessor dependence chain: one of
$W \xrightarrow{co'} R \xrightarrow{po} RW$
$RW \xrightarrow{spo} \{A \xrightarrow{sco} B \xrightarrow{spo}\} + RW$
$W \xrightarrow{sco} R \xrightarrow{spo} \{A \xrightarrow{sco} B \xrightarrow{spo}\} + R$
then $X(i) \xrightarrow{xo} Y(i)$ for all i.

Figure 2-9: Aggressive conditions for SC [Gha95].

---

addition, a condition in the form of $X(i) \xrightarrow{xo} Y(j)$ for all i, j means sub-operation $X(i)$ must appear before sub-operation $Y(i)$ in execution order $\xrightarrow{xo}$ for any processor P_i, P_j. Such conditions are used to determine valid execution orders among memory accesses.

Figures 2-9 and 2-10 show the specifications of sequential consistency model and total store ordering (TSO), specified for SPARC, provided by [Gha95]. In the specifications, they commonly require underlying conditions 1, 2, and 3 to realize the underlying behaviors of shared-memory multiprocessors, while the condition (b) constrains execution orders by expressing relations. For instance, given two conflicting write operations w_1 and w_2 executed in multiprocessor $P = P_1, P_2$, if write sub-operation $w_1(1)$ appears before $w_2(1)$ in the execution order $\xrightarrow{xo}$, the coherence condition, $W \xrightarrow{co} W$, constrains that $w_1(2)$ must appear before $w_2(2)$ in the execution order $\xrightarrow{xo}$. Note that the relation expressed by curly brackets, such as $\left\{A \xrightarrow{sco} B \xrightarrow{spo}\right\}+$, means the relation in the brackets must appear at least once.

Intuitively, sequential consistency model maintains the results by concurrent programs to be the same that programs are executed in a sequential way. In the specification, significant program order $\xrightarrow{spo}$ is defined for any memory accesses must maintain their results, and the side effect of the following accesses would not affect the prior memory accesses. In contrast to sequential consistency model, the significant program order of TSO memory model does not preserve the program order $W \xrightarrow{po} R$ unless there is a fence MEMBAR(WR) between them. With these conditions, a valid execution can be decided by considering the existing memory accesses and the program order among them.

define $\xrightarrow{spo}, \xrightarrow{spo'}, \xrightarrow{spo''}$:

$X \xrightarrow{spo''} Y$ if X and Y are the first and last operations in one of: $R \xrightarrow{po} RW$, $W \xrightarrow{po} W$, $W \xrightarrow{po}$ MEMBAR(WR) $\xrightarrow{po} R$.

$X \xrightarrow{spo'} Y$ if X and Y are the first and last operations in: W(in RMW) $\xrightarrow{po} R$

$X \xrightarrow{spo} Y$ if $X \left\{ \xrightarrow{spo'} \middle| \xrightarrow{spo''} \right\} + Y$

define $\xrightarrow{sco}$: $X \xrightarrow{sco} Y$ if X and Y are the first and last operations on one of:

$X \xrightarrow{co'} Y$,

$R \xrightarrow{co'} W \xrightarrow{co'} R$

Conditions on $\xrightarrow{xo}$:

(a) the following conditions must be obeyed:

Condition 1: termination condition for writes.

Condition 2: return value for read sub-operations.

Condition 3: atomicity of read-modify-write instructions.

(b) given memory operation X and Y, if X and Y conflict and X, Y are the first and last operations in one of:

uniprocessor dependence: $RW \xrightarrow{po} W$

coherence: $W \xrightarrow{co} W$

multiprocessor dependence chain: one of

$W \xrightarrow{co'} R \xrightarrow{po} RW$

$RW \xrightarrow{spo} \{A \xrightarrow{sco} B \xrightarrow{spo}\} + RW$

$W \xrightarrow{sco} R \xrightarrow{spo} \{A \xrightarrow{sco} B \xrightarrow{spo}\} + R$

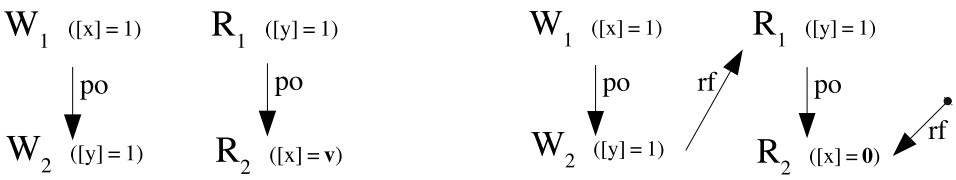
then $X(i) \xrightarrow{xo} Y(i)$ for all i.

Figure 2-10: Aggressive conditions for TSO+[Gha95].

2.3.2 Herding Cats Framework

In this framework [AMT14], memory accesses are considered as events. In addition, the instances of operations which are not memory accesses, such as fence operations, are also considered as events to capture the existing events in the system. The modeling framework uses a *control flow* of concurrent program to represent the issued events occurring in the system and the program order among them. Nevertheless, a control flow does not realize the evaluated values of each read event, yet. Then, a *data flow* is a valuation of the control flow, in which a data flow represents how the data is transferred to each read event. This would represent the communication of data between memory events.

Control flow A control flow of concurrent programs is a symbolic representation of the programs to be performed in the system. A control flow could consist of events, which can be a representation of memory access, fence instruction, or branch decision. Note that the return values and/or evaluations on events are represented by symbolic values, in which a valuation of those symbolic values is considered as a *data flow*.



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

Figure 2-11: An abstraction of message passing

Atomic: empty($\xrightarrow{atom} \cap \xrightarrow{fre;coe}$)
Sequential Consistency: acyclic($\xrightarrow{po} \cup \xrightarrow{rf} \cup \xrightarrow{fr} \cup \xrightarrow{co}$)

Figure 2-12: SC constraints for Herding cats framework

Data flow A data flow of a control flow can be represented by adding a read-from relation $\xrightarrow{rf}$ to the control flow. Read-from relation $\xrightarrow{rf}$ is a relation between write and read events, in which the value of the write event is transferred to the read event. For example, Figure 2-11(a) shows a control flow of the message passing programs, in which $\xrightarrow{po}$ represents the program order among events, and Figure 2-11(b) is a data flow of the control flow, in which the value v of read event R_2 is evaluated to be 0. Note that the read-from relation that has no source, that means the read is got a value form initial value, which is usually 0. Note that the data flow is an anomalous result that could not occur in sequential consistency model.

Constraint specification Given a data flow, such as Figure 2-11, the constraint specification is used to decide whether it is valid on the memory model or not. Constraints are usually either *acyclic*, *irreflexive*, or *empty* on a relation, in which the relation can be constructed from basic relations.

For instance, Figure 2-12 shows the constraint specification for sequential consistency model. In this specification, there are 2 constraints, which are the atomic constraint and sequential consistency constraint. The constraints consider the relations, in which the relations are constructed based on union and intersection on the basic relations, such as $\xrightarrow{po}$ and $\xrightarrow{rf}$. In the second constraint, the constructed relation $\xrightarrow{po} \cup \xrightarrow{rf} \cup \xrightarrow{fr} \cup \xrightarrow{co}$ represents the communication of a data flow that can be constructed, in which constraint `acyclic` is provided to prevent a loop communication occur. Note that the relation $\xrightarrow{fre;coe}$ represents concatenation of two relations: $\xrightarrow{fre}$ and $\xrightarrow{coe}$.

For the basic relations, relation $\xrightarrow{po}$ and $\xrightarrow{rf}$ are program order and read-from relations, which are explained earlier. Relation $\xrightarrow{co}$ is a coherence relation to define the coherence order of conflicting write operations. In addition, relation $\xrightarrow{fr}$ is a from-read relation, which can be derived from read-from relation $\xrightarrow{rf}$ and coherence relation $\xrightarrow{co}$ to represent the conflicting writes that its value is not taken by the read and following the taken write

```
empty atom & (fre;coe) as atomic
acyclic po | rfe | co | fr as sc
```

Figure 2-13: SC constraints in cat language

```
let po-loc = po & loc
acyclic po-loc | rf | co | fr as scpv
let ppo = po \ (W*R)
acyclic ppo | rfe | co | fr as tso
```

Figure 2-14: TSO constraints in cat language

event in the coherence order.

Moreover, the constraint specification of each memory model can be written in cat language [ACM16], which is provided for Herding Cats framework. In this language, users can freely define relations and define constraints on existing relations. Figures 2-13 and 2-14 show constraint specifications for sequential consistency model (SC) and total store ordering (TSO), respectively.

2.4 Program Verification

Formal verification is an approach to ensure whether the property is satisfied with the system or not. Generally, it is used to proved and/or disproved the given property by using a formal method. *Formal method* refers to mathematically rigorous techniques and tools for the specification, design, and verification of software and hardware systems. The phrase *mathematically rigorous* means that the specifications used in formal methods are well-formed statements in a mathematical logic and the formal verification is rigorously deduced in that logic (i.e. each step follows from a rule of inference and, hence, can be checked by a mathematical process).

There are approaches for formal verification. One approach is *model checking*, in which we formalize a target system as a *mathematical model* and explore all reachable states and transitions of the model. The target property is then used to check against each reachable state. Most of the research could define the target property in a temporal logic to determine liveness property of the system. The great advantage of this technique is the process is done automatically to explore the states of the model. However, it could be suffered by the state-explosion problem during the exploring process. The state-explosion problem usually occurs in model checking of a complex system. To deal with the state-explosion problem, there are various works to reduce the states to be explored or provide a bound for exploring.

Another one is *deductive verification*, which constructs the collection of mathematical proof obligations, the truth of which imply conformance of the system to its specification. Then, proofs are provided for those proof obligations to verify the correctness. Generally,

we usually use either *interactive theorem prover*, automatic theorem provers, or satisfiability modulo theories (SMT) solvers. The disadvantage of this technique is that it requires users to understand the system in detail, and how to convey the information for program verification on the target system.

To ensure the program correctness, the program property must be defined for ensuring the programs executed on relaxed memory models must always satisfy the given conditions. Thus, Section 2.4.1 shows the motivation to define the property for specific concurrent programs executed on relaxed memory models. Then, Section 2.4.2 shows the SMT-based program verification approach that is used in our research.

2.4.1 Program Property

The program property can be categorized as safety property and liveness property. Liveness property required 'something good should occur', while safety property requires 'something bad must not occur'. In this research, safety property is target property to be ensured as the anomalous result must not occur in any program execution on relaxed memory models. In particular, the value of memory locations observed by a processor must be in the scope that we expected. For example, the following assembly program uses load instructions to read the memory locations `[X]` and `[Y]` in the order to register `r1`.

```
1 ldr r1, [X]
2 ldr r1, [Y]
```

Let the program property requires the read value of memory location `[X]` must greater than 10, while the read value of memory location `[Y]` must less than 10. To ensure the program execution satisfying the property, the assertion would be injected into the program as the following.

```
1 ldr r1, [X]
2 assert(r1 > 10)
3 ldr r1, [Y]
4 assert(r1 < 10)
```

Then, the assertion conditions can be used to define the program property to ensure the program correctness.

In usual concurrent programs, the programs are executed in an interleaving way, and each statement is expected to be completed immediately. This means the effect of the completed statement can be observed by any processor immediately. Thus, if an assertion condition is not satisfied, the program is expected to be terminated immediately. However, the effect of performed statement on systems using relaxed memory models would not be seen by any processor immediately. This means it would be difficult to determine the interleaving step and considering the terminating state. Thus, the assertion conditions injected in the programs would be used as program invariant to ensure the conditions must always be satisfied for any program execution no matter which completion order. In addition, assumption conditions are also expected to be injected in the programs using

assume statements, such as **assume**($v > 5$). For instance, the following programs ensure that the value v must always equal 1 if the value if u is 1.

```
1 [X] := 1                1 [Y] := 1
2 u := [Y]                2 v := [X]
3 assume(u = 1)           3 assert(v = 1)
```

In this research, the assertion conditions and assumption conditions are then used to define the desired program property for ensuring the program correctness.

2.4.2 Satisfiability Modulo Theories (SMT)

In program verification using satisfiability modulo theories, the verification property of the program P is supposed to be satisfied by system S, written as $S \models P$. This means there is a valuation in a system S that satisfy P, in which the interpretation of system S must be provided in a formal way.

Basically, the problem in satisfiability modulo theories (SMT) is a decidable problem [Wik18], in which the problem is usually expressed by a first-order formula regarding the background theories to deduce the formula, such as the theory of real number and theory of arrays. For instance, if there is a problem expressed by $x^2 = y \wedge y > 1$, the background theorems help to show there is a valuation(or interpretation) of the variables to satisfy the problem. However, if there is a contradiction, the background theorems also disprove the problem, in which there is no valuation found.

[AMP06] shows the way to verify a sequential program using SMT solver, which is more efficient than using SAT solver. The work considers a sequential program as a sequential execution, and abstract each statement as an instance to appear in a system, in which the program is transformed into static single assignment form to realize the data flow of the program. Then, the program is encoded into a first-order formula to be deduced by the background theories provided by an SMT solver.

For instance, Figure 2-15(a) shows a sequential program, in which the program property ensures the value of `x` is in the range between 1 and 9, expressed by **assert**($x > 0 \wedge x < 10$). In the approach of [AMP06], the program is then transformed into a static single assignment (SSA) form, shown in Figure 2-15(b), in which the value of variable x is abstracted by x_0, x_1, x_2, x_3, and x_4 to capture the state of variable `x` at each program point after an assignment to `x`. Besides, the transformed program is then normalized as shown in Figure 2-15(c) before the encoding process. Then, the encoding process can directly be translated as the sets of formulas.

$$
\begin{aligned}
C &= \{x_1 = (x_0 > 0 \wedge x_0 < 10)?x_0 + 1 : x_0, \\
&\quad\; x_2 = (x_0 > 0 \wedge \neg(x_0 < 10))?x_0 - 1 : x_1, \\
&\quad\; x_3 = (x_0 > 0)?((x_0 < 10)?x_1 : x_2)) : x_2, \\
&\quad\; x_4 = (x_0 > 0)?x_3 : x_0\} \\
P &= \{x_4 > 0 \wedge x_4 < 10\}
\end{aligned}
$$

where set C represents the formulas of program behavior represented in a first-order logic,

```
1 if( x > 0 ){
2   if(x < 10){
3     x = x + 1;
4   }else{
5     x = x - 1;
6   }
7 }
8 assert(x > 0 ∧ x < 10);
```

(a) A sequential program

```
1 if( x0 > 0 ){
2   if(x0 < 10){
3     x1 = x0 + 1;
4   }else{
5     x2 = x0 - 1;
6   }
7   x3 = (x0 < 10)? x1:x2;
8 }
9 x4 = (x0 > 0)? x3:x0;
10 assert(x4 > 0 ∧ x4 < 10);
```

(b) a static single assignment (SSA)

```
1 if(x0 > 0 ∧ x0 < 10) x1 = x0 + 1;
2 if(x0 > 0 ∧ ¬(x0 <  10)) x2 = x0 - 1;
3 if(x0 > 0) x3 = (x0 < 10)? x1:x2;
4 if(⊤) x4 = (x0 > 0)? x3:x0;
5 if(⊤) assert(x4 > 0 ∧ x4 < 10);
```

(c) a normalized program

Figure 2-15: Transformation of a sequential program for SMT-based program verification

and P represents the formulas of assertion conditions. In particular, the assignment x_3 = (c)?x_1;x_2 is realized as

$$x_3 = \begin{cases} x_1 & \text{If } c \\ x_2 & \text{Otherwise} \end{cases}$$

Then, the formulas are used to check satisfaction whether the property formulas P is satisfied under the program behavior C, written by

$$C \models_T \bigwedge P$$

where T is a set of background theories to support the deduction and $\bigwedge P$ is the conjunction of formulas in set P. However, to adopt SMT solver, a formula is used to find a valuation on free variables in the first-order formula such that

$$\bigwedge (C \cup P)$$

In the approach of [AMP06], the behavior of a sequential program is deterministic and there is only one interpretation of program behavior C. On the other hand, concurrent programs are able to execute statements in an interleaving way. Thus, there might be various evaluations, in which one of them violate the property formula $\bigwedge P$. Thus, the formula $\bigwedge (C \cup P)$ might not be used in some program verification using SMT solver.

2.5 Symbolic Analysis for SMT-based Program Verification

Given a program, an execution of the program is considered based on the computation of the variables used in the conditions. *Symbolic analysis* of program executions is considered to extract the possible control flows from the programs to be considered in further steps. In particular, the computed valued of each variable is considered as a symbolic value which is not interpreted yet. The benefit of symbolic values is the concrete value is not needed to consider the behavior in the program, in which the scope of variables is used to consider the reachable behaviors of the program.

For instance, the following program shows the control flows of the program by using `if` statement.

```
v = A;
if(v > 5){
  u = 0;
}else{
  u = 1;
}
v = u + v;
```

By considering the way to execute programs, there are 2 ways using assumption statements instead of if condition as followings:

```
v = A;
assume(v > 5);
u = 0;
v = u + v;
```

and

```
v = A;
assume(!(v > 5));
u = 1;
v = u + v;
```

To realize the symbolic values, *static single assignment (SSA)* can be the form to model the data flow in the program. Then, the above programs can be in the following forms to describe the data flow on the variables using symbolic values.

```
v^0= A;
assume(v^0 > 5);
u^0 = 0;
v^1 = u^0 + v^0;
```

and

```
1 v^0 = A;
2 assume(!(v^0 > 5));
3 u^0 = 1;
4 v^1 = u^0 + v^0;
```

Consequently, the values flown in the program are captured by SSA form. Note that, in concurrent programs, the data flow on a variable can come from other programs, in which the interleaving behavior must be considered. Thus, the analyzing of SSA form could be more complicated for concurrent programs.

For SMT-based program verification, symbolic executions to be considered should cover all practical executions that can occur in target programs. A direct way is to construct a control flow graph of each program, and then collect a control flow from each of them to be considered as a symbolic execution. In the static symbolic analysis, a loop caused in the control flow graph can provide the infinite number of control flows to be considered. Typically, the symbolic analysis of a loop behavior can provide a bound to restrict the number of control flows for program verification.

2.5.1 Static Single Assignment (SSA)

In general, *static single assignment* is the form of programs that all variables in the programs must be assigned a value at most once. For example, the following program is in SSA form.

```
1 v^0 = 5;
2 if(v^0 == 2){
3   v^1 = v^0 * 2;
4 }else{
5   v^2 = v^0 + 2;
6 }
7 v^3 = (v^0 == 2)?v^1:v^2;
```

Note that assignment "$v^3 = (v^0 = 2)?v^1{:}v^2$" is the condition assignment that assign the value v^1 to v^3 if condition $(v^1 = 2)$ is satisfied, otherwise v^1 is assigned by v^3. For the condition assignment, the input data are v^1 and v^2 to be considered for v^3, in which the consideration is done only one time.

The programs in static single assignment form are applicable for program verification in SMT-based program verification, in which the data flow can be abstracted by the first-order formula directly. For example, the formula for the above example is $(v^0 = 5 \wedge (v^0 = 2 \implies v^1 = v^0 * 2) \wedge (\neg(v^0 = 2) \implies v^2 = v^0 + 2) \wedge (v^0 = 2 \implies v^3 = v^1) \wedge (\neg(v^0 = 2) \implies v^3 = v^2))$. Based on the provided formula, SMT solver can ensure the program property is satisfied or not.

Given a sequential program, the transformation process of the program into an SSA form is not so complicated. However, for concurrent programs, the interleaving behavior must be considered to capture the data flow from other programs. The approach to realizing the SSA form of concurrent programs can be considered by [LMP98]. In contrast, this the global variables of this research is affected by a relaxed memory model, in which

the concurrent static single assignment approach cannot be used in our research. However, this research considers only the SSA form of local variables in the programs, in which the generation mechanism is not complicated as the concurrent SSA.

2.5.2 Control Flow Analysis

Constructing a control flow graph for a program is a direct way to consider all possible symbolic executions for program verification. A graph usually defined as a tuple $\langle N, E, n_0 \rangle$, where N is the set of nodes, E is the set of directed edges and $n_0 \in N$ is the initial node. Basically, each statement in a program is considered as a node appearing in the control flow graph, in which there are directed edges between each sequential statement. For the control flow statement, such as `goto` and `if`, the direction of edges could not be in the sequence or there is a condition on the directed edges. For instance, the following program contains a loop and `goto` statement to exit the loop.

```
1 v = 0;
2 do{
3   if(v == 5)
4     goto L
5   v = v + 1;
6 }while(true);
7 L:
8 u = v;
```

The control flow graph of the above program can be represented by Figure 2-16.

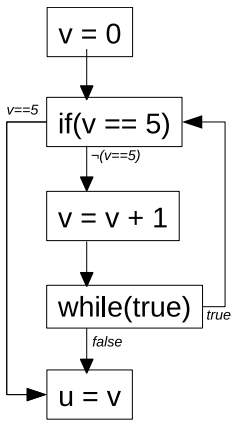
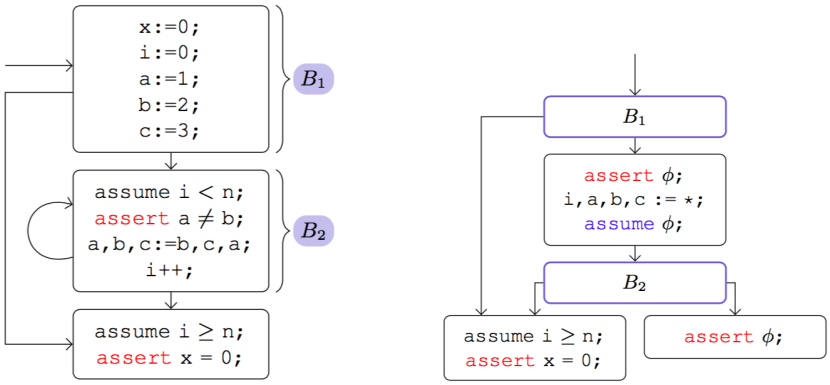


Figure 2-16: An example of control flow graph

To analyze symbolic executions, the control flows from the graph is extracted and replace each control statement as an assumption statement, such as `assume(v == 5)` in the following execution.

```
1 v = 0;
2 assume(v == 5)
3 u = v;
```



(a) Original CFG (b) CFG after loop cutting

Figure 2-17: A program, and CFG obtained using the inductive invariant approach [DHKR11]

To realize the symbolic execution, the control flow must be in SSA form, such as

```
1 v^0 = 0;
2 assume(v^0 == 5)
3 u^0 = v^0;
```

By using this approach, the program described by either structured programming style or unstructured programming style can be considered to extract the corresponding symbolic executions systematically. However, if the control flow graph contains a cycle, the number of control flows becomes infinite. Thus, for the program verification purpose, the bound must be given to restrict the state space to verify the program property on the loop behavior.

2.5.3 Invariant Analysis

Due to the limitation of control flow analysis, the number of loop iterations must be restricted by a bound to generate the finite number of symbolic executions automatically. As the loop behavior can be infinite because the condition is not evaluated yet, we would like to find a way to abstract the behavior of the loop.

The inductive invariant approach is described in [DHKR11], which can be directly used to verify concurrent programs in multiprocessor systems using sequential consistency model. The approach is originally proposed to analyze the program execution symbolically for program verification using SAT/SMT solver. Especially, the behavior of loop is considered in an abstraction regarding a loop invariant.

Figure 2-17 shows the overview of the original approach. First, the control flow graph in Figure 2-17(a) is considered to point out the cut-points of the graph. According the graph, B_1 is a program fragment before entering a loop, B_2 is a loop body, and the loop condition is $i < n$. After loop cutting, shown in Figure 2-17(b), an invariant condition ϕ is used to assume an arbitrary loop iteration before the loop body. After the loop body B_2 in Figure 2-17(b), there is 2 possible control flows: (1) ensuring the invariant condition

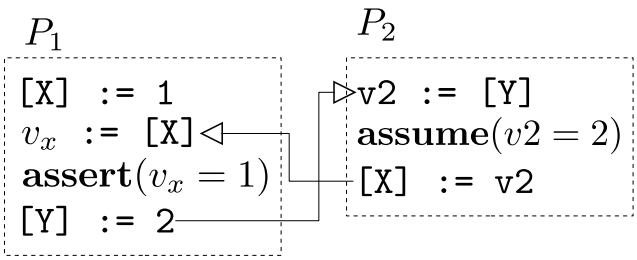


Figure 2-18: A data flow of concurrent programs allowed by POWER

is always preserved for any loop iteration and (2) exiting from an arbitrary loop iteration. This approach is always correct for the program executed on sequential consistency model because the effect of every program execution always follows the program order. However, we suspect the approach could be used for some relaxed memory models.

Figure 2-18 illustrates the data flow in which the non-conflicting writes could affect the assertion, which is permitted by POWER memory model regarding axiomatic semantics in [MHMS+12]. Note that this data flow was tested using herd7 tool[1] by providing a litmus test. The arrows in the figure mean the direction of the value of a write event is flown to a read event. This shows the assertion of $P1$ is violated due to the value of v_x can be 2 which is affected by the following non-conflicting writes.

However, there might be some relaxed memory models that are applicable to adopt the inductive invariant approach. The problem of this approach on relaxed memory models is the read access could be delayed, in which the invariant is not preserved anymore if the following effect of memory access can change the computation of prior operations. Therefore, the relaxed memory models that are applicable for this approach should not allow the effect of following operations can affect the computation of prior operations, in which total store ordering (TSO) and partial store ordering (PSO) models are applicable.

Total Store Ordering (TSO) is the memory model that relaxes some behaviors of memory accesses in which the completing order of write accesses, or stores, is always preserved. Intuitively, TSO allows the write access to be completed after the following read access if the write access cannot be completed, yet, so as to reduce the memory latency of program execution. Although there is no standard specification of the model in general, there are two types of specifications of TSO provided by Gharachorloo and Herding cats frameworks, shown in Figure 2-4 and Figure 2-10, respectively. According to both specification for TSO memory model, the order of a write access following by a read access is allowed to be completed out-of-order regarding the whole system.

Partial Store Ordering (PSO) is an extension of TSO memory model to allow the non-conflicting write accesses to be completed out-of-order. Figure 2-6 shows the specification for PSO memory model provided by Gharachorloo framework, in which the non-conflicting write accesses are allowed to be completed out-of-order, in which the memory latency can be reduced by delayed the completion of a write access after a following non-conflicting write access if the access cannot be completed immediately.

[1]herd7 is available at http://diy.inria.fr

According to the inductive invariant approach, the loop behavior is abstracted by ignoring the remaining memory access outside the loop as the sequential consistency model does not allow the following memory accesses to affect the prior computation. On the other hand, the proposed operation structure considers the computation on local variables and the local variables can be affected by a read access. In particular, as our assertion language is expressed on local variables, the effect of read access must be a concern on relaxed memory models. However, as the effect of TSO and PSO do not allow the read access to be completed by any memory access; thus, the computation always depends on the existing memory accesses in the system. Consequently, the inductive invariant approach is deemed to be sound for TSO and PSO based on our operation structure.

Chapter 3

Bounded Method for SMT-based Program Verification

3.1 Motivation

To ensure the correctness of concurrent programs on relaxed memory models, the requirement is expected to be provided on the program variables. In our research, the requirement is then considered as the safety property to be verified to ensure the undesired values cannot be provided. Besides, as relaxed memory models are considered, the exact value of a shared-memory location could not be determined globally at that same time. Thus, the program property is determined by local variables on each processor.

As the target memory models are at the hardware-level, assembly programs which facilitate the behavior of processors are considered. In particular, as the objective of this method is to ensure the correctness of programs executed on relaxed memory models, we would like to provide an abstraction level of assembly programs that can capture the essential behavior of programs affected by relaxed memory models. Thus, Section 3.2 shows the abstraction of an assembly program, called operation structure. Especially, this abstraction also defined the program property to be verified by itself.

In traditional program verification of concurrent programs, the effect of every memory access is always completed in the order following the program order among processors. Obviously, the effect of computations relies on the prior memory accesses performed in a system. However, the behavior of programs on a relaxed memory model permits the completing order so as not to follow the program order, the computations could be affected by the latter memory accesses if the completion of a memory access is delayed. Our research would like to propose a method that can realize the effect of program executions for program verification on relaxed memory model.

Due to there is no standard description of memory models, one could consider the program semantics on a specific memory model, such as [Rid10, LV15]. However, this research would like to realize the program behavior on a variety of memory models. Currently, there are frameworks, such as [AMT14, Gha95], to model relaxed memory models to realize the effect of program execution on their abstraction. The idea of those

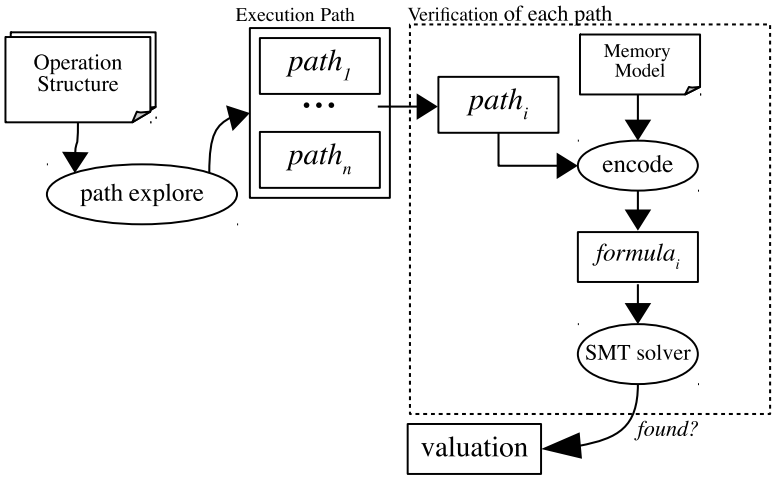


Figure 3-1: Overview of Bounded SMT-based Verification

frameworks is to give the constraints on the program behavior and realize the effect of program execution on the behavior. The constraints are given by a specification of a memory model provided in the standard of a modeling framework. Hence, adopting the memory model specifications would be an appropriate way to realize the effect of program execution for verification.

SMT-based Program Verification is an appropriate approach to realize a valid effect of program executions based on the given constraints. Using an SMT solver in program verification, the behavior to be verified must be represented in a first-order formula, in which the read value is represented by a free-variable to be evaluated with respect to a memory model. Besides, the program property is also abstracted in the formula as the condition must not occur to ensure the correctness. For instance, if the read value of read R_1 is abstracted as variable val_1 and there is an assertion condition $val > 1$, the formula consists of $(val_1 = \text{ReadVal}(R_1)) \wedge (\neg(val > 1))$, where ReadVal is an uninterpreted function to realize the read value of a read access based on a memory model. Using that shape of the formula, if the solver cannot find any valuation, we can ensure there is no valuation that violates the program property $val > 1$.

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 operation and write operation. However, operation structures cannot solely express the program behavior on a relaxed memory model. Thus, the abstraction of program executions, called execution path, is considered.

Figure 3-1 shows the overview of the approach to verify the program property of the given assembly concurrent programs on target memory model. First of all, given a sequence of assembly programs, each program must be translated into a corresponding *operation structure*, defined in Section 3.2, 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 operation and write operation. However, operation structures cannot solely express the program behavior on a relaxed memory model. Thus, the abstraction of program executions, called *execution path*, is considered.

The sequence of operation structures is then explored as a set of *execution paths*, in which the execution path represents a way to perform operation into a system in the program order. To verify the program property, an SMT solver is used to automatically realize the valid effect of program executions under the constraints regarding target memory model. In particular, an execution path is encoded with the constraints representing the specification of target memory model into a first-order formula. Especially, the program property of an execution path is accumulated from the property statements appearing in the execution path. In program verification, let the encoded formula represents the effect that violates the program property, then, the SMT solver must not provide a valuation of the effect that violates the program property. Therefore, to ensure the program correctness, the encoded formula of any corresponding execution path must not have a valuation provided by the SMT solver.

3.2 Abstractions of Assembly Programs

In our research, the abstraction is used to provide the sufficient information of the concurrent programs for program verification on relaxed memory models. As the target of memory models is at the hardware level, the behavior of programs performed at the hardware level is our concern. Besides, due to assembly programs is close to the hardware behavior, the behavior of assembly programs are also considered to provide the abstraction. However, in practice, the assembly programs could provide various behavior to facilitate the processors, while the target of abstraction focuses on the behavior that produces significant behavior at the hardware level for relaxed memory models. Therefore, Section 3.2.1 shows the assumptions on assembly programs for program verification.

Then, our research introduces an *operation structure*, as shown in Section 3.2.2, as a representation of an assembly program. Note that the representation focuses on the operations, which are granules of instructions, that affect the calculation of local variables in a program. Figure 3-2(a) shows a fragment of Spinlock implementation in FMP kernel of TOPPERS project for ARM processors. Figure 3-2(b) then shows the operation structure of the assembly program, in which interruption instruction **wfene** and coprocessor instruction **wfene** are not considered. Although interruption and coprocessor instructions are used to change the behavior of processors, the side effects of those do not affect the calculation of programs. Thus, our operation structure is an abstraction of an assembly program, which focuses on the behavior that is sufficient for program verification on relaxed memory models.

For program verification, the program property of concurrent programs is then defined by assertion conditions and assumption conditions as the *program invariant*. This means the conditions must be satisfied for any program execution on the target memory model. In particular, as the value of shared-memory locations could not be the same as the

```
1 mov r2, #0x01
2 ldreq r1, [lock]
3 cmp r1, #0x00
4 msrne cpsr_c, %2
5 wfene
6 msrne cpsr_c, %3
7 strexeq r1, r2, [lock]
```

(a) assembly program

```
1 instr{ //move r2, #0x01
2   val := 1;
3   r2 := val
4 };
5 instr{ // ldreq r1, [lock]
6   val_z := z;
7   if(val_z = 1){
8     ll(val, [lock]);
9     r1 := val
10  }
11 };
12 instr{ // cmp r1, #0x00
13  (rd := r1 ‖ rt := 0);
14  (val_z := (rd =rt)? 1:0 ‖
15   val_n := (rd =rt)? 0:1) ;
16  (z:=val_z ‖ n := val_n)
17 };
18 instr{  //strexeq r1, r2, [lock]
19  val_z := z;
20  if(val_z = 1){
21    rt := r2;
22    sc(val, [lock], rt);
23    r1 := val
24  }
25 }
```

(b) Operation Structure

Figure 3-2: Representation of an assembly program

same value at the same time on each processor, the local variables are used to define the program property.

Figure 3-3 shows a sequence of operation structures for store buffer (sb) programs in Figure 2-5 and the program property defined by assertion and assumption. In our operation structure, assertion and assumption annotations are allowed to be injected in the structures at appropriate program locations. In this program property, assumption **assume**`(val = 0)` restricts the state space of program verification, in which the read value of location `[Y]` observed by structure γ_1 equals 0. On the other hand, assertion **assert**(`val = 1`) ensures the return value of read access to location `[X]` must be 1 for every valid evaluation of read values.

Nevertheless, a sequence of operation structures cannot be used in SMT-based program verification directly. Thus, an execution of operation structures is defined in Section 3.2.3 in a formal way. Then, the executions of the given sequence can be considered using the semantics of operation structures, shown in Section 3.2.4. The semantics of operation structures considers the executions on operations to be performed in an abstract way

```
1 instr{ // mov r1, #0x01
2   val := 1;
3   r1 := val
4 };
5 instr{ //str r1, [X]
6   val := r1;
7   [X] := val
8 };
9 instr{ // ldr r2, [Y]
10  val := [Y];
11  r2 := val
12 };
13 assume(val = 0)
```

(a) Operation Structure γ_1

```
1 instr{ // mov r1, #0x01
2   val := 1;
3   r1 := val
4 };
5 instr{ //str r1, [Y]
6   val := r1;
7   [Y] := val
8 };
9 instr{ // ldr r2, [X]
10  val := [X];
11  r2 := val
12 };
13 assert(val = 1)
```

(b) Operation Structure γ_2

Figure 3-3: Example of program property

because the computed value on a read access depends on target memory model. However, the semantics can represent a way to performed operations to the system.

3.2.1 Assumptions on Assembly Programs

In practice, there are various assembly instructions to facilitate processors, such as interruption and supervisor call. Besides, assembly programs could use macro instructions and/or assembler directives in the programs to describe the behavior of the programs. However, as the target of this research is *to verify the concurrent assembly programs on relaxed memory models*, the essential behaviors of assembly programs are considered in an abstraction of assembly programs. Thus, this subsection introduces the assumptions on assembly programs to be considered.

Assumption on Computation

For the computation of assembly programs, there are various factors such, as using word or byte, to be considered for calculating. However, for the simplicity of program verification, those values are considered as natural numbers in the abstraction. Besides, the basic operators + and - are used for calculation.

Among the assembly instructions, the data-processing instructions are the concern for realizing the program computation. In practice, there are various computations can be done on registers, such as multiplication and shift instructions. However, as the assumption restricts the operators to be + and - for the computation, the complicated instructions are not considered in the abstraction of assembly programs. For instance, multiplication instruction `MUL` of ARM instruction is not considered in this research. On the other hand, addition instruction `ADD` of ARM instruction is considered in this research.

Besides, as the data type to be used in the abstraction is natural numbers, all data type

in assembly programs is considered as the same type. Thus, the abstraction would be the same, even if there is a variety of assembly instructions for data type. For example, POWER instruction for word data type `lwz` and POWER instruction for by byte data type `lbz` are considered as loading the integer value from the memory location.

Assumption on Assembly Instructions

In this research, the target of program property is *to ensure the computation of concurrent programs executed on relaxed memory models.* According to Section 2.2.1, although there are various assembly instructions provided by processor architectures, the instructions that are affected by relaxed memory models is our concern. For instances, supervision calls are not considered in the abstraction for program verification. In multiprocessors using relaxed memory models, the instructions accessing shared-memory locations are directly affected, such as load and store instructions. In addition, there are also instructions that facilitate the computation of concurrent programs on multiprocessors systems, such as fence instructions.

Load/Store instruction. For program verification, the instructions that access the shared-memory locations are the primary concern in this research. According to Section 2.2.1, *load/store instructions* are the instructions to be considered for the abstraction of assembly programs. For instance, the behavior of `ldr r1, [0x02]` that access the memory location `[0x02]` is taken into account. In addition, the load-exclusive and store-exclusive instructions are considered in this research as *load-link* and *store-condition* instructions, which could be more general for other multiprocessors. For instance, the following program shows the usage of load-link instruction ldrex and store-condition strex. These instructions are supposed to be used as a pair.

```
1 ldrex r1, [X]
2 cmp r1, #0x00
3 moveq r2, #0x04
4 strex r3, r2, [X]
```

The write access issued by strex can succeed if there is no intermediate write access to memory location `[X]` after instruction ldrex. If the write access cannot be completed, the result of `r3` becomes 1, while the result of `r3` equals 0 if the write can succeed. Note that the intermediate write access can be external write access from other programs in which the behavior of load-link and store-condition helps to avoid anomalous results among concurrent programs. Consequently, the program verification can realize the effect of program execution regarding the relaxed memory models.

Swap and **test-and-set** instructions are also considered because those instructions can access the shared-memory locations. As these instructions could be used in the concurrent programs, the behavior of these instructions must be taken into account for program verification. Note that these instructions are considered as read-modify-write in this research.

Read-modify-write instructions are instructions that issue both read and write accesses to the same memory location and require the behavior of both accesses *appears* to be

atomic. A practical instruction for this kind of instructions is swap instruction, which is used for some concurrent programs to prevent interruption between read and write accesses.

Predicated Instruction. In addition, the behavior of condition executions is also considered in the abstraction of assembly programs. In particular, this behavior is considered as predicated instruction in this research. In practice, if the load/store instructions contain the behavior of condition execution, the instruction can access the target memory location if the condition is satisfied during program execution. For instance, `ldreq r1, [X]` is an assembly instruction for ARM processors that has a condition, denoted by `eq`, flag register `z` must equal 1 to perform instruction `ldr r1, [X]`. Note that the condition usually relies on local registers, such as flag register, to be evaluated as 0 or 1. This behavior is attached to almost instructions in recent processors. Thus, this behavior must be considered as the decision to access the memory locations could affect the computation of the programs executed on relaxed memory models.

Synchronizing Instruction. Besides, as the behavior of multiprocessors is needed for program verification, the instructions for synchronization are the significant instructions to be considered. In the implementation of concurrent programs, there are various concerns, such as data race and race condition, to be considered to prevent anomalous behaviors occurring in program executions. In particular, these instructions are used to maintain the result of program execution as not to be affected by relaxed memory models. For instance, memory barrier or fence instructions can prevent the side effects of prior instructions to be taken or their memory accesses must be completed before the following instructions. In some processors, there might be various fence instructions to prevent anomalous on the specific type of behaviors, such as `dmb` and `dsb` in ARM processors. Obviously, the behavior of a fence is provided based on specific memory models and specific processor architectures.

Branch Instruction is a standard instruction for assembly instructions for any processor. The behavior of a branch is to change the program counter of a processing unit to fetch the instruction at the target branch if its condition is satisfied. Normally, the target of the branch is abstracted as a label in the assembly program, which represents the specific line number of programs. The branch is used with predicated instruction or no condition to go to the target location. For instance, instructions `bne CS` and `b L` are branch instructions to jump onto the label `CS` and `L`, respectively, if the condition is satisfied.

```
1 L:
2 sub r1, r1, 1
3 cmp r1, #0x10
4 beq CS
5 mov r2, r1
6 add r1, r2, 2
7 b L
8 CS:
```

Normally, the condition of each processor architecture relies on a flag register, such as `z`

and `n`. In the above program, the condition is extracted by decoded instruction, such as `beq` has condition `(z = 1)`, while `b` has no condition which is always satisfied.

Branch to subroutine is not considered in this research yet. As the primary concern is the control flow of the program, only the basic control flows in considered for simplicity of program verification. Besides, due to the behavior of a branch, an assembly program is considered as unstructured programming, in which a spaghetti code can occur during implementation.

Assumption on Memory Locations

In our program verification, as the target to be verified is the property of concurrent programs, the abstraction to be considered assumes the shared-memory locations to be used must be known beforehand. Intuitively, the scope of memory locations is already defined during implementation. Thus, the assumption that memory locations to be considered in program verification must be provided explicitly.

For instance, given the following programs, the addressing mode of each load/store instructions indicates the memory locations to access directly.

```
1 ldr r1, [0x01]
2 str r2, [0x02]
```

```
1 ldr r3, [0x02]
2 str r4, [0x01]
```

In this example, ones explicitly know the memory locations `[0x01]` and `[0x02]` are used to communicate between concurrent programs. On the other hand, the following example uses register `r1` to indicate memory location `[0x01]`, which can be known by program analysis.

```
1 mov r1, 0x01
2 ldr r3, [r1]
```

```
1 mov r1, 0x01
2 str r4, [r1]
```

This means the scope of memory locations to be accessed is restricted finitely. Thus, these behaviors are considered in this research.

Memory Allocation should not occur. Due to the assumption that the memory locations must be known beforehand, the concurrent programs to be considered must not access to a fresh memory location. This means the memory locations must be known before providing the abstraction of the concurrent assembly programs.

Limitation of the assumptions

In programs' point-of-view, there is missing information that cannot be represented by the abstraction. According to send event (`sev`) and wait for an event (`wfe`), these instructions would cause the starvation problem for program verification. Intuitively, the behavior of interruptions is not taken into account for program verification. For example, the following program uses to wait for event (`wfe`) instruction to enter the low-power state and wait for an event. If there is a possibility to wait forever, the starvation property must be a concern for this program.

```
L:
ldr r1, [0x04]
cmp r1, #1
wfeeq
beq L
; critical section
sev
```

Among load/store instructions, pop and push instructions are also included in this category. However, the abstraction does not intent to realize the stack behavior during the program execution. Using stack behavior, the memory allocation could occur, which is not supposed to occur in the abstraction. For example, the following assembly program uses push instruction to store the values of `r1` and `r2` to the memory locations indicated by stack pointer (`sp`). During the pushing process, a fresh memory location is used in the program.

```
; Save sp before push.
mov r0, sp
; Push.
mov r1, #1
mov r2, #2
push {r1, r2}
```

Thus, the program containing the pop and push instructions cannot be represented by the proposed abstraction.

However, in our program verification approach, the SMT-based program verification approach is supposed to be used. Hence, a verification condition of the target programs must be decidable. By the assumptions: (1) memory locations are known beforehand, and (2) the operators + and - are used for computations, the verification condition becomes decidable because the scope of instances to be realized by SMT solvers is finite. Although the abstraction to be proposed does not represent the whole behavior of practical assembly programs, the essential behaviors to be considered on relaxed memory models are expected to be captured in the abstraction. Note that the discussion of the advantages and disadvantages of the proposed abstractions are shown in Section 5.3.4.

3.2.2 Operation Structure

At the hardware-level, an assembly instruction is not always executed atomically. Besides, one instruction can do more than one actions, such as multiple write accesses. Thus, an instruction is assumed to be a collection of *operations* to be executed in the hardware. Note that an operation might not be executed atomically in the hardware. For example, once an operation is performed by a processor, it might appear to each processor in a different time as a micro-operation.

In an assembly language, instructions can access *register*, *memory locations* and *temporal registers* using for calculation. For temporal register, this research assumes the data flow of a processor should have an intermediate storage to temporally keep the value for

the further process in the behavior of an instruction. For example, instruction `ldr r1, [X]` reads the value of location `[X]` and store in a temporal register before writing back to register `r1`. In this research, the target accesses are considered as *variables* to be used in our abstraction, which are shown in Definition 3.1. Note that, in practice, the components to be accessed are different on each processor. For simplicity, the sets of target accesses are defined similarly to the followings for the explanation in the thesis.

$$\begin{aligned} \mathrm{Reg} &= \{\mathrm{r0}, \mathrm{r1}, \ldots, \mathrm{r13}\} \cup \{\mathrm{z}, \mathrm{n}, \mathrm{v}, \mathrm{c}\} \\ \mathrm{Loc} &= \{[\mathrm{A}], [\mathrm{B}], [x], [y]\} \\ \mathrm{Tmp} &= \{\mathrm{val}, \mathrm{v}_1, \mathrm{v}_2, \mathrm{result}\} \end{aligned}$$

For memory locations, in an assembly program, a memory address could be calculated during the program execution to determine the memory location. However, this research assumes that the shared-memory locations to be accessed should be known beforehand. Besides, the behavior of program executed on relaxed memory models can be determined on the known memory locations using the memory model specification. Therefore, the memory locations are defined in set Loc as symbolic values, in which each value is different from each other.

Definition 3.1 (Variable)**.** A *variable* is either *register*, *memory location* and *temporal register*. Let V be the set of variables, Reg be the set of registers, Loc be the set of memory location and Tmp be the set temporal registers, such that $\mathcal{V} = \mathrm{Reg} \cup \mathrm{Loc} \cup \mathrm{Tmp}$ and $\mathrm{Reg} \cap \mathrm{Loc} \cap \mathrm{Tmp} = \emptyset$.

To define computation of programs, *expression* and *boolean expression* is defined on *temporal registers*. The main reasons not to consider on memory locations and registers are: (1) the consistency of memory locations is not the same for any memory model, and (2) we would like to separate the behavior of reading/writing to a register and computation on a value even if the computation is seemed to be done immediately in a processor unit. Consequently, the sets of expressions and boolean expressions are shown in Definition 3.2 and Definition 3.3, respectively.

Definition 3.2 (Expression)**.** An expression $e \in \mathrm{Exp}$ over temporal register Tmp is defined on operators $+$ and $-$ for verification purpose. Set Exp is the smallest set X with the properties:

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

where $+$ and $-$ are connectives, $(,), :$, and $?$ are auxiliary symbols.

Definition 3.3 (Boolean Expression)**.** Set 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 $(\varphi \wedge \psi), (\varphi \vee \psi) \in X$,

4. $\varphi \in X$ implies $\neg(\varphi) \in X$

where $=, <, >, \wedge, \vee$, and $\neg$ are connectives, (and) are auxiliary symbols.

The usage of expressions is to compute the arithmetic calculation of assembly instructions. For instance, ARM instruction `add r2, r1, 2` computes the value of register `r1` by adding 2 and then save the result into register `r2`. To abstract the behavior, the value of register `r1` is loaded into a temporal register, and the computation is done on the temporal register by adding 2; After the computation, the value of the temporal register is saved into register `r2`. Note that, according to Section 2.1.2, POWER and ARM multiprocessors requires dependencies on the memory accesses to consider the behavior of program execution. As temporal registers are introduced and the arithmetic calculation is done on the temporal registers, the dependency on POWER and ARM multiprocessors can be determined on the read and write operations directly.

In a system, *operations* are supposed to be granules of assembly instructions to be performed implicitly. Especially, operations representing memory accessing are our concern, which is affected by relaxed memory models. Definition 3.5 shows the types of an operation to be used in an operation structure. For simplicity, the representation of read operation, write operation and arithmetic operation is considered as an *assignment*, shown in Definition 3.4. Note that there is a restriction on an assignment that either v or e must be a temporal register, which allows us to distinguish the type of an operation as either read operation, write operation or arithmetic operation.

Definition 3.4 (Assignment)**.** An assignment is of the form $v := e$, where $v \in \mathcal{V}, e \in \text{Exp} \cup \mathcal{V}$ such that $v \in \text{Tmp} \vee e \in \text{Tmp}$.

Definition 3.5 (Operation)**.** An *operation* $op \in \text{Opr}$ is a granule of assembly instructions, which is either:

- Read operation, which is an assignment $v := e$, where $e \notin \text{Tmp}$,
- Write operation, which is an assignment $v := e$, where $v \notin \text{Tmp}$,
- Arithmetic operation, which is an assignment $v := e$, where $v \in \text{Tmp}$ and $e \in \text{Exp}$,
- Branch operation $\mathbf{branch}(c, l)$, where $c \in \text{BExp}$ and l is a label annotation,
- Fence operation $f \in \text{Fence}$,
- Load-Link operation $\mathbf{ll}(v, loc)$, where $v \in \text{Tmp}$ and $loc \in \text{Loc}$, or
- Store-Condition operation $\mathbf{sc}(v_1, loc, v_2)$, where $v_1, v_2 \in \text{Tmp}$, $loc \in \text{Loc}$.

where Fence is the set of fence operations.

Branch operation $\mathbf{branch}(c, l)$ represents the behavior of a branch instruction that condition $c \in \text{BExp}$ must be satisfied to jump onto program location at label l, shown in Definition 3.6. This is a decision branch for program flow to control the execution of operations based on the evaluation of condition c.

Definition 3.6 (Label)**.** A label annotation is of the form $\mathbf{label}(l)$ where $l \in \text{Lid}$ is a label identifier.

Fence operation $f \in \text{Fence}$ corresponds to a fence instruction, or memory barrier, appearing in an instruction set architecture. Set Fence could be different from each processor, such as $\text{Fence}_{\text{ARM}} = \{\mathbf{dmb}, \mathbf{dsb}\}$. Especially, the semantics of each fence operation is also different from each other. However, our operation structure only focuses on the way to perform operations in a system, excluding the concrete semantics of each operation.

Load-link $\mathbf{ll}(v, loc)$ and store-condition $\mathbf{sc}(v_1, loc, v_2)$ 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 at hardware-level, such as C++ and Power [SMO+12], in which the semantics of these varies on a practical processor. Generally, load-link $\mathbf{ll}(v, loc)$ produces a read access to location loc; store-condition $\mathbf{sc}(v_1, loc, v_2)$ writes the value of v_2 to location loc and assigns a flag to v_1 if the write access fails.

In relaxed memory models, the operations proposed in Definition 3.5 would be sufficient to realize the effect of program execution. For example, a write operation issues a corresponding write access, and a read operation issues a corresponding read access. These issued accesses are necessary to determine the effect of relaxed memory models. Besides, store-condition operation and load-link operation are motivated by the synchronizing instructions proposed in ARM and POWER. These operations provide a read access and a write access with conditions. Then, the operations also proposed to cover the behavior of basic assembly instructions for the purpose of program verification.

In the instruction semantics of a processor architecture, the corresponding operations of an instruction are supposed to be performed in a partial order. In addition, there might be instructions that restrict the behavior of the operations, e.g., predicated instruction and read-modify-write instruction. Thus, an *execution structure* shown in 3.7 is used to capture the way to perform the corresponding operations.

Definition 3.7 (Execution Structure)**.** Given operation op, Boolean expression c and execution structures γ_1, γ_2, an *execution structure* is either an *operation*,

- $\mathbf{nil}$,
- Sequential Execution $\gamma_1; \gamma_2$,
- Parallel Execution $\gamma_1 \parallel \gamma_2$,
- Atomic Execution $\mathbf{atom}(op)$, where op is either read operation or write operation, or

- Condition Execution **if**$(c)\{\gamma_1\}$.

An **execution structure** is defined based on operations regarding operators and auxiliary symbols to indicate the way to perform operations and the relations on operations, such as atomic requirement. First of all, **nil** is a basic structure indicating there is no operation to be performed. For sequential execution and parallel execution, these are used to indicate the partial order of operations to be performed. As we suspect operations of an instruction are not needed to be performed in a sequential way, this means some operations are allowed to be performed simultaneously. Hence, the parallel execution is introduced for our operation structure.

As for condition execution **if**$(c)\{\gamma_1\}$, the behavior of a predicated instruction is then captured by this execution. The motivation of this execution comes from the behavior of predicated instruction which performed its own behavior once the condition is satisfied. For example, load instruction `ldreq r1, [X]` issues a read access if the flag register `z` equals 1.

For atomic execution **atom**(op), wrapper **atom** is used to indicate which operations must appear as a read-modify-write manner. The motivation of this notation comes from the read-modify-write instructions that require its read and write accesses to appear atomically, such as swap instruction. Instead of using a direct operation to represent such behavior, the wrapper is used to indicate a pair of read and write operations to allows the calculation before the write access is issued.

In practical, there might be compare-and-swap instruction that is an atomic instruction to load a memory location to be compared before exchanging the values between register and the memory location. By the definitions of condition execution and atomic execution, we would adapt the definition to represent the behavior as the following.

```
atom(val_l := [L]);
if( val_l = 0){
    val_r := r1;
    r1 := val_l;
    atom([L] := val_r)
}
```

where the operations `val_l := [L]` and `[L] := val_r` are required to appear atomically if the operation `[L] := val_r` is performed.

Given an assembly instruction, the corresponding operations to be performed is defined in γ, 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, instruction `cmp r1, r2` that could read the values of `r1` and `r2` concurrently can be represented by *parallel execution* (`v1 := r1` $\|$ `v2 := r2`), where `v1` and `v2` are temporal registers such that `v1`, `v2` $\in$ Tmp. In addition, to define the scope of those operations that are performed by the same assembly instruction, execution structure γ must be the element of instruction execution **instr**$\{\gamma\}$.

```
1 instr{
2   val_z = z;
3   if(val_z = 1){
4     val := [X];
5     r1 := val
6   }
7 }
```

ldreq r1, [x]

```
1 instr{
2   atom(val := [x]);
3   r2 := val;
4   atom([x] := 1)
5 }
```

ldstub [x], r2

Figure 3-4: Examples of corresponding execution structures for instructions

Definition 3.8 (Instruction Execution)**.** Given an execution structure γ, an *instruction execution* is of the form **instr**$\{\gamma\}$.

Figure 3-4 illustrates the instruction executions of a predicated instruction and an atomic instruction. The behavior of a predicated instruction, in which its execution occurs if the condition holds, can be represented by *condition execution*. For instance, `ldreq r1, [x]` can be represented as the left execution structure in Figure 3-4. To illustrate more behavior, let's consider read-modify-write instruction `ldstub [x], r2` that reads the value of memory location `[x]` into register `r2` and then writes 1 to memory location `[x]` atomically. The instruction can be represented as the right execution structure in Figure 3-4. Atomic wrapper **atom** is used to indicate read and write operations that are required to appear atomically.

Definition 3.9 (Property Statement)**.** Let $c \in$ Bexp be a Boolean expression, a *property statement* is either **assume**(c) or **assert**(c).

For the purpose of program verification, the requirement of programs is described in the programs using assertion and assumption statements. Note that the property statements is to define the invariant condition of programs. Assumption **assume**(c) restricts the scope of behavior in which the effect of the behavior always satisfies condition c. If there is no behavior that satisfies the condition, this means there is no behavior to be considered for program verification. Note that, by using assumptions, users could be able to analysis the cases to be verified.

For assertion **assert**(c), the effect of program behavior at the specific location is required to satisfies the assertion condition c. In contrast to usual assertion statements, the assertion conditions in the programs must be used together with assumption conditions to determine the *invariant condition* of the programs. In relaxed memory models, the interleaving step of program execution is not the matter for program verification, while the invariant condition to always be satisfied is the concern. This means the termination during the program execution does not the matter for program verification.

Figure 3-5 shows the difference between assertion and assumption. Both of them read memory location `[X]`, in which the left-side requires the return value that stored in temporal register `val` must equal 0 for any effect of program behavior, while the right-side

```
1 instr{
2   val := [X]
3   r1 := val
4 };
5 assert(val = 0)
```

```
1 instr{
2   val := [X]
3   r1 := val
4 };
5 assume(val = 0)
```

Figure 3-5: The difference between two statements

```
1 instr{ [X] := 1 };
2 instr{ val_y := [Y] };
3 assume(val_y = 1)
```

```
1 instr{ [Y] := 1 };
2 instr{ val_x := [X] };
3 assert(val_x = 1)
```

Figure 3-6: Example of Program Property

just restricts the scope of program behavior such that the behavior that causes `val` equals 0 is of interest for program verification.

In addition, Figure 3-6 shows the way to define the program property using property statements. Assumption condition `val_y = 1` is used to restrict the execution to be considered must have return value `val_y` equals 1. For program verification, the assumption condition is given to ensure that the executions satisfying the assumption condition must satisfy assertion condition `val_x = 1`. Thus, the program property does not consider the interleaving step of program execution, however, the effect of program executions is considered regarding the program invariant defined by the property statements.

Definition 3.10 and Definition 3.11 represent the way to define operation structures for representing a sequence of assembly programs. Figure 3-7 shows the corresponding operation structures of the message passing programs in Figure 1-2. Note that an assignment such as `val_z := (rd = rt)?1:0` is a condition assignment in which 1 is assigned to `val_z` if `(rd = rt)` is satisfied, and 0 is assigned for otherwise. Each instruction is represented by instruction execution **instr**$\{\ldots\}$ to represent the way to execute with respect to its processor.

Definition 3.10 (Operation Structure)**.** An operation structure is a sequence of instruction execution, property statement, and/or label.

Definition 3.11 (A sequence of operation structure)**.** Let $P_1, P_2, \ldots, P_n$ be n operation structures corresponding to n programs to be verified. The sequence of operation structures is of the form $P_1 \cdot P_2 \cdot \ldots \cdot P_n$.

By given n concurrent programs, these programs are expected to be performed concurrently on multiprocessors, in which n operation structures are used to represent these programs for program verification. Not that this means n operation structures can be considered on *at most* n multiprocessors for program verification. However, if n concurrent programs are executed on a single processor by context switching, the programs are not suffered by relaxed memory models.

```
1 instr{
2   val := 1;
3   r1 := val
4 };
5 instr{
6   val := r1
7   [x] := val
8 };
9 instr{
10  val := r1
11  [y] := val
12 }
```

Operation structure γ_1

```
1 label(L);
2 instr{
3   val := [y];
4   r1 := val
5 };
6 instr{
7   (rd := 1‖rt := r2);
8   val_z := (rd = rt)?1:0;
9   z := val_z;
10  val_n := (rd = rt)?0:1;
11  n := val_n
12 };
13 instr{
14  val_n := n;
15  branch(val_n = 1,label(l))
16 };
17 instr{
18  val := [x];
19  r1 := val
20 };
21 assert(val = 1)
```

Operation structure γ_2

Figure 3-7: An operation structure for message passing

3.2.3 Executions of Operation Structures

In program execution, an event shown in Definition 3.12 is supposed to be a granule instance in the system, which is issued by an operation. Once an operation, excepting arithmetic operation, is performed, an event is supposed to be issued by a processing unit with a unique identifier $eid \in \text{Eid}$ among events in the system.

Definition 3.12 (Event)**.** An *event* $ev \in \text{Event}$ is tuple $\langle eid, op \rangle$ of event identifier $eid \in \text{Eid}$ and performed operation op, which is categorized as either:

- Read event $\langle eid, R_{op} \rangle \in \text{R}_{ev}$ where R_{op} is either read operation or load-link operation,
- Write event $\langle eid, W_{op} \rangle \in \text{W}_{ev}$ where W_{op} is either write operation or store-condition operation,
- Fence event $\langle eid, f \rangle \in \text{Fence}_{ev}$, where $eid \in \text{Eid}$ and $f \in \text{Fence}$, or
- Branch event $\langle eid, \mathbf{br} \rangle$, where $eid \in \text{Eid}$ and $\mathbf{br}$ is a branch operation,

Where $\text{R}_{ev}, \text{W}_{ev}, \text{Fence}, \text{Branch} \subseteq \text{Event}$. For simplicity, the set of event identifiers is assumed to be a subset of natural numbers, such that $\text{Eid} \subseteq \mathbb{N}$.

Read event and write event represent read access and write access, respectively, to access either memory location $loc \in \text{Loc}$ or register $r \in \text{Reg}$. For write event, there always is a write value to store at target memory location. For read event, the read value is supposed to be either initial value 0 or the write value of a conflicting write event in a system. In particular, uninterpreted functions shown in Definition 3.14 are used to describe the effect of the read events and write events in an abstract way. Note that partial function $\mathbb{S}$ shown in the definition is used to indicate the success flag of the write event issued by a store-condition operation. If write event w issued by a store-condition operation succeeded, the value of function $\mathbb{S}[\![w]\!]$ is 0, while $\mathbb{S}[\![w]\!]$ is 1 for otherwise. In addition, the location of the read event and write event can be indicated by the following function.

Definition 3.13 (Location of Memory Events)**.** Let $\mathbb{L} : \text{R}_{ev} \cup \text{W}_{ev} \rightarrow \text{Loc}$ such that

$$\begin{aligned}
\mathbb{L}(\text{ev}(eid, \texttt{v:=e})) &= \begin{cases} e & \text{if } e \in \text{Loc} \\ v & \text{if } v \in \text{Loc} \end{cases} \\
\mathbb{L}(\text{ev}(eid, \mathbf{ll}(v, loc))) &= loc \\
\mathbb{L}(\text{ev}(eid, \mathbf{sc}(v_1, loc, v_2))) &= loc
\end{aligned}$$

Definition 3.14 (Uninterpreted Functions for Memory Events)**.** Let $\mathbb{R}_{\text{val}}, \mathbb{W}_{\text{val}}$ and $\mathbb{S}$ be uninterpreted functions.

$$\begin{aligned}
\mathbb{R}_{\text{val}} &: \text{R}_{ev} \rightarrow \mathbb{N} \\
\mathbb{W}_{\text{val}} &: \text{W}_{ev} \rightarrow \mathbb{N} \\
\mathbb{S} &: \text{W}_{ev} \hookrightarrow \{0, 1\}
\end{aligned}$$

where $\hookrightarrow$ represents a partial function from write event. This means the success flag is considered on only write events corresponding to a store-condition operation.

Figure 3-8 shows a usage of synchronize operations in operation structure `A1`. From these operation structures, let the read events issued by `ll(val, [X])` and `val := [X]` are considered as R_{ll} and R_x, respectively. For write events, the events of `[X] := 1`, `sc(val 1, [X], 3)` and `[Y] := 2` are considered as W_x, W_{sc} and W_y, respectively. To indicate the value of memory event for computation, $\mathbb{R}_{\text{val}}[\![r]\!]$ and $\mathbb{W}_{\text{val}}[\![w]\!]$ are used to indicate the read value of read event r and write value w, respectively. In addition, there is a special case for write event W_{sc} issued by store-condition operation such as `sc(val 1, [X], 3)`, in which $\mathbb{S}[\![W_{sc}]\!]$ represents the success write flag to be returned to temporal register `val_1`.

According to Figure 3-8, read event R_{ll} and write event W_{sc} are supposed to be used as a pair in the program order. In the semantics, write event W_{sc} finds the recent read issued by load-link operation such as R_{ll} in the program order first, then, the write can succeed if there is no write operation appear after the read event is completed. According to the figure, the write seemed to always succeed because there is no other conflicting write to memory location `[X]`. However, in practical system, write event W_y would cause the write not to succeed due to the processor implementation.

```
instr{
  [X] := 1;
};
instr{
  ll(val, [X]);
};
instr{
  [Y] := 2;
};
instr{
  sc(val_1, [X], 3);
};
```

(a) Operation Structure A1

```
instr{
  val := [X]
};
```

(b) Operation Structre A2

Figure 3-8: Example operation structures contain synchronize operations

A fence event is used to restrict the behavior of the read events and write events to behave in an expected manner. For instance, **dmb** instruction for ARM memory model ensures a specific group of events issued before the instruction must be completed, in some manner, before completing the following events in the program order. In this research, the behavior of fence event is not considered in the detail because there are various implementations regarding processor architectures.

A branch event is considered as a decided branch appearing in a system. In some relaxed memory models such as POWER, a branch event causes control dependency which restricts the behavior of program execution. Intuitively, the event let us know the following operations of the branch event in the program order is issued after the evaluation of loop condition was resolved, thus, we also consider this event to appear in the system for considering the behavior precisely.

An *event state* shown in Definition 3.15 represents the state of events appearing in a system including the information of program order, intra-instruction casual order, and requirement for atomic instructions. The intention of an even state is to keep the event that appearing in the system regarding the program order on them. Especially, the program order is a partial order so as to allow the parallelism of events issued by the same assembly instruction.

Definition 3.15 (Event State)**.** An *event state* ε is tuple $\langle e, po, iico, atom\rangle$ consisting of event set $e \subseteq$ Event, program order po and intra-instruction casual order $iico$ on e, and the set of atomic pairs $atom \subseteq (\mathrm{R} \times \mathrm{W})$.

Adding an event to an event state can be done by the following using operator $\prec$, shown in Definition 3.16. The operator is proposed to append a new event to the event state in the program order. In addition, for instantiating operation op in event state ε, function $\mathrm{ev}^*(op, \varepsilon)$ shown in Definition 3.17 is used to find a unique event id $eid \in$ Eid to instantiate an event.

Definition 3.16 (Event Adding Operator)**.** Let $\prec$ be an operator for an event state and event to produced a new state that include the event in a program order. Given event state $\langle s, po, iico, atom\rangle$ and event ev, the operator is defined as the following.

$$\langle s, po, iico, atom\rangle \prec ev = \langle s \cup ev, po \cup po', iico, atom\rangle$$

where $po' = \{(a, ev) \mid a \in s\}$ and ev is an event.

Definition 3.17 (Unique Event)**.** Let $\mathrm{ev}^*(op, \varepsilon)$ be a function to instantiate a new event issued by operation op regarding the existing events in ε, such that

$$\mathrm{ev}^*(op, \langle s, po, iico, atom\rangle) = \langle \mathrm{Max}(\{eid \mid \langle eid, op'\rangle \in s\}) + 1, op\rangle$$

where $\mathrm{Max}(s)$ is a maximum value of elements in set s.

Besides, addition relation *intra-instruction casual order* (iico) is supposed to defined the scope of events issued by the same assembly instruction regarding the program order. For instance, the following operation structure is supposed to issue events R_1, $Wreg_1$, R_2, $Wreg_2$ in the program order.

```
1 instr{
2   val := [X]; //  R1
3   r1 := val   // Wreg1
4 };
5 instr{
6   val := [Y]; // R2
7   r2 := val   // Wreg2
8 }
```

The relation *iico* is then represented by $\{(R_1, Wreg_1), (R_2, Wreg_2)\}$. Moreover, there might be relations on pairs of read event and write event, denoted by *atom*, to restrict the effect of program executions so as to appear atomically without any interruption from any event. Let's consider the following sequence of operation structures, wrapper **atom** defines a relation *atom* on events R_1 and W_1, such that $atom = \{(R_1, W_1)\}$.

```
1 instr{                          1 instr{
2   atom(val := [L]); // R1        2   [L] := 2  // W'
3   val := val + 1                3 }
4   atom([L]:= val);  // W1
5 }
```

The relation restricts write event W' is completed either before or after read event R_1 and write event W_1.

An *execution state* shown in Definition 3.18 represents the intermediate state during execution which keeps the information of each local processor and issued events in the systems, represented by an event state; The local information is supposed to include the current performing instruction on each processing units and the state of local registers.

To consider the execution of operation structures $P = P_1 \cdot \ldots \cdot P_n$, each operation structure is supposed to be executed by a processor independently from each other, such

that operation structure P_i is executed on processor i, where $1 \leq i \leq n$. Thus, set Pid is defined as the set of processor identifiers such that $\text{Pid} = \{1, \ldots, n\}$.

Definition 3.18 (Execution State)**.** An execution state is tuple $\langle exec, reg, \varepsilon \rangle$ of execution units *exec*, register state *reg*, and event state ε. Given execution state ς, its elements can be indicated by the following abbreviations: $\varsigma.\text{exec}, \varsigma.\text{reg}$, and $\varsigma.\text{es}$.

An *execution unit* shown in Definition 3.19 represents the storage of a processor unit that fetches the corresponding of an instruction to be performed. The unit is abstracted as either **ready** state, **complete** state, or an execution structure. The states are used for determining the semantics of operation structures, while an execution structure represents the behavior of the fetched instruction.

Definition 3.19 (Execution Units)**.** An execution units is a mapping from processor identifier to either execution structure or a state of a processor unit, such that

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

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.

A *register state* shown in Definition 3.20 represents the local information of each processor. The local information includes the state of registers, program pointer, and next program pointer of each processor. For program pointer pc and next program pointer nPC, the state of these is captured for realizing the fetching behavior of an assembly program, which allows branch instruction jump to anywhere in the program. Moreover, there are the semantics of an expression $\mathcal{N}$ and the semantics of a Boolean expression $\mathcal{B}$, shown in Figure 3-9 and Figure 3-10, in which $n \in \mathbb{N}, x \in \text{Tmp}, e_1, e2 \in \text{Exp}, c_1, c_2 \in \text{BExp}$, and ρ is the register state for a processor i, e.g., $\rho = \varsigma.\text{reg}(i)$.

Definition 3.20 (Register State)**.** A register state is a mapping of each program executed in a processor, such that

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

In practical, 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. In our research, event state ε shown in Definition 3.15 is used to capture the issued event in an abstract way. However, the evaluation during execution is abstracted by evaluation function $\mathcal{R}[\![r_{ev}]\!]\varepsilon$ of memory model $\mathcal{M}$ is used to evaluate the return value of read access r_{ev} based on the issued operation appearing in ε.

$$
\begin{aligned}
\mathcal{N}[\![n]\!]\rho &= n \\
\mathcal{N}[\![x]\!]\rho &= \rho(x) \\
\mathcal{N}[\![e_1 + e_2]\!]\rho &= \mathcal{N}[\![e_1]\!]\rho + \mathcal{N}[\![e_2]\!]\rho \\
\mathcal{N}[\![e_1 - e_2]\!]\rho &= \mathcal{N}[\![e_1]\!]\rho - \mathcal{N}[\![e_2]\!]\rho \\
\mathcal{N}[\![(c)?e_1 : e_2]\!]\rho &= \begin{cases} e_1 & \text{if } \mathcal{B}[\![c]\!]\rho = \top \\ e_2 & \text{if } \mathcal{B}[\![c]\!]\rho = \bperp \end{cases}
\end{aligned}
$$

Figure 3-9: The semantics of an expression

3.2.4 Semantics of Operation Structures

Let P be a sequence of operation structures and ς be an execution state, the execution can either reach a terminated state, represented as an execution state ς', or violate state, written by $\lightning$. Thus, the semantics is proposed to express the execution behavior in a system.

To define the semantics of operation structure, we use an operational semantics to describe the behavior of the operation structure in a system. The semantics borrows the definitions in a structural operational semantics [RN07], which describes the individual steps of the execution. In the semantics, a transition shown in Definition 3.21 is used to define a rule. Each rule is used to describe the step of an execution of P from execution state ς. The possible outcomes of θ are:

- θ is of the form $\langle P', \varsigma' \rangle$: This means the execution is not finished.
- θ is of the form ς' : This means the execution from state ς is finished.
- θ is $\lightning$: This means the execution violates the program property.

Definition 3.21 (Transition of an execution step)**.** A transition to describe an execution step has the form

$$\langle P, \varsigma \rangle \rightarrow \theta$$

where θ is either $\lightning$ (violation), of the form $\langle P', \varsigma' \rangle$, or of the form ς'.

Given execution structure γ, the order to perform operations could be defined using sequential execution and/or parallel operation. To simplify the representation, we would like to use an *evaluation context*[1] shown in Definition 3.22 to indicate the fragment of execution structure to be evaluated.

Definition 3.22 (Evaluation Context of Execution Structure)**.** Evaluation context E of an execution structure is defined as following.

$$\mathrm{E} \ ::= \ \Box \mid \mathrm{E}; \gamma \mid \mathrm{E} \parallel \gamma \mid \gamma \parallel \mathrm{E}$$

[1]This term is adopted from reduction semantics with evaluation contexts, which is an alternative representation of operational semantics

$$
\begin{aligned}
\mathcal{B}[\![\top]\!]\rho &= \top \\
\mathcal{B}[\![\bot]\!]\rho &= \bot \\
\mathcal{B}[\![e_1 = e_2]\!]\rho &= \begin{cases} \top & \text{if } \mathcal{N}[\![e_1]\!]\rho = \mathcal{N}[\![e_2]\!]\rho \\ \bot & \text{if } \mathcal{N}[\![e_1]\!]\rho \neq \mathcal{N}[\![e_2]\!]\rho \end{cases} \\
\mathcal{B}[\![e_1 > e_2]\!]\rho &= \begin{cases} \top & \text{if } \mathcal{N}[\![e_1]\!]\rho > \mathcal{N}[\![e_2]\!]\rho \\ \bot & \text{if } \mathcal{N}[\![e_1]\!]\rho \leq \mathcal{N}[\![e_2]\!]\rho \end{cases} \\
\mathcal{B}[\![e_1 < e_2]\!]\rho &= \begin{cases} \top & \text{if } \mathcal{N}[\![e_1]\!]\rho < \mathcal{N}[\![e_2]\!]\rho \\ \bot & \text{if } \mathcal{N}[\![e_1]\!]\rho \geq \mathcal{N}[\![e_2]\!]\rho \end{cases} \\
\mathcal{B}[\![c_1 \wedge c_2]\!]\rho &= \begin{cases} \top & \text{if } \mathcal{B}[\![c_1]\!]\rho = \top \text{ and } \mathcal{B}[\![c_2]\!]\rho = \top \\ \bot & \text{if } \mathcal{B}[\![c_1]\!]\rho = \bot \text{ or } \mathcal{B}[\![c_2]\!]\rho = \bot \end{cases} \\
\mathcal{B}[\![c_1 \vee c_2]\!]\rho &= \begin{cases} \top & \text{if } \mathcal{B}[\![c_1]\!]\rho = \top \text{ or } \mathcal{B}[\![c_2]\!]\rho = \top \\ \bot & \text{if } \mathcal{B}[\![c_1]\!]\rho = \bot \text{ and } \mathcal{B}[\![c_2]\!]\rho = \bot \end{cases} \\
\mathcal{B}[\![\neg c_1]\!]\rho &= \begin{cases} \top & \text{if } \mathcal{B}[\![c_1]\!]\rho = \bot \\ \bot & \text{if } \mathcal{B}[\![c_1]\!]\rho = \top \end{cases}
\end{aligned}
$$

Figure 3-10: The semantics of a Boolean expression

where $\square$ is a placeholder to place an execution structure into the context.

For example, given evaluation context $(\gamma_1 \parallel (\square; \gamma_3))$ and execution structure γ_2, thus, $(\gamma_1 \parallel (\square; \gamma_3))[\gamma_2] = (\gamma_1 \parallel (\gamma_2; \gamma_3))$. Besides, the execution structure $(\gamma_1 \parallel (\gamma_2; \gamma_3))$ can be separated to be context $(\square \parallel (\gamma_2; \gamma_3))$ and γ_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 execution structure in instruction execution, such as `instr`$\{E[$`val_z := 1`$]\}$ where $E = (\square \parallel$ `val_n := 0`$)$.

In addition to execution structure, we also introduce the substitutions of operation structures shown in Definition 3.23 to simplify the semantics.

Definition 3.23 (Substitutions of Operation Structures)**.** Let P be a sequence of operation structures, such that $P = \varphi_1 \cdot \ldots \cdot \varphi_n$ and each operation structure is of a form $\varphi_i = \gamma_1; \ldots ; \gamma_{m_i}$, where $0 \leq i \leq n$ and $n, m_i \in \mathbb{N}^+$. The substitutions of operation structure and a sequence of operation structures are defined as followings.

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

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

According to the definition, the intention of the former substitution is to replace element γ' into an operation structure $\gamma_1;\gamma_2;\ldots;\gamma_{m_i}$ at index j^{th}, and then access the element at index k^{th}. For the latter substitution, the element γ' replaces the element at index j^{th} of structure φ_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 $P_2 = \gamma_1;\gamma_2;\gamma_3$, $P^{2,3}[\texttt{assume}(z=1)]$ replaces the element γ_3 with $\texttt{assume}(z=1)$.

Moreover, we could define a generic substitution shown in Definition 3.24 to update a state in the semantics definition. Besides, the substitutions of execution units and register states to processor $i \in \text{Pid}$ are abbreviated as Definition 3.25 to simplify the expression in semantics. In the definition, superscripts $\text{r}(i)$ and $\text{e}(i)$ are used to indicate the register state and the execution unit of processor i to be substituted by reg' and γ', respectively.

Definition 3.24 (Generic Substitution)**.** Let $\mathcal{X}, \mathcal{Y}$ be any arbitrary sets, and ϱ is any mapping $\mathcal{X} \to \mathcal{Y}$,

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

where $v \in \mathcal{Y}$ and $x, y \in \mathcal{X}$.

Definition 3.25 (Substitution of Execution Units and Register State)**.** Let $\langle \text{exec}, \text{reg}, \varepsilon \rangle$ be an execution state, there are substitutions as followings.

$$\begin{aligned} \langle \text{exec}, \text{reg}, \varepsilon \rangle^{\text{r}(i)}[\mathit{reg}'] &= \langle \text{exec}, \text{reg}[i \mapsto \mathit{reg}'], \varepsilon \rangle \\ \langle \text{exec}, \text{reg}, \varepsilon \rangle^{\text{e}(i)}[\gamma'] &= \langle \text{exec}[i \mapsto \gamma], \text{reg}, \varepsilon \rangle \end{aligned}$$

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

[nil] $\langle P, \varsigma^{\text{e}(i)}[\texttt{nil}] \rangle \to \langle P, \varsigma^{\text{e}(i)}[\texttt{complete}] \rangle$

[seq] $\langle P, \varsigma^{\text{e}(i)}[E[\texttt{nil};\gamma]] \rangle \to \langle P, \varsigma^{\text{e}(i)}[E[\gamma]] \rangle$

[par-l] $\langle P, \varsigma^{\text{e}(i)}[E[\texttt{nil} \parallel \gamma]] \rangle \to \langle P, \varsigma^{\text{e}(i)}[E[\gamma]] \rangle$

[par-r] $\langle P, \varsigma^{\text{e}(i)}[E[\gamma \parallel \texttt{nil}]] \rangle \to \langle P, \varsigma^{\text{e}(i)}[E[\gamma]] \rangle$

[fetch] $\langle P^{i,j}[\texttt{instr}\{\gamma\}], \varsigma^{\text{e}(i)}[\texttt{ready}] \rangle \to \langle P^{i,j}[\texttt{instr}\{\gamma\}], (\varsigma_{\text{nPC++}})^{\text{e}(i)}[\gamma] \rangle$

[terminate] $\langle P, \varsigma \rangle \to \varsigma$ if $\forall i \in \text{Pid}.(P(i)(\varsigma.\text{reg(i)(pc)}) = \bot)$

[next] $\langle P, \varsigma^{\text{e}(i)}[\texttt{complete}] \rangle \to \langle P, (\varsigma_{\text{pc}\mapsto\text{nPC}})^{\text{e}(i)}[\texttt{ready}] \rangle$

where $\sigma_{\text{nPC++}}$ and $\varsigma'_{pc \mapsto nPC}$ are defined as followings.

$$\begin{aligned} j &= \varsigma.\text{reg}(i)(\text{pc}) \\ \sigma_{\text{nPC++}} &= \varsigma^{\text{r}(i)}[\varsigma.\text{reg}(i)[\text{nPc} \mapsto j+1]] \\ nPC &= \varsigma.\text{reg}(i)(\text{nPC}) \\ \sigma_{\text{pc}\mapsto\text{nPC}} &= \varsigma^{\text{r}(i)}[\varsigma.\text{reg}(i)[\text{pc} \mapsto nPC]] \end{aligned}$$

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

[read] $\langle P, \varsigma^{\mathrm{e}(i)}[E[\texttt{v:=e}]]\rangle \rightarrow \langle P, \varsigma^{+r}\rangle$ if $e \in \mathcal{V} \setminus \mathrm{Tmp}$

where result $\varsigma^{+r}$is defined as the followings.

$$\begin{array}{rcl} R_{ev} & = & \mathrm{ev}^*(\texttt{v := e}, \varsigma.\mathrm{es}) \\ \varsigma_{es} & = & \varsigma.\mathrm{es} \prec R_{ev} \\ reg_i & = & \varsigma.\mathrm{reg}(i)[v \mapsto \mathcal{R}[\![R_{ev}]\!]\varsigma_{es}] \\ \varsigma^{+r} & = & (\langle \varsigma.\mathrm{exec}, \varsigma.\mathrm{reg}, \varsigma_{es}\rangle^{\mathrm{r}(i)}[reg_i])^{\mathrm{e}(i)}[E[\texttt{nil}]] \end{array}$$

According to rule [read], $\mathcal{R}[\![r]\!]\varepsilon$ is a return value of read event r considered on event state ε. 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.

[write] $\langle P, \varsigma^{\mathrm{e}(i)}[E[\texttt{v:=e}]]\rangle \rightarrow \langle P, \varsigma^{+w}\rangle$ if $v \in \mathcal{V} \setminus \mathrm{Tmp}$

where result ς^{+w} is defined as the followings.

$$\begin{array}{rcl} \varsigma_{es} & = & \varsigma.\mathrm{es} \prec \mathrm{ev}^*(\texttt{v := e}, \varsigma.\mathrm{es}) \\ \varsigma^{+w} & = & \langle \varsigma.\mathrm{exec}, \varsigma.\mathrm{reg}, \varsigma_{es}\rangle^{\mathrm{e}(i)}[E[\texttt{nil}]] \end{array}$$

[arith] $\langle P, \varsigma^{\mathrm{e(i)}}[E[\texttt{v:=e}]]\rangle \rightarrow \langle P, \varsigma_{v:=e}\rangle$ if $e, v \in \mathrm{Tmp}$

where result $\varsigma_{v:=e}$ is defined as the followings.

$$\begin{array}{rcl} reg_i & = & \varsigma.\mathrm{reg}(i)[v \mapsto \mathcal{N}[\![e]\!]\varsigma.\mathrm{reg}(i)] \\ \varsigma_{v:=e} & = & (\varsigma^{\mathrm{r}(i)}[reg_i])^{\mathrm{e}(i)}[E[\texttt{nil}]] \end{array}$$

[fence] $\langle P, \varsigma^{\mathrm{e}(i)}[E[f]]\rangle \rightarrow \langle P, \varsigma^{+f}\rangle$ if $f \in \mathrm{Fence}$

where $\varsigma^{+f} = \langle \varsigma.\mathrm{exec}, \varsigma.\mathrm{reg}, \varsigma.\mathrm{es} \prec \mathrm{ev}^*(f, \varsigma.\mathrm{es})\rangle^{\mathrm{e}(i)}[E[\texttt{nil}]]$.

[ll] $\langle P, \varsigma^{\mathrm{e}(i)}[E[\texttt{ll}(v, loc)]]\rangle \rightarrow \langle P, (\varsigma^{+ll})^{\mathrm{e}(i)}[E[\texttt{nil}]]\rangle$

where ς^{+ll} is defined as followings.

$$\begin{array}{rcl} ll_{ev} & = & \mathrm{ev}^*(\texttt{ll}(v, loc), \varsigma.\mathrm{es}) \\ \varsigma_{es+ev} & = & \varsigma.\mathrm{es} \prec ll_{ev} \\ reg_i & = & \varsigma.\mathrm{reg}(i)[v \mapsto \mathcal{R}[\![ll_{ev}] \\ \varsigma^{+ll} & = & \langle \varsigma.\mathrm{exec}, \varsigma.\mathrm{reg}, \varsigma_{es+ev}\rangle^{\mathrm{r}(i)}[reg_i] \end{array}$$

[sc-fail] $\langle P, \varsigma^{\mathrm{e}(i)}[E[\mathtt{sc}(v_1, loc, v_2)]]\rangle \rightarrow \langle P, \varsigma_f\rangle$ if $\neg C_{succ}$

[sc-suc] $\langle P, \varsigma^{\mathrm{e}(i)}[E[\mathtt{sc}(v_1, loc, v_2)]]\rangle \rightarrow \langle P, \varsigma_s^{+sc}\rangle$ if C_{succ}

where $\varsigma_f, \varsigma_s^{+sc}$ are defined as followings.

$$
\begin{array}{rcl}
sc_{ev} & = & \mathrm{ev}^*(\mathtt{sc}(v_1, loc, v_2), \varsigma.\mathrm{es}) \\
reg_i & = & \varsigma.\mathrm{reg}(i) \\
\varsigma_{es+sc} & = & \varsigma.\mathrm{es} \prec sc_{ev} \\
C_{succ} & \text{iff} & (\mathrm{Sync}[\![sc_{ev}]\!]\varsigma.\mathrm{es} = \top) \\
\varsigma_f & = & (\varsigma^{\mathrm{r(i)}}[reg_i[v_1 \mapsto 1]])^{\mathrm{e}(i)}[E[\mathtt{nil}]] \\
\varsigma_{temp}^{+sc} & = & \langle \varsigma.\mathrm{exec}, \varsigma.\mathrm{reg}, \varsigma_{es+sc}\rangle^{\mathrm{r}(i)}[reg_i[v_1 \mapsto 0]] \\
\varsigma_s^{+sc} & = & (\varsigma_{temp}^{+sc})^{\mathrm{e}(i)}[E[\mathtt{nil}]]
\end{array}
$$

In these rules, $\mathrm{Sync}[\![ev]\!]\varepsilon$ is evaluated to be either $\top$ or $\bot$ to checks whether there is the event corresponding a load-link operation in ε to event ev and the target location of the events is not written by any write access yet. If the condition is not satisfied, the write event does not appear in the system and flag v_1 is set. On the other hand, the write event appears in the system and flag v_1 is clear. Note that the function $\mathrm{Sync}[\![ev]\!]\varepsilon$ depends on an implementation, in which some processors would use multiple locations on the same cache-line for this synchronize behavior. This means if there is a non-conflicting write event following the event corresponding a load-link operation, the store-condition operation can fail if the target locations are located at the same cache-line. Thus, to generalize the semantics, the function Sync is given for target processor, which relies on the implementation of the processor.

[if-$\top$] $\langle P, \varsigma^{\mathrm{e}(i)}[E[\mathtt{if}(c)\{\gamma'\}]]\rangle \rightarrow \theta_\top$ if $\mathcal{B}[\![c]\!]\rho = \top$

[if-$\bot$] $\langle P, \varsigma^{\mathrm{e}(i)}[E[\mathtt{if}(c)\{\gamma'\}]]\rangle \rightarrow \theta_\bot$ if $\mathcal{B}[\![c]\!]\rho = \bot$

where $\theta_\top, \theta_\bot$ are defined as followings.

$$
\begin{array}{rcl}
\rho & = & \varsigma.\mathrm{reg}(i) \\
\theta_\top & = & \langle P, \varsigma^{\mathrm{e}(i)}[E[\gamma']]\rangle \\
\theta_\bot & = & \langle P, \varsigma^{\mathrm{e}(i)}[E[\mathtt{nil}]]\rangle
\end{array}
$$

These rules, [if-$\top$] and [if-$\bot$], represent the behavior of predicated instruction, which is abstracted by our operation structure.

[br-$\top$] $\langle P, \varsigma^{\mathrm{e}(i)}[E[\mathtt{branch}(c, l)]]\rangle \rightarrow \theta_\top$ if $\mathcal{B}[\![c]\!]\rho = \top$

[br-$\bot$] $\langle P, \varsigma^{\mathrm{e}(i)}[E[\mathtt{branch}(c, l)]]\rangle \rightarrow \theta_\bot$ if $\mathcal{B}[\![c]\!]\rho = \bot$

where $\theta_\top$ and $\theta_\bot$ are defined as followings.

$$\begin{array}{rcl}
\rho & = & \varsigma.\mathrm{reg}(i) \\
\varsigma_{+br} & = & \langle \varsigma.\mathrm{exec}, \varsigma.\mathrm{reg}, \varsigma.\mathrm{es} \prec \mathrm{ev}^*(\mathtt{branch}(c,l), \varsigma.\mathrm{es})\rangle \\
P & = & Q^{i,k}[l] \\
reg_i & = & \varsigma.\mathrm{reg}(i)[\mathrm{nPC} \mapsto k] \\
\theta_\top & = & \langle P, ((\varsigma_{+br})^{\mathrm{r}(i)}[reg_i])^{\mathrm{e}(i)}[E[\mathtt{nil}]]\rangle \\
\theta_\bot & = & \langle P, (\varsigma_{+br})^{\mathrm{e}(i)}[E[\mathtt{nil}]]\rangle
\end{array}$$

Rules [br-$\top$] and [br-$\bot$] represent the behavior of a branch operation that can change the next fetching instruction of the processor. To simulate such behavior, register nPC is used to indicate the 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.

[atm-R] $\langle P, \varsigma^{\mathrm{e}(i)}[E[\mathtt{atom}(v_1 := e_1)]]\rangle \rightarrow \theta_{+R}$ if $e_1 \in \mathcal{V} \setminus \mathrm{Tmp}$

[atm-W] $\langle P, \varsigma^{\mathrm{e}(i)}[E[\mathtt{atom}(v_2 := e_2)]]\rangle \rightarrow \theta_{+W}$ if $v_2 \in \mathcal{V} \setminus \mathrm{Tmp}$

where θ_{+R} and θ_{+W} are defined as followings.

$$\begin{array}{rcl}
\varsigma_{es+r} & = & \varsigma.\mathrm{es} \prec \mathrm{ev}^*(\mathtt{atom}(v_1 := e_1), \varsigma.\mathrm{es}) \\
reg_{i,r} & = & \varsigma.\mathrm{reg}(i)[v_1 \mapsto \mathcal{R}[\![R_{ev}]\!]\sigma_{es+r}] \\
\varsigma_{+R} & = & \langle \varsigma.\mathrm{exec}, \varsigma.\mathrm{reg}, \varsigma_{es+r}\rangle^{\mathrm{r}(i)}[reg_{i,r}] \\
\theta_{+R} & = & \langle P, (\varsigma_{+R})^{\mathrm{e}(i)}[E[\mathtt{nil}]]\rangle \\
\varsigma_{es+w} & = & \varsigma.\mathrm{es} \prec \mathrm{ev}^*(\mathtt{atom}(v_2 := e_2), \varsigma.\mathrm{es}) \\
\sigma_{+W} & = & \langle \varsigma.\mathrm{exec}, \varsigma.\mathrm{reg}, \varsigma_{es+w}\rangle \\
\theta_{+W} & = & \langle P, (\varsigma_{+W})^{\mathrm{e}(i)}[E[\mathtt{nil}]]\rangle
\end{array}$$

For the remaining, the behaviors of label and property statements are defined as followings.

[label] $\langle P^{i,j}[\mathtt{label}(l)], \varsigma\rangle \rightarrow \langle P^{i,j}[\mathtt{label}(l)], \varsigma'\rangle$

[assume] $\langle P^{i,j}[\mathtt{assume}(c)], \varsigma\rangle \rightarrow \langle P^{i,j}[\mathtt{assume}(c)], \varsigma'\rangle$ if $\mathcal{B}[\![c]\!]\varsigma.\mathrm{reg}(i) = \top$

[assert-$\top$] $\langle P^{i,j}[\mathtt{assert}(c)], \varsigma\rangle \rightarrow \langle P^{i,j}[\mathtt{assert}(c)], \varsigma'\rangle$ if $\mathcal{B}[\![c]\!]\varsigma.\mathrm{reg}(i) = \top$

[assert-$\bot$] $\langle P^{i,j}[\mathtt{assert}(c)], \varsigma\rangle \rightarrow \lightning$ if $\mathcal{B}[\![c]\!]\varsigma.\mathrm{reg}(i) = \bot$

where

$$\begin{array}{rcl}
j & = & \varsigma.\mathrm{reg}(i)(\mathrm{pc}) \\
reg_{new} & = & \varsigma.\mathrm{reg}(i)[\mathrm{pc} \mapsto j+1][\mathrm{nPC} \mapsto j+1] \\
\varsigma' & = & \varsigma^{\mathrm{r(i)}}[reg_{new}]
\end{array}$$

Then, a *derivation sequence* shown in Definition 3.26 is used to represent the seqeunce to represent how the programs are executed.

Definition 3.26 (Derivation Sequence)**.** Given sequence of operation structures P and execution state ς, a derivation state is either:

- a finite sequence:
$$\theta_1 \rightarrow \theta_2 \rightarrow \ldots \rightarrow \theta_k$$
such that $\theta_1 = \langle P, \varsigma \rangle, \theta_i \rightarrow \theta_{i+1}$ for $1 \leq i \leq k$, and $k > 1$.

- an infinite sequence:
$$\theta_1 \rightarrow \theta_2 \rightarrow \ldots$$
such that $\theta_1 = \langle P, \varsigma \rangle$ and $\theta_i \rightarrow \theta_{i+1}$ for $i \geq 1$.

We could write $\theta_1 \xrightarrow{i} \theta_{i+1}$ to indicate i^{th} steps to reach θ_{i+1} from θ_1. In addition, $\theta_1 \xrightarrow{*} \theta_{i+1}$ is written to indicate that there is a finite sequence from θ_1 to θ_{i+1}. Thus, we could write a semantics function to define the semantics of operation structure as the following definition.

Definition 3.27 (Semantics Function of Operation Structures)**.**

$$S[\![P]\!]^{\mathcal{R}}\varsigma = \begin{cases} \varsigma' & \text{If } \langle P, \varsigma \rangle \xrightarrow{*} \varsigma' \\ \lightning & \text{If } \langle P, \varsigma \rangle \xrightarrow{*} \lightning \\ \underline{\text{undef}} & \text{Otherwise} \end{cases}$$

where $\mathcal{R}$ is a function to realize the return value form existing events regarding memory model $\mathcal{M}$.

Note that result <u>undef</u> means the execution is not terminated 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 $\lightning$ to ensure the program correctness.

3.3 SMT-based Program Verification

In SMT-based program verification, the events are supposed to be considered. The abstraction of programs is analyzed to extract the possible ways to instantiate operations, called *execution paths*. Besides, as assembly language is described in the unstructured programming style, we construct the corresponding control flow graph to consider the possible instantiating of the programs. Intuitively, an *execution path* represents the sequence of control flows to instantiate the assembly programs.

In the executions of operation structures, there could be an *infinite derivation sequence* produced by the structure if there is a loop caused by branch operations. Consequently, the number of events becomes *infinite* as well as the number of free variables in the encoded formula which corresponding to existing events. In SMT-based program verification, the number of free variables should be finite, thus, we consider the executions that are eventually *terminated* for the purpose of program verification

Besides, a direct method to explore the corresponding execution paths is *loop unwinding*, which is a systematic way to expand the loop from a control flow graph if there is a loop. Nevertheless, the number of execution paths for SMT-based program verification is infinite if a loop appears. To limit the state space, the bounded method to unwind loop in an operation structure is given. Therefore, the exploration can be done automatically to produce a finite set of execution paths.

For the encoding method, the specification of target memory model provided by an existing modeling framework is used to consider the possible effects on an execution path. According to a modeling framework, the behavior of program execution is valid if it satisfies the memory model specification. Thus, to realize the effects of an execution path, the execution path is encoded as a first-order formula regarding the memory model specification provided by a modeling framework. In our research, a framework provided by Adve and Gharachorloo [Gha95] and Herding Cats framework [AMT14] are considered to encode an execution path.

3.3.1 Execution Path

An *execution path* shown in Definition 3.28 is the operation structures that has no condition to be considered whether operations should be performed or not. Given a sequence of operation structure P, there could be various candidate executions of P to be considered in program verification. By the representation of an execution path, the possible executions is restricted based on existing events regarding the program order on the events.

Definition 3.28 (Execution Path)**.** An execution path is a sequence of operation structures such that each operation structure always perform the same operations into a system.

In an operation structure, branch operation $\mathbf{branch}(c, l)$ and condition execution $\mathbf{if}(c)\{\gamma\}$ cause the number of operations cannot be determined systematically regarding the evaluation of condition c. Therefore, an *execution path* is introduced to represent the operation structures in which the operations are performed without any decision.

The operation structure that always performs operations, in the same way, is considered as *unique operation structure* shown in Definition 3.29. Thus, every execution can be constructed by an execution path always contain the same events. In addition, every branch in an execution path must be *unique branch* shown in Definition 3.30, in which the next performing operations must be the same in every execution. For instance, branch operation $\mathbf{branch}(\top, l)$ is an always branch that goes to only label l. Otherwise, branch operation $\mathbf{branch}(c, l)$ is a unique branch if condition c is always either *satisfied* or *unsatisfied* in every execution. To make a unique branch, a tricky way is to add an assumption $\mathbf{assume}(c)$ or assumption $\mathbf{assume}(\neg c)$ before the branch, such as `assume(`c`); branch(`c`, l)`.

Definition 3.29 (Unique operation structure)**.** An operation structure is *unique* if there is no a condition structure and all branches are unique branches.

Definition 3.30 (Unique branch)**.** `branch(`c`, `l`)` is a unique branch if condition c is always either satisfied or unsatisfied by every execution.

Figure 3-11 shows an execution path of message passing in Figure 3-7. A unique operation structure ψ_1^1 is same to γ_1 in Figure 3-7, while operation structure ψ_2^1 is a unique operation structure of γ_2. Assumption `assume(`$\neg(val_n$ `= 1))` at line 31 make `branch(n = 1, label(L))` at line 27 is always unsatisfied.

In program verification, the set of execution paths is used to represent the behavior of the original operation structures. The behavior of each operation structure can be represented by the set of unique operation structures, that should cover original behaviors. Thus, the set of execution paths is the combinations of those sets, in which Definition 3.31 shows a way to construct the set. Note that, if there is an infinite set of unique operation structures, the set of execution paths is also infinite. For automatically verifying, a bounded unwinding method is used to construct the finite set of execution paths.

Definition 3.31 (The set of execution paths)**.** Given a sequence of operation structures $\Gamma = \Gamma_1 \cdot \ldots \cdot \Gamma_n$, let Ψ_{Γ_i} be the set of unique operation structures of operation structure Γ_i. An *execution path* is a combination of sets $\Psi_{\Gamma_1}, \ldots, \Psi_{\Gamma_n}$, in which the set of execution paths is $\Pi_\Gamma = \{(\psi_1 \cdot \ldots \cdot \psi_n) \mid \psi_1 \in \Psi_1 \wedge \cdots \wedge \psi_n \in \Psi_n\}$.

3.3.2 Bounded Loop Unwinding

An *execution path* is an essential component in this method, in which the set of execution paths should cover every execution of the given operation structures. A way to explore execution paths is to use a bounded model checking approach [AMP09] 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) shown in Definition 3.32 describes the way in which the operation structure can execute. A node is either: *nil*, *instruction execution*, or *annotation*. Note that 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 the program order or the target label annotation. Also, Definition 3.33 and Definition 3.34 show auxiliary notations on nodes in a control flow graph for loop detecting in path exploring algorithm. In particular, these notations are used to detect a loop in the control flow graph.

Definition 3.32 (Control Flow Graph of Operation Structure)**.** A control flow graph of operation structure is 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.

Definition 3.33 (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 $1 \leq k < n$.

```
instr{
  val := 1;
  r1 := val
};
instr{
  val := r1
  [x] := val
};
instr{
  val := r1
  [y] := val
}
```

Operation structure ψ_1^1

```
label(l);
instr{
  val := [y];
  r1 := val
};
instr{
  (rd := 1 || rt := r2);
  val_z := (rd = rt)?1:0;
  z := val_z;
  val_n := (rd = rt)?0:1;
  n := val_n
};
instr{
  val_n := n;
};
assume(¬(val_n = 1));
instr{
  val_n := n;
  branch(val_n = 1, label(l))
};
instr{
  val := [x];
  r1 := val
};
assert(r1 = 1)
```

Operation structure ψ_2^1

Figure 3-11: Execution path $\pi_1 = (\psi_1^1 \cdot \psi_2^1)$

Definition 3.34 (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 [WG] 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 sequentially fetch instructions, 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 [br-$\top$] and [br-$\bot$] proposed in Section 3.2.4. Instead of realizing the behavior of fetching execution in 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 Γ_2 in Figure 3-7 has a sequence of 6 elements. Figure 3-12 is the output of CFG(Γ_2), in which the

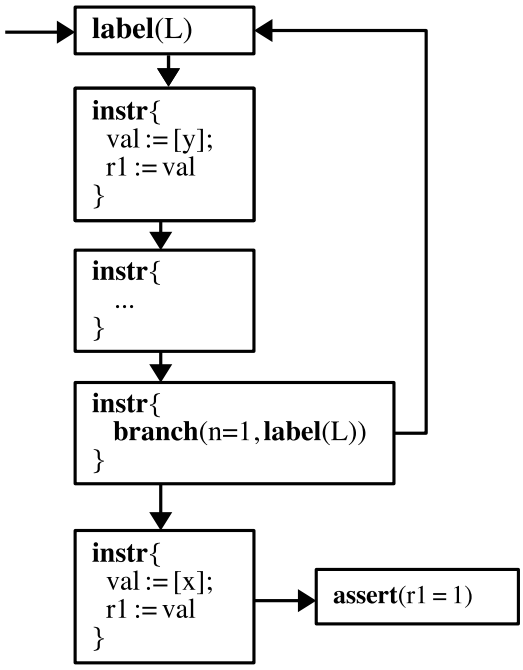


Figure 3-12: The control flow graph for operation structure Γ_2

nodes corresponding to elements in Γ_2, and the edges represent the flows to fetch the next element into the system. Note that the detail of instruction `cmp r1, r2` is omitted in this control flow graph.

Algorithm 1 CFG Generation for Basic Instruction Cycle

1: **procedure** CFG$((e_1; e_2; \ldots; e_n))$
2: $(v_1, \ldots v_n) \leftarrow$ (**new** NODE$(e_1), \ldots,$ **new** NODE(e_n))
3: $n_0 \leftarrow v_1$
4: $V \leftarrow \{v_1, \ldots, v_n\}$
5: $E \leftarrow \{(v_1, v_2), (v_2, v_3), \ldots, (v_{n-1}, v_n)\}$
6: **for** $i = 1$ to n **do**
7: **if** v_i.body **contains** **branch**(*cond*, *label*) **then**
8: $E \leftarrow E \cup \{(v_i, v) \mid v \in V \wedge v.\text{body is } \mathit{label}\}$
9: **end if**
10: **end for**
11: **return** $\langle V, E, n_0 \rangle$
12: **end procedure**

An approach to loop unwinding under a bound is a way to explore program behaviors for program verification using an SAT/SMT solver [AKT13, AMP09, DHKR11]. Although there are transformations for while-loop and reducible CFG shown in [AMP09, DHKR11], 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. Besides, the behavior of predicated instruction described as a condition execution is also extracted as two control flows through 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 Figure 3-7, 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; then, 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, such as Figure 3-12, is then extracted as unique operation structures under a bound using procedure EXPLORE shown in Algorithm 3. However, the input graph of procedure EXPLORE is required to be prepared because each control flow is expected to be a unique operation structure. These preparations are done by PREPARECOND and PREPAREBRANCH appearing in Algorithms 4 and 5.

Algorithm 2 Path Exploring algorithm

1: **procedure** PATHEXPLORING($\mathcal{G}, k$)
2: $s \leftarrow \emptyset$
3: **for all** $\langle V, E, n\rangle$ in $\mathcal{G}$ **do**
4: $\langle V, E, n\rangle \leftarrow$PREPARECOND($\langle V, E, n\rangle$)
5: $(\langle V, E, n\rangle, \mathit{loopEdges}) \leftarrow$PREPAREBRANCH($\langle V, E, n\rangle$)
6: $s \leftarrow s \cup \{$EXPLORE($\langle V, E, n\rangle, n, k, \mathit{loopEdges}$)$\}$
7: **end for** ▷ Cartesian product of every set of control flows
8: **return** $\{(P_1 \cdot P_2 \cdot \ldots \cdot P_n | (P_1, P_2, \ldots, P_n) \in \prod_{i \in S} i\}$
9: **end procedure**

For the behavior of condition execution `if`$(c)\{\gamma\}$, rules [if-$\top$] and [if-$\bot$] show that there are two ways to perform condition execution `if`$(c)\{\gamma\}$, in which the number of events could be affected by this behavior. Thus, procedure PREPARECOND shown in Algorithm 4 replaces every condition execution in the graph by two additional control flows, in which one for taking the condition execution and another one for taking nothing. In each additional flow, a proper assumption annotation is added. Figure 3-13(a) shows a node containing execution execution **`if`**`(z = 1){val := [A]; r1:= val}`. This node represents the execution of instruction `ldreq r1, [A]` of ARM architecture. Figure 3-13(b) shows the result of replacing node by PREPARECOND.

For the behavior of branch operation `branch`(c, l), rules [br-$\top$] and [br-$\bot$] 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

Algorithm 3 Explore unique operation structures

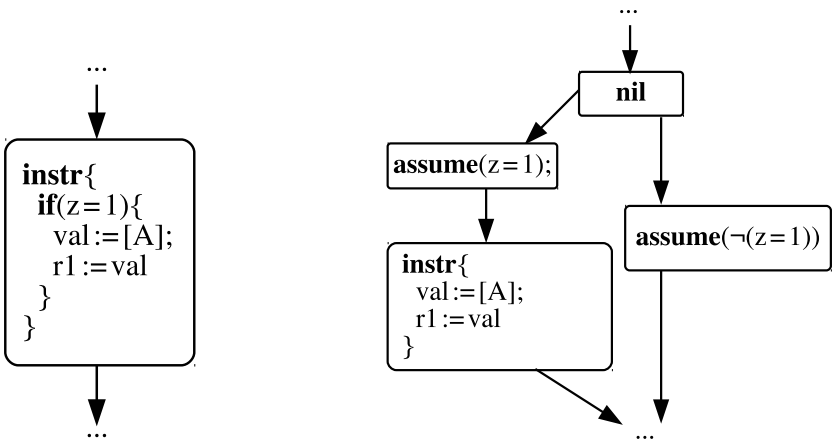
```
1: procedure EXPLORE(⟨V, E, n⟩, v, k, loopEdges)
2:     next ← {u | (v, u) ∈ E}
3:     if next = ∅ then
4:         return {v.body}
5:     else if |next| > 1 and ∃(v, u) ∈ next.(v, u) ∈ loopEdges then
6:         if k ≤ 0 then
7:             next ← {u | u ∈ next ∧ (v, u) ∉ loopEdges}
8:         end if
9:         R ← ⋃_{i∈next} Explore(⟨V, E, n⟩, i, max(k − 1, 0), loopEdges)
10:        return {(v.body; u) | u ∈ R}
11:    else
12:        R ← ⋃_{i∈next} Explore(⟨V, E, n⟩, i, k, loopEdges)
13:        return {(v.body; u) | u ∈ R}
14:    end if
15: end procedure
```

Algorithm 4 Prepare condition execution

```
1: procedure PREPARECOND(⟨V, E, n⟩)
2:     for all v in V do
3:         if v.body = instr {if(cond){γ}} then
4:             v_0 ← new NODE( assume(cond) )
5:             v_1 ← new NODE( instr{γ} )
6:             v_2 ← new NODE( assume(¬cond))
7:             E_1 ← {(v_0, v_1), (v_1, n), (v_2, n) | (v, n) ∈ E ∧ n ∈ V}
8:             E_2 ← {(v, v_0), (v, v_2)}
9:             E_0 ← E \ {(v, n) | (v, n) ∈ E ∧ n ∈ V}
10:            E ← E_0 ∪ {(v_0, v_1)} ∪ E_1 ∪ E_2
11:            V ← V ∪ {v_0, v_1, v_2}
12:            v.body← nil
13:        end if
14:    end for
15:    return ⟨V, E, n⟩
16: end procedure
```

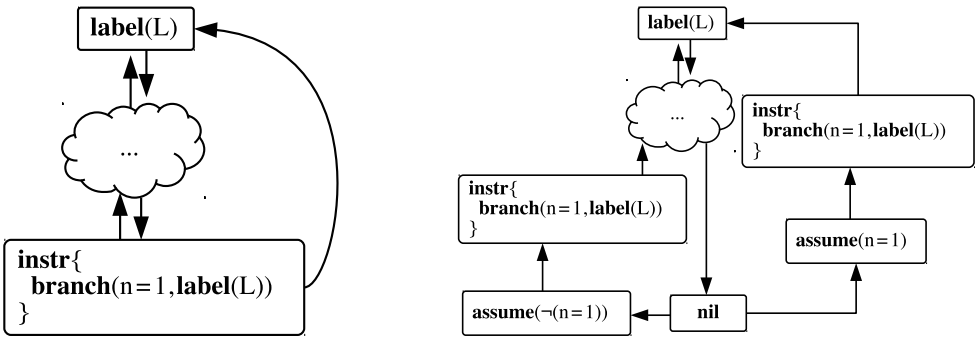
Algorithm 5 Eliminate branch

1: **procedure** PREPAREBRANCH($\langle V, E, n\rangle$)
2: $\quad$ *visited* $\leftarrow \emptyset$
3: $\quad$ *loopEdges* $\leftarrow \emptyset$
4: $\quad$ **for all** v in V **do**
5: $\quad\quad$ **if** v.body **contains branch**(*cond*, *label*) **and** $v \notin$ *visited* **then**
6: $\quad\quad\quad$ $v_1 \leftarrow$ **new** NODE(**instr**{ **assume**(*cond*) })
7: $\quad\quad\quad$ $v_2 \leftarrow v$.clone()
8: $\quad\quad\quad$ $E_1 \leftarrow \{(v, v_1), (v_1, v_2), (v_2, n) \mid (v, n) \in E \wedge n \in V \wedge n$ **contains label**$(l)\}$
9: $\quad\quad\quad$ *loopEdges* $\leftarrow$ *loopEdges* $\cup \{(v, v1) | (v, n) \in E \wedge n$ **dominates** $v\}$
10: $\quad\quad\quad$ $v_3 \leftarrow$ **new** NODE(**instr**{ **assume**($\neg$*cond*) })
11: $\quad\quad\quad$ $v_4 \leftarrow v$.clone()
12: $\quad\quad\quad$ $E_2 \leftarrow \{(v, v_3), (v_3, v_4), (v_4, n) \mid (v, n) \in E \wedge n \in V \wedge \neg(n$ **contains label**$(l))\}$
13: $\quad\quad\quad$ $E_0 \leftarrow E \setminus \{(v, n) \mid (v, n) \in E \wedge n \in V\}$
14: $\quad\quad\quad$ $E \leftarrow E_0 \cup E_1 \cup E_2$
15: $\quad\quad\quad$ $V \leftarrow V \cup \{v1, v2, v3, v4\}$
16: $\quad\quad\quad$ *visited* $\leftarrow$ *visited* $\cup \{v1, v2, v3, v4\}$
17: $\quad\quad\quad$ v.body$\leftarrow$ **nil**
18: $\quad\quad$ **end if**
19: $\quad$ **end for**
20: $\quad$ **return** $(\langle V, E, n\rangle, loopEdges)$
21: **end procedure**



(a) Original node (b) Result of elimination

Figure 3-13: Eliminating execution condition



(a) Original branch node (b) Result of preparing

Figure 3-14: Preparing unique branches

by the semantics. This means among rules [br-⊤] 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 enforces 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 have only a derivation sequence for those control decisions. Note that this procedure also collects the edges that causes a backward branch in the set *loopEdges*. Figure 3-14 illustrates the translation of the given graph. Figure 3-14(a) shows the original graph that contains a branch node; Figure 3-14(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 $\langle V, E, n\rangle$ 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

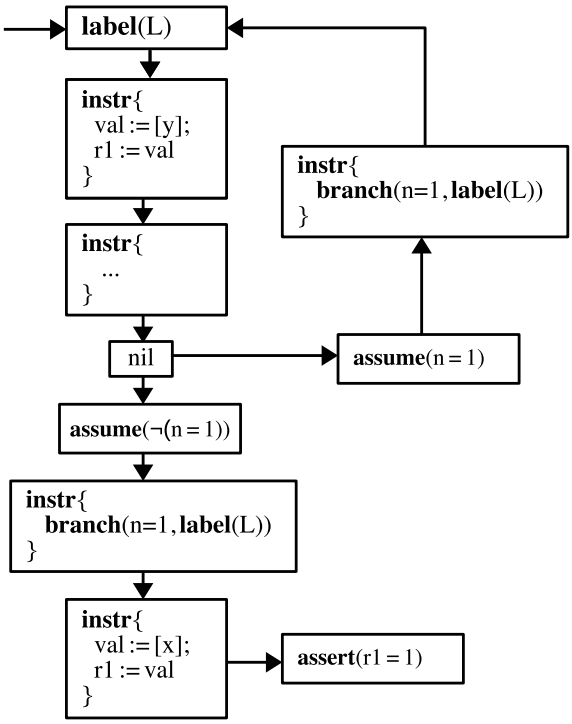


Figure 3-15: A prepared CFG for procedure Explore

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 the 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 $\mathcal{G} = \{G_1, \ldots, G_n\}$ corresponding to operation structures for program verification, PathExploring$(\mathcal{G}, k)$ in Algorithm 2 explores a set of execution paths of $\mathcal{G}$ under bound k. For each graph $G \in \mathcal{G}$, procedures PrepareCond and PrepareBranch are applied to the graph G. For instance, Figure 3-15 shows the result after applying PrepareCond and PrepareBranch to the graph in Figure 3-12. 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 combined 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 $\mathcal{G}$. However, one would propose an alternative approach to exploring execution paths that cover every execution of the programs.

3.3.3 Encoding Scheme

By given an execution path, we first transform the execution path into a *static single assignment* form (SSA) to represent a symbolic execution, in which the evaluation of variables represented by symbolic values. Then, given an execution path $\psi = \psi_1 \cdot \ldots \cdot \psi_n$ in SSA form, the program property of execution path ψ is of the form $(p \implies a)$ where p is the property condition and a is the assertion condition accumulated from the execution path. In addition, event state ε_ψ of execution path ψ can be constructed by considered the operation to be performed. To ensure the program correctness, the valuation of symbolic values must *always* satisfy $p \implies a$. However, to adopt an SMT solver in program verification, we would like to ensure there is no valuation such that $\neg(p \implies a)$ is satisfied on memory model $\mathcal{M}$, written by $\varepsilon_\psi \not\models_{\mathcal{M}} \neg(p \implies a)$.

In static single assignment form (SSA), the variables appearing in a program are supposed to be assigned at most one time. For instance, the following program is in SSA form, in which the variable can represent how the data is flown in the program.

```
1 a^1 = 2;
2 v^1 = 1;
3 v^2 = v^1 + a^1;
4 assert(v^2 > a^1);
```

In an execution path, we rename only local variable names: temporal registers Tmp and registers Reg, excluding memory locations Loc. For instance, Figure 3-16 shows the corresponding SSA form of an execution path from Figure 3-11. Consequently, the variables in the operations can appear in the formula directly. According to SSA form, the program condition p and assertion condition a can be accumulated from assumption conditions and assertion conditions, respectively, such as $p \iff \neg(val_n^0 = 1)$ and $a \iff (\mathrm{r1}^2 = 1)$.

To provide an event state from an execution path in SSA form, the operations in each operation structure are then performed in the program order (po) and collected them in the event state. 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;B`, or nested execution structures, such as **instr**{`A`};**instr**{`B` $\|$ ...}, the events of `A` must be ordered before the events of `B` in the program order (po). As the parallel execution, such as `A` $\|$ `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 **instr**{`A;B`}, the events of `A` must be ordered before the events of `B` in the intra-instruction casual order (iico).

Consequently, a first-order formula is then required to find a valuation on the free variables. Thus, the encoded formula must include the property $\neg(p \implies a)$, event state describing the behavior of programs symbolically ε, and the first-order formula to constraint the effect on event state regarding memory model $\mathcal{M}$. The way to encode the behavior is then explained in the followings regarding a memory model for using in an

```
1  instr{
2    val^0 := 1;
3    r1^0 := val^0
4  };
5  instr{
6    val^1 := r1^0
7    [x] := val^1
8  };
9  instr{
10   val^2 := r1^0
11   [y] := val^2
12 }
```

The SSA form of ψ_1^1

```
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)?1:0;
9    z^0 := val_z^0;
10   val_n^0 := (rd^0 = rt^0)?0:1;
11   n^0 := val_n^0
12 };
13 instr{
14   val_n^1 := n^0
15 };
16 assume(¬(val_n^0 = 1));
17 instr{
18   val_n^2 := n^0;
19   branch(val_n^2 = 1, label(L))
20 };
21 instr{
22   val^4 := [x];
23   r1^2 := val^4
24 };
25 assert(r1^2 = 1)
```

The SSA form of ψ_2^1

Figure 3-16: The SSA form of execution path π_1

SMT solver. Note that the encoding methods proposed in this research rely on Herding cats [AMT14] and Gharachorloo framework [Gha95].

Encoding Scheme of Gharachorloo Framework

In this framework, the execution order is expected to be realized explicitly, in which the order relies on the sub-operations appearing to the target processor. In this framework, there is an abstraction of a multiprocessor system consisting of n processors, such that a write access must be separated into n sub-operations to appear on each processor. Then, the write access can be completed once all sub-operations are updated the target memory location. For read access, there is a corresponding sub-operation to be considered regarding the sub-operations appear to its processor that issued the read access. To control the behavior regarding memory model $\mathcal{M}$, the program information which is extracted from ε is used with the constraints on the execution order to consider which order is allowed to occur on the target memory model. Note that the effect of execution can be

realized by the execution order.

In the previous work [MCA17], the way to abstract the behavior as a formula is proposed for Gharachorloo framework [Gha95]. In particular, *predicated functions* are used to represent the program information that can be extracted from the event state, while the axioms and/or conditions are abstracted on the predicated functions to realize the possible execution orders. This means the predicated functions are also considered to be realized under some constraints.

Given even state ε, property condition p, assertion condition a and memory model $\mathcal{M}$, the encoded formula is of the form

$$C_\varepsilon \wedge A^\varepsilon_{\mathcal{M}} \wedge p \wedge \neg a$$

where C_ε represents the *program behavior* and $A^\varepsilon_{\mathcal{M}}$ represents *constraints* of memory model $\mathcal{M}$ for event state ε. The encoded formula represents the condition of execution state σ whether there is a computation violating assertion a.

Program Behavior Program behavior C_ε captures the program semantics, such as arithmetic semantics, and memory accesses in event state ε. In particular, C_ε contains the encoded program information Info_ε and the behavior properties Basis_ε. The formulas in Info_ε represent what was extracted from event state ε directly such as program order and representations of memory accesses. While Basis_ε of event state ε is constructed by analyzing behavior properties of programs such as conflicting operations and conflict orders. The formulas in Basis_ε are necessary to determine a possible execution order of event state ε. Therefore, the program behavior C_ε consists of Info_ε and Basis_ε such that:

$$C_\varepsilon = \text{Info}_\varepsilon \wedge \text{Basis}_\varepsilon$$

Program Information: Given event state $\varepsilon = \langle s, po, iico, rmw \rangle$, every read and write accesses in s are considered specifically for each processor. Let's assume there are n processors, every write access w eventually appears as sub-operation $w(i)$ on each processor i for any $i \in \{1, \ldots, n\}$. For read access r issued by processor i, there is a sub-operation $r(i)$ appear on processor i to access. Note that, although there is no processor identifier information in an event state, the identifier can be analyzed dynamically from program order *po* information; If the events are issued by the same processor, those events might be related either directly or indirectly. For instance, if there are $a \xrightarrow{po} b$ and $a \xrightarrow{po} c$, events a, b, and c are issued by the same processor. Those sub-operations are then used by uninterpreted functions, such as program order and execution order, to realize an execution order and the return values of the read accesses.

Definition 3.35 (Sub-Operation of Memory Event)**.** Let W_ε be the set of sub-operations of the write events, R_ε be the set of sub-operations of the read events, and RW_ε be the set of sub-operations, such that $RW_\varepsilon \setminus R_\varepsilon = W_\varepsilon$,

$$\begin{aligned} W_\varepsilon &= \{w(i) \mid \varepsilon = \langle s, po, iico, rmw \rangle \text{ and } w \in s \text{ and } w \text{ is a write event and } 1 \leq i \leq n\}, \\ R_\varepsilon &= \{r(i) \mid \varepsilon = \langle s, po, iico, rmw \rangle \text{ and } r \in s \text{ and } r \text{ is a read event and} \\ &\qquad r \text{ is issued by processor } i\ \}. \end{aligned}$$

Besides, uninterpreted functions shown in the following definition represent the information of those events, where $\mathrm{Ev}_\varepsilon = RW_\varepsilon \cup \mathrm{Fence}_\varepsilon$ and $\mathrm{Fence}_\varepsilon$ is the set of fence events appearing in event state ε.

Definition 3.36 (Uninterpreted Functions for Gharachorloo framework)**.**

$$\begin{aligned}
\mathrm{loc} &: RW_\varepsilon \to \mathrm{Loc} && \text{(target location of a memory access)}\\
\mathrm{write_value} &: W_\varepsilon \to \mathbb{N} && \text{(write value of sub-operation)}\\
\mathrm{read_value} &: R_\varepsilon \to \mathbb{N} && \text{(read value of sub-operation)}\\
\mathrm{pid} &: \mathrm{Ev}_\varepsilon \to \{1,\ldots,n\} && \text{(processor identifier)}\\
\mathrm{po} &: (\mathrm{Ev}_\varepsilon \times \mathrm{Ev}_\varepsilon) \to \{\top,\bot\} && \text{(program order)}\\
\mathrm{rmw} &: (R_\varepsilon \times W_\varepsilon) \to \{\top,\bot\} && \text{(read-modified-write requirement)}
\end{aligned}$$

Then, the following formulas are built to capture the information of event state $\varepsilon = \langle s, po, iico, rmw\rangle$.

$$\begin{aligned}
e_{write} &= \textstyle\bigwedge_{w(i)\in W_\varepsilon}(\mathbb{L}(w) = \mathrm{loc}(w(i)) \wedge \mathbb{W}_{\mathrm{val}}[\![w]\!] = \mathrm{write_val}(w(i)))\\
e_{read} &= \textstyle\bigwedge_{r(i)\in R_\varepsilon}(\mathbb{L}(r) = \mathrm{loc}(r(i)) \wedge \mathbb{R}_{\mathrm{val}}[\![r]\!] = \mathrm{read_val}(r(i)))\\
e_{pid} &= \textstyle\bigwedge_{ev\in \mathrm{Ev}_\varepsilon}(\mathrm{pid}(ev) = j) && \text{where } j \text{ is the issued processor of } ev\\
e_{po} &= \textstyle\bigwedge_{a,b\in s}((a,b) \in po \implies \mathrm{po}(a,b))\\
e_{rmw} &= \textstyle\bigwedge_{r(i),w(i)\in RW_\varepsilon}((r,w) \in rmw \textbf{ iff } \mathrm{rmw}(r(i),w(i)))
\end{aligned}$$

Consequently, formula $\mathrm{Info}_\varepsilon$ is $e_{write} \wedge e_{read} \wedge e_{pid} \wedge e_{po} \wedge e_{rmw}$. Note that this formula captures only the information of events to be represented in the first-order formula. The uninterpreted functions, such as read_val, are then evaluated under the constraints using an SMT solver later.

Behavior Property The program order information is given in formula $\mathrm{Info}_\varepsilon$ which is extracted directly. For program orders that are not given, the solver can assume the existence of an order by itself. For instance, if there are $\mathrm{pid}(c) \neq \mathrm{pid}(b)$, $\mathrm{po}(a,b)$ and $\mathrm{po}(c,d)$, an SMT solver could assume $\mathrm{po}(c,b)$ occur even if it should not. To avoid this, formula $e_{po'}$ is given such that

$$e_{po'} = \bigwedge_{x,y\in RW_\varepsilon} (\mathrm{pid}(x) \neq \mathrm{pid}(y) \implies \neg(\mathrm{po}(x,y)))$$

Two memory operations are the *conflict* operations if one of them is a write operation and they access the same location. For every events x, y appearing in event state ε, the uninterpreted function $\mathrm{conflict}(x,y)$ is defined such that

$$e_{conf} = \bigwedge_{x(i),y(j)\in RW_\varepsilon} (\mathrm{conflict}(x,y) \textbf{ iff } (\mathrm{loc}(x(i)) = \mathrm{loc}(y(j)) \wedge (x(i) \in W_\varepsilon \vee y(j) \in W_\varepsilon)))$$

Besides, there are two basic orders for constraining the execution order, which are *conflicting order*, denoted by $\xrightarrow{co}$, and *inter-conflicting order*, denoted by $\xrightarrow{co'}$. These orders are defined regarding the execution order $\xrightarrow{xo}$. Thus, let co, coe, and xo be uninterpreted

functions of those orders, such that

$$\begin{array}{rcl} e_{co} & = & \bigwedge_{x(i),y(j)\in RW_\varepsilon}(\mathrm{co}(x,y)\ \textbf{iff}\ \mathrm{conflict}(x,y)\wedge\exists k.\mathrm{xo}(x(k),y(k))) \\ e_{coe} & = & \bigwedge_{x(i),y(j)\in RW_\varepsilon}(\mathrm{coe}(x,y)\ \textbf{iff}\ \mathrm{co}(x,y)\wedge\mathrm{pid}(x)\neq\mathrm{pid}(y)) \end{array}$$

Then,

$$\mathrm{Basic}_\varepsilon = e_{po'}\wedge e_{conf}\wedge e_{co}\wedge e_{coe}$$

Constraints of a memory model The specification of a memory model is of the form similar to Figure 2-10, which consists of the definition of significant orders, underlying requirements, and conditions on an execution order $\xrightarrow{xo}$, such that

$$A^{\varepsilon}_{\mathcal{M}} = \mathcal{M}_{def}\wedge\mathcal{U}_\varepsilon\wedge\mathcal{M}_{cond}$$

where $\mathcal{M}_{def}$ represents the definitions of significant orders, such as $\xrightarrow{spo}$ and $\xrightarrow{sco}$, $\mathcal{U}_\varepsilon$ represents the underlying behaviors of multiprocessor systems, and $\mathcal{M}_{cond}$ represents the conditions on an execution order of memory model $\mathcal{M}$. Note that the sub-formulas of these formulas usually contain the axioms to constraints the valid execution regarding memory model $\mathcal{M}$.

Underlying behavior: The formula of underlying behavior consists of the formulas of conditions 1, 2, 3 shown in Section 2.3.1, such that

$$\mathcal{U}_\varepsilon = \mathrm{Cond}^{\varepsilon}_1\wedge\mathrm{Cond}^{\varepsilon}_2\wedge\mathrm{Cond}^{\varepsilon}_3$$

The termination condition, Condition 1, of writes events ensures every corresponding sub-operation of a write event must eventually appear in the execution order $\xrightarrow{xo}$, which is described by the following formula.

$$\mathrm{Cond}^{\varepsilon}_1 = \bigwedge_{w(i)\in W_\varepsilon, y(j)\in RW_\varepsilon}(\mathrm{xo}(w(i),y(j))\ \textbf{xor}\ \mathrm{xo}(y(j),w(i)))$$

where operator **xor** is used to choose one of orders must occur in a system.

To preserve Condition 2, the return value of read sub-operations, the corresponding sub-operation $r(i)\in R_\varepsilon$ must appear in an execution order $\xrightarrow{xo}$, thus, the following formula must be included in formula $\mathrm{Cond}^{\varepsilon}_2$.

$$\bigwedge_{r(i)\in R_\varepsilon, y(j)\in RW_\varepsilon}(\mathrm{xo}(r(i),y(j))\ \textbf{xor}\ \mathrm{xo}(y(j),r(i)))$$

Besides, return value function is then defined regarding condition 2 as following definition.

Definition 3.37 (Return value function for Gharchorloo framework)**.**

$$\mathrm{read\_val}(r(i)) = \begin{cases} \mathrm{init}(\mathrm{loc}(r(i))) & \text{If } \mathrm{no\_w_{po}}(r(i))\wedge\mathrm{no\_w_{xo}}(r(i)) \\ \mathrm{W_{xo}}(r(i)) & \text{If } \mathrm{no\_w_{po}}(r(i))\wedge\neg\mathrm{no\_w_{xo}}(r(i)) \\ \mathrm{W_{po}}(r(i)) & \text{Otherwise} \end{cases}$$

where init(L) represents the initial value of location L, which can be an arbitrary value, $\mathrm{W_{xo}}$ and $\mathrm{W_{po}}$ are the return values from write sub-operations that satisfy their conditions, which are defined later. The following definition of $\mathrm{no\_w_{po}}(r(i))$ checks that there is no conflicting write operation that appears before read operation r in the program order and appears in execution order before read event r. The definition $\mathrm{no\_w_{xo}}(r(i))$ checks that there is no conflicting write operation appear in the execution order before read sub-operation $r(i)$.

Definition 3.38 (Return value predicates for Gharachorloo framework)**.**

$$\mathrm{no\_w_{po}}(r(i)) \quad \textbf{iff} \quad \bigwedge_{w(i)\in W_\varepsilon} (\mathrm{conflict}(w,r) \wedge \mathrm{po}(w,r) \implies \mathrm{xo}(w(i),r(i)))$$

$$\mathrm{no\_w_{xo}}(r(i)) \quad \textbf{iff} \quad \bigwedge_{w(i)\in W_\varepsilon} (\mathrm{conflict}(r,w) \implies \mathrm{xo}(r(i),w(i)))$$

The value $\mathrm{W_{po}}(r)$ is returned by the conflict write operation that has no other conflicting write operations appear between that read and write operations in the program order. For the value $\mathrm{W_{xo}}(r)$, it is returned by the conflict write operation that has no another conflict write operations appear between that read and write operation in the execution order. To determine those values, the following formulas are used.

$$e_{W_{\mathrm{po}}} = \bigwedge_{r(i)\in R_\varepsilon} \left(\exists w(i) \in W_\varepsilon . \left(\begin{array}{l} \mathrm{conflict}(r,w) \wedge \mathrm{xo}(r(i),w(i)) \wedge \\ \mathrm{Conseq_{po}}(w,r) \wedge \mathrm{write\_val}(w(i)) = \mathrm{W_{po}} \end{array} \right) \right)$$

$$e_{W_{\mathrm{xo}}} = \bigwedge_{r(i)\in R_\varepsilon} \left(\exists w(i) \in W_\varepsilon . \left(\begin{array}{l} \mathrm{conflict}(r,w) \wedge \mathrm{xo}(w(i),r(i)) \wedge \\ \mathrm{Conseq_{xo}}(w(i),r(i)) \wedge \mathrm{write\_val}(w(i)) = \mathrm{W_{xo}} \end{array} \right) \right)$$

Predicate $\mathrm{Conseq_{po}}(w,r)$ is used to check that write event w is followed by conflicting read event r in the program order consecutively. Similarly, $\mathrm{Conseq_{xo}}(w(i),r(i))$ is used to check whether write sub-operation $w(i)$ is followed by conflicting read sub-operation $r(i)$ immediately in execution order $\xrightarrow{xo}$. The following formulas are then used to defined predicates $\mathrm{Conseq_{po}}(w,r)$ and $\mathrm{Conseq_{xo}}(w(i),r(i))$.

$$e_{\mathrm{conseq\_po}} = \bigwedge_{w(i)\in W_\varepsilon, r(i)\in R_\varepsilon} \left(\begin{array}{l} \mathrm{Conseq_{po}}(w,r) \ \textbf{iff} \\ \left(\begin{array}{l} \mathrm{po}(w,r) \wedge \bigwedge_{w'(i)\in W_\varepsilon \setminus \{w(i)\}} \\ (\mathrm{conflict}(w',r) \implies \mathrm{notInterrupt}(\mathrm{po},w,r,w')) \end{array} \right) \end{array} \right)$$

$$e_{\mathrm{conseq\_xo}} = \bigwedge_{w(i)\in W_\varepsilon, r(i)\in R_\varepsilon} \left(\begin{array}{l} \mathrm{Conseq_{xo}}(w(i),r(i)) \ \textbf{iff} \\ \left(\begin{array}{l} \mathrm{xo}(w(i),r(i)) \wedge \bigwedge_{w'(i)\in W_\varepsilon \setminus \{w(i)\}} \\ (\mathrm{conflict}(w',r) \implies \mathrm{notInterrupt}(\mathrm{xo},\mathrm{w(i)},\mathrm{r(i)},\mathrm{w'(i)})) \end{array} \right) \end{array} \right)$$

Note that function $\mathrm{notUnterrupt}(rel,x,y,z)$ is used to check whether z does not appear between x and y in relation rel.

$$\mathrm{notInterrupt}(rel,x,y,z) \ \textbf{iff} \ (rel(z,x) \wedge rel(z,y)) \ \textbf{xor} \ (rel(x,z) \wedge rel(y,z))$$

Therefore, the formula for $\text{Cond}_2^{\varepsilon}$ is defined as

$$\text{Cond}_2^{\varepsilon} = e_{xo\_r} \wedge e_{W_{\text{po}}} \wedge e_{W_{xo}} \wedge e_{\text{conseq\_po}} \wedge e_{\text{conseq\_xo}}$$

For condition 3, atomicity of read-modify-write operation, $\text{Cond}_3^{\varepsilon}$ is then defined as the following equation.

$$\text{Cond}_3^{\varepsilon} = \bigwedge_{w(i),w'(j)\in W_{\varepsilon},r(i)\in R_{\varepsilon}} \left(\begin{array}{l} \text{rmw}(r(i), w(i)) \wedge \text{pid}(w') \neq \text{`pid}(w) \implies \\ \text{notInterrupt}(\text{xo}, r(i), w(i), w'(i)) \end{array} \right)$$

Definitions of significant orders: Each specification in [Gha95] usually has its own definitions for specific orders, such as $\xrightarrow{spo}$ and $\xrightarrow{sco}$, for significant program order and significant conflict order, respectively. These orders are usually used to consider the valid execution order in the conditions. However, each model has different definitions of these orders and conditions. Constraints should be defined independently for each model. For example, the definition of $\xrightarrow{spo''}$ of TSO in Figure 2-10 is defined by formula $\text{TSO}_{spo''}$.

$$\text{TSO}_{spo''} = \bigwedge_{x(i),y(j)\in RW_{\varepsilon}} \left(\begin{array}{l} \text{spo}''(x, y) \textbf{ iff } (\\ (x(i) \in R_{\varepsilon} \wedge \text{po}(x, y))\vee \\ (x(i), y(j) \in W_{\varepsilon} \wedge \text{po}(x, y))\vee \\ (x(i) \in W_{\varepsilon} \wedge y(i) \in R_{\varepsilon}\wedge \\ \quad \exists m \in \text{MemWR}_{\varepsilon}.(\text{po}(x, m) \wedge \text{po}(m, y)))) \end{array} \right)$$

where MemWR is the set of memory barrier events issued by SPARC instruction such that $\text{MemWR} \subseteq \text{Fence}_{\varepsilon}$.

Conditions on execution orders: Lets consider the condition $W \xrightarrow{sco} R \xrightarrow{spo} \{A \xrightarrow{sco} B \xrightarrow{spo}\} + R$. The condition consists of a transitive relation $x\{\xrightarrow{sco} A \xrightarrow{spo}\} + y$ that can be formalized as a relation function loop by the following formula.

$$\text{TSO}_{loop} = \bigwedge_{x(i),y(j)\in RW_{\varepsilon}} \left(\begin{array}{l} \text{loop}(x, y) \textbf{ iff } (\\ (\exists z(k) \in RW_{\varepsilon}.(\text{sco}(x, z) \wedge \text{spo}(z, y)))\vee \\ (\exists z(k) \in RW_{\varepsilon}.(\text{loop}(x, z) \wedge \text{loop}(z, y)))) \end{array} \right)$$

Condition $W \xrightarrow{sco} R \xrightarrow{spo} \{A \xrightarrow{sco} B \xrightarrow{spo}\} + R$ is then formalized using loop using the following formula.

$$\text{TSO}_{\text{mDep3}} = \bigwedge_{x(i)\in W_{\varepsilon},y(i),z(j)\in R_{\varepsilon},a(k)\in RW_{\varepsilon}} \left(\begin{array}{l} \text{conflict}(x, y) \wedge \text{sco}(x, z)\wedge \\ \text{spo}(z, a) \wedge \text{loop}(a, y) \implies \text{xo}(x(i), y(i)) \end{array} \right)$$

Encoding Scheme of Herding Cats Framework

In this framework, the communication between events in the system is a concern to determine the data flow of the read accesses and the write accesses. In contrast to the previous framework, herding cats framework uses *relations* instead of *orders*. Besides, as the events are directly used to determine the data flow, the completing order of each event

is not realized explicitly in this framework. However, it is sufficient to consider the effect of program executions for program verification.

In the encoding method, *predicated functions* are used to represent the relations on the existing events. In the constraint of this framework, the basis relations are provided for all constraints, such as read-from relation, and additional relations for relaxed memory models could be constructed from the basis relations. Then, the constraints are considered as the property on those relations, such as acyclic of a relation.

Given even state ε, property condition p, assertion condition a and memory model $\mathcal{M}$, the encoded formula is of the form

$$\text{Basis}_\varepsilon \wedge \text{Cons}^\varepsilon_{\mathcal{M}} \wedge p \wedge \neg a$$

where Basis_ε represents the *basis relations* on event state ε and $\text{Cons}^\varepsilon_{\mathcal{M}}$ represents *constraints* of memory model $\mathcal{M}$ for event state ε. Note that, although the form of encoded formula is similar to the previous framework, the way to encode relies on relations of existing events.

According to cats specifications [ACM16], the constraints rely on the relations of existing events. Besides, there might be additional relations for the constraints in formula $\text{Cons}^\varepsilon_{\mathcal{M}}$. Our approach also provides the way to constructs those additional relations for realizing a specification in term of first-order formula.

Basis Relations Given event state $\varepsilon = \langle s, po, iico, rmw\rangle$, every read or write event are considered as a single-event, which is different to Gharachorloo framework. However, each event also has its processor identifier, which can be analyzed as same as Gharachorloo framework. In addition, read and write events are categorized for registers and shared-memory to distinguish the behavior, in which there are the following basic sets.

Definition 3.39 (Basic sets for Herding Cats framework)**.**

$$\begin{aligned}
\text{W}_\varepsilon &= \{w \mid \varepsilon = \langle s' \cup \{w\}, po, iico, rmw\rangle \wedge w \text{ is a write event to shared memory}\} \\
&\quad \cup \{w^0_L \mid w^0_L \text{ writes an initial value to share memory } L \in \text{Loc}\} \\
\text{R}_\varepsilon &= \{r \mid \varepsilon = \langle s' \cup \{r\}, po, iico, rmw\rangle \wedge w \text{ is a read event to shared memory}\} \\
\text{Wreg}_\varepsilon &= \{w \mid \varepsilon = \langle s' \cup \{w\}, po, iico, rmw\rangle \wedge w \text{ is a write event to register}\} \\
&\quad \cup \{w^0_R \mid w^0_R \text{ writes an initial value to register } R \in \text{Reg}\} \\
\text{Rreg}_\varepsilon &= \{r \mid \varepsilon = \langle s' \cup \{r\}, po, iico, rmw\rangle \wedge w \text{ is a read event to register}\}
\end{aligned}$$

where w^0_L and w^0_R are write events to shared memory L and register R, which usually write 0 and have processor identifier 0.

The additional writes w^0_L and w^0_R are used to realize the initial value of every target access because the read value of read events relies on a relation from the existing write events, represented by read-from relation $\xrightarrow{rf}$.

Besides, uninterpreted formulas are also used in this encoding to represent relations in an encoded formula. First four basis uninterpreted functions are po, po$-$loc, iico, and

rmw, which are abstracted by the following formulas.

$$
\begin{aligned}
e_{po} &= \bigwedge_{x,y \in \mathrm{Ev}_\varepsilon} (\mathrm{po}(x,y) \ \textbf{iff} \ (x,y) \in po) \\
e_{po\text{-}loc} &= \bigwedge_{x,y \in \mathrm{Ev}_\varepsilon} (\mathrm{po}{-}\mathrm{loc}(x,y) \ \textbf{iff} \ (\mathrm{po}(x,y) \wedge \mathrm{Loc}(x) = \mathrm{Loc}(y))) \\
e_{iico} &= \bigwedge_{x,y \in \mathrm{Ev}_\varepsilon} (\mathrm{iico}(x,y) \ \textbf{iff} \ (x,y) \in iico) \\
e_{rmw} &= \bigwedge_{x,y \in \mathrm{Ev}_\varepsilon} (\mathrm{rmw}(x,y) \ \textbf{iff} \ (x,y) \in rmw)
\end{aligned}
$$

where po$-$loc represents the program order of events that accesses to the same location.

To realize the read value of a read event, read-from relation $\xrightarrow{rf}$ is used to represent the data flow from a write event to the read event. The relation considers all conflicting writes, represented by $\mathrm{cf}(W,r)$ from set of write event and conflict with read event r. The representation of read-from relation $\rightarrow rf$ is abstracted by uninterpreted function rf, which is defined by the following formulas.

$$
\begin{aligned}
e_{rf} &= \bigwedge_{r \in R_\varepsilon} \left(\bigvee_{w \in \mathrm{cf}(W_\varepsilon, r)} (\mathrm{rf}(w,r)) \right) \wedge \bigwedge_{e1 \in \mathrm{Ev}_\varepsilon \setminus R_\varepsilon} (\neg \forall e. \neg(\mathrm{rf}(e, e1))) \\
e_{rf\text{-}val} &= \bigwedge_{r \in R_\varepsilon, w \in \mathrm{cf}(W_\varepsilon, r)} (\mathrm{rf}(w,r) \implies \mathbb{R}_{\mathrm{val}}(r) = \mathbb{W}_{\mathrm{val}}(w)) \\
e_{rfe} &= (\mathrm{rfe}(x,y) \ \textbf{iff} \ (\mathrm{rf}(x,y) \wedge \mathrm{pid}(x) \neq \mathrm{pid}(y))) \\
\mathrm{cf}(W,r) &= \{w \mid w \in W \wedge \mathrm{Loc}(w) = \mathrm{Loc}(r)\}
\end{aligned}
$$

Note that uninterpreted function rfe is also defined to indicate the read-from relation that the value is transferred across processors.

There is also conflicting order definition in this encoding approach, which relies on the relations instead of execution order. Uninterpreted functions co and coe are used to indicate conflicting order and inter-conflicting order, which are defined by following equations.

$$
\begin{aligned}
e_{co} &= \bigwedge_{x,y \in \mathrm{Ev}_\varepsilon \setminus \mathrm{Wsc}_\varepsilon} (\mathrm{co}(x,y) \ \textbf{iff} \ (x \neq y \wedge x,y \in W_\varepsilon \wedge \neg \mathrm{co}(y,x) \wedge \mathrm{pid}(y) \neq 0)) \\
e_{coe} &= \bigwedge_{x,y \in \mathrm{Ev}_\varepsilon} (\mathrm{coe}(x,y) \ \textbf{iff} \ (\mathrm{co}(x,y) \wedge \mathrm{pid}(x) \neq \mathrm{pid}(y)))
\end{aligned}
$$

where Wsc_ε is the set of write events issued by store-condition operations, such that $\mathrm{Wsc}_\varepsilon = \{(eid, \mathbf{sc}(v_1, loc, v_2)) \in \mathrm{W}_{ev} \mid eid \in \mathrm{Eid} \wedge v_1 \in \mathrm{Tmp} \wedge v_2 \in \mathrm{Tmp}\}$.

Moreover, there is a relation from-read, denoted by $\xrightarrow{fr}$, to indicate the conflicting write events following the read event. This relation is used for basic information to build other relations for constraining the behavior. The relation is then defined by the following

formulas. Note that uninterpreted function fre also appear to indicate the relation across processors.

$$e_{fr} = \bigwedge_{x,y \in \mathrm{Ev}_\varepsilon} (\forall e1, e2, e3 \in \mathrm{Ev}_\varepsilon((\mathrm{rf}(e2, e1) \wedge \mathrm{co}(e2, e3)) \implies fr(e1, e3)))$$

$$e_{fre} = \bigwedge_{x,y \in \mathrm{Ev}_\varepsilon} (\mathrm{fre}(x, y) \textbf{ iff } (\mathrm{fr}(x, y) \wedge \mathrm{pid}(x) \neq \mathrm{pid}(y)))$$

To realize the semantics of the store-condition operation, the following formulas are provided to consider the behavior in which the write accesses provided by store-condition operations can arbitrarily fail.

$$e_{sc} = \bigwedge_{w \in \mathrm{Wsc}_\varepsilon} (r = \mathrm{syncR}(w, \varepsilon) \implies (sc\_cond_1(w, r) \vee sc\_cond_2(w, r)))$$

$$sc\_cond_1(w, r) \textbf{ iff } (\forall wp \in \mathrm{W}_{ev}.(\mathrm{rf}(wp, r) \wedge w \neq wp \implies (\forall w' \in \mathrm{W}_{ev}.(\mathrm{co}(wp, w') \wedge w \neq w' \implies \mathrm{co}(wp, w) \wedge \mathrm{co}(w, w'))))) \wedge (\mathbb{S}[\![w]\!] = 0)$$

$$sc\_cond_2(w, r) \textbf{ iff } (\forall w' \in \mathrm{W}_{ev}.(\neg\mathrm{co}(w', w) \wedge \neg\mathrm{co}(w, w')) \wedge \forall r' \in \mathrm{R}_{ev}.(\neg\mathrm{rf}(w, r')) \wedge \mathbb{S}[\![w]\!] = 0)$$

where $\mathrm{syncR}(w, \varepsilon)$ is the recent load-link event in the program order prior write event w.

Consequently, the basis relations include uninterpreted functions: po, po$-$loc, iico, rmw, co, coe, rf, rfe, fr, and fre. These functions are abstracted in first-order formula $\mathrm{Basis}_\varepsilon$, such that

$$\mathrm{Basis}_\varepsilon = e_{po} \wedge e_{po\text{-}loc} \wedge e_{iico} \wedge e_{rmw} \wedge e_{co} \wedge e_{coe} \wedge e_{rf} \wedge e_{rf\text{-}val} \wedge e_{rfe} \wedge e_{fr} \wedge e_{fre} \wedge e_{sc}$$

Additional Relations According to specifications for herding cats framework, there are additional relations to realize the behavior regarding a memory model. However, those relations are basically constructed from the basis relations. For instance, relation $\xrightarrow{fre;coe}$ appearing in Figure 2-12 is constructed by concatenation of relations $\xrightarrow{fre}$ and $\xrightarrow{coe}$. Thus, to construct additional relations, there are four operators are used: $\cup$ (union), ; (sequence), $\cap$ (intersection), and $\setminus$ (set difference). The following formulas are used to define those operations on relations $r1$ and $r2$. Note that these definitions are just guidance for defining additional relations on specific specification.

$$e_{union} = \bigwedge_{x,y \in \mathrm{Ev}_\varepsilon} (union(x, y) \textbf{ iff } (r1(x, y) \vee r2(x, y)))$$

$$e_{seq} = \bigwedge_{x,y \in \mathrm{Ev}_\varepsilon} \left(seq(x, y) \textbf{ iff } (\bigvee_{z \in \mathrm{Ev}_\varepsilon} (r1(x, z) \wedge r2(z, y))) \right)$$

$$e_{intersec} = \bigwedge_{x,y \in \mathrm{Ev}_\varepsilon} (intersec(x, y) \textbf{ iff } (r1(x, y) \wedge r2(x, y)))$$

$$e_{diff} = \bigwedge_{x,y \in \mathrm{Ev}_\varepsilon} (diff(x, y) \textbf{ iff } (r1(x, y) \wedge \neg r2(x, y)))$$

Moreover, most of the modern memory models usually use memory barrier or fence operation to prevent anomalous behaviors. In specification for herding cats framework,

the relation of two events are ordered by a fence operation is usually used. For instance, $a \xrightarrow{\text{dmb}} b$ represents event a is ordered before b, in which there is event dmb between those events. To define an additional relation to indicate such behavior, the following formula can be adopted to define uninterpreted function for fence $\mathcal{F}$, where f is a fence instance $f \in \mathcal{F}$ and $\mathcal{F} \subseteq \text{Fence}_\varepsilon$.

$$e_{fence} = \bigwedge_{x,y \in \text{Ev}_\varepsilon} \left(\mathit{fence}(x,y) \textbf{ iff } (\bigvee_{f \in \mathcal{F}} (\text{po}(x,f) \wedge \text{po}(f,y))) \right)$$

Constraints Relations Given event state $\varepsilon = \langle s, po, iico, rmw \rangle$ and the specification of memory model $\mathcal{M}$, the relations of ε represented by basis relations and additional relations must be constrained to realize the valid read values of read events. The way to constrain the relations in specifications for herding cats framework relies on constraints: *acyclic*, *irreflexive*, and *empty*.

Acyclic constraint ensures there is no loop of the relation can occur in a system. To check the relation, addition relation *trans* is used to represent relation r, instead of check the relation directly. Then, the transitive closure of relation *trans* is introduced by a formula. Finally, the acyclic constraint checks there is no irreflexive of relation *trans* occur in a system. Thus, the following formulas are used to represent the acyclic constraint.

$$\begin{aligned}
e_{acyclic\text{-}1} &= \bigwedge_{x,y \in \text{Ev}_\varepsilon} (r(x,y) \implies \mathit{trans}(x,y)) \\
e_{acyclic\text{-}2} &= \forall x,y,z \in \text{Ev}_\varepsilon.((\mathit{trans}(x,y) \wedge \mathit{trans}(y,z)) \implies \mathit{trans}(x,z)) \\
e_{acyclic\text{-}3} &= \forall e.\neg(\mathit{trans}(e,e))
\end{aligned}$$

For irreflexive constraint and empty constraint, the following formulas can be used directly for any relation r.

$$\begin{aligned}
e_{irreflexive} &= \forall e.\neg(r(e,e)) \\
e_{empty} &= \forall x,y.\neg(r(x,y))
\end{aligned}$$

3.4 Conclusions

This chapter proposes an SMT-based program verification method to verify concurrent programs executed on a multiprocessor system. Primarily, the effect of programs caused by a relaxed memory model is considered to verify the program property. In this method, we can deal with a variety of assembly languages and the program behavior on a variety of relaxed memory models. In particular, the contributions of this method are (1) the abstraction of concurrent programs to deal with a variety of assembly language and (2) an SMT-based method to ensure the program correctness based on the specification of target memory model.

3.4.1 Achievements

Among assembly languages, an abstraction, called *operation structure*, is introduced to capture the essential behavior of an assembly program for program verification on a relaxed memory model. In this abstraction, *operations* are assumed to be granules of programs to be performed by a processing unit, and the way to perform those also represented by the abstraction, such as parallelisms of performing and unstructured programming of assembly language. The considered operations are the side effects of assembly instructions that could change the computation of the programs because of relaxed memory models. Mainly, the effects of memory instructions are the primary concern of this research, while the effect of instructions such as interruptions is excluded in our abstraction. However, as the target program property to be verified is the effect caused by relaxed memory models, our abstraction is sufficient to capture those effect for program verification.

In SMT-based program verification, the corresponding program executions are abstracted by *execution path*. To ensure the program correctness, the program property of every execution path must be verified with the behavior of the execution path on target memory model. Thus, the encoding scheme is provided to represent the target verification property as a first-order formula to be used in an SMT solver.

In general, an assembly program usually has branch instructions and predicate instructions to decide the control flow based on the computation of the program. Especially, the computation is also affected by relaxed memory models. To explore the corresponding execution paths without considering the computation, the *bounded loop unwinding method* is introduced for the programs to restrict the number of state-space if programs contain loops.

To provide a first-order formula for program verification, the specification of target memory models is adopted as a standard description of memory model to realize the effect of program execution. In this research, two encoding schemes are proposed based on Gharachorloo framework [Gha95] and Herding cats framework [AMT14]. Although the program behavior of each framework is different from each other, the effect of program executions can be realized based on those behaviors.

In summary, by given a sequence of operation structures corresponding to target concurrent programs, the method can *automatically* verify the program property of the programs on target memory model. By using bounded loop unwinding, the corresponding execution paths can be explored automatically. The coverage of the provided execution paths can capture all program executions if there is no loop in the programs, while the executions are assumed to be eventually terminated if there is a loop. For encoding method, a given execution path is also encoded as a first-order formula to be verified automatically by an SMT solver. As a result, if there is a valuation found by the solver, the effect of the valuation can *disprove* the correctness of the assembly programs. On the other hand, the program correctness is ensured if there is no valuation founded by the solver for any execution path of *programs containing no loop*. Otherwise, if the programs contain at least one loop, the correctness cannot be ensured because the execution of the program is bounded.

3.4.2 Limitations

Due to the consistency of shared-memory of a relaxed memory model, the global variables cannot be expressed in our assertion language. Although the local variables can be used to express safety property of the programs, the expressiveness of the program property is limited as the property of concurrent programs can rely on the effect on global variables. For instance, the mutual exclusion property usually uses a global counter to count the number of process entering its critical section.

In addition, if the programs contain a loop, a bounded method is used to consider a finite number of execution paths. Besides, the loop is assumed to be eventually terminated to consider a finite number of events in SMT-based program verification. Thus, the bounded method cannot ensure the program correctness of programs contain at least one loop, such that (1) each loop is eventually terminated and (2) the number of execution paths is finite. However, some programs could use a local variable to determine the number of loop iterations, such as

```
1 i := 0;
2 do{
3   // do something
4   i := i + 1;
5 }while(i < 10)
```

ones could be able to determine the number of loop iterations explicitly if the local variables used for loop condition are not affected by other programs.

Chapter 4

Inductive Invariant Method for SMT-based Program Verification

4.1 Motivation

In our SMT-based program verification, an SMT solver is adopted to find a valuation of free variables appearing in the given formula. Due to SMT solvers cannot handle the infinite number of free variables, the previous method proposed in Chapter 3 assumes the corresponding executions are eventually terminated. In addition, the bounded method described in Chapter 3 has been used to systematically analyze the corresponding executions to be verified in SMT-based program verification. Obviously, the bounded method limits the corresponding executions to be verified by using a bound if there is a loop in the program. Consequently, the program property is then verified automatically.

According to our bounded method proposed in Chapter 3, if the number of loop iterations can be determined systematically, the program correctness can be ensured automatically using the number of loop iterations as a bound. Regarding the loop behavior, the branch condition that causes a loop relies on the local variables. If the local variables used in the loop condition are not affected by other programs, the bounded method can be used due to the number of iterations could be considered beforehand. However, if the local variables can be affected by the computation of other processing units, the number of iterations could not be determined beforehand, and the infinite iterations can occur. As our research focuses on the program verification for concurrent programs, the latter effect must be considered, in which the number of iterations could be infinite.

Due to the limitation of the bounded method, this method adopts the inductive invariant approach using in software verification [DHKR11], which considers a sequential program. Our work intends to use loop invariant to abstract the infinite iterations for program verification on relaxed memory models. In addition to the effect of loop iteration for the following iterations, the effect of write events to other processing units is also captured in our method. Consequently, the effect of infinite iterations is abstracted to produce finite events for program verification.

In this chapter, first, the overview of proposing method is explained to show the overall

idea to abstract the behavior for SMT-based program verification. Then, the abstractions of program executions for this method is introduced which is a modified version of the abstraction for the bounded method. After that, the inductive invariant method is proposed using transform functions instead of control flow graphs used in Chapter 3.

4.2 Overview of Inductive Invariant Method

To use the loop invariant for abstracting the loop behavior, the description of a loop behavior is of the form $\mathbf{do}\{\gamma\langle inv\rangle\}\mathbf{while}(c)$ where γ is the loop body, inv is the invariant condition and c is the loop condition. In particular, the program is required to be described in structured programming style, in which the branch is not allowed to jump into/out a loop. Then, the invariant condition is provided to each loop to abstract the behavior of the loop. Thus, an operation structure used in the method must not allow unstructured programming style, and the invariant has to be placed explicitly for each loop.

Due to the limitation of the bounded method, the inductive invariant method focuses on the loop behavior that causes infinite loop iterations for program verification. Especially, the loop condition that relies on the computation of other programs to exit the loop is considered in this method. For instance, the following loop behavior waiting for the result of memory location `[X]` becomes 1 to exit the loop.

```
1 do{
2   v:=[X]
3 ⟨v = 1 ∨ v = 0⟩
4 }while(v = 0);
```

In the loop behavior, the infinite iterations can occur, which is the target behavior to be abstracted in our inductive invariant method.

4.2.1 Issues for Program Verification

As the original approach [DHKR11] is proposed for a sequential program, the invariant is expected to abstract the loop iterations for the next iteration for considering arbitrarily. For instance, the following program provides the invariant for the loop.

```
1 do{
2   v:=v+1
3 ⟨v ≤ 4⟩
4 }while(v < 4)
```

Invariant condition $v \leq 4$ is used to ensure the value of local variable v always less than or equal 4. In original approach uses this fact to provide the following description to simulate an arbitrary loop iteration.

```
1 v := *;
2 assume(v ≤ 4);
3 v:=v+1
```

where `v := *` is an assignment that $*$ refers to an arbitrary value and then **assume**$(v \leq 4)$ constrains the arbitrary value to be in the scope described by the invariant condition. These two statements are expected to abstract the effect of previous iterations using invariant and statement `v:=v+1` is then an arbitrary behavior of a loop iteration. Although this is an appropriate approach for verifying a sequential program using SAT/SMT-solvers, the concurrent programs cannot be abstracted by this approach directly because the information could be lost.

```
1 v:=0;
2 do{
3   v := v+1;
4   [X] := v;
5   ⟨v ≤ 4⟩
6 }while(v < 4);
```

```
1 u:=[X];
2 v:=[X];
3 assert(u < v);
```

Figure 4-1: Concurrent Programs

For instance, Figure 4-1 shows concurrent programs containing a loop and another program has two read operations to memory location `[X]`. By inductive invariant approach, an arbitrary loop iteration would be described as

```
1 v := *
2 assume(v ≤ 4);
3 v:=v+1;
4 [X] := v;
```

Consequently, the read events issued by another program can see only the effect of a single write event from this arbitrary iteration. This means the write events are missing due to the abstraction. Thus, our method considers this issue for program verification of concurrent programs.

In addition to the missing information, the program verification for relaxed memory models has another concern that the read events and write events could be completed out-of-order. This means some relaxed memory models would allow events issued after a loop iteration could be completed before the events issued by the iteration. For instance, POWER allows a read operation to be delayed if it is independent to the following operations. If the target program verification is the following:

```
1 v:=0;
2 do{
3   v := [X]
4 ⟨v ≤ 5⟩
5 }while(v < 5);
6 [Y] := 10
```

Intuitively, write operation `[Y] := 10` can be completed before `v:=[X]` in any loop iteration due to it is independent to the read operation. Consequently, if there is another program that gets the write value of `[Y] := 10` and reflect the result to read operation `v:=[X]`, the invariant condition is not satisfied for this situation. This means the abstraction of loop behavior would be affected by the following loop iterations or the memory operations appearing after the loop.

4.2.2 Overview of Method for Relaxed Memory Models

Regarding a loop behavior, the loop body can be described arbitrarily regarding its invariant condition to abstract the effect of previous loop iterations. Besides, as concurrent programs are the target of our method, the abstraction must also cover the effect of the other concurrent programs which can see the effect of each iteration. In our case, the effect to other programs corresponds to the write operations inside the loop body. Thus, the effect of write events issued by infinite iterations must be abstracted for other programs appropriately. Besides, the motivation of this method is to provide the first-order formula for SMT-based program verification automatically. Thus, the abstraction of infinite iterations is expected to be generated automatically.

```
1 γ1;
2 do{
3   v := [Y];
4   [X] := ψ(v);
5   ⟨inv⟩
6 }while(ϕ(v));
7 γ2;
```

```
1 r1:=[X];
2 r2:=[X];
3 ...
4 rk:=[X]
```

Figure 4-2: Infinite Loop Programs with k reads

Although infinite iterations are allowed to occur, there is an assumption that k loop iterations would be sufficient for program verification if there are k read events issued by other processors. For instance, Figure 4-2 shows the concurrent programs where the loop body writes the value which is calculated from $\psi(v)$ to memory location `[X]` and there are k read operations accessing to memory location $[X]$ on another program. In program verification on relaxed memory models, the program execution is assumed to be eventually terminated. This means there are i iterations to instantiate the loop body where $i \geq 0$. However, as there are only k read events to memory location `[X]` issued by

the programs, the number of write events to memory location [X] is at most k in which the read events get the values from different write events. Also, according to Figure 4-3, although there are i^{th} iterations that can occur, at most k iterations would be considered to determine the computation on other programs as there are only k read events.

```
1  γ1;
2  // iteration 0
3  v := [Y];
4  [X] := ψ(v);
5  assume(ϕ(v));
6  ...
7  // iteration 1
8  assume(ϕ(v));
9  v := [Y];
10 [X] := ψ(v);
11 ...
12 // iteration k
13 assume(ϕ(v));
14 v := [Y];
15 [X] := ψ(v);
16 ...
17 // iteration i
18 assume(ϕ(v));
19 v := [Y];
20 [X] := ψ(v);
21 assume(¬ϕ(v));
22 γ2
```

```
1 r1:=[X];
2 r2:=[X];
3 ...
4 rk:=[X]
```

Figure 4-3: k iteration with k reads

According to the assumption, the infinite iterations for k read events are expected to abstracted as an arbitrary effect of k iterations that satisfies the loop invariant. The following description shows the target abstraction which abstracts the effect of write events from $k+1$ loop iterations where $inv(v, w_x)$ is the invariant condition that relies on the value of local variable v and write value w_x.

```
// arbitrary iteration
v := *;
assume(φ(v) ∧ inv(v, w_x));
// iteration 1
[X] := w_x;
assume(φ(v) ∧ inv(v, w_x));
...
// iteration k+1
[X] := w_x;
assume(φ(v) ∧ inv(v, w_x));
v := [Y];
[X] := ψ(v);
```

By the assumption, the number of the read events on other programs is the concern to abstract a loop iteration. Besides, iteration $(k+1)^{\text{th}}$ is used to abstract the prior iteration for the read events in an arbitrary loop iteration. However, if there is a loop on other programs, the number of events could be infinite that also affects the number of abstraction.

Regarding the number of the read events caused by loop behaviors, the infinite number of the read events would be caused by the behavior that waits for the read values from outside not to satisfy the loop condition. In the purpose of program verification, arbitrary behavior of loop body is considered in which the read value can be affected by outside memory events. Thus, the number of the read events to be considered caused by the loop equals the number of the read operations. For instance, the following description is an arbitrary loop iteration of $\mathbf{do}\{v := [X]; u := [Y]\langle inv(v,u)\rangle\}\mathbf{while}(\phi(v,u))$.

```
// arbitrary effect
v := *;
u := *;
assume(φ(v, u) ∧ inv(v, u));
// arbitrary iteration
v := [X];
u := [Y];
```

If there is a loop on another program to write values to memory locations `[X]` and `[Y]`, at most two loop iterations can be considered to consider this arbitrary value of this arbitrary loop iteration.

Given concurrent programs $P = P_1 \cdot Q$ where $Q = P_2 \cdot \ldots \cdot P_n$ and each P_i is executed on processing unit i such that $1 \leq i \leq n$. Let P_1 be $\gamma_1; \mathbf{do}\{\gamma_s\langle inv\rangle\}\mathbf{while}(c); \gamma_2$ and Q be arbitrary programs that can read and write to the same memory locations that P_1 uses. Thus, the method derives the control flows to be verified regarding the loop invariant in P_1 and the number of read events that access the same memory locations that γ_s access. For simplicity, the control flow graph shown in Figure 4-4 derives the execution of P_1 regarding invariant condition inv where arb_v is the arbitrary assignments of local variables and arb_w^i is the arbitrary assignment of memory locations for iteration i. In

addition, k is the number of read events appearing in Q that access the same memory locations written by the write operations in the loop body.

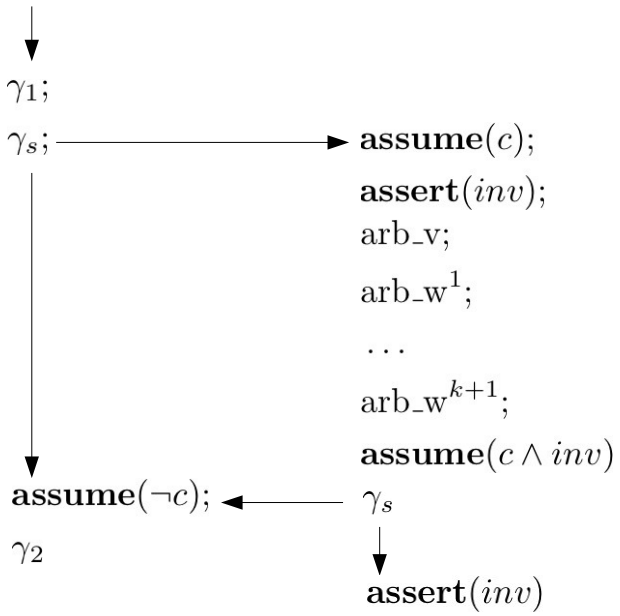


Figure 4-4: Overview of Inductive Invariant Method

For the arbitrary assignment of local variables arb_v, we collect the assigning target of the read operation and load-link operation to the local variables. Those local variables are assigned by arbitrary values for considering the arbitrary effect from the previous iteration. Then, assumption **assume**($c \wedge inv$) is used to restrict the scope of assigning values to those local variables.

For the arbitrary assignment of memory locations arb_w, we consider the target assignment of write operations and store-condition operations to the memory locations. Due to the behavior of store-condition operation, there is a possibility the write event issued by the operation can fail. Besides, the sequence of write operations would be the matter because of the behavior of store-condition operations. Thus, the possible ways to issue the write events from the loop body are considered for the arbitrary assignment of memory locations arb_w. For instance, the following description illustrates the loop body to be abstracted for arb_w.

```
1 // loop body
2 u := v+z;
3 [X] := u;
4 if(z = 1){
5   sc(z, [X], u);
6   [X] := v;
7 }
```

Figure 4-5 shows the control flow graph by analyzing the possible ways to issue the write events regarding the program order. Note that there are restrictions in this method that

(1) the local variable for assigning the result of store-condition operation must not be used in the read operations and load-link operations, and (2) the loop behavior is not allowed as a nested loop. Then, the arbitrary assignment to memory locations must consider every case to issue the write events.

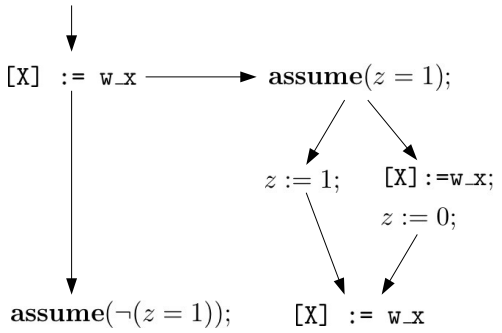


Figure 4-5: Control flow graph for considering arbitrary assignments to memory locations

To summarize, each control flow derived from the graph is then considered regarding the target memory model as the same as the bounded method. In this method, the way to abstract the effect of infinite iterations is the primary concern for program verification on relaxed memory models. This means after the loop behavior is abstracted, the encoding method for program verification using SMT-solver is the same. However, as this method also use the invariant condition to restrict the scope of write values to memory locations, the assertion language is also improved to support such requirement.

Nevertheless, the proposed inductive invariant method did not take the memory events occurring after a loop iteration into account. This means this method would not be sound for any relaxed memory models. However, as partial store ordering (PSO) doesn't allow a read operation to be delayed; thus, the method is sound for PSO and stronger memory models.

4.3 Abstractions for Program Execution

In the inductive invariant method, a program to be considered must be described by structured programming style as so not to allow jump into/out a loop which could violate the loop invariant. In the structured programming, the control flows of a program are usually determined by if-branch and loop-branch explicitly, while unstructured programming allows branch statements to jump onto any place in the program which is the usual behavior of an assembly program.

In this section, the operation structure is modified to be described in a structured programming style to support inductive invariant method. Although the behavior of an assembly program would not be captured directly, the infinite loop iterations can be abstracted for program verification on relaxed memory models. Then, the executions produced derived by the modified operation structure is described for program verification.

4.3.1 Abstractions of Assembly Programs

First of all, the inductive invariant method expects the invariant conditions to define the scope of local variables and write values of a loop iteration. However, the boolean expression used in the bounded method covers only the scope of local variables. Thus, definition 4.1 extends the boolean expression to determine the write value of the recent write operation in the program order. For instance, the following description uses an assertion expression as the assumption condition to restrict the value of u and v such that $v = 3 \wedge u > 2$.

```
1 [X] := 2;
2 [X] := u;
3 [Y] := v;
4 assume( [Y] = 3 ∧ [X] > 2 )
```

Note that the intention to indicate memory locations in an assertion expression is not to ensure the value of the memory locations because the value of global variables cannot be determined due to relaxed memory models. In this description, the memory locations in an assertion expression are used to indicate the recent write operations appearing before the condition. Then, this expression can be used in the inductive invariant method.

Definition 4.1 (Assertion Expression)**.** the set Aexp is the smallest set X with the properties:

1. $\top, \bot \in X$,
2. $e_1, e_2 \in \text{Exp} \cup \text{Loc}$ implies $(e_1 = e_2), (e_1 < e_2), (e_1 > e_2) \in X$,
3. $\varphi, \psi \in X$ implies $(\varphi \wedge \psi), (\varphi \vee \psi) \in X$,
4. $\varphi \in X$ implies $\neg(\varphi) \in X$

where $=, <, >, \wedge, \vee$, and $\neg$ are connectives, (and) are auxiliary symbols.

To describe a loop invariant condition, we modify the abstraction of an operation structure described in Section 3.2 to distinguish the behavior of if-branch and loop-branch explicitly. In particular, we introduce *control flow structure* shown in Definition 4.2 to indicate the if-branch and loop-branch explicitly.

Definition 4.2 (Control Flow Structure)**.** Let γ be an execution structure, $inv \in$ AExp be an assertion expression and $c \in$ BExp be a boolean expression, a control flow structure is either: (1) Forward branch **ifBr**$(c)\{\ \gamma\ \}$ or (2) Loop branch **do**$\{\gamma\langle inv\rangle\}$**while**$(c)$.

For the forward branch, description **ifBr**(c)$\{\gamma\}$ is used to indicate branch condition c as a condition for performing the operations appearing in γ if the evaluation of the condition is satisfied. Otherwise, the operations appearing in γ are not performed.

For the backward branch, description **do**$\{\gamma\langle inv\rangle\}$**while**$(c)$ is used to describe the loop behavior such that operations appearing in γ are performed once and check condition c to

```
1 instr{
2   val_z := z
3 };
4 ifBr(¬(val_z = 1)){ //beq
    L
5   instr{
6     val := [X]
7     r1 := val
8   }
9 } // L
```

(a) Forward branch behavior

```
1 do{ // L
2   instr{
3     val := [X]
4     r1 := val
5   };
6   instr{
7     val_z := z
8   }
9 }while(val_z = 1) // beq L
```

(b) Backward branch behavior

Figure 4-6: Examples of branch behavior

repeat the performing those operations again. As for invariant condition inv, this boolean expression is used for the inductive invariant method later. Note that this description intends to represent the behavior of label and branch that cause a loop in an assembly program directly. For instance, Figure 4-6 shows the difference between a forward branch and a backward branch of instruction `beq L` where label `L` is located in different places.

According to control flow structure, the branch operation should not be used in the operation structure, which is already abstracted by if-branch and loop-branch. Thus, the elements of an operation structure for the inductive invariant method must include control flow structure and exclude branch operation $\mathbf{branch}(c, l)$ and label $\mathbf{label}(l)$. Figure 4-7 shows the corresponding operation structures of the message passing programs in Figure 1-2 for the inductive invariant method. In contrast to Figure 3-7, the label and brach are replaced by loob branch explicitly.

4.3.2 Execution of Operation Structures

For program executions, an event state defined in Chapter 3 is also used to represent the abstraction of issued events to the system. However, as the description of operation structures for the inductive invariant method is in structured programming style, the control flow graph of an operation structure is not necessary for considering the possible issued operations into a system. Thus, instead of constructing a control flow graph, the *transform function* of operation structures is proposed directly to expand the possible executions regarding the loop unwinding approach. Consequently, we define transformation function to transform a set of states based on program description to generate the corresponding set of states that represent all symbolic executions directly based on the program description. For instance, let a state to be considered be a formula, such as $v = k$, and $\mathcal{F}[\![\mathbf{if}(v > 1)\{\gamma\}]\!]$ be a transform function based on if condition, the computation of the transform function could be defined as the following formula.

$$\mathcal{F}[\![\mathbf{if}(v > 1)\{\gamma\}]\!]\{(v = k)\} = \mathcal{F}[\![\gamma]\!]\{(v > 1 \wedge v = k) \cup \{(\neg(v > 1) \wedge v = k)\}$$

```
instr{
  val := 1;
  r1 := val
};
instr{
  val := r1
  [x] := val
};
instr{
  val := r1
  [y] := val
}
```

Operation structure γ_1

```
do{
 instr{
  val := [y];
  r1 := val
 };
 instr{
  (rd := 1‖rt := r2);
  val_z := (rd = rt)?1:0;
  z := val_z;
  val_n := (rd = rt)?0:1;
  n := val_n
 };
 instr{
  val_n := n
 }
}while(val_n = 1);
instr{
  val := [x];
  r1 := val
};
assert(val = 1)
```

Operation structure γ_2

Figure 4-7: An operation structure for message passing

where the output is the set of formulas by analyzing condition $v > 1$. However, this is the overview of transform function; The target of the transform function used in this method is to extract the symbolic execution directly instead of using control flow analysis shown in the bounded method.

To abstract the program execution in the transform function, *symbolic execution state* shown in Definition 4.3 is used to represent the way to representation instead of execution path described in previous chapter. A symbolic execution state consists of *event state* ε, *property condition* p and *assertion condition* a that required by the execution, in which the undetermined values in the conditions are *symbolic values* shown in definition 4.4. The intention of p and a is to represent *property* and *assertion* on symbolic variables appearing in ε. Property p on given event state ε represents how the data of local variables are flown among events. While assertion a ensures the values of local variables satisfy the condition to show the program correctness. Consequently, the elements of a symbolic execution state can be used in SMT-based program verification directly.

Definition 4.3 (Symbolic Execution State)**.** A *symbolic execution state* $\sigma \in \Sigma$ is a tuple $\langle \varepsilon, p, a \rangle$ where ε is an event state, $p, a \in$ Bexp is a boolean expression.

Definition 4.4 (Symbolic Value)**.** Given SymVal be the set of symbolic values, a *symbolic value* is either an *arbitrary value*, written by $*$, or the value of a temporal register, written

by t^j for temporal register $t \in \text{Tmp}$ and identifier $j \in \mathbb{N}^+$.

A *symbolic value* is intended to represent the intermediate value of each step a temporal register is changed, in which the value is not evaluated yet. To indicate each step, we use the superscript on a temporal variable to indicate the step by using a natural number. Hence, to capture the step of a variable during generation, the *variable state* shown in Definition 4.5 is used to save the state of counters on temporal registers. In addition, the value might be an arbitrary value of natural number $\mathbb{N}$.

Definition 4.5 (Variable State)**.** *Variable state* $v \in \mathbb{V}$ is a mapping from temporal register Tmp to a positive natuaral number $\mathbb{N}^+$ where $\mathbb{V}$ is the set of variable states.

In addition to variable states, a write variable state shown in Definition 4.6 is used to capture the recent temporal variable to a memory location for checking in an assertion expression. For instance, the following description produces three write events to memory locations `[X]` and `[Y]`.

```
1 [X]  := w_x^1;
2 [Y]  := w_y^1;
3 [X]  := w_x^2;
4 assert([X] = 2);
```

For the assertion, the assertion expression is expected to ensure the value of w_x^2 equals 2 for any program execution. Thus, the intention of the write variable state is to remember the last symbolic variable used by the write operation.

Definition 4.6 (Write Variable State)**.** *Write variable state* $\mu \in \Xi$ is a mapping from memory location Loc to a symbolic value SymVal where Ξ is the set of write variable states.

Definition 4.7 (Symbolic Expression)**.** Let $\text{SymExp}[\![e]\!]v$ be either an expression or a boolean expression that can contain symbolic values in the expression, where e be either an expression or a boolean expression, and v is a variable state, such that

$$\text{SymExp}[\![c]\!]\rho = c$$

$$\text{SymExp}[\![t]\!]\langle v, \mu, \theta\rangle = \begin{cases} t^{v(t)-1} & \text{If } v(t) \geq 1 \\ \mu(t) & \text{If } t \in \text{Loc} \\ * & \text{Otherwise} \end{cases}$$

$$\text{SymExp}[\![e_1 \Box e_2]\!]\rho = \begin{cases} * & \text{If SymExp}[\![e_1]\!]\rho = * \text{ or SymExp}[\![e_2]\!]\rho = * \\ \text{SymExp}[\![e_1]\!]\rho & \text{Otherwise} \\ \Box\ \text{SymExp}[\![e_2]\!]\rho & \end{cases}$$

$$\text{SymExp}[\![\neg e_1]\!]\rho = \neg(\text{SymExp}[\![e_1]\!]\rho)$$

$$\text{SymExp}[\![(e_1)?e_2 : e_3]\!]\rho = (\text{SymExp}[\![e_1]\!]\rho)?(\text{SymExp}[\![e_2]\!]\rho) : (\text{SymExp}[\![\neg e_3]\!]\rho)$$

where $*$ is an arbitrary value, which can be any value $v' \in \mathbb{N}$, and $\Box \in \{+, -, \wedge, \vee\}$.

In some situations, the symbolic execution state is a merging state between two or more states to represent the parallelism of among event states and requirement on them. Thus,

the operator $\oplus$ is a binary operator to merge the behavior of two symbolic execution states, such that

Definition 4.8 (State Merging Operator)**.** Let $\oplus$ be a binary operator on symbolic execution states to consider the concurrent execution of events on those states, such that

$$\langle \varepsilon_1, p_1, a_1 \rangle \oplus \langle \varepsilon_2, p_2, a_2 \rangle = \langle \varepsilon', p_1 \wedge p_2, a_1 \wedge a_2 \rangle$$

where: $\varepsilon_1 = \langle e_1, po_1, iico_1, atom_1 \rangle$,
$\varepsilon_2 = \langle e_2, po_2, iico_2, atom_2 \rangle$,
$\varepsilon' = \langle e_1 \cup e_2, po_1 \cup po_2, iico_1 \cup iico_2, atom_1 \cup atom_2 \rangle$, and
$\emptyset = e_1 \cap e_2$

To explore the possible symbolic execution states of a sequence of operation structures $P = P_1 \cdot \ldots \cdot P_n$, an intermediate state ρ shown in 4.9 is introduced for the loop unwinding method. Then, we consider the set of intermediate states ϑ as a configuration shown in Definition 4.10 to define a transform function. Transform function $\mathcal{E}[\![P]\!]\vartheta$ represents the set of intermediate states that consider the behavior of P from configuration ϑ.

Definition 4.9 (Intermediate State)**.** An intermediate state ρ is a pair $\langle v, \mu, \theta \rangle$ where v is a variable state, μ is a write variable state and θ is either symbolic execution state σ or the *abortion* of symbolic execution state, written by $\texttt{abort}(\sigma)$.

Definition 4.10 (Configuration)**.** A configuration is a set of intermediate states such that $\vartheta \subseteq (\mathbb{V} \times \Xi \times \Sigma)$. Let v_0 be the initial variable state and σ_0 be the initial symbolic execution state, initial configuration $\vartheta_0 = \{\langle v_0, \mu_0, \sigma_0 \rangle\}$ such that $v_0(t) = 0$ for any $t \in \text{Tmp}$, $\mu_0(l) = *$ for any $l \in \text{Loc}$ and $\sigma_0 = \langle \langle \emptyset, \emptyset, \emptyset, \emptyset \rangle, \top, \top \rangle$.

Th abortion of a symbolic execution state σ, written by $\texttt{abort}(\sigma)$, means a termination of a program, in which the further operations are not performed anymore. Given a configuration ϑ, let ϑ^{exec} and ϑ^{abort} be the set containing executable state and the set containing aborted state, respectively, such that

$$\begin{aligned} \vartheta^{\text{exec}} &= \{\langle v, \mu, \sigma \rangle \mid \langle v, \mu, \sigma \rangle \in \vartheta\} \\ \vartheta^{\text{abort}} &= \{\langle v, \mu, \text{abort}(\sigma) \rangle \mid \langle v, \mu, \text{abort}(\sigma) \rangle \in \vartheta\} \end{aligned}$$

Transform function $\mathcal{E}$ is used to explore the possible symbolic execution states for program verification. First of all, the semantics of an execution structure γ, defined by semantics function $\mathcal{I}[\![\gamma]\!]\vartheta$, is defined to systematically capture the events issued by the elements of operation structures.

Let $\mathcal{I}[\![\gamma]\!]\vartheta$ is a configuration containing the possible intermediate states that can be produced by γ from configuration ϑ. The following definitions represent the semantics of execution structure.

$$\mathcal{I}[\![\texttt{v:=e}]\!]\vartheta = \begin{cases} \text{WriteEv}(v, e, \texttt{v:=e})\vartheta & \text{if } v \in \mathcal{V} \setminus \text{Tmp} \\ \text{ReadEv}(v, e, \texttt{v:=e})\vartheta & \text{if } e \in \mathcal{V} \setminus \text{Tmp} \\ \text{Assn}(v, e)\vartheta & \text{if } v, e \in \text{Tmp} \end{cases}$$

$$
\begin{aligned}
\mathrm{Abort}(\vartheta) &= \{\langle v, \mu, \mathrm{abort}(\sigma)\rangle \mid \langle v, \mu, \sigma\rangle \in \vartheta^{\mathrm{exec}}\} \cup \vartheta^{\mathrm{abort}} \\
\mathrm{Prop}(c)(\vartheta) &= \left\{\langle v, \mu, \langle \varepsilon, p \wedge (c\ \rho), a\rangle\rangle \;\middle|\; \rho = \langle v, \mu, \langle \varepsilon, p, a\rangle\rangle \wedge \rho \in \vartheta^{\mathrm{exec}} \right\} \cup \vartheta^{\mathrm{abort}} \\
\mathrm{Assert}(c)(\vartheta) &= \left\{\langle v, \mu, \langle \varepsilon, p, a \wedge (c\ \rho)\rangle\rangle \;\middle|\; \rho = \langle v, \mu, \langle \varepsilon, p, a\rangle\rangle \wedge \rho \in \vartheta^{\mathrm{exec}} \right\} \cup \vartheta^{\mathrm{abort}} \\
\mathrm{Atom}(r, w)(\vartheta) &= \{\langle v, \mu, \langle\langle s, po, iico, atom \cup \{(r, w)\}\rangle, p, a\rangle\rangle \mid \\
&\quad \langle v, \mu, \langle\langle s, po, iico, atom\rangle, p, a\rangle\rangle \in \vartheta^{\mathrm{exec}}\} \cup \vartheta^{\mathrm{abort}} \\
\mathrm{ReadEv}(t, e, op)(\vartheta) &= \{\langle v', \mu, \langle \varepsilon', p \wedge (t^{v(t)} = e'), a\rangle\rangle \mid e' = \mathbb{R}_{\mathrm{val}}[\![R_{ev}]\!] \wedge R_{ev} = ev^*(op, \varepsilon) \wedge \\
&\quad \rho = \langle v, \mu, \langle \varepsilon', p, a\rangle\rangle \wedge (\varepsilon' = \varepsilon \prec R_{ev}) \wedge \langle v, \mu, \langle \varepsilon, p, a\rangle\rangle \in \vartheta^{\mathrm{exec}}\} \\
&\quad \cup \vartheta^{\mathrm{abort}} \\
\mathrm{WriteEv}(t, e, op)(\vartheta) &= \{\langle v, \mu[t \mapsto e'], \langle \varepsilon', p \wedge (\mathbb{W}_{\mathrm{val}}[\![W_{ev}]\!] = e'), a\rangle\rangle \mid \\
&\quad e' = \mathrm{SymExp}[\![e]\!]\langle v, \mu, \langle \varepsilon, p, a\rangle\rangle \wedge W_{ev} = \mathrm{ev}^*(op, \varepsilon) \wedge \\
&\quad \varepsilon' = \varepsilon \prec W_{ev} \wedge \langle v, \mu, \langle \varepsilon, p, a\rangle\rangle \in \vartheta^{\mathrm{exec}}\} \cup \vartheta^{\mathrm{abort}} \\
\mathrm{Assn}(t, e)(\vartheta) &= \{\langle v[t \mapsto v(t) + 1], \mu, \langle \varepsilon, (p \wedge t^{v(t)} = \mathrm{SymExp}[\![e]\!]\langle v, \mu, \langle \varepsilon, p, a\rangle\rangle), a\rangle\rangle \mid \\
&\quad \langle v, \mu, \langle \varepsilon, p, a\rangle\rangle \in \vartheta^{\mathrm{exec}}\} \cup \vartheta^{\mathrm{abort}}
\end{aligned}
$$

Figure 4-8: Auxiliary functions

where ReadEv, WriteEv, and Assn are auxiliary functions, defined in Figure 4-8, to add read event, add write event, and define computation for a symbolic assignment for configuration ϑ.

$$
\begin{aligned}
\mathcal{I}[\![\mathbf{ll}(v, loc)]\!]\vartheta &= \mathrm{ReadEv}(v, loc, \mathbf{ll}(v, loc))\vartheta \\
\mathcal{I}[\![\mathbf{sc}(v_1, loc, v_2)]\!]\vartheta &= \{\langle v, \mu[loc \mapsto e'], \langle \varepsilon', p \wedge (\mathbb{W}_{\mathrm{val}}[\![W_{ev}]\!] = e') \wedge (v_1 = \mathbb{S}[\![W_{ev}]\!]), a\rangle\rangle \mid \\
&\quad e' = \mathrm{SymExp}[\![v_2]\!]\langle v, \mu, \langle \varepsilon, p, a\rangle\rangle \wedge W_{ev} = \mathrm{ev}^*(\mathbf{sc}(v_1, loc, v_2), \varepsilon) \wedge \\
&\quad \varepsilon' = \varepsilon \prec W_{ev} \wedge \langle v, \mu, \langle \varepsilon, p, a\rangle\rangle \in \vartheta^{\mathrm{exec}}\} \cup \vartheta^{\mathrm{abort}} \\
\mathcal{I}[\![fence]\!]\vartheta &= \{\langle v, \mu, \langle \varepsilon \prec \mathrm{ev}^*(fence, \varepsilon), p, a\rangle\rangle \mid \langle v, \mu, \langle \varepsilon, p, a\rangle\rangle \in \vartheta\} \quad \text{if } fence \in \mathrm{Fence}
\end{aligned}
$$

where fence operation *fence* is performed to issue an fence event, denoted by $\mathrm{ev}^*(fence, \varepsilon)$, to be added in event state ε.

$$
\begin{aligned}
\mathcal{I}[\![\mathbf{instr}\{\gamma\}]\!]\vartheta &= \mathrm{Instr}(\mathcal{I}[\![\gamma]\!])\vartheta \\
\text{where: } \mathrm{Instr}(s)\vartheta &= \{\langle v', \mu', \langle \varepsilon''', p', a'\rangle\rangle \mid \langle v, \mu, \langle \varepsilon, p, a\rangle\rangle \in \vartheta \wedge \\
&\quad \langle v', \mu', \langle \varepsilon'', p', a'\rangle\rangle \in s\{\langle v, \mu, \langle \varepsilon', p, a\rangle\rangle\}\} \\
\varepsilon &= \langle ev, po, iico, atom\rangle \\
\varepsilon' &= \langle ev, \emptyset, \emptyset, \emptyset\rangle \\
\varepsilon'' &= \langle ev', po', iico', atom'\rangle \\
\varepsilon''' &= \langle ev', po \cup po' \cup \{(a, b) \mid a \in ev \wedge b \in ev'\}, iico \cup po', atom\rangle
\end{aligned}
$$

$$
\mathcal{I}[\![\mathbf{atom}(\texttt{v1:=e1}); \gamma; \mathbf{atom}(\texttt{v2:=e2})]\!]\vartheta = (\mathrm{Atom}(R_{ev}, W_{ev}) \circ addR \circ \mathcal{I}[\![\gamma]\!] \circ addR)\vartheta
$$

where $addR = \mathrm{ReadEv}(v1, e1, \texttt{v1:=e1})$, $addW = \mathrm{WriteEv}(v2, e2, \texttt{v2:=e2})$, and Atom is an auxiliary function to put atomic constraint to configuration ϑ.

$$
\begin{aligned}
\mathcal{E}[\![\mathbf{instr}\{\gamma\}]\!]\vartheta &= \mathcal{I}[\![\mathbf{instr}\{\gamma\}]\!]\vartheta^{\text{exec}} \cup \vartheta^{\text{abort}} \\
\mathcal{E}[\![P_1 \cdot P_2]\!]\vartheta &= \{\langle v'', \mu_0, \theta' \oplus \theta'' \rangle \mid \langle v'', \mu'', \theta'' \rangle \in \mathcal{E}[\![P_2]\!]\{\langle v', \mu_0, \theta \rangle\} \wedge \\
&\qquad \langle v', \mu', \theta' \rangle \in \mathcal{E}[\![P_1]\!]\{\langle v, \mu_0, \theta \rangle \mid \langle v, \mu, \theta \rangle \in \vartheta\}\} \\
\mathcal{E}[\![\gamma_1; \gamma_2]\!]\vartheta &= (\mathcal{E}[\![\gamma_2]\!] \circ \mathcal{E}[\![\gamma_1]\!])\vartheta \\
\mathcal{E}[\![\mathbf{ifBr}(c)\{\gamma\}]\!]\vartheta &= \text{ifBrCond}(\text{SymExp}[\![c]\!], \mathcal{E}[\![\gamma]\!])\vartheta \\
\text{where IfBrCond}(c, s)\vartheta &= (\text{Prop}(\neg c))\vartheta \cup (s \circ \text{Prop}(c))\vartheta \\
\mathcal{E}[\![\mathbf{do}\{\gamma\}\mathbf{while}(c)]\!]\vartheta &= \textstyle\bigcup_{i \geq 0}(\text{Prop}(\neg\text{SymExp}[\![c]\!]) \circ \text{loop}(\mathcal{E}[\![\gamma]\!] \circ \text{Prop}(c'), i) \circ \mathcal{E}[\![\gamma]\!])\vartheta \\
\text{where } c' &= \text{SymExp}[\![c]\!] \\
\text{loop}(s, i) &= \begin{cases} s & \text{if } i = 1 \\ s \circ \text{loop}(s, i-1) & \text{if } i > 1 \\ \text{id}_\vartheta & \text{Otherwise} \end{cases} \\
\mathcal{E}[\![\mathbf{assume}(c)]\!]\vartheta &= \text{Prop}(\text{SymExp}[\![c]\!])\vartheta \\
\mathcal{E}[\![\mathbf{assert}(c)]\!]\vartheta &= \text{Assert}(\text{SymExp}[\![c]\!])\vartheta
\end{aligned}
$$

Figure 4-9: Transform Function $\mathcal{E}$ of Operation Structure

$$
\begin{aligned}
\mathcal{I}[\![\mathbf{nil}]\!]\vartheta &= \vartheta \\
\mathcal{I}[\![\gamma_1; \gamma_2]\!]\vartheta &= (\mathcal{I}[\![\gamma_2]\!] \circ \mathcal{I}[\![\gamma_1]\!])\vartheta \\
\mathcal{I}[\![\gamma_1 \parallel \gamma_2]\!]\vartheta &= \{\langle v'', \mu'', \sigma' \oplus \sigma'' \rangle \mid \langle v'', \mu'', \sigma'' \rangle \in \mathcal{I}[\![\gamma_2]\!]\{\langle v', \mu', \sigma \rangle\} \wedge \\
&\qquad \langle v', \mu', \sigma' \rangle \in \mathcal{I}[\![\gamma_1]\!]\{\langle v, \mu, \sigma \rangle\} \wedge \langle v, \mu, \sigma \rangle \in \vartheta\} \\
\mathcal{I}[\![\mathbf{if}(c)\{\gamma\}]\!]\vartheta &= \text{Prop}(\neg\text{SymExp}[\![c]\!])\vartheta \cup (\mathcal{I}[\![\gamma]\!] \circ \text{Prop}(\text{SymExp}[\![c]\!]))\vartheta
\end{aligned}
$$

Given a sequence of operation structures $P = P_1 \cdot \ldots \cdot P_n$, the transform function is defined by $\mathcal{E}[\![P]\!]\vartheta$ where ϑ is a configuration, in which transform function $\mathcal{E}$ is defined in Fig. 4-9. This function systematically explores the symbolic execution states by expanding a loop if any.

4.4 Inductive Invariant Method

This section would like to introduce derivation executions for programs that containing loop behavior for program verification. Mainly, the derived executions are then used in our SMT-based program verification approach in which a derived execution is encoded as a formula for an SMT solver. Firstly, the inductive invariant method is explained using transform functions, in which the functions are described to abstract the loop behavior in the programs to produce the finite executions to be considered in program verification. Then, the way to encode each of executions regarding the memory model is as the same as the bounded method.

4.4.1 Derivation of Programs containing Loop

In the inductive invariant method, the abstraction of loop behavior relies on the number of the read events that need to get that computation of a loop iteration. In other words, the effect of write operations in each loop iteration is needed to be considered for the read events issued by other programs. Thus, we would like to define a transform function to derive the executions regarding the number of the read events issued by other programs.

First of all, to determine the number of the read events for considerations, the decision of if-branches and predicated conditions are eliminated to determine the number of the read events to be considered explicitly. However, the descriptions of loop-branches are still as the number of the read operations appearing in the loop body are determined explicitly. In program verification, all decisions of branch conditions and predicated conditions are captured by the set of operation structures, in which each operation structure in the set represents a decision case for program executions. Thus, transform function $\mathcal{S}[\![\gamma]\!]\Gamma$ shown in Definition 4.11 is given to explore the cases of if-branch and predicated conditions to be considered where Γ is the set of operation structures in which initial set $\Gamma_0 = \{\mathbf{nil}\}$.

Definition 4.11 (Condition Extraction Function)**.** Given an operation structure γ and set of operation structures Γ, transform function $\mathcal{S}[\![\gamma]\!]\vartheta$ is defined as the following.

$$
\begin{aligned}
\mathcal{S}[\![\gamma_1;\gamma_2]\!]\Gamma &= (\mathcal{S}[\![\gamma_2]\!] \circ \mathcal{S}[\![\gamma_1]\!])\Gamma \\
\mathcal{S}[\![\mathbf{ifBr}(c)\{\gamma\}]\!]\Gamma &= \mathcal{S}[\![\mathbf{assume}(\neg c)]\!]\Gamma \cup (\mathcal{S}[\![\gamma]\!] \circ \mathcal{S}[\![\mathbf{assume}(c)]\!])\Gamma \\
\mathcal{S}[\![\mathbf{assume}(c)]\!]\Gamma &= \{s;\mathbf{assume}(c) \mid s \in \Gamma\} \\
\mathcal{S}[\![\mathbf{assert}(c)]\!]\Gamma &= \{s;\mathbf{assert}(c) \mid s \in \Gamma\} \\
\mathcal{S}[\![\mathbf{do}\{\gamma\langle inv\rangle\}\mathbf{while}(c)]\!]\Gamma &= \{s;\mathbf{do}\{\gamma\langle inv\rangle\}\mathbf{while}(c) \mid s \in \Gamma\} \\
\mathcal{S}[\![\mathbf{instr}\{\gamma\}]\!]\Gamma &= \{s;\mathbf{instr}\{\gamma'\} \mid \gamma' = \mathcal{I}'[\![\gamma]\!]\{\mathbf{nil}\} \wedge s \in \Gamma\}
\end{aligned}
$$

As for transform function $\mathcal{I}'$ appearing in Definition 4.11, this function is used to extract the cases to perform operations in an instruction without any decision. Definition 4.12 shows the semantics of the function to eliminate the predicated condition appearing in an instruction if any. By this elimination, the number of the read events can be counted explicitly.

Definition 4.12 (Condition Execution Extraction Fucntion)**.** Given an instruction execution $\mathbf{instr}\{\gamma\}$ and γ is its execution strucutre, transformation function $\mathcal{I}'[\![\gamma]\!]$ can be defined in the followings.

$$
\begin{aligned}
\mathcal{I}'[\![\mathbf{nil}]\!]\Gamma &= \Gamma \\
\mathcal{I}'[\![\mathtt{v:=e}]\!]\Gamma &= \{\gamma;\mathtt{v:=e} \mid \gamma \in \Gamma\} \\
\mathcal{I}'[\![\mathbf{ll}(v_1,l)]\!]\Gamma &= \{\gamma;\mathbf{ll}(v_1,l) \mid \gamma \in \Gamma\} \\
\mathcal{I}'[\![\mathbf{sc}(v_1,l,v_2)]\!]\Gamma &= \{\gamma;\mathbf{sc}(v_1,l,v_2) \mid \gamma \in \Gamma\} \\
\mathcal{I}'[\![fence]\!]\Gamma &= \{\gamma;fence \mid \gamma \in \Gamma\} \\
\mathcal{I}'[\![\gamma_1;\gamma_2]\!]\Gamma &= (\mathcal{I}'[\![\gamma_2]\!] \circ \mathcal{I}'[\![\gamma_1]\!])\Gamma \\
\mathcal{I}'[\![\gamma_1 \parallel \gamma_2]\!]\Gamma &= \{\gamma;(\gamma_1 \parallel \gamma_2) \mid \gamma \in \Gamma\} \\
\mathcal{I}'[\![\mathbf{if}(c)\{\gamma\}]\!]\Gamma &= \mathcal{I}'[\![\mathbf{assume}(c);\gamma]\!]\Gamma \cup \mathcal{I}'[\![\mathbf{assume}(\neg c)]\!]\Gamma
\end{aligned}
$$

By given the set of operation structures for counting the number of the read events, transform function info shown in Definition 4.14 is introduced to accumulate the number of the read events to be considered. To accumulate the number of events, mapping $\delta : \mathrm{Loc} \to \mathbb{N}^+$ shown in Definition 4.13 is used to remember the number of read events to a specific memory location. Then, the loop body can determine the significant read events affected by the write events issued by each loop iteration.

Definition 4.13 (Counter of Read Events)**.** Let $\delta \in \Delta$ be a mapping from memory locations Loc to positive natural numbers $\mathbb{N}^+$ where Δ is the set of the mapping functions $\mathrm{Loc} \to \mathbb{N}^+$. The initial mapping is defined as δ_0 where $\delta_0(l) = 0$ for any $l \in \mathrm{Loc}$.

Definition 4.14 (Information of Read Event Counters)**.** Given set of operation structure Γ and mapping $\delta : \mathrm{Loc} \to \mathbb{N}^+$, function $\mathrm{info}[\![\Gamma]\!]\delta$ is defined as the followings.

$$\mathrm{info}(\Gamma) = \begin{cases} \delta_0 & \text{if } \Gamma = \emptyset \\ \mathrm{countR}[\![\gamma]\!](\mathrm{info}(\Gamma \setminus \{\gamma\})) & \text{if } \Gamma = \Gamma' \cup \{\gamma\} \end{cases}$$

For function countR appearing in Definition 4.14, the function shown in Definition 4.15 is used to count the number of read events to be considered in a single operation structure. Although the loop would appear on the operation structure and the number of the read events becomes infinite, the number of the read events to be considered equals the number of the read operations and load-link operations appearing in the loop body. Thus, in the definition, the number of the read operations appearing in the loop body is considered directly.

Definition 4.15 (Read Counter Function)**.** Given an operation structure γ and mapping $\delta : \mathrm{Loc} \to \mathbb{N}^+$, function $\mathrm{countR}[\![\gamma]\!]\delta$ is defined as the followings.

$$\begin{aligned}
\mathrm{countR}[\![\mathbf{nil}]\!]\delta &= \delta \\
\mathrm{countR}[\![\mathbf{assume}(c)]\!]\delta &= \delta \\
\mathrm{countR}[\![\mathbf{assert}(c)]\!]\delta &= \delta \\
\mathrm{countR}[\![\gamma_1; \gamma_2]\!]\delta &= (\mathrm{countR}[\![\gamma_2]\!] \circ \mathrm{countR}[\![\gamma_1]\!])\delta \\
\mathrm{countR}[\![\gamma_1 \parallel \gamma_2]\!]\delta &= (\mathrm{countR}[\![\gamma_2]\!] \circ \mathrm{countR}[\![\gamma_1]\!])\delta \\
\mathrm{countR}[\![\mathbf{instr}\{\gamma\}]\!]\delta &= \mathrm{countR}[\![\gamma]\!]\delta \\
\mathrm{countR}[\![\mathbf{ll}(v, l)]\!]\delta &= \delta[l \mapsto \delta(l) + 1] \\
\mathrm{countR}[\![v := e]\!]\delta &= \begin{cases} \delta[v \mapsto \delta(v) + 1] & \text{If } v \in \mathrm{Loc} \\ \delta & \text{Otherwise} \end{cases} \\
\mathrm{countR}[\![\mathbf{sc}(v_1, l, v_2)]\!]\delta &= \delta \\
\mathrm{countR}[\![fence]\!]\delta &= \delta && \text{If } fence \in \mathrm{Fence} \\
\mathrm{countR}[\![\mathbf{do}\{\gamma\langle inv\rangle\}\mathbf{while}(c)]\!]\delta &= \mathrm{countR}[\![\gamma]\!]\delta \\
\mathrm{countR}[\![\mathbf{ifBr}(c)\{\gamma\}]\!]\delta &= \mathrm{countR}[\![\gamma]\!]\delta \\
\mathrm{countR}[\![\mathbf{if}(c)\{\gamma\}]\!]\delta &= \mathrm{countR}[\![\gamma]\!]\delta
\end{aligned}$$

Given a sequence of operation structure $P = P_1 \cdot \ldots \cdot P_n$, Inductive Invariant transform function $Inv[\![P]\!]$ shown in Definition 4.16 is given to analyzing the cases to be considered for program verification. Firstly, the function uses function $\mathcal{S}$ and function info to determine the read events to memory locations for specific operation structure P_i by considering every operation structure P_j where $j \neq i$. Then, transformation function $\mathcal{L}^{\delta}[\![\gamma]\!]\vartheta$ shown in Definition 4.17 is used to abstract the loop behavior in operation structure γ with the information of read events described in δ for program verification.

Definition 4.16 (Inductive Invariant Transform Function)**.** Given a sequence of operation structure $P = P_1 \cdot P_2 \cdot \ldots \cdot P_n$, the inductive invariant transform function $Inv[\![P]\!]$ for sequence P is defined as the following.

$$
\begin{aligned}
Inv[\![P_1 \cdot P_2 \cdot \ldots \cdot P_n]\!] = \{\sigma_1 \oplus \sigma_2 \oplus \ldots \oplus \sigma_n \mid \\
& s_1 \in \mathcal{S}[\![P_1]\!]\{\mathbf{nil}\} \wedge s_2 \in \mathcal{S}[\![P_2]\!]\{\mathbf{nil}\} \wedge \ldots \wedge s_n \in \mathcal{S}[\![P_n]\!]\{\mathbf{nil}\} \wedge \\
& \Gamma = \{s_1, s_2, \ldots, s_n\} \wedge \delta_1 = \text{info}(\Gamma \setminus \{s_1\}) \wedge \\
& \delta_2 = \text{info}(\Gamma \setminus \{s_2\}) \wedge \ldots \wedge \delta_n = \text{info}(\Gamma \setminus \{s_n\}) \wedge \\
& \langle v_1, \mu_1 \sigma_1 \rangle \in \mathcal{L}^{\delta_1}[\![s_1]\!]\{\langle v_0, \mu_0, \sigma_0 \rangle\} \wedge \\
& \langle v_2, \mu_2 \sigma_2 \rangle \in \mathcal{L}^{\delta_2}[\![s_2]\!]\{\langle v_1, \mu_0, \sigma_0 \rangle\} \wedge \ldots \\
& \wedge \langle v_n, \mu_n, \sigma_n \rangle \in \mathcal{L}^{\delta_n}[\![s_n]\!]\{\langle v_{n-1}, \mu_0, \sigma_0 \rangle\}\}
\end{aligned}
$$

In particular, transform function $\mathcal{L}^{\delta}[\![\mathbf{do}\{\gamma\langle inv \rangle\}\mathbf{while}(c)]\!]$ provides the cases regarding the provided invariant condition inv and the number of read events described in mapping function δ. The main idea is to provide the cases to be considered in the inductive invariant approach, in which the number of the read events is needed to consider the effect of write events to be abstracted in the inductive cases.

Definition 4.17 (Loop Abstraction Transformation)**.** Given operation structure γ, mapping function δ, and configuration ϑ, transform function $\mathcal{L}^{\delta}[\![\gamma]\!]\vartheta$ is defined as the follow-

ings.

$$
\begin{aligned}
\mathcal{L}^{\delta}[\![\text{nil}]\!]\vartheta &= \vartheta \\
\mathcal{L}^{\delta}[\![\gamma_1;\gamma_2]\!]\vartheta &= (\mathcal{L}^{\delta}[\![\gamma_2]\!] \circ \mathcal{L}^{\delta}[\![\gamma_1]\!])\vartheta \\
\mathcal{L}^{\delta}[\![\mathbf{assume}(c)]\!]\vartheta &= \text{Prop}(\text{SymExp}[\![c]\!])\vartheta^{\text{exec}} \cup \vartheta^{\text{abort}} \\
\mathcal{L}^{\delta}[\![\mathbf{assert}(c)]\!]\vartheta &= \text{Assert}(\text{SymExp}[\![c]\!])\vartheta^{\text{exec}} \cup \vartheta^{\text{abort}} \\
\mathcal{L}^{\delta}[\![\mathbf{instr}\{\gamma\}]\!]\vartheta &= \mathcal{I}[\![\mathbf{instr}\{\gamma\}]\!]\vartheta^{\text{exec}} \cup \vartheta^{\text{abort}} \\
\mathcal{L}^{\delta}[\![\mathbf{do}\{\gamma\langle inv\rangle\}\mathbf{while}(c)]\!]\vartheta &= (\text{Prop}(\neg\text{SymExp}[\![c]\!]) \circ \mathcal{L}^{\delta}[\![\gamma]\!])\vartheta \cup \\
&\quad (\text{Abort} \circ \text{Assert}(\text{SymExp}[\![inv]\!]) \circ body \circ \\
&\qquad \text{Prop}(\text{SymExp}[\![c \wedge inv]\!]) \circ \text{loop}(arb\_w, k+1) \circ \text{Arbitr}(\gamma) \circ \\
&\qquad \text{Assert}(inv) \circ \text{Prop}(\text{SymExp}[\![c]\!]) \circ \mathcal{L}^{\delta}[\![\gamma]\!])\vartheta \cup \\
&\quad (\text{Prop}(\neg\text{SymExp}[\![c]\!]) \circ body \circ \text{Prop}(\text{SymExp}[\![c \wedge inv]\!]) \circ \\
&\qquad \text{loop}(arb\_w, k+1) \circ \text{Arbitr}(\gamma) \circ \text{Assert}(inv) \circ \\
&\qquad \text{Prop}(\text{SymExp}[\![c]\!]) \circ \mathcal{L}^{\delta}[\![\gamma]\!])\vartheta \\
\text{where } k &= \Sigma_{l \in \text{locW}(\gamma)}\delta(l), \\
arb\_w &= \text{Prop}(\text{SymExp}[\![c \wedge inv]\!]) \circ \text{ArbW}[\![\gamma_s]\!]\text{id}_{\vartheta}, and \\
body &= (\mathcal{L}^{\delta}[\![\gamma]\!] \circ \text{Prop}(\text{SymExp}[\![c \wedge inv]\!]))\vartheta \\
\text{loop}(s,i) &= \begin{cases} s & \text{if } i = 1 \\ s \circ \text{loop}(s, i-1) & \text{if } i > 1 \\ \text{id}_{\vartheta} & \text{Otherwise} \end{cases}
\end{aligned}
$$

To determine the induction cases, the arbitrary state of a loop iteration is constructed by considering the arbitrary effect of previous loop iterations. As the method concerns the infinite loop iterations, there could be any i iterations before exiting the loop behavior where $i \geq 0$. To consider the arbitrary effect of previous iterations, the effect of assignments to temporal registers and the effect of the write events to memory locations are considered. According to Definition 4.17, $\text{Arbtr}(\gamma)$ is the arbitrary effect of assignments to temporal registers, while $\text{ArbW}[\![\gamma]\!]\text{id}_{\vartheta}$ is the transformation function representing the arbitrary performing of write operations to memory locations regarding γ where id_{ϑ} is the identity function of configuration ϑ.

For the arbitrary effect of assignments to temporal registers, transform function $\text{Arbitr}(\gamma)$ shown in Definition 4.18 represents the arbitrary effect in which the arbitrary value would be assigned to temporal registers. To represent an arbitrary value in a first-order formula, a fresh variable of a natural number is used in the formula without any constraint on it. Consequently, the variable appearing in the formula will be evaluated by an SMT solver to find an instance that satisfying the provided constraint in the formula. Thus, all of the variables appearing in γ, explored by function $\text{AssnV}(\gamma)$ shown in Definition 4.19, are used to instantiate the fresh variables for use in the formula. In the definition, a fresh variable can be instantiated by increasing the specific counters in the variable state.

Definition 4.18 (Arbitrary Assignment)**.** Given operation structure γ, transform function $\text{Arbitr}(\gamma)$ is defined as the following.

$$\text{Arbitr}(\gamma)\vartheta = \{\langle v[l \mapsto v(l)+1], \mu, \sigma\rangle \mid l \in \text{AssnV}(\gamma) \wedge \langle v, \mu, \sigma\rangle \in \vartheta^{\text{exec}}\} \cup \vartheta^{\text{abort}}$$

Definition 4.19 (Assignment Target)**.** Given operation structure γ, function $\mathrm{AssnV}(\gamma)$ is defined as the followings to capture the target assignments appearing in the operation structure.

$$\mathrm{AssnV}(op) = \begin{cases} \{v\} & \text{if } op = \texttt{v:=e} \wedge v \in \mathrm{Tmp} \\ \{v\} & \text{if } op = \mathbf{atom}(\texttt{v:=e}) \wedge \\ & v \in \mathrm{Tmp} \\ \emptyset & \text{Otherwise} \end{cases}$$

$$\begin{aligned} \mathrm{AssnV}(\mathbf{nil}) &= \emptyset \\ \mathrm{AssnV}(\gamma_1;\gamma_2) &= \mathrm{AssnV}(\gamma_1) \cup \mathrm{AssnV}(\gamma_2) \\ \mathrm{AssnV}(\gamma_1 \parallel \gamma_2) &= \mathrm{AssnV}(\gamma_1) \cup \mathrm{AssnV}(\gamma_2) \\ \mathrm{AssnV}(\mathbf{instr}\{\gamma\}) &= \mathrm{AssnV}(\gamma) \end{aligned}$$

$$\begin{aligned} \mathrm{AssnV}(\mathbf{if}(c)\{\gamma\}) &= \mathrm{AssnV}(\gamma) \\ \mathrm{AssnV}(\mathbf{ifBr}\{c\}\{\gamma\}) &= \mathrm{AssnV}(\gamma) \end{aligned} \qquad \mathrm{AssnV}\left(\ \mathbf{do}\{\gamma\langle inv\rangle\}\mathbf{while}(c)\ \right) = \mathrm{AssnV}(\gamma)$$

where op is an assignment, $c, inv \in \mathrm{BExp}$, $\gamma, \gamma_1, \gamma_2$ are either operation structures or execution structures.

For the effect of the write events to memory locations, function $\mathrm{ArbW}[\![\gamma]\!]F$ shown in Definition 4.20 represents the set of transformation functions to capture the possible effects of write events from a loop iteration where γ is an operation structure and F is the set of transformation functions. Due to the effect of a store-condition operation can arbitrary fail, this means the write event issued by the operation would not be considered as the effect of the loop behavior. In the abstraction, transformation functions produced by $\mathrm{ArbW}[\![\gamma]\!]\mathrm{id}_\vartheta$ are the possible instantiations of write events in which the write values are arbitrary values where id_ϑ is the identity function for configuration ϑ.

Definition 4.20 (Arbitrary Write Events)**.** Given operation structure γ and transform function for configuration ϑ, transform fuction $\mathrm{ArbW}[\![\gamma]\!]f$ is defined as the followings.

$$\begin{aligned} \mathrm{ArbW}[\![\mathbf{nil}]\!]f &= f \\ \mathrm{ArbW}[\![\mathbf{assume}(c)]\!]f &= f \\ \mathrm{ArbW}[\![\mathbf{assert}(c)]\!]f &= f \\ \mathrm{ArbW}[\![\mathbf{ifBr}(c)\{\gamma\}]\!]f &= \mathrm{AltF}(\mathrm{ArbW}[\![\gamma]\!]f, f) \\ \mathrm{ArbW}[\![\mathbf{instr}\{\gamma\}]\!]f &= \mathrm{ArbW}[\![\gamma]\!]f \\ \mathrm{ArbW}[\![\gamma_1;\gamma_2]\!]f &= (\mathrm{ArbW}[\![\gamma_2]\!] \circ \mathrm{ArbW}[\![\gamma_1]\!])f \\ \mathrm{ArbW}[\![\gamma_1 \parallel \gamma_2]\!]f &= \mathrm{ParallelF}(\mathrm{ArbW}[\![\gamma_1]\!]\mathrm{id}_\vartheta, \mathrm{ArbW}[\![\gamma_2]\!]\mathrm{id}_\vartheta) \circ f \\ \mathrm{ArbW}[\![\mathbf{if}(c)\{\gamma\}]\!]f &= \mathrm{AltF}(\mathrm{ArbW}[\![\gamma]\!]f, f) \\ \mathrm{ArbW}[\![\texttt{v := e}]\!]f &= \begin{cases} \mathrm{WriteEv}*(v) \circ f & \text{If } v \in \mathrm{Loc} \\ f & \text{Otherwise} \end{cases} \\ \mathrm{ArbW}[\![\mathbf{ll}(v,l)]\!]f &= f \\ \mathrm{ArbW}[\![\mathbf{sc}(v_1,l,v_2)]\!]f &= \mathrm{AltF}(\mathrm{WriteEv}*(l) \circ \mathrm{Assn}(v_1,0) \circ f, \mathrm{Assn}(v_1,1) \circ f) \end{aligned}$$

where id_ϑ is the identity function of configuration ϑ, $\mathrm{AltF}(f_1, f_2)$ is the function representing alternative cases for consideration, $\mathrm{ParalellF}(f_1, f_2)$ is the function to consider the behavior of parallelism of operations in an instruction, and $\mathrm{WriteEv}*(l)$ is an abstract

write instance to memory location l such that

$$\begin{aligned}
\text{AltF}(f_1, f_2)\vartheta &= f_1\ \vartheta \cup f_2\ \vartheta \\
\text{ParallelF}(F_1, F_2)\vartheta &= \{\langle v'', \sigma_1 \oplus \sigma_2\rangle \mid \langle v, \sigma\rangle \in \vartheta \wedge \langle v', \sigma_1\rangle \in F_1\{\langle v, \sigma\rangle\} \wedge \\
&\qquad \langle v', \sigma_2\rangle \in F_2\{\langle v', \sigma\rangle\}\} \\
\text{WriteEv}*(l)\vartheta &= \{v[w_l \mapsto v(w_l) + 1], \mu[l \mapsto w_l^{v(w_l)+1}] \mid \langle v, \mu, \sigma\rangle \in \vartheta^{\text{exec}}\} \cup \vartheta^{\text{abort}}
\end{aligned}$$

To summarize, given a sequence of operation structures $P = P_1 \cdot \ldots \cdot P_n$, the symbolic execution states derived from $Inv[\![P]\!]$ can be used for SMT-based program verification directly. In particular, each of the symbolic execution states is automatically encoded as a first-order formula regarding a memory model, in which the formula represents the executions that cause a violation of the program property. Thus, to ensure the program correctness, there must be no valuation of the formula for any symbolic execution states derived from the inductive invariant method.

4.4.2 Soundness of Inductive Invariant Method

Given a sequence of operation structures P for the inductive invariant method, sequence P is *partially correct* on memory model $\mathcal{M}$ if all of the executions always satisfy the program property of P where the execution is eventually terminated. Note that, in our SMT-based program verification approach, a symbolic execution state is expected to be terminated to determine the effect on any relaxed memory model.

To determine whether symbolic execution state $\langle \varepsilon, p, a\rangle$ is correct on memory model M, the corresponding first-order formula is provided as the following.

$$\text{Encode}(\varepsilon, \mathcal{M}) \wedge p \wedge \neg a$$

where Encode is the encoding function to translate event state ε regarding memory model $\mathcal{M}$ as a first-order formula to represent the program behavior, as explained in Chapter 3. Hence, the correctness of a symbolic execution state on a memory model is shown in Definition 4.21 by determining the corresponding first-order formula.

Definition 4.21 (Correctness of Symbolic Execution)**.** Given a symbolic execution state $\sigma = \langle \varepsilon, p, a\rangle$ and corresponding formula $\text{Encode}(\varepsilon, \mathcal{M}) \wedge p \wedge \neg a$, the symbolic execution state is correct on memory model $\mathcal{M}$ if there is no valuation of symbolic values in the corresponding formula.

Consequently, given a sequence of operation structures P, the correctness of P on a memory model can be determined by considering the correctness of the corresponding symbolic execution states of P. In particular, the partial correctness shown in Definition 4.22 is considered in which the corresponding executions are supposed to be eventually terminated.

Definition 4.22 (Partial Correctness of Operation Structures)**.** Given a sequence of operation structures P and memory model $\mathcal{M}$, P is partial correct on memory model $\mathcal{M}$ is written by $\mathcal{M} \models P$ such that

$$\mathcal{M} \models P \textbf{ iff } \forall \langle v, \sigma \rangle \in \mathcal{E}[\![P]\!]\vartheta_0.\sigma \textit{ is correct on memory model } \mathcal{M}$$

where the corresponding symbolic executions of P are derived by transform function $\mathcal{E}[\![P]\!]\vartheta_0$.

Regarding transform function $\mathcal{E}[\![P]\!]$ of sequence P, the number of loop iterations is a positive natural number. In particular, if the number cannot be determined in a systematic way regarding the programs, the number of loop iterations to be considered becomes infinite. For instance, the loop condition that relies on the effect of other programs can be waiting for an arbitrary number of loop iterations before exiting the loop behavior. Instead of using transform function $\mathcal{E}[\![P]\!]$ of sequence P, the derived symbolic execution states from the inductive invariant method is used to determine the partial correctness of P. Definition 4.23 shows the partial correctness of P using the inductive invariant method in which the symbolic execution states to be considered are derived by $Inv[\![P]\!]$.

Definition 4.23 (Partial Correctness using Indutive Invariant Method)**.** Given a sequence of operation structures P and the symbolic execution states of P are derived by $Inv[\![P]\!]$, the partial correctness of P is written by $\mathcal{M} \vdash P$ such that

$$\mathcal{M} \vdash P \textbf{ iff } \forall \sigma \in Inv[\![P]\!].\sigma \text{ is correct on memory model } \mathcal{M}$$

According to the out-of-order executions occurring on relaxed memory models, the inductive invariant method seems not to be sound for any relaxed memory model. For instance, some relaxed memory models such as POWER allows the completion of a read operation to be delayed if the following memory operations do not rely on the effect of the read operation. Hence, in concurrent programs, the operations after the loop behavior could affect the prior read operations. However, in the inductive invariant method, the behavior after an arbitrary loop iteration is not considered. Thus, the invariant condition cannot be ensured for any relaxed memory model.

However, some relaxed memory models that do not allow any read operation to be delayed, the inductive invariant approach would be sound for those relaxed memory models. Thus, we consider partial store ordering (PSO) as the relaxed memory model to consider the soundness of the inductive invariant method. In particular, to determine the program behavior on partial store ordering, the specification shown in Figure 4-10 provided for herding cats [AMT14] is used. Intuitively, the specification requires: (1) the data flow of conflicting events issued by the same program must follow the program order, and (2) the completing order of memory operations after a read operation must follow the program order. Consequently, the inductive invariant method seems to be sound on partial store ordering (PSO).

let po-loc = po & loc
acyclic po-loc | rf | co | fr
let ppo = po \ (W*RW)
acyclic ppo | rfe | co | fr

Figure 4-10: PSO specification in cat language

In addition, for simplicity of determining the soundness, operator $\triangleleft$ shown in Definition 4.24 is used instead of transform functions to represent the way to constructing a symbolic execution state. Moreover, we use curly brackets { and } to represent the repetition of operator $\triangleleft$. For instance, $\sigma_0 \triangleleft \{\sigma_1 \triangleleft\}^2 \sigma_n$ can be interpreted as $\sigma_0 \triangleleft \sigma_1^1 \triangleleft \sigma_1^2 \triangleleft \sigma_n$ where the content of σ_1 is repeated 2 times and the symbolic variables in σ_1^1 is different from σ_1^2. On the other words, the curly brackets correspond to transform function $\text{loop}(f_1, 2)$ where the symbolic execution states σ_1 corresponds to f_1.

Definition 4.24 (Adding Symbolic Execution State)**.** Given two symbolic execution states $\langle \varepsilon_1, p_1, a_1 \rangle$ and $\langle \varepsilon_2, p_2, a_2 \rangle$, operator $\triangleleft$ is defined as the following.

$$\langle \varepsilon_1, p_1, a_1 \rangle \triangleleft \langle \varepsilon_2, p_2, a_2 \rangle = \langle \varepsilon', p_1 \wedge p_2, a_1 \wedge a_2 \rangle$$

where $\varepsilon_1 = \langle s_1, po_1, iico_1, atom_1 \rangle$, $\varepsilon_1 = \langle s_2, po_2, iico_2, atom_2 \rangle$, and $\varepsilon' = \langle s_1 \cup s_2, po_1 \cup po_2, iico_1 \cup iico_2, atom_1 \cup atom_2 \rangle$.

First of all, Lemma 1 is provided to ensure the program execution on a single processor must be preserved as the same as the program executed in the program order. Note that this behavior is expected to be preserved on any relaxed memory model to maintain the program correctness on a uniprocessor. However, in the lemma, the specification of PSO is used as a reference to determine the behavior.

Lemma 1 (The effect on Uniprocessors)**.** *Given the program order of conflicting write events issued by the same program, a read event can get the write value from the last write events appear in the program order before the read event.*

Proof. To ensure the lemma, we consider two cases: (1) the write events appear before the recent write event in the program order must not be seen by the following read event, and (2) the write events appear after the read event must not be seen by the prior read event.

Given two write events w_1 and w_2, and read event r accessing to the same memory location, such that $\mathbb{L}[\![w_1]\!] = \mathbb{L}[\![w_2]\!]$ and $\mathbb{L}[\![w_1]\!] = \mathbb{L}[\![r]\!]$. Besides, the events are issued in the program order, abstracted by predicated function po such that $\text{po}(w_1, w_2) \wedge \text{po}(w_1, r) \wedge \text{po}(w_2, r)$. Given a contradiction case that the write value w_1 can be flown to read event r, such that $\text{rf}(w_1, r)$. This means the following formula must be satisfied.

$$\text{po}(w_1, w_2) \wedge \text{po}(w_1, r) \wedge \text{po}(w_2, r) \wedge \text{rf}(w_1, r) \tag{4.1}$$

In this proof, we use a restriction of PSO specification shown in Figure 4-10, which is `acyclic po-loc | rf | co | fr`. This restriction is then presented as the following

formulas.

$$\begin{gathered}\forall x, y \in \{w_1, w_2, r\}.\mathrm{rel}(x, y) \textbf{ iff } (\mathrm{po{-}loc}(x, y) \vee \mathrm{rf}(x, y) \vee \mathrm{co}(x, y) \vee \mathrm{fr}(x, y)) \\ \forall x, y, z \in \{w_1, w_2, r\}.\mathrm{rel}(x, y) \wedge \mathrm{y, z} \implies \mathrm{rel}(x, z) \\ \forall x \in \{w_1, w_2, r\}.\neg(\mathrm{rel}(x, x))\end{gathered} \tag{4.2}$$

where rel is a fresh relation to detect a cycle of `po-loc | rf | co | fr`. As $\mathrm{rf}(w_1, r)$ is the assumption for contradiction case, this means $\mathrm{rf}(w_1, r) \vee \mathrm{po{-}loc}(w_1, r) \vee \mathrm{co}(w_1, r) \vee \mathrm{fr}(w_1, r)$ must not cause a cycle relation, denoted by predicated function rel. For $\mathrm{po{-}loc}$ relation, as all the events access the same memory location, the relation is the same as po where the following formula is derived.

$$\mathrm{po{-}loc}(w_1, w_2) \wedge \mathrm{po{-}loc}(w_1, r) \wedge \mathrm{po{-}loc}(w_2, r) \tag{4.3}$$

According to the formula for fr relation and $\mathrm{rf}(w_1, r)$ is given, the following formula is derived.

$$\forall w' \in \{w_1, w_2\}.\mathrm{rf}(w_1, r) \wedge \mathrm{co}(w_1, w') \implies \mathrm{fr}(r, w') \tag{4.4}$$

According to the formula for co relation, write event w' of relation $\mathrm{fr}(r, w')$ can be evaluated as w_2, such that the following formula is derived.

$$\mathrm{co}(w_1, w_2) \wedge \mathrm{fr}(r, w_2) \tag{4.5}$$

Consequently, the derived formulas cause a cycle of relation rel where $\mathrm{fr}(r, w_2)$ and $\mathrm{po{-}loc}(w_2, r)$ are derived. Thus, this shows the contradiction to the restriction `acyclic po-loc | rf | co | fr`.

On the other hand, given two write events w_1 and w_2, and read event r accessing to the same memory location, where w_1, w_2, and r are issued in the program order, such that the following formula is derived.

$$\mathrm{po}(w_1, r) \wedge \mathrm{po}(r, w_2) \wedge \mathrm{w_1, w_2} \tag{4.6}$$

Let's assume the contradiction case that the value of write event w_2 that occur after read event r in the program order can be read by read event, such that $\mathrm{rf}(w_2, r)$. This means the following formula is satisfied on the behavior of uniprocessors.

$$\mathrm{po}(w_1, r) \wedge \mathrm{po}(r, w_2) \wedge \mathrm{po}(w_1, w_2) \wedge \mathrm{rf}(w_2, r) \tag{4.7}$$

For $\mathrm{po{-}loc}$ relation, as all the events access the same memory location, the relation is the same as po, such that

$$\mathrm{po{-}loc}(w_1, r) \wedge \mathrm{po{-}loc}(r, w_2) \wedge \mathrm{po{-}loc}(w_1, w_2) \tag{4.8}$$

According to the restruction represented by Equation 4.2, the fact $\mathrm{po{-}loc}(r, w_2)$ and $\mathrm{rf}(w_2, r)$ cause a cycle which is prohibited by the restriction. Thus, this also shows a contradiction. Consequently, we can conclude that (1) the write event that is not the last write before the read event cannot be seen by the read event, and (2) the write occur later than the read event cannot be seen by the prior read event. These implies that only the last write can be seen by the conflicting read event occur in the program order. □

Moreover, Lemma 2 shows the required property for ensuring the soundness of the method that the effect of following write operations cannot be seen by the prior read issued by the same program. Intuitively, `ppo` relation appearing in the specification restricts the effect of the read event must be completed before any read events and write events after the read event.

Lemma 2 (Read is not delayed in PSO)**.** *In partial store ordering, the effect of a write event occur after a read event in the program order cannot be seen by the prior read issued by the same program.*

Proof. According to Lemma 1, the write effect appearing after a read event cannot be seen directly. In addition, the effect on concurrent programs contains more than 1 programs is considered for this proof.

Given read event r and write event w are issued in the program order, such that $\mathrm{po}(r, w)$, and the events access to the same memory location. In addition, there are read event r' and write event w' on another program accessing to the same memory location of read event r and write event w. We give a contradiction case that read event r can be affected by write event w indirectly through other program, such that

$$\mathrm{rf}(w, r') \wedge \mathrm{po}(r', w') \wedge \mathbb{W}_{\mathrm{val}}[\![w']\!] = f(\mathbb{R}_{\mathrm{val}}[\![r']\!]) \wedge \mathrm{rf}(w', r) \tag{4.9}$$

where events w' and r' are issued by other program in the program order such that $\mathrm{po}(r', w')$, and function f computes a new value regarding the read value of event r' such that $\mathbb{R}_{\mathrm{val}}[\![r']\!]$. Regarding the formula to realize the read value, the following formula is derived.

$$\mathrm{rf}(w, r') \implies \mathbb{R}_{\mathrm{val}}[\![r']\!] = \mathbb{W}_{\mathrm{val}}[\![w]\!] \tag{4.10}$$

and

$$\mathrm{rf}(w', r) \implies \mathbb{R}_{\mathrm{val}}[\![r]\!] = \mathbb{W}_{\mathrm{val}}[\![w']\!] \tag{4.11}$$

This case assumes read event r is affected by the following w indirectly, such that $\mathbb{R}_{\mathrm{val}}[\![r]\!] = \mathbb{W}_{\mathrm{val}}[\![w']\!]$ where $\mathrm{W}_{\mathrm{val}}[\![w']\!] = f(\mathbb{R}_{\mathrm{val}}[\![r']\!])$ and $\mathbb{R}_{\mathrm{val}}[\![r']\!] = \mathbb{W}_{\mathrm{val}}[\![w]\!]$.

According to the specification of PSO shown in Figure 4-10, the restriction `acyclic ppo | rfe | co | fr` can be derived as the following formulas

$$\begin{gathered}\forall x, y \in \{r, w, r', w'\}.\mathrm{rel}_2(x, y) = (\mathrm{ppo}(x, y) \vee \mathrm{rfe}(x, y) \vee \mathrm{co}(x, y) \vee \mathrm{fr}(x, y)) \\ \forall x, y, z \in \{r, w, r', w'\}.\mathrm{rel}_2(x, y) \wedge \mathrm{rel}_2(y, z) \implies \mathrm{rel}_2(x, z) \\ \forall x \in \{r, w, r', w'\}.\neg\mathrm{rel}_2(x, x)\end{gathered} \tag{4.12}$$

where rel_2 is a fresh relation to detect the cycle of relation `ppo | rfe | co | fr`. According to ppo relation, defined by `let ppo = po \ (W*RW)` and the existing events, the relation can be derived as the following formula.

$$\mathrm{ppo}(r, w) \wedge \mathrm{ppo}(r', w') \tag{4.13}$$

For rfe relation and the existing assumption on rf, the following formula is derived.

$$\mathrm{rfe}(w, r') \wedge \mathrm{rfe}(w', r) \tag{4.14}$$

Then, the derived formulas are contradict where $\text{ppo}(r, w)$, $\text{rfe}(w, r')$, $\text{ppo}(r', w')$ and $\text{rfe}(w', r)$ cause a cycle, which contradict the formulas 4.12.

□

According to the inductive invariant method, the number of abstraction arb_w to be used in the method depends on the number of read events issued by other programs. Thus, Lemma 4 shows k read events on other programs need at most k abstractions to be considered. In addition, Lemma 3 derives the fact of Herding cats framework to support the Lemma 4.

Lemma 3 (At most k write events are read by k read events)**.** *In herding cats framework, at most k write events are accessed if there are k read events.*

Proof. According to herding cat framework [AMT14], the read-from relation rf is defined based on conflicting write events in the system. Besides, our encoding method provides initial write event w_L^0 for memory location L. Thus, given read event r, at least one write event in the system must be accessed by the read event, such that

$$\bigvee_{w \in \text{cf}(W,r)} .(\text{rf}(w, r)) \tag{4.15}$$

where $\text{cf}(W, r)$ is the conflicting write events of event r among the write events in W. Thus, if we have n read events and m write events in the system, at most n write events are read by the read event separately if $m \geq n$.

On the other hand, if the number of write events is less than the number of read events where $m < n$, there is at least two read events that access to the same write events by the pigeonhole principle[1]. □

Lemma 4 (The number of write abstractions in the method)**.** *In the inductive invariant method, if there is k read events issued by other operation structures to memory locations, at most k abstractions of arbitrary loop iterations are needed for realizing the write values produced by any m loop iterations where $m > k$.*

Proof. Given an operation structure $P_1 = \gamma_1; \mathbf{do}\{\gamma_s\langle inv\rangle\}\mathbf{while}(c); \gamma_2$ and P_2 contains k read events to the same memory locations accessed by γ_s. For the target of inductive invariant method, the number of loop iterations is any arbitrary number due to the loop condition relies on the read events. In our method, arb_w represents an arbitrary effect of write events appearing in loop body γ_s that satisfying invariant condition inv, where $\text{arb\_w} = \text{Prop}(\text{SymExp}[\![c \wedge inv]\!]) \circ \text{ArbW}[\![\gamma_s]\!]\{\text{id}_\vartheta\}$. Note that the effect of arb_w is over-approximation approach that realizes the arbitrary computations satisfying the invariant condition.

In each abstract iteration, the effect of arb_w would be different from each other. Assume that there are m possible computations can be produced by m abstractions of loop iteration separately, without any order. According to Lemma 3, k read events on other

[1] https://en.wikipedia.org/wiki/Pigeonhole_principle

programs need at most k write events to be considered. As m computations are produced by m abstractions, at most k computations are needed for k read events on other processors. This means at most k abstractions of arbitrary loop iteration are needed for k read events issued by other programs even if there are m possible computations.

□

Consequently, Theorem 1 shows if the derived symbolic execution states from the inductive invariant method are correct on partial store ordering (PSO), the corresponding symbolic execution states that are eventually terminated are also correct on partial store ordering (PSO). This means if sequence P is correct by the inductive invariant method, the partial correctness can be ensured on PSO.

Theorem 1 (Soundness of Inductive Invariant Method for PSO)**.** *Given a sequence of operation structures P, $\mathcal{M} \vdash P \implies \mathcal{M} \models P$ in which $\mathcal{M}$ is PSO memory model.*

Proof. First of all, let's consider an arbitrary sequence of operation structures P, there are two cases: only one operation structure and n operation structures. For the case 1 structure, the effect of any execution is always as the same as the program executed on sequential consistency model. Consequently, $\mathcal{M} \vdash P \implies \mathcal{M} \models P$ for any memory model M if P consists of an operation structure. This means only case $P = P_1 \cdot \ldots \cdot P_n$ must be taken into account.

Let Q be a sequence of operation structures such that $Q = P_2 \cdot \ldots \cdot P_n$, where Q is considered as the arbitrary effect of computation in the system. For sequence $P = P_1 \cdot Q$, structure P_1 is considered to contain arbitrary loop inside the structure such that $P_1 = \gamma_1; \mathbf{do}\{\gamma_{s1}\}\mathbf{while}(c1); \gamma_2$, where γ_1, γ_{s1} and γ_2 are arbitrary structures. In the proof, we consider 2 cases: (1) Q is an arbitrary effect that consists of finite read instances to shared-memory locations and (2) there is a loop in Q that *could cause infinite read instances* to the system. For instance, we consider P^1 and P^2 such that

Case 1: $P^1 = (\gamma_1; \mathbf{do}\{\gamma_{s1}\}\mathbf{while}(c1); \gamma_2) \cdot Q$
Case 2: $P^2 = (\gamma_1; \mathbf{do}\{\gamma_{s1}\}\mathbf{while}(c1); \gamma_2) \cdot (\gamma_3; \mathbf{do}\{\gamma_{s2}\}\mathbf{while}(c2); \gamma_4) \cdot Q$

where each element is considered arbitrary.

Let's consider the target proof PSO $\models P$ where P is either P^1 or P^2. By the definition, any arbitrary σ such that $\langle v, \sigma \rangle \in \mathcal{E}[\![P]\!]\vartheta_0$ must be correct on PSO. In particular, there must be no evaluation of $\text{Encode}(\varepsilon, \mathcal{M}) \wedge p \wedge \neg a$ for any symbolic execution state $\sigma = \langle \varepsilon, p, a \rangle$ where $\text{Encode}(\varepsilon, \mathcal{M})$ is the encoded formula to realize the effect of program execution on memory model $\mathcal{M}$. For each case of P, there is assumption that P derived by inductive invariant method is correct on PSO, written by PSO $\vdash P$. According to the assumption, both cases must be shown to conclude soundness of the inductive invariant method.

Case 1 For the case P^1, let $P_1 = \gamma_1; \mathbf{do}\{\gamma_{s1}\}\mathbf{while}(c); \gamma_2$, an arbitrary σ is considered from $\mathcal{E}[\![P_1 \cdot Q]\!]\vartheta_0$ such that

$$\mathcal{E}[\![P_1 \cdot Q]\!]\{\langle v_0, \mu_0, \sigma_0 \rangle\} = \{\langle v'', \mu_0, \sigma_1 \oplus \sigma_Q \rangle \mid \langle v'', \mu'', \sigma_1 \rangle \in \mathcal{E}[\![P_1]\!]\{\langle v', \mu_0, \sigma_0 \rangle\} \wedge \langle v', \mu', \sigma_Q \rangle \in \mathcal{E}[\![Q]\!]\{\langle v_0, \mu_0, \sigma_0 \rangle\}\}$$

This means arbitrary state σ to be considered is $\sigma_1 \oplus \sigma_Q$ in which $\sigma_Q = \langle \varepsilon_Q, p_Q, a_Q \rangle$ for arbitrary case. On the other hand, σ_1 is considered from $\mathcal{E}[\![P_1]\!]$ and $P_1 = \gamma_1; \mathbf{do}\{\gamma_{s1}\}\mathbf{while}(c1); \gamma_2$, where γ_1, γ_{s1} and γ_2 are arbitrary structures, such that

$$
\begin{aligned}
& \mathcal{E}[\![\gamma_1; \mathbf{do}\{\gamma_{s1}\}\mathbf{while}(c1); \gamma_2]\!]\{\langle v', \mu_0, \sigma_0 \rangle\} \\
= & \textstyle\bigcup_{i \geq 0} (\mathcal{E}[\![\gamma_2]\!] \circ \mathrm{Prop}(\neg c') \circ \mathrm{loop}(\mathcal{E}[\![\gamma_{s1}]\!] \circ \mathrm{Prop}(c'), i) \circ \mathcal{E}[\![\gamma_{s1}]\!] \circ \mathcal{E}[\![\gamma_1]\!])\{\langle v', \mu_0, \sigma_0 \rangle\} \\
= & (\mathcal{E}[\![\gamma_2]\!] \circ \mathrm{Prop}(\neg c') \circ \mathcal{E}[\![\gamma_{s1}]\!] \circ \mathcal{E}[\![\gamma_1]\!])\{\langle v', \mu_0, \sigma_0 \rangle\} \cup \\
& \textstyle\bigcup_{i \geq 1} (\mathcal{E}[\![\gamma_2]\!] \circ \mathrm{Prop}(\neg c') \circ \mathrm{loop}(\mathcal{E}[\![\gamma_{s1}]\!] \circ \mathrm{Prop}(c'), i) \circ \mathcal{E}[\![\gamma_{s1}]\!] \circ \mathcal{E}[\![\gamma_1]\!])\{\langle v', \mu_0, \sigma_0 \rangle\}
\end{aligned}
$$

Hence, there are 2 arbitrary cases of σ_1 to be considered:

Case 1-1: $(\mathcal{E}[\![\gamma_2]\!] \circ \mathrm{Prop}(\neg c') \circ \mathcal{E}[\![\gamma_{s1}]\!] \circ \mathcal{E}[\![\gamma_1]\!])\{\langle v', \sigma_0 \rangle\}$
Case 1-2: $\bigcup_{i \geq 1} (\mathcal{E}[\![\gamma_2]\!] \circ \mathrm{Prop}(\neg c') \circ \mathrm{loop}(\mathcal{E}[\![\gamma_{s1}]\!] \circ \mathrm{Prop}(c'), i) \circ \mathcal{E}[\![\gamma_1]\!])\{\langle v', \sigma_0 \rangle\}$

Then, arbitrary state $\sigma = \sigma_1 \oplus \sigma_Q$ can be considered from those 2 cases. For simplicity, the cases of σ are explained in the following forms:

Case 1-1: $\sigma^0 = (\sigma_0 \lhd \sigma^0_{\gamma 1} \lhd \sigma^0_{s1} \lhd \sigma^0_{\neg c} \lhd \sigma^0_{\gamma 2}) \oplus \sigma_Q$

Case 1-2: $\sigma^{inf} = (\sigma_0 \lhd \sigma^0_{\gamma 1} \lhd \sigma^0_{s1} \lhd \{\sigma_c \lhd \sigma_{s1} \lhd\}^i \sigma^{inf}_{\neg c} \lhd \sigma^{inf}_{\gamma 2}) \oplus \sigma_Q$ For any $i \geq 1$

There is an assumption that PSO $\vdash P_1 \cdot Q$, in which the symbolic execution states derived by indutive invariant method are correct on PSO, where $Q = P_2 \cdot \ldots \cdot P_n$. Thus, arbitrary state σ is considered from $Inv[\![P_1 \cdot P_2 \cdot \ldots \cdot P_n]\!]$ such that

$$
\begin{aligned}
Inv[\![P_1 \cdot P_2 \cdot \ldots \cdot P_n]\!] = \{ & \theta_1 \oplus \theta_2 \oplus \ldots \oplus \theta_n \mid \\
& s_1 \in \mathcal{S}[\![P_1]\!]\{\mathbf{nil}\} \wedge s_2 \in \mathcal{S}[\![P_2]\!]\{\mathbf{nil}\} \wedge \ldots \wedge s_n \mathcal{S}[\![P_n]\!]\{\mathbf{nil}\} \wedge \\
& \delta_1 = \mathrm{info}(\{s_2, \ldots, s_n\}) \wedge \ldots \wedge \delta_n = \mathrm{info}(\{s_1, \ldots, s_{n-1}\}) \wedge \\
& \langle v^1, \mu_1, \theta_1 \rangle \in \mathcal{L}^{\delta_1}[\![s_1]\!]\{\langle v_0, \mu_0, \sigma_0 \rangle\} \wedge \ldots \wedge \\
& \langle v^n, \mu_n, \theta_n \rangle \in \mathcal{L}^{\delta_n}[\![s_n]\!]\{\langle v_0, \mu_0, \sigma_{n-1} \rangle\}
\end{aligned}
$$

As $P_2 \cdot \ldots \cdot P_n$ is considered in an arbitrary way, let $\sigma_Q = \langle \varepsilon_Q, p_Q, a_Q \rangle$ be an abstraction of $\theta_2 \oplus \ldots \oplus \theta_n$ for consideration. Besides, we assume that Q has k read accesses to shared-memory locations, in which could be written by the write accesses in P_1. Thus, let arbitrary state to be considered for P^1 be $\sigma^1 = \theta_1 \oplus \sigma_Q$ where θ_1 is considered from $L^{\delta_1}[\![\gamma_1; \mathbf{do}\{\gamma_{s1}\}\mathbf{while}(c1); \gamma_2]\!]\{\langle v_0, \mu_0, \sigma_0 \rangle\}$ such that

$$
\begin{aligned}
& \mathcal{L}^{\delta}_1[\![\gamma_1; \mathbf{do}\{\gamma_{s1}\}\mathbf{while}(c1); \gamma_2]\!]\{\langle v_0, \mu_0, \sigma_0 \rangle\} \\
= & (\mathcal{L}^{\delta}_1[\![\gamma_2]\!] \circ \mathcal{L}^{\delta}_1[\![\mathbf{do}\{\gamma_{s1}\}\mathbf{while}(c1)]\!] \circ \mathcal{L}^{\delta}_1[\![\gamma_1]\!])\{\langle v_0, \mu_0, \sigma_0 \rangle\} \\
= & (\mathcal{L}^{\delta}_1[\![\gamma_2]\!] \circ \mathcal{L}^{\delta}_1[\![\mathbf{do}\{\gamma_{s1}\}\mathbf{while}(c1)]\!])\{\langle v_1, \mu_1, \sigma_0 \lhd \sigma^0_{\gamma 1} \rangle\}
\end{aligned}
$$

where $\sigma^0_{\gamma 1}$ is an arbitrary symbolic execution state corresponding to transform function $\mathcal{L}^{\delta}_1[\![\gamma_1]\!]$ such that $\sigma^0_{\gamma 1} = \langle \varepsilon^0_{\gamma 1}, p^0_1, a^0_1 \rangle$. After that, by case analysis of $\mathcal{L}^{\delta}_1[\![\mathbf{do}\{\gamma_{s1}\}\mathbf{while}(c)]\!]$, arbitrary state θ_1 can be considered from either of the following cases:

$$
\begin{aligned}
& \langle v'_1, \mu'_1, \theta^a_{s1} \rangle \in (\mathrm{Prop}(\neg c') \circ \mathcal{L}^{\delta_1}[\![\gamma_s]\!])\{\langle v_1, \mu_1, \sigma_0 \lhd \sigma^0_{\gamma 1} \rangle\} \\
& \langle v'_2, \mu'_2, \theta^a_{s2} \rangle \in (\mathrm{Abort} \circ \mathrm{Assert}(inv') \circ body \circ \mathrm{Prop}(c' \wedge inv') \circ \mathrm{loop}(arb\_w, k+1) \circ \\
& \qquad \mathrm{Arbitr}(\gamma_s) \circ \mathrm{Assert}(inv') \circ \mathrm{Prop}(c') \circ \mathcal{L}^{\delta_1}[\![\gamma_s]\!])\{\langle v_1, \mu_1, \sigma_0 \lhd \sigma^0_{\gamma 1} \rangle\} \\
& \langle v'_3, \mu'_3, \theta^a_{s3} \rangle \in (\mathrm{Prop}(\neg c') \circ body \circ \mathrm{Prop}(c' \wedge inv') \circ \mathrm{loop}(arb\_w, k+1) \circ \mathrm{Arbitr}(\gamma_s) \circ \\
& \qquad \mathrm{Assert}(inv') \circ \mathrm{Prop}(c') \circ \mathcal{L}^{\delta_1}[\![\gamma_s]\!])\{\langle v_1, \mu_1, \sigma_0 \lhd \sigma^0_{\gamma 1} \rangle\}
\end{aligned}
$$

where k is the number of read events issued by Q, $c' = \mathrm{SymExp}[\![c]\!]$, $inv' = \mathrm{SymExp}[\![inv]\!]$,

$arb\_w = \text{ArbW}[\![\gamma_s]\!]\{\text{id}_\vartheta\}$ and $body = (\mathcal{L}^{\delta_1}[\![\gamma_s]\!] \circ \text{Prop}(\text{SymExp}[\![c \wedge inv]\!]) \circ \text{Arbitr}(\gamma_s))$. On the other words, θ_1 is either the followings.

$$\begin{aligned}
\theta^1 &= (\sigma_0 \lhd \sigma^0_{\gamma 1} \lhd \sigma^0_{s1} \lhd \sigma^0_{\neg c} \lhd \sigma^0_{\gamma 2}),\\
\theta^2 &= \text{abort}(\sigma_0 \lhd \sigma^0_{\gamma 1} \lhd \sigma^0_{s1} \lhd \sigma_{c1} \lhd \sigma_{inv} \lhd \sigma^a_{arb_v} \lhd \{\sigma_{arb_w} \lhd\}^{k+1}\ \sigma_{c\&inv} \lhd \sigma^a_{s1} \lhd \sigma^a_{inv}), \text{ or}\\
\theta^3 &= (\sigma_0 \lhd \sigma^0_{\gamma 1} \lhd \sigma^0_{s1} \lhd \lhd \sigma_{c1} \lhd \sigma_{inv} \lhd \sigma^a_{arb_v} \lhd \{\sigma_{arb_w} \lhd\}^{k+1}\ \sigma_{c\&inv} \lhd \sigma^a_{s1} \lhd \sigma^a_{\neg c} \lhd \sigma^a_{\gamma 2}).
\end{aligned}$$

where each symbolic execution state corresponds the transform function to generate the state. Consequently, arbitrary state $\sigma^1 = \theta_1 \oplus \sigma_Q$ which is correct on PSO is either:

$$\begin{aligned}
\sigma_{h0} &= (\sigma_0 \lhd \sigma^0_1 \lhd \sigma^0_{s1} \lhd \sigma^0_{\neg c} \lhd \sigma^0_{\gamma 2}) \oplus \sigma_Q,\\
\sigma_{inv} &= \text{abort}(\sigma_0 \lhd \sigma^0_{\gamma 1} \lhd \sigma^0_{s1} \lhd \sigma_{c1} \lhd \sigma^{a0}_{inv} \lhd \sigma^a_{arb_v} \lhd \{\sigma_{arb_w} \lhd\}^{k+1}\ \sigma_{c\&inv} \lhd \sigma^a_{s1} \lhd \sigma^a_{inv})\\
&\quad \oplus \sigma_Q \text{ , or}\\
\sigma_t &= (\sigma_0 \lhd \sigma^0_{\gamma 1} \lhd \sigma^0_{s1} \lhd \sigma_{c1} \lhd \sigma^{a0}_{inv} \lhd \sigma^a_{arb_v} \lhd \{\sigma_{arb_w} \lhd\}^{k+1}\ \sigma_{c\&inv} \lhd \sigma^a_{s1} \lhd \sigma^a_{\neg c} \lhd \sigma^a_{\gamma 2}) \oplus \sigma_Q
\end{aligned}$$

According to the first case, $\sigma^0 = (\sigma_0 \lhd \sigma^0_{\gamma 1} \lhd \sigma^0_{s1} \lhd \sigma^0_{\neg c} \lhd \sigma^0_{\gamma 2}) \oplus \sigma_Q$, we must show σ^0 must be correct for any arbitrary execution on PSO. By assumption σ_{h0} is correct on PSO, this can conclude the correctness of σ^0 in which the executions of σ_{h0} covers the cases of σ^0. Thus, state σ^0 is also correct on PSO.

For case 1-2, $\sigma^{inf} = (\sigma_0 \lhd \sigma^0_{\gamma 1} \lhd \sigma^0_{s1} \lhd \{\sigma_c \lhd \sigma_{s1} \lhd\}^i \sigma^{inf}_{\neg c} \lhd \sigma^{inf}_{\gamma 2}) \oplus \sigma_Q$, we must show the program property of σ^{inf} must always be satisfied for any valuation. According to the assumption of σ_{inv} is correct on PSO, where $\sigma_{inv} = \text{abort}(\sigma_0 \lhd \sigma^0_{\gamma 1} \lhd \sigma^0_{s1} \lhd \sigma_{c1} \lhd \sigma_{inv} \lhd \sigma^a_{arb\_v} \lhd \{\sigma_{arb\_w} \lhd\}^{k+1}\ \sigma_{c\&inv} \lhd \sigma^a_{s1} \lhd \sigma^a_{inv}) \oplus \sigma_Q$ where k is the number of read events appearing in σ_Q. Intuitively, the idea of σ_{inv} is to ensure the invariant condition inv is always satisfied for any arbitrary loop iteration.

According to σ_{inv}, state σ^{a0}_{inv} is of the form $\langle \varepsilon_0, \top, f_{inv}(\vec{v}_{a0}) \rangle$ where $\vec{v}_{a0}$ is the local variables appearing before assertion statement **assert**(inv) and f_{inv} is function to represent the assertion condition inv regarding the input variables. Thus, if assumption $PSO \vdash P^1$ is proved, assertion condition $f_{inv}(\vec{v}_{a0})$ is also preserved regarding the behavior of $\sigma_0 \lhd \sigma^0_{\gamma 1} \lhd \sigma^0_{s1} \lhd \sigma_{c1}$. Intuitively, this ensures the behavior before entering the loop satifies the invariant condition inv.

For a loop behavior, each iteration i is abstracted as $\sigma^a_{arb\_v} \lhd \{\sigma_{arb\_w} \lhd\}^j\ \sigma_{c\&inv} \lhd \sigma^a_{s1}$ where $j \leq i$ to abstract the effect of write events in j iterations before iteration i. By Lemma 1, the arbitrary effect of write events from iteration $i-1$ is enough for realizing the effect of previous iterations. Besides, Lemma 4 shows k abstract iterations $\{\sigma_{arb\_w} \lhd\}^{k+1}$ are enough for realizing the effect of write events to k read events issued by other processing units and the effect of write events of the last iteration before the arbitrary iteration. If there are write events on other processing units that affected by those write events in the prior iterations, the effect of external write events is covered as all write events appeared in the σ_{inv}. Note that, according to Lemma 2, the effect of write operations appearing after the considering iteration i can be ignored. Consequently, the invariant condition is ensured as the given symbolic execution state σ_{inv}, shown as the following, is correct on PSO.

$$\sigma_{inv} = \text{abort}(\sigma_0 \lhd \sigma^0_{\gamma 1} \lhd \sigma^0_{s1} \lhd \sigma_{c1} \lhd \sigma^{a0}_{inv} \lhd \sigma^a_{arb\_v} \lhd \{\sigma_{arb\_w} \lhd\}^{k+1} \sigma_{c\&inv} \lhd \sigma^a_{s1} \lhd \sigma^a_{inv}) \oplus \sigma_Q$$

From state σ_{inv}, σ^a_{inv} is of the form $\langle \varepsilon_0, \top, f_{inv}(\vec{v}_a) \rangle$ where $\vec{v}_a$ represents the neccessary

variables appearing in $\sigma^0_{\gamma 1}$ and σ^a_{s1}. Thus, as $PSO \vdash P^1$, assertion condition $f_{inv}(\vec{v}_a)$ is ensured where states $\sigma^a_{arb\_v}$ and $\{\sigma_{arb\_w} \lhd\}^{k+1}$ are used to consider arbitrary previous loop iterations at most $k+1$ abstract iterations. Therefore, the invariant condition inv is also ensured on PSO during loop iterations as Lemma 2 also ensure the effect on PSO. In other words, if assertion **assert**(inv) is added at the end of the loop body, symbolic execution state $(\sigma_0 \lhd \sigma^0_{\gamma 1} \lhd \sigma_{s1} \lhd \sigma_{inv} \lhd \{\sigma_c \lhd \sigma_{s1} \lhd \sigma_{inv} \lhd\}^{i \geq 1}\ \sigma') \oplus \sigma_Q$ is correct on PSO where σ' is an arbitrary symbolic execution state.

Besides, assumption σ_t attempts to show P is correct based on derived program executions, such that

$$\sigma_t = (\sigma_0 \lhd \sigma^0_{\gamma 1} \lhd \sigma^0_{s1} \lhd \sigma_{c1} \lhd \sigma^{a0}_{inv} \lhd \sigma^a_{arb\_v} \lhd \{\sigma_{arb\_w} \lhd\}^{k+1}\ \sigma_{c\&inv} \lhd \sigma^a_{s1} \lhd \sigma^a_{\neg c} \lhd \sigma^a_{\gamma 2}) \oplus \sigma_Q$$

Once the invariant condition is ensured by σ_{inv}, arbitrary loop iteration $(\sigma^a_{arb\_v} \lhd \{\sigma_{arb\_w} \lhd\}^{k+1}$ $\sigma_{c\&inv} \lhd \sigma^a_{s1})$ always satisfies the loop invariant. In symbolic execution state σ_t, arbitrary loop iteration σ^a_{s1} eventually exit the loop by $\sigma^a_{\neg c}$ and then the behavior of γ_2 is captured arbitrary by $\sigma^a_{\gamma 2}$. By Lemma 4, all write events issued by P_1 can be covered by σ^t as the k abstract iterations $\{\sigma_{arb\_w} \lhd\}^{k+1}$ are the most iterations for realizing the write events from the loop iterations for k read events from other processing units and read events from arbitrary iteration σ^a_{s1}. Thus, the computation on other processing units can compute the write values to the P_1 regarding the possible values form P_1. Consequently, as σ^t is correct on PSO, σ^{inf} is also correct on PSO.

Case 2 For the case $P^2 = (\gamma_1; \mathbf{do}\{\gamma_{s1}\}\mathbf{while}(c1); \gamma_2) \cdot (\gamma_3; \mathbf{do}\{\gamma_{s2}\}\mathbf{while}(c2); \gamma_4) \cdot Q$, the number of read events is countable infinite because of loop behavior on other programs. Consequently, the necessary arbitrary iterations also become countable infinite. Let $P_1 = (\gamma_1; \mathbf{do}\{\gamma_{s1}\}\mathbf{while}(c1); \gamma_2)$ and $P_2 = (\gamma_3; \mathbf{do}\{\gamma_{s2}\}\mathbf{while}(c2); \gamma_4)$, an arbitrary σ is considered from $\mathcal{E}[\![P_1 \cdot P_2 \cdot Q]\!]\vartheta_0$ such that

$$\begin{aligned}\mathcal{E}[\![P_1 \cdot P_2 \cdot Q]\!]\{\langle v_0, \mu_0, \sigma_0\rangle\} = \{\langle v''', \mu_0, \sigma_1 \oplus \sigma_2 \oplus \sigma_Q\rangle \mid \langle v', \mu', \sigma_1\rangle \in \mathcal{E}[\![P_1]\!]\{\langle v_0, \mu_0, \sigma_0\rangle\} \wedge \\ \langle v'', \mu'', \sigma_2\rangle \in \mathcal{E}[\![P_2]\!]\{\langle v', \mu_0, \sigma_0\rangle\} \wedge \\ \langle v''', \mu''', \sigma_Q\rangle \in \mathcal{E}[\![Q]\!]\{\langle v'', \mu_0, \sigma_0\rangle\}\}\end{aligned}$$

This means arbitrary state σ to be considered is $\sigma_1 \oplus \sigma_2 \oplus \sigma_Q$ in which $\sigma_Q = \langle \varepsilon_Q, p_Q, a_Q\rangle$ for arbitrary case. On the other hand, σ_1 and σ_2 are considered from $\mathcal{E}[\![P_1]\!]$ and $\mathcal{E}[\![P_2]\!]$, respectively, where $\gamma_1, \gamma_{s1}, \gamma_2, \gamma_3, \gamma_{s2}, \gamma_4$ are considered arbitrary. Symbolic execution state σ_1 is considered from $\mathcal{E}[\![\gamma_1; \mathbf{do}\{\gamma_{s1}\}\mathbf{while}(c1); \gamma_2]\!]$ such that

$$\begin{aligned}&\mathcal{E}[\![\gamma_1; \mathbf{do}\{\gamma_{s1}\}\mathbf{while}(c1); \gamma_2]\!]\{\langle v_0, \mu_0, \sigma_0\rangle\} \\ &= \textstyle\bigcup_{i \geq 0}(\mathcal{E}[\![\gamma_2]\!] \circ \mathrm{Prop}(\neg c') \circ \mathrm{loop}(\mathcal{E}[\![\gamma_{s1}]\!] \circ \mathrm{Prop}(c'), i) \circ \mathcal{E}[\![\gamma_{s1}]\!] \circ \mathcal{E}[\![\gamma_1]\!])\{\langle v_0, \mu_0, \sigma_0\rangle\} \\ &= (\mathcal{E}[\![\gamma_2]\!] \circ \mathrm{Prop}(\neg c') \circ \mathcal{E}[\![\gamma_{s1}]\!] \circ \mathcal{E}[\![\gamma_1]\!])\{\langle v_0, \mu_0, \sigma_0\rangle\} \cup \\ &\quad \textstyle\bigcup_{i \geq 1}(\mathcal{E}[\![\gamma_2]\!] \circ \mathrm{Prop}(\neg c') \circ \mathrm{loop}(\mathcal{E}[\![\gamma_{s1}]\!] \circ \mathrm{Prop}(c'), i) \circ \mathcal{E}[\![\gamma_{s1}]\!] \circ \mathcal{E}[\![\gamma_1]\!])\{\langle v_0, \mu_0, \sigma_0\rangle\}\end{aligned}$$

There are 2 arbitrary cases of σ_1 to be considered:

Case P1-1: $(\mathcal{E}[\![\gamma_2]\!] \circ \mathrm{Prop}(\neg c') \circ \mathcal{E}[\![\gamma_{s1}]\!] \circ \mathcal{E}[\![\gamma_1]\!])\{\langle v', \mu_0, \sigma_0\rangle\}$
Case P1-2: $\bigcup_{i \geq 1}(\mathcal{E}[\![\gamma_2]\!] \circ \mathrm{Prop}(\neg c') \circ \mathrm{loop}(\mathcal{E}[\![\gamma_{s1}]\!] \circ \mathrm{Prop}(c'), i) \circ \mathcal{E}[\![\gamma_{s1}]\!] \circ \mathcal{E}[\![\gamma_1]\!])\{\langle v', \mu_0, \sigma_0\rangle\}$

Besides, the arbitrary cases of σ_2 are also considered in a similar way such that

Case P2-1: $(\mathcal{E}[\![\gamma_4]\!] \circ \mathrm{Prop}(\neg c') \circ \mathcal{E}[\![\gamma_{s2}]\!] \circ \mathcal{E}[\![\gamma_3]\!])\{\langle v', \mu_0, \sigma_0\rangle\}$

Case P2-2: $\bigcup_{i\geq 1}(\mathcal{E}[\![\gamma_4]\!] \circ \mathrm{Prop}(\neg c') \circ \mathrm{loop}(\mathcal{E}[\![\gamma_{s2}]\!] \circ \mathrm{Prop}(c'), i) \circ \mathcal{E}[\![\gamma_{s2}]\!] \circ \mathcal{E}[\![\gamma_3]\!])\{\langle v', \mu_0, \sigma_0\rangle\}$

Then, arbitrary state $\sigma = \sigma_1 \oplus \sigma_2 \oplus \sigma_Q$ can be considered as 4 cases by combining each case of σ_1 and σ_2. For simplicity, the cases of σ are explained in the following forms:

Case 2-1: $\sigma^{0,0} = (\sigma_0 \lhd \sigma^0_{\gamma 1} \lhd \sigma^0_{s1} \lhd \sigma^0_{\neg c1} \lhd \sigma^0_{\gamma 2}) \oplus (\sigma_0 \lhd \sigma^0_{\gamma 3} \lhd \sigma^0_{s2} \lhd \sigma^0_{\neg c2} \lhd \sigma^0_{\gamma 4}) \oplus \sigma_Q$

Case 2-2: $\sigma^{0,i} = (\sigma_0 \lhd \sigma^0_{\gamma 1} \lhd \sigma^0_{s1} \lhd \sigma^0_{\neg c1} \lhd \sigma^0_{\gamma 2}) \oplus$
$(\sigma_0 \lhd \sigma^0_{\gamma 3} \lhd \sigma^0_{s2} \lhd \{\sigma_{c2} \lhd \sigma_{s2} \lhd\}^{i\geq 1} \sigma^{inf}_{\neg c2} \lhd \sigma^{inf}_{\gamma 4}) \oplus \sigma_Q$

Case 2-3: $\sigma^{i,0} = (\sigma_0 \lhd \sigma^0_{\gamma 1} \lhd \sigma^0_{s1} \lhd \{\sigma_{c1} \lhd \sigma_{s1} \lhd\}^{i\geq 1} \sigma^{inf}_{\neg c1} \lhd \sigma^{inf}_{\gamma 2}) \oplus$
$(\sigma_0 \lhd \sigma^0_{\gamma 3} \lhd \sigma^0_{s2} \lhd \sigma^0_{\neg c2} \lhd \sigma^0_{\gamma 4}) \oplus \sigma_Q$

Case 2-4: $\sigma^{i,j} = (\sigma_0 \lhd \sigma^0_{\gamma 1} \lhd \sigma^0_{s1} \lhd \{\sigma_{c1} \lhd \sigma_{s1} \lhd\}^{i\geq 1} \sigma^{inf}_{\neg c1} \lhd \sigma^{inf}_{\gamma 2}) \oplus$
$(\sigma_0 \lhd \sigma^0_{\gamma 3} \lhd \sigma^0_{s2} \lhd \{\sigma_{c2} \lhd \sigma_{s2} \lhd\}^{j\geq 1} \sigma^{inf}_{\neg c2} \lhd \sigma^{inf}_{\gamma 4}) \oplus \sigma_Q$

An assumption PSO $\vdash P_1 \cdot P_2 \cdot Q$ is given in which the symbolic execution states derived by inductive invariant method are correct on PSO. The arbitrary symbolic execution states are considred from $Inv[\![P_1 \cdot P_2 \cdot Q]\!]$. The arbitrary symbolic execution state is of the form $\theta_1 \cdot \theta_2 \cdot \theta_Q$, where θ_Q is considered in an arbitrary way. For θ_1 and θ_2, the arbitrary states is considered from $P_1 = \gamma_1; \mathbf{do}\{\gamma_{s1}\langle inv1\rangle\}\mathbf{while}(c1); \gamma_2$ and $P_2 = \gamma_3; \mathbf{do}\{\gamma_{s2}\langle inv2\rangle\}\mathbf{while}(c2); \gamma_4$, respectively. The way to realize both arbitrary states is similar to case P^1. For simplicity, aribitrary state θ_1 is either the followings.

$$\begin{aligned}
\theta^0_1 &= (\sigma_0 \lhd \sigma^0_{\gamma 1} \lhd \sigma^0_{s1} \lhd \sigma^0_{\neg c1} \lhd \sigma^0_{\gamma 2}) \\
\theta^{inv}_1 &= \mathrm{abort}(\sigma_0 \lhd \sigma^0_{\gamma 1} \lhd \sigma^0_{s1} \lhd \sigma^0_{c1} \lhd \sigma^0_{inv1} \lhd \sigma^a_{\mathrm{arb_v1}} \lhd \{\sigma_{\mathrm{arb_w1}} \lhd\}^{k1+1}\ \sigma_{inv\&c1} \lhd \sigma^a_{s1} \lhd \sigma^a_{inv1}) \\
\theta^t_1 &= (\sigma_0 \lhd \sigma^0_{\gamma 1} \lhd \sigma^0_{s1} \lhd \sigma^0_{c1} \lhd \sigma^0_{inv1} \lhd \sigma^a_{\mathrm{arb_v1}} \lhd \{\sigma_{\mathrm{arb_w1}} \lhd\}^{k1+1}\ \sigma_{inv\&c1} \lhd \sigma^a_{s1} \lhd \sigma^a_{\neg c1} \lhd \sigma^0_{\gamma 2})
\end{aligned}$$

where $k1$ are the numbers of read operations to access the shared memory locations which is also accessed by write operaions in loop body $s1$. For state θ_2, the way to realize the arbitrary state is the same as θ_1, where the arbitrary state θ_2 is either the followings.

$$\begin{aligned}
\theta^0_2 &= (\sigma_0 \lhd \sigma^0_{\gamma 3} \lhd \sigma^0_{s2} \lhd \sigma^0_{\neg c2} \lhd \sigma^0_{\gamma 4}) \\
\theta^{inv}_2 &= \mathrm{abort}(\sigma_0 \lhd \sigma^0_{\gamma 3} \lhd \sigma^0_{s2} \lhd \sigma^0_{c2} \lhd \sigma^0_{inv2} \lhd \sigma^a_{\mathrm{arb_v2}} \lhd \{\sigma_{\mathrm{arb_w2}} \lhd\}^{k2+1}\ \sigma_{inv\&c2} \lhd \sigma^a_{s2} \lhd \sigma^a_{inv2}) \\
\theta^t_2 &= (\sigma_0 \lhd \sigma^0_{\gamma 3} \lhd \sigma^0_{s2} \lhd \sigma^0_{c2} \lhd \sigma^0_{inv2} \lhd \sigma^a_{\mathrm{arb_v2}} \lhd \{\sigma_{\mathrm{arb_w2}} \lhd\}^{k2+1}\ \sigma_{inv\&c2} \lhd \sigma^a_{s2} \lhd \sigma^a_{\neg c2} \lhd \sigma_{\gamma 4})
\end{aligned}$$

where $k2$ are the numbers of read operations to access the shared memory locations which is also accessed by write operaions in loop body $s2$. Consequently, arbitrary state $\theta_1 \oplus \theta_2 \oplus \theta_Q$ that is correct on PSO is either:

$$\begin{aligned}
\sigma_{0,0} &= \theta^0_1 \oplus \theta^0_2 \oplus \theta_Q, \\
\sigma_{0,inv} &= \theta^0_1 \oplus \theta^{inv}_2 \oplus \theta_Q, \\
\sigma_{0,t} &= \theta^0_1 \oplus \theta^t_2 \oplus \theta_Q, \\
\sigma_{inv,0} &= \theta^{inv}_1 \oplus \theta^0_2 \oplus \theta_Q, \\
\sigma_{inv,inv} &= \theta^{inv}_1 \oplus \theta^{inv}_2 \oplus \theta_Q, \\
\sigma_{inv,t} &= \theta^{inv}_1 \oplus \theta^t_2 \oplus \theta_Q,
\end{aligned}$$

$\sigma_{t,0} = \theta_1^t \oplus \theta_2^0 \oplus \theta_Q$,
$\sigma_{t,inv} = \theta_1^t \oplus \theta_2^{inv} \oplus \theta_Q$, or
$\sigma_{t,t} = \theta_1^t \oplus \theta_2^t \oplus \theta_Q$.

According to case 2-1, $\sigma^{0,0} = (\sigma_0 \lhd \sigma_{\gamma 1}^0 \lhd \sigma_{s1}^0 \lhd \sigma_{\neg c1}^0 \lhd \sigma_{\gamma 2}^0) \oplus (\sigma_0 \lhd \sigma_{\gamma 3}^0 \lhd \sigma_{s2}^0 \lhd \sigma_{\neg c2}^0 \lhd \sigma_{\gamma 4}^0) \oplus \sigma_Q$ is correct on PSO because the assumption shows $\sigma_{0,0}$ is correct on PSO. As for cases of $\sigma^{0,i}$ and $\sigma^{i,0}$, if one of symbolic states producing a finite instances of read events, the correctness of those cases can be ensured as same as case P^1. In particular, arbitrary states $\sigma_{0,inv}, \sigma_{0,t}, \sigma_{inv,0}$, and $\sigma_{t,0}$, which are correct on PSO, are used as the assumption to show symbolic execution states $\sigma^{0,i}$ and $\sigma^{i,0}$ are correct on PSO.

For case $\sigma^{i,j} = (\sigma_0 \lhd \sigma_{\gamma 1}^0 \lhd \sigma_{s1}^0 \lhd \{\sigma_{c1} \lhd \sigma_{s1} \lhd\}^{i \geq 1} \sigma_{\neg c1}^{inf} \lhd \sigma_{\gamma 2}^{inf}) \oplus (\sigma_0 \lhd \sigma_{\gamma 3}^0 \lhd \sigma_{s2}^0 \lhd \{\sigma_{c2} \lhd \sigma_{s2} \lhd\}^{j \geq 1} \sigma_{\neg c2}^{inf} \lhd \sigma_{\gamma 4}^{inf}) \oplus \sigma_Q$, the given assumption expects to abstract the loop behavior in both P_1 and P_2 using invariant conditions. This means the invariant conditions derive the sufficient behavior from the loop behavior $\{\sigma_{c1} \lhd \sigma_{s1} \lhd\}^{i \geq 1}$ and $\{\sigma_{c2} \lhd \sigma_{s2} \lhd\}^{j \geq 1}$ for program verification. To show that, symbolic execution states $\sigma_{inv,inv}, \sigma_{inv,t}, \sigma_{t,inv}$, and $\sigma_{t,t}$ are used to conclude $\sigma^{i,j}$ is correct on PSO.

According to $\sigma_{inv,inv}, \sigma_{inv,t}$ and $\sigma_{t,inv}$ are correct on PSO, invariant conditions $inv1$ and $inv2$ are used to abstract the effect of writes and local variable assignment by using $\sigma_{\text{arb}_\text{w}}$ and $\sigma_{\text{arb\_v}}$. Although the number of $\sigma_{\text{arb}_\text{w}}$ to abstract the effect of write events should equal the number of read events on other processing units, the number of read events issued by the loop to be considered is only the number of read operations and load-link operations inside the loop body.

In the derived symbolic execution states by the method, the read events issued by the loop behavior $\mathbf{do}\{\gamma_s \langle inv \rangle\}\mathbf{while}(c)$ are only considered in an arbitrary loop iteration. Thus, the number of read events to realize an arbitrary iteration equals the number of read operations and load-link operations appearing in the loop body γ_s. Thus, considering only the number of read events in the loop is enough to determine the behavior of arbitrary loop behavior. Therefore, loop behaviors on other programs must realize the sufficent number of possible effects from a loop iteration aribtrarily regarding their invariant condition.

Then, state $\sigma_{inv,inv}$ implies both invariant conditions, $inv1$ and $inv2$, are ensured on PSO for each loop iteration, while states $\sigma_{inv,t}$ and $\sigma_{t,inv}$ implies the invariant of a program is still preserved even if there are additional writes after the loop of other processing units. Consequently, the assumption that $\sigma_{inv,inv}, \sigma_{inv,t}$, and $\sigma_{t,inv}$ are correct on PSO can imply the invariant conditions are always satisfied for any iterations of both processing units.

Regarding symbolic execution state $\sigma_{t,t}$, the state considers both P_1 and P_2 are eventually terminated, such that

$$\sigma_{t,t} = \begin{pmatrix} (\sigma_0 \lhd \sigma_{\gamma 1}^0 \lhd \sigma_{s1}^0 \lhd \sigma_{c1}^0 \lhd \sigma_{inv1}^0 \lhd \sigma_{\text{arb\_v1}}^a \lhd \{\sigma_{\text{arb\_w1}} \lhd\}^{k1+1} \sigma_{inv\&c1} \lhd \sigma_{s1}^a \lhd \sigma_{\neg c1}^a \lhd \sigma_{\gamma 2}^0) \oplus \\ (\sigma_0 \lhd \sigma_{\gamma 3}^0 \lhd \sigma_{s2}^0 \lhd \sigma_{c2}^0 \lhd \sigma_{inv2}^0 \lhd \sigma_{\text{arb\_v2}}^a \lhd \{\sigma_{\text{arb\_w2}} \lhd\}^{k2+1} \sigma_{inv\&c2} \lhd \sigma_{s2}^a \lhd \sigma_{\neg c2}^a \lhd \sigma_{\gamma 4}) \oplus \theta_Q \end{pmatrix}$$

As the invariant conditions $inv1$ and $inv2$ are correct on PSO by the previous explanation, the effect of

$$\sigma_0 \lhd \sigma_{\gamma 1}^0 \lhd \sigma_{s1}^0 \lhd \sigma_{c1}^0 \lhd \sigma_{inv1}^0 \lhd \sigma_{\text{arb\_v1}}^a \lhd \{\sigma_{\text{arb\_w1}} \lhd\}^{k1+1}$$

and

$$\sigma_0 \lhd \sigma_{\gamma 3}^0 \lhd \sigma_{s2}^0 \lhd \sigma_{c2}^0 \lhd \sigma_{inv2}^0 \lhd \sigma_{\text{arb\_v2}}^a \lhd \{\sigma_{\text{arb\_w2}} \lhd\}^{k2+1}$$

are the abstractions of

$$\sigma_0 \lhd \sigma^0_{\gamma 1} \lhd \sigma^0_{s1} \lhd \{\sigma_{c1} \lhd \sigma_{s1} \lhd\}^{i-1}$$

and

$$\sigma_0 \lhd \sigma^0_{\gamma 3} \lhd \sigma^0_{s2} \lhd \{\sigma_{c2} \lhd \sigma_{s2} \lhd\}^{j-1}$$

, respectively, for any $i, j \in \mathbb{N}$. Then, σ^a_{s1} and σ^a_{s2} are considered as arbitrary state of any iteration i and j. Note that the abstraction is an over-approximation model regarding the invariant conditions $inv1$ and $inv2$. Regarding the invariants are given as the assumption on the computation on PSO, the assertion condition of each loop iteration is always satisfied because the behavior of each loop iteration is covered by the derived executions by the inductive invariant method. Then

$$\sigma_{inv\&c1} \lhd \sigma^a_{s1} \lhd \sigma^a_{\neg c1} \lhd \sigma^0_{\gamma 2}$$

and

$$\sigma_{inv\&c2} \lhd \sigma^a_{s2} \lhd \sigma^a_{\neg c2} \lhd \sigma^0_{\gamma 4}$$

are used to represent the last iterations of

$$\sigma^i_{c1} \lhd \sigma^i_{s1} \lhd \sigma^{inf}_{\neg c1} \lhd \sigma^{inf}_{\gamma_2}$$

and

$$\sigma^j_{c2} \lhd \sigma^j_{s2} \lhd \sigma^{inf}_{\neg c2} \lhd \sigma^{inf}_{\gamma_4}$$

, respectively. This means state $\sigma_{t,t}$ is the over-approximation state of $\sigma^{i,j}$ for PSO, where

$$\sigma^{i,j} = \begin{pmatrix} (\sigma_0 \lhd \sigma^0_{\gamma 1} \lhd \sigma^0_{s1} \lhd \{\sigma_{c1} \lhd \sigma_{s1} \lhd\}^{i \geq 1} \sigma^{inf}_{\neg c1} \lhd \sigma^{inf}_{\gamma 2}) \oplus \\ (\sigma_0 \lhd \sigma^0_{\gamma 3} \lhd \sigma^0_{s2} \lhd \{\sigma_{c2} \lhd \sigma_{s2} \lhd\}^{j \geq 1} \sigma^{inf}_{\neg c2} \lhd \sigma^{inf}_{\gamma 4}) \oplus \sigma_Q \end{pmatrix}$$

Intuitively, the all write events and read events to consider the evaluation of programs containing the loop behaviors are captured in state $\sigma_{t,t}$. Consequently, if the assumption is given, the assertion conditions appearing in the loop or outside the loop of state $\sigma^{i,j}$ are ensured on PSO. □

4.5 Conclusions

Due to a limitation of the bounded method in Chapter 3, the program could not be proved if there is a loop. Hence, this chapter adopts the inductive invariant approach, which is originally described in [DHKR11]. The original approach is proposed for a sequential program, which does not affect on relaxed memory models occur. However, we adopt the approach to our SMT-based program verification on a sequence of operation structures. In this method, the partial correctness of a sequence of operation structures can be ensured for partial store ordering (PSO); however, the stronger memory models would also be applicable for this method. The contribution of this method is the partial correctness can be ensured on PSO if the derived symbolic execution states are correct.

4.5.1 Contrary to Bounded Method

In contrast to the bounded method, first of all, the operation structure to be used in the inductive invariant method must be described in structured programming style. This would allow us to define the invariant condition to each loop and restrict the jump into/out the loop that could violate the invariant condition. However, there is a trade-off between the style of representations.

In the bounded method, term *execution path* is used to represent the whole path regarding the control flows to perform operations. Contrary, the inductive invariant method uses term *symbolic execution state* which is more abstract than execution path. Although the target of these terms is to be used in an SMT solver, the motivation of a symbolic execution state is to capture the intermediate step during the performing of operations. Besides, the inductive invariant method does not consider the whole path; however, the abstract states are considered for the loop behavior regarding the invariant condition. Therefore, the symbolic execution state could be a proper term for this method.

4.5.2 Achievements

By adopting the inductive invariant approach, the proposed inductive invariant method can show the partial correctness of a sequence of operation structures P on partial store ordering (PSO) if $\mathcal{M} \vdash P$, such that

$$\mathcal{M} \vdash P \implies \mathcal{M} \models P \quad \text{(where } \mathcal{M} \text{ is either TSO, PSO, or stronger memory models)}$$

In particular, the inductive invariant approach supposes the infinite loop iterations could be abstracted regarding the invariant condition to consider an arbitrary loop iteration. In theory, the arbitrary value is supposed to assign to the temporal registers and the memory locations appearing in the loop body for realizing the arbitrary effect of infinite iterations. However, to derive the abstraction for SMT-based program verification, the arbitrary effect is realized by considering the sufficient events to be considered. Consequently, those events and their relations can be encoded as the first-order formula for an SMT solver directly.

Besides, transform function *Inv* can automatically produce the symbolic execution states to be used in SMT-based program verification with a finite number of states. Thus, a bound is not needed for a loop anymore for SMT-based program verification on partial store ordering (PSO).

Moreover, as each of loop iterations is considered regarding the invariant condition, the method is an *over-approximation* approach which uses the invariant condition to determine the scope of write values for a loop iteration. In other words, the order of loop iterations is not considered in this method, in which there might be an order of updating values. This means if a violation of the program property is found, we cannot conclude there is a bug in the programs.

4.5.3 Limitations

In practical, assembly language usually allow a branch instruction to jump onto any program location, which is unstructured programming style. However, the operation structure used in this method restricts the expressiveness to structured programming style to support inductive invariant approach. Thus, an assembly program cannot be translated into an operation structure directly. Besides, the invariant conditions of loops are not generated automatically. Thus, the method needs a user to provide the invariant condition to each loop.

Besides, as the realization of write events excluding the loop behavior, the nested loop is not allowed in the proposed method. Moreover, although the method seems applicable to abstract the loop behavior regarding the invariant condition, the method is not applicable for any relaxed memory models, such as ARM and POWER.

Chapter 5

Experiment and Discussion

5.1 Case Study

Message passing the message passing is a famous program that could raise a bug in relaxed memory models. Figure 1-2 shows an implementation of message passing in ARM assembly language, and their corresponding abstractions can be shown by either Figure 3-7 or Figure 4-7 which are used in different methods. In program verification, assertion **assert**(`r2 = 1`) is added at line 6 in program `R2` in Figure 1-2. For the representation in an operation structure, the assertion condition has been changed to be `val = 1`, in which the condition to be ensured must rely on local variables Tmp. Besides, invariant condition ($val_n = 0 \vee val_n = 1$) is added to the loop to abstract the loop behavior for the inductive invariant method, as shown in Figure 5-1. For the bounded method, bound 1 is used to extract the execution paths from the loop behavior, then, each path is encoded with constraints of target specification provided by a modeling framework. As for the inductive invariant method, the corresponding execution states are used to construct the formulae instead of execution paths. Consequently, this research uses Z3 solver to find a valuation of each formula, if any.

A violation of this program can occur if the read value is not as same as the write value by the first write event to memory location `[X]`. This means the completing order of the write events and the completing order of the read events should be preserved to ensure the program correctness. As a result, the program property should not be proved on PSO and weaker memory models, while it can be proved on TSO and the stronger memory models.

Message passing with fence To preserve the correctness of message passing on PSO memory model, a proper fence operation must be added between write operations for preventing the reordering of the completing order of write events. For instance, the following execution structure must be added after line 8 of operation structure γ_1 shown in Figure 5-1.

```
instr{
  val := 1;
  r1 := val
};
instr{
  val := r1
  [x] := val
};
instr{
  val := r1
  [y] := val
}
```

Operation structure γ_1

```
do{
 instr{
  val := [y];
  r1 := val
 };
 instr{
  (rd := 1‖rt := r2);
  val_z := (rd = rt)?1:0;
  z := val_z;
  val_n := (rd = rt)?0:1;
  n := val_n
 };
 instr{
  val_n := n
 }
 ⟨val_n = 0 ∨ val_n = 1⟩
}while(val_n = 1);
instr{
  val := [x];
  r1 := val
};
assert(val = 1)
```

Operation structure γ_2

Figure 5-1: An operation structure of message passing for inductive invariant method

```
instr{
  STBar
};
```

`STBar` refers to fence instruction "stbar" in SPARC architecture to restrict the order of writes to be completed in the order. By adding this fence, the program property is then proved by PSO memory model, while it might not be proved on weaker memory models, such as ARM and POWER. In the case of ARM memory model, a fence, or memory barrier, must be added between the write operations and also between the read operations to prevent the effect of reordering. This means the following execution structure also injects after the loop.

```
instr{
  DMB
};
```

Spinlock for TOPPERS TOPPERS is a platform for embedded real-time systems [TOP10], such as automotive systems. The lock mechanisms of a Spinlock program in TOPPERS/FMP kernel is implemented for ARM processors, which is shown in Figure

```
L:  mov r2, #1
    ldrex r1, [lock]
    cmp r1, #0
    strexeq r1, r2, [lock]
    cmpeq r1, #0
    bne L
CS:
```

Figure 5-2: Mutex Lock mechanism of TOPPERS Spinlock

```
instr{
  ll(val, lock);
  r1 := val
}
```

ldrex r1, [lock]

```
instr{
  if(val_z = 1){
    val:= r2;
    sc(result, lock, val);
    r1 := result
  }
}
```

strex r1, r2, [lock]

Figure 5-3: Execution Structures for synchronize instructions

5-2. The key of this program is using synchronizing instructions, **ldrex** and **strex**, represented by execution structures in Figure 5-3. However, the program omits the interrupt instructions, `wfe`, because the correctness focuses on the read value of a read event.

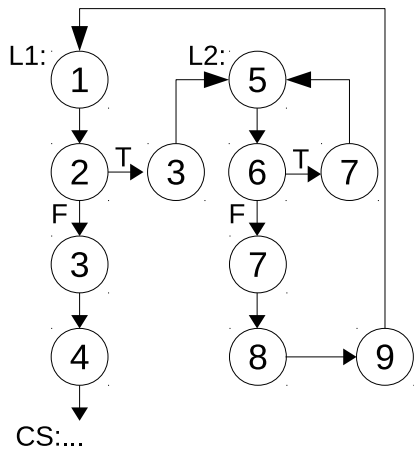
To ensure the program correctness, the mutual exclusion property must be preserved by every execution using our methods. However, the semantics of the synchronizing instruction, `ldrex` and `strex`, is not formally defined in neither herding cats [AMT14] nor Gharachorloo framework [Gha95]. For the verification purpose, the semantics of the synchronize instructions is defined regarding the semantics described in [SMO+12]. As the implementation of the synchronize instructions varies on processors, the semantics includes the arbitrary failure of a store condition instruction even if the condition is satisfied.

Spinlock for SPARC In prior work [MCA17], a Spin lock program implemented in Linux Kernel for SPARC was verified using the bounded method on the specifications of SC, TSO and PSO provided by Gharachorloo framework [Gha95]. In this manuscript, the program is also represented in an operation structure for the inductive invariant method to abstract the loop behavior. In particular, SPARC processors usually have the behavior of delayed branch instructions, in which the following instruction of a branch is executed before the branch is decided.

Figure 5-4(a) shows the corresponding of Spinlock programs written in an assembly language implemented in Linux Kernel for SPARC architecture. Spinlock program uses `ldstub` instruction which is a primitive instruction for SPARC. The read access and

```
1 L1: ldstub  [L],  r5
2   brnz, pn r5, L2
3   nop
4   ba CS
5 L2: ldub  [L], r5
6   brnz, pt  r5, L2
7   nop
8   ba, a, pt L1
9   nop
10 CS: ...
11 Unlock: stb g0, [L]
```

(a) Spinlock mechanism for SPARC processors



(b) Control flow graph of Spinlock for SPARC

Figure 5-4: Spinlock Implementation for SPARC

write access issued by the instruction appear to be performed atomically. Because of the behavior of delayed branch instructions in SPARC architecture, Figure 5-4(b) shows the control flow graph of the program that preserves the fetch-cycle semantics of SPARC architecture. The label of each node represents the line number of an instruction.

For the behavior of delayed branch instructions, the bounded method constructs the control flow graph corresponding to Figure 5-4 to capture the exact flows to be considered. For the inductive invariant method, the following instructions that contain a branch instruction and the following instruction

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

is abstracted by either forward branch

```
1 ifBr(¬(¬(val_r5 = 0))){
2   instr{ /* ldub [L], r5 */}
3   //L2
4 };
5 instr{ /* ldub [L], r5 */}
```

or backward branch

```
1 do{
2   //L2
3   // ...
4   instr{ /* ldub [L], r5 */}
5 }while(¬(¬(val_r5 = 0)));
```

Given two Spinlock programs, the lock mechanism of each program uses swap instruction `ldstub [L], r5` to atomically read and write to the same memory location `[L]`. By using

Table 5.1: Bounded Gharachorloo Framework

Program (bound)	*Run time (s)*			*Violation*		
	SC	**TSO**	**PSO**	**SC**	**TSO**	**PSO**
Message passing (1)	0.683	1.921	0.718	n	n	y
Message passing with fence (1)	0.761	1.722	1.776	n	n	n
SPARC Spinlock (1)	7.441	7.542	8.004	n	n	n
Dekker (1)	32.499	34.496	32.477	n	y	y
Peterson (1)	37.621	204.873	113.963	n	y	y
Known PSO bug (0)	2.896	125.707	2265.563	n	n	y

such atomic instruction, the mechanism seems not be affected by relaxed memory models.

Also, as an operation structure represents the program, the property of Spinlock mechanism could be verified on ARM memory model, which is memory model weaker than PSO memory model. One would notice that a program for the specific architecture is not needed to be checked under other memory models, such as ARM memory model. However, this case study also shows that our operation structure can capture the essential behavior issued the program, which could appear on other architecture.

Known PSO bug Figure 5-5 shows a real-world bug program, extracted from a massive program that can occur on a system using PSO memory model [kno]. The program can be executed safely on SC and TSO memory models, while an error occurs on PSO memory model. The error is caused by the write access to object `curPosition` can be happened before the write access to the field of the object, which is allowed by PSO memory model.

Although Java language has its memory model, called JMM [MPA05], the memory model of a programming language is orthogonal with the hardware memory models. This means the reordering caused at high-level memory model is not known in the low-level memory model. In Java Memory Model, the writes are allowed to be delayed as same as TSO and PSO memory model; the Java Virtual Machine also has no fence or barrier to disable the effect of reordering on a system using TSO and/or PSO memory model. Consequently, the reordering in PSO memory model causes a bug in Figure 5-5.

Figure 5-6(a) was introduced in [HH16] to capture the significant behavior that causes an error on PSO memory model, in which the initial values of the memory locations are $x = 1, y = 2$, and $z = 0$. In the programs, print statement at line 9 is supposed not to be performed. To verify the programs, our research considers the operation structures shown in Figure 5-6(b), in which the value of `val_z` is assumed to be 1 and the assertion requires `val_x` $= 2 \wedge$ `val_y` $= 4$ for all execution. Consequently, the programs are supposed to have a bug on PSO memory model, or weaker memory models.

```
class A {
    static Point currentPos = new Point(1,2);
    static class Point {
        int x;
        int y;
        Point(int x, int y) {
            this.x = x;
            this.y = y;
        }
    }
    public static void main(String[] args) {
        new Thread() {
            void f(Point p) {
                synchronized(this) {}
                if (p.x+1 != p.y) {
                    System.out.println(p.x+" "+p.y);
                    System.exit(1);
                }
            }
            @Override
            public void run() {
                while (currentPos == null);
                while (true)
                    f(currentPos);
            }
        }.start();
        while (true)
          currentPos = new Point(currentPos.x+1, currentPos.y+1);
    }
}
```

Figure 5-5: A real PSO bug in an electron microscope software [kno]. This bug caused a $12 million loss of equipment.

```
1 z=0
2 x=0
3 y=0
4 x=2
5 y=3
6 z=1
7 if(z == 1)
8   if(x+1 != y)
9     print(x,y)
```

(a) Simplified version [HH16]

```
1 instr{
2   [Z] := 0;
3   [X] := 0;
4   [Y] := 0;
5   [X] := 2;
6   [Y] := 3;
7   [Z] := 1;
8 }
1 instr{
2   val_z := [Z];
3 };
4 assume(val_z = 1);
5 instr{
6   val_x := [X];
7   val_y := [Y];
8 };
9 assert(val_x = 2 ∧
     val_y = 3)
```

(b) Simplified operation structure

Figure 5-6: Simplified programs for Known PSO bug

Table 5.2: Bounded Herding Cats

Program (bound)	*Run time (s)*			*Violation*		
	SC	**TSO**	**ARM**	**SC**	**TSO**	**ARM**
Message passing (1)	5.614	6.549	14.368	n	n	y
Message passing with fence (1)	5.724	6.721	36.595	n	n	n
TOPPERS Spinlock (1)	461.081	535.861	3385.248	n	n	n
SPARC Spinlock (1)	32.282	38.497	150.107	n	n	n
Dekker (1)	435.416	561.755	3697.089	n	y	y
Peterson (1)	267.575	74.058	523.214	n	y	y
Known PSO bug (0)	4.238	5.559	135.116	n	n	y

5.2 Experiment

In our research, an experimental tool was developed according to the proposed methods to verify operation structures with the program property under a relaxed memory model. The tool has been modified from the previous work [MCA17] to use the inductive invariant method and also encode the behavior based on herding cats framework. The tool encodes the symbolic execution states with the axioms of a memory model, based on a modeling framework into a first-order formula, and then uses the Z3 solver, an SMT solver, to solve the formula. The valuation of the formula found by the solver can be considered as a violation of program property.

For the bounded method, Tables 5.1 and 5.2 show the result of the tool to check whether the program, described by operation structures, is violated under a memory model within a bound or not. The encoding methods are implemented based on Gharachorloo framework [Gha95] and Herding cats framework [AMT14], in which the abstractions of programs executed on a memory model are different from each other. The bound value of each test program is also shown in its parentheses.

Table 5.3: Inductive Invariant Herding Cats (Runtime)

Program	*Run time (s)*			
	SC	**TSO**	**PSO**	**ARM**
Message passing	15.17	19.52	3.38	11.43
Message passing with fence	14.75	19.12	18.98	90.38
TOPPERS Spinlock	15089.42 (≈ 5 hours)	19584.4 (≈ 6 hours)	19219.8 (≈ 6 hours)	136009.13 (≈ 38 hours)
Peterson	4557.97	311.11	310.06	2459.23

Table 5.4: Inductive Invariant Herding Cats (Violation)

Program	*Violation*			
	SC	**TSO**	**PSO**	**ARM**
Message passing	n	n	y	y
Message passing with fence	n	n	n	n
TOPPERS Spinlock	n	n	n	n
Peterson	n	y	y	y

As Section 5.1 explains the details of programs: (1) Message passing, (2) Message passing with fence, (3) Spinlock TOPPERS, (4) Spinlock SPARC, and (5) Known PSO bug, tables 5.1 and 5.2 show the experiments on those programs and also Dekker's and Peterson's algorithms. The result in *violation* column show if there is an assertion violation, denoted by y and n for otherwise. From the experiment result, all bugs are raised as we expected. For *run time* column, the time includes symbolic analysis, the encoding process, and solving time for the encoded formula by Z3 solver [DMB08]. However, if there is any violating assertion, the program is then stopped immediately. All test results for the bounded method were produced on a MacBook Air with a 1.4 GHz Intel Core i5 processor running OS X 10.10.5 with memory 4 GB. As the state space to be verified is restricted by a bound, the program property cannot be proved though there is no violation; however, the bug founded by the method can disprove the program property on a memory model.

For the inductive invariant method, the experiment focuses on the program containing loops, which are: (1) Message passing, (2) Message passing with fence, (3) TOPPERS Spinlock, and (4) Petersons algorithm. Note that the tool considers the maximum number of the read events that can be produced from other programs to simplify the implementation. Although the implementation modifies the proposed method, it still is an over-approximation approach. The test results shown in Tables 5.3 and 5.4 were produced on RedHat Enterprise Server with 2.10 GHz Intel Xeon Silver 4116 processors with Memory 15.3 GiB. Each program was tested on the following relaxed memory models: SC, TSO, PSO, and ARM [1]. In each program, an invariant condition is added to the loop. For

[1] the specification is adopted from http://diy.inria.fr/cats/model-arm/herd.cat

instance, $val\_n = 1 \vee val\_n = 0$ is used as the invariant for message passing and message passing with fence, defined in the form of:

```
do{
  γ
  ⟨inv⟩
}while(c)
```

where inv is the loop invariant. For Peterson program, invariant $val\_z = 1 \vee val\_z = 0$ is used.

According to TOPPERS Spinlock shown in Figure 5-2, a store-condition instruction appearing in the loop body would affect the abstraction of the previous iterations whether there are write events issued by the instruction or not. To ensure the program correctness, the invariant must ensure the previous iterations must have no success write event issued by the instruction, in which the loop is terminated if a write occurs. Thus, the return flag to register r1 from instruction `strexeq r1, r2, [lock]` must be considered. Then, the operation structure to be verified is of the following form.

```
do{
  instr{ /* mov r2, 1 */ };
  instr{ /* ldrex r1, [lock] */};
  instr{ /* cmp r1, 0 */ };
  instr{
    val_z := z;
    if(val_z = 1){
      val := r2;
      sc(res, [lock], val);
      r1 := res
    }
    instr{ /* cmpeq r1, 0 */ };
  }
⟨(val_r1 = 1 ∨ val_r1 = 0) ∧ (res = 1 ∨ res = 0)⟩
}while(val_r1 = 1 ∧ res = 1)
```

In this operation structure, the condition $val\_r1 = 1 \wedge res = 1$ is used to ensure the values of val_r1 and res equal 1. This means there is no write event in the previous iterations if we need to continue the loop. In particular, if there is no res variable appearing in the execution state, the valuation of res will be an arbitrary value $*$ that can be evaluated to be $\top$ which is always satisfied.

As the loop invariant can abstract the behavior of the loop, the program property can be proved on PSO memory model, or stronger memory model, if there is no valuation found by the Z3 solver. However, we cannot disprove the program property on the memory model even if the solver found a valuation.

In program verification of mutual exclusion, the program property requires a program can enter its critical section, while other programs are not in their critical section. One could say no more than one program can enter its critical section simultaneously. In general program verification of mutual exclusion, the program property is defined by LTL

property or the assertion ensures a global variable always satisfied the conditions, such as:

```
mutex_lock(&lock);
//CS
cnt = cnt + 1;
assert(cnt = 1);
cnt = cnt -1;
mutex_unlock(&lock)
```

where `cnt` is a global variable. However, the consistency of a global variable in each memory model is different from each other. In other words, each program could observe a different result of the same global variable at the same time in a relaxed memory model. Thus, the program property for mutual exclusion could not be entirely verified on relaxed memory models.

To verify mutual exclusion, as our assertion language considers the local variables as the program property and an execution of each program is assumed to be eventually terminated, the following fragment of operation structure is added in each critical section of a program.

```
instr{
  val := [flag]
};
assert(val = 0);
instr{
  [flag] := 1
}
```

The initial value of global variable `flag` is supposed to be 0, and the variable is not used in any place, except the critical section. As read and write operations on variable `[flag]` is conflict, the completing order of these operations could be preserved on the most of relaxed memory models. As the assumption is added to each `mutex_lock(&lock)` to eventually lock the variable, the assumption has to be contradicted because two programs using `mutex_lock(&lock)` are supposed not to enter the critical section at the same time eventually. If there is a case that the assumption is not contradicted, there must be a case that a read access to `[flag]`can return value 1 as the read value, which violates the program property. To summarize, this verification property considers only the case of `mutex_lock(&lock)` is implemented correctly on a memory model, while the correctness does not include the case of `mutex_unlock(&lock)`.

5.3 Discussion

5.3.1 Encoded Formula

According to the encoded formula, event state ε is used to determine the data flow on the existing events. In both proposed methods, the finite events are considered for ensuring the evaluation of the computation of the program execution. Although there are quantifiers

such as $\forall$ and $\exists$, the scope of the quantifiers are restricted by the existing events provided by the methods, such as $\exists x \in \mathrm{Ev}_{\varepsilon}$. This means *whether a valuation of the formula exists* is decidable because there is a scope on instances using in the quantifiers.

Besides, the arithmetic computation used in the formula relies on the operator plus and minus. As the target of program verification is at the hardware-level, the arithmetic calculation on such operators would be sufficient. This means the encoded formula is decidable.

As for SMT solvers, the most of solvers would support quantifier-free formulae to be solved. In our experiment, Z3 solver supporting quantifier formulae is used to solve our encoded formula. However, as our quantifier formulae rely on the number of events which is finite, those formulae can be translated to quantifier-free formulae. For example, $\forall x, y, z \in \mathrm{Ev}_s.(f(x, y, z))$ can be translated as $\bigwedge_{x,y,z \in \mathrm{Ev}_s}(f(x, y, z))$. This means the formula would be able to use on other SMT solvers supporting first-order formulae.

5.3.2 Preciseness

In the bounded method, the valuation founded by the SMT solver can be considered as a flaw to contradict the program correctness. On the other hand, if the structures contain no loop, the correctness of programs can be ensured. This means our method is applicable to (1) verify assembly programs, represented by our operation structures, on target relaxed memory model and (2) automatically detect a flaw of the programs that can occur on target relaxed memory model.

In practice, a program can be either executed infinitely or eventually terminated due to the existence of a loop. However, as the modeling frameworks are adopted in our research to verify the program, the number of instances in an execution to be considered must be finite. Thus, our research assumes any execution to be considered must eventually be terminated. For instance, if the program to be verified contains an infinite loop as shown in Figure 5-7, our methods assume the execution is eventually terminated by adding assumption **assume**$(\neg(\top))$. However, the assumption condition makes the whole execution not to considered in the program verification, such that $\bot \implies a$ is always valid. This means if the program is probably executed infinitely the program property cannot be proved for the infinite case using our research. However, among the terminated programs, the research considers two methods to consider the program executions for verification.

By using the bounded method, a bound is given for restricting the number of instances to be considered in program verification. Due to the fact that (1) the loop can cause infinite executions and (2) the symbolic execution states is needed beforehand, in which the number of loop iterations cannot be determined systematically, the bound must be provided by users for program verification. Although the method is an under-approximation to consider the executions of loop behavior, the experiment in Tables 5.1 and 5.2 show the violation, denoted by y, is a bug that could be raised in the original program. Intuitively, for a relaxed memory model, this method can ensure (1) the execution is proved under a certain number of loop iterations, (2) the program property is disproved if the SMT solver finds a valuation.

```
1 label(L);
2 instr{  // ldr r1, [X]
3   val := [X];
4   r1 := val
5 };
6 branch(⊤, label(L))
```

(a) For bounded method

```
1 do{ // L
2   instr{  // ldr r1, [X]
3     val := [X];
4     r1 := val
5   }
6 }while(⊤)
```

(a) For inductive invariant method

Figure 5-7: Infinite Program

On the other hand, the inductive invariant approach used in software verification [DHKR11] is then adopted to abstract the loop behavior in our method. Originally, the inductive invariant considers abstracted control flows of a program execution using loop invariant to be preserved for every iteration of the program executed in a sequential way. By adopting in our method, the method is applicable for some memory models that do not permit the effect of following memory accesses in the program order to appear before a read access, in which total store ordering (TSO) and partial store ordering (PSO) are applicable for this approach. In other words, the partial correctness of programs containing loops can be ensured on TSO and PSO using *inductive invariant method*.

5.3.3 Expressiveness of Assertion Language

In our methods, assertion conditions and assumption conditions are injected into the operation structures to ensure the computation of local variables at the specific program points. As the interleaving of program executions is not considered explicitly, the conditions are used as program invariant to ensure the program property. Besides, the proposed assertion language can express a safety property in our target property, in which the values of the local variables at the specific program point affected by target relaxed memory model must satisfy the assertion condition. In addition, using assumption statements, the behavior of specific programs can be restricted to consider the safety property on such restricted behaviors. Intuitively, users are also allowed to make assumptions about specific program points to prove or disprove the safety property.

In our assertion language, the condition is expressed as a Boolean expression on temporal registers. Because of the consistency of a memory location, the value of a memory location could not be used in the assertion language as each program could see a different value on the same location at the moment. Thus, to check the value of a memory location, a processor must use a read access to the memory location and check the value returned by the read access.

Besides, as an execution of a program is assumed to be eventually terminated, some property such as liveness property could not be checked by the assertion language. However, as the assumption condition can help to restrict the behavior to be considered, users could make an assumption to the specific situations to consider the desired property.

In summarize, the assertion conditions and assumption conditions appearing in the

programs are used as the invariant of the programs, in which the conditions at the specific program locations can determine the intermediate computation at those points. To ensure the program correctness, users would add these conditions at the stable points of the programs, which is the program locations to ensure the property is always satisfied as expected.

5.3.4 Expressiveness of Operation Structure

In this research, an operation structure is proposed as an abstraction of an assembly program for program verification. In particular, the behavior of operation structures is sufficient to realize the effect regarding the memory model specifications provided by modeling frameworks. However, as the target of operation structures is to represent the behavior of assembly programs for program verification. Therefore, this section would like to show the expressiveness of our abstraction in describing the behavior of assembly programs.

Assumptions of Operation Structures

This research defines the operation structures regarding the assumptions in Section 3.2.1 to provide an abstraction of assembly programs. Due to the program verification focuses on the program property of computations that are affected by relaxed memory models, the abstraction regarding the assumptions is proposed to simplify only the necessary behavior for program verification. Although an operation structure could not represent most of the assembly program, the behavior represented by operation structures is useful for program verification on relaxed memory models.

For the assumption on assembly instructions to be considered, the instructions that affect the computation and the execution of concurrent programs are taken into account. The essential instructions that affect the computation of concurrent programs are: (1) arithmetic calculation instructions, (2) load and store instructions, and (3) read-modify-write instructions. For effect on program execution, branch instructions and predicated instructions are the concerns that affect the next computations during the program execution. Therefore, these behaviors are sufficient for describing the concurrent assembly programs that communicate with each other using load and store instructions, such as the following.

```
1 ldr r1, [0x01]
2 add r2, r1, 2
3 str r2, [0x02]
```

```
1 L:
2 ldr r1, [0x02]
3 str r1, [0x01]
4 cmp r1, 0
5 beq L
```

Besides, the assumption of granularity considers operations are granules of assembly instructions. In hardware's point-of-view, more than one action could be performed for an assembly instruction. By introducing an operation as an abstraction of action at the hardware level, various assembly instructions could be defined by the corresponding

collection of operations. For instance, the instructions `ldr r1, [0x01]` and `lwz r1, (0x01)` of different multiprocessors can be described by the same operation structure for program verification.

```
instr{
  val := [0x01];
  r1 := val
}
```

On the other hand, the indirect address of instruction `ldr r1, [r2]` could be defined as the following.

```
instr{
  address := r2;
  if( address = 1 ) val := [0x01];
  if( address = 2 ) val := [0x02];
  r1 := val
}
```

where the possible addresses are assumed to be `[0x01]` and `[0x02]`.

For the assumption of using natural numbers for computations and using basic operators, the abstraction can consider the simple calculations in the program verification. In practice, the data types such as word and byte would be used in different instructions, such as `lwz` and `lbz`. However, our abstraction considers those types for computation as an integer. For example, `lbz r1, (X)` and `lwz r1, (X)` are described in the same operation structure.

```
instr{
  val := [X];
  r1 := val
}
```

As the effect of relaxed memory model is the primary concern, the calculation is not considered in details for the simplicity of program verification.

For the assumption that shared-memory locations are known beforehand, the program verification can realize the memory locations to be considered explicitly. Besides, the number of memory locations used in concurrent programs is expected to be finite. Then, the decidable formula can be provided to realize the behavior based on the known memory locations. In practical concurrent programs, although the memory address could be calculated during the executions, the possible addresses for program verifications should be known beforehand for communication between concurrent programs.

Applicable Assembly Instructions

First of all, the read and write operations are the basic operations to be captured as the main concern for program verification on relaxed memory models. For example, `ldr r3,`

[X], load a value of address [X], can be represented by execution structure

```
instr{
  val := [X];
  r3 := val
}
```

However, our research captures the load and store operations, or read and write operations, in an abstraction such that the value of val is an integer number for simplicity instead of considering the way to encode the value as a bit-strings in a practical system. Besides, the physical address of a memory location is also abstracted by the bracket of the identity of a memory location, such as [X].

For the order of performing operations, called program order, our operation structure also captures the simultaneous performing of operations of an assembly instruction in addition to the performing in a sequential way. For example, the following execution structure can represent assembly instruction cmp r1, r2.

```
instr{
  (rt := r1 ‖ rd := r2);
  (val_z := (rt = rd)?1:0  ‖
  val_n := (¬(rt=rd))?1:0);
  (z := val_z ‖ n := val_n)
}
```

The purpose of using parallel connective ‖ is to indicate two execution structures can be performed simultaneously. In other words, this defines the program order as a partial order of the operations of a program. In this research, the operations are allowed to be performed simultaneously if there is no dependency on those operations.

The calculation of an operation structure relies on temporal registers, represented by arithmetic assignments. In this research, as the read and write operations are used to capture only the data flow, the arithmetic calculation is considered separately, such as val := val + 1.

For predicated instructions used in modern processors, the behavior of such instruction can be represented by the following execution structure, which is the corresponding structure of streq r1, [X].

```
instr{
  val_z := z;
  if(val_z = 1){
    val := r1;
    [X] := val
  }
}
```

As there is the limitation of a boolean expression, which considers on temporal registers, there must be a read operation to a register first, such as val_z := z, condition execution **if**(val_z := z){ ... } is then used to represent the decoded condition of eq in streq r1, [X].

In addition to read and write operations, there are special kinds of instructions that used for multiprocessor system: read-modify-write and synchronize instructions. For read-modify-write, the atomic annotation is used to indicate the read operation and write operation must appear that the completing order of the operations is not interrupted. For instance, test-and-set instruction `ldstub [L], r1` for SPARC architecture that reads memory location `[L]` and set the value 1 to memory location `[L]` can be represented by the following execution structure.

```
1 instr{
2   atom(val := [L]);
3   atom([L] := 1);
4   r1 := val
5 }
```

where annotation **atom** is considered as a wrapper to indicate two operations that are required for atomic instruction. For the synchronize instruction, load-link operation and store-condition operation are proposed to represent the behavior of a load-link and store-condition instruction directly.

In addition to test-and-set instruction and swap instruction, there could be compare-and-swap instruction that tests the value before the next access is instantiated. For instance, the compare-and-swap instruction `CMPXCHG r1, [L]` in x86 architecture could be described as the following description.

```
1 instr{
2   atom(val := [L]);
3   if(val = 1){
4     val := r1;
5     atom([L] := val)
6   }
7 }
```

In the description, the read value of the read operation is used in the condition in an instruction. If the condition is satisfied, there will be the write operation to be performed. In this case, the read operation and the write operation are required to appear atomically.

Applications for Program Verification

First of all, as the target for program verification is concurrent programs executed on relaxed memory models, a sequence of operation structures can be used to represent the sequence of concurrent programs directly. In program verification, the possible program executions are considered regarding the desired program property. To determine the program execution, as an assembly program can be described in an unstructured programming style, the operation structure defined in the bounded method uses branch operation to represent the effect of a branch instruction directly. For instance, branch instruction bne L where L is a label can be translated as the following description.

```
instr{
   val_n := n;
   branch(val_n = 1, label(L))
}
```

On the other hand, the inductive invariant method uses control flow statements explicitly for considering the loop behavior. To represent the control flow of the programs, users must manually provide the corresponding operation structure of an assembly program. For instance, assembly programs

```
L: ldr r1, [0x01]
     cmp r1, r2
     bne L
```

can be represented by

```
do{
   instr{ /* ldr r1, [0x01] */ };
   instr{ /* cmp r1, r2 */ };
   instr{
       val_n := r
   }
   ⟨val_n = 0 ∨ val_n = 1⟩
}while(val_n = 1)
```

where the execution structures of instructions `ldr r1, [0x01]` and `cmp r1, r2` are omitted. Then, our methods can uses these corresponding operation structures of practical assembly programs for program verification on relaxed memory models.

To determine the program correctness, the program property in our research is defined by property statements. As our research does not determine the step of program execution explicitly, the program property is considered as the invariant of programs to be preserved for any effect of program execution. In addition to assertion statements used in usual program testing, assumption statement is used in our research for program verification. In particular, the assumption statements are used to restrict the cases to be considered in program verification. For instances, the following concurrent assembly programs would have the program property that `r2` must always equal 1 for the execution that `r1` equals 1.

```
mov r3, #1
str r3, [X]
str r3, [Y]
```

```
ldr r1, [Y]
ldr r2, [X]
```

To defined the desired program property, the following sequence of operation structures can be used for program verification.

```
instr{ /* mov r3 */ };
instr{ /* str r3, [X] */ };
instr{ /* str r3, [Y] */ }
```

```
instr{ /* ldr r1, [Y] */};
instr{ val_r1 := r1 };
assume( val_r1 = 1);
instr{ /* ldr r2, [X] */};
instr{ val_r2 := r2 };
assert( val_r2 = 1)
```

In addition, as the target of SMT-based program verification is to provide a decidable formula, the basic operators + and - are considered for the computation during program executions. According to assembly programs, the basic operators have seemed to be sufficient for practical assembly programs. Besides, the computation is considered on natural numbers without restriction on the limitation of the maximum value or minimum value. For instance, the following assembly program

```
ldr r1, [0x02]
add r1, r1, 2
```

can be represented by the following operation structure

```
instr{
  val := [0x02];
  r1 := val
};
instr{
  val := r1;
  val := val + 2;
  r1 := val
}
```

Although the restrictions could be necessary for computation on practical processors, using natural numbers to realize the computation is sufficient for program verification, in which the behaviors on relaxed memory models is the main concern. However, it is possible to add such limitation or realize the restriction on the computation in future work, if the program property relates to such behavior.

Moreover, most of the concurrent programs would access the same memory locations, in which the memory locations to be accessed should be known beforehand. If the memory locations are not in scope, a bug might occur. For instance, the following concurrent assembly programs use memory locations `[X]` and `[Y]` to communicate with each other.

```
str r1, [X]
ldr r2, [Y]
```

```
str r3, [Y]
ldr r4, [X]
```

The corresponding operation structures can be represented as the following.

```
instr{
  val := [X];
  r1 := val
};
instr{
  val := [Y];
  r2 := val
}
```

```
instr{
  val := [Y];
  r3 := val
};
instr{
  val := [X];
  r4 := val
}
```

In practice, the memory location is indicated by a memory address which can be calculated during the program execution. Ones would notice our operation structures consider the memory locations symbolically without any computation on the memory address. However, as the memory locations are assumed to be known beforehand, the operation structures could use case analysis to determine the possible computation of memory address, such as

```
instr{ // ldr r1, [r2]
   val_r2 := r2;
   if(val_r2 = 1){ val := [0x01] };
   if(val_r2 = 2){ val := [0x02] };
   r1 := val
}
```

where `[0x01]` and `[0x02]` are symbolic memory locations that are provided beforehand. This means the calculation of memory address to indicate memory locations is done indirectly.

Limitations of Operation Structures

In practice, an assembly language is described in the unstructured programming style, in which the branch behavior can jump from any place of a program. However, the operation structure for the inductive invariant method restricts the behavior to be in structured programming style so as to support inductive invariant approach. In other words, the program is not allowed to jump into/out of a loop, which could violate the loop invariant conditions. Consequently, an assembly program cannot be transformed into an operation structure for inductive invariant method directly.

As for the expressiveness on instructions, there are remaining assembly instructions such as supervisor calls (SVC), interrupt instructions and instructions that change the processor status that cannot be represented by the proposed operation structure. As the main concern is to find the method that applicable for program verification on relaxed memory models, the instructions that issue the read access and write access have the highest priority for realizing the behavior in this research. However, additional instructions would be considered as a future work. For example, the interrupt instructions would affect the liveness property on embedded systems.

In practical assembly programs, the computation of a memory address can be known during the program execution. However, our abstraction uses a symbolic value to indicate

Table 5.5: Solving time of 3 execution paths of Dekker under SC

Gharachorloo		*Herding cats*	
# Events	Solving (s)	# Events	Solving(s)
12	0.007699	67	0.007831
14	0.003791	82	0.002714
10	0.003179	62	0.005685

Table 5.6: Encoding time of 3 execution paths of Dekker under SC

Gharachorloo		*Herding cats*	
# Events	Encoding (s)	# Events	Encoding (s)
12	2.344894	67	22.253717
14	3.091485	82	32.933337
10	1.298048	62	18.748727

the memory location such as `[X]` and `[Y]`. Although there is an assumption that memory locations would be known beforehand for concurrent programs, the programs could be allowed to compute the memory address in the scope. For instance, if the possible value of register `r2` is either 1 or 2, assembly instruction `str r1, [r2]` can access only memory address 1 or 2. However, it is possible to support the computation of memory address as a future work, in which the abstraction must be improved. For the encoding method for an SMT solver, the encoded formula would still be decidable because (1) the computation of memory address relies on the plus and minus operators, and (2) the number of memory addresses would be finite as the shared-memory locations should be known beforehand for concurrent algorithms.

5.3.5 Scalability

Tables 5.5 and 5.6 show the solving time and encoding time of three symbolic execution states of Dekker's algorithm executed under sequential consistency model; According to Table 5.5, the solving time of both modeling frameworks is not too different, however, the encoding time shown in Table 5.6 is quite different. In addition, Figure 5-8 shows the comparison graph between two modeling frameworks in bounded method to encode the Dekker's algorithm into a first-order formula. In the setting, the bound is set to be 1 and consider on sequential consistency model. For each sample, the encoding time of Gharachorloo framework seems less than the encoding method for Herding cats framework. In fact, the number of events (#Events) is affected by the formalization of a framework. The encoding method for Gharachorloo [Gha95] considers only shared-memory accesses as events, while the encoding method for herding cats [AMT14] also consider register accesses as events. Besides, the constraints that decide valid executions are also proposed in different ways; The ways to encode is also implemented in a different style, in which

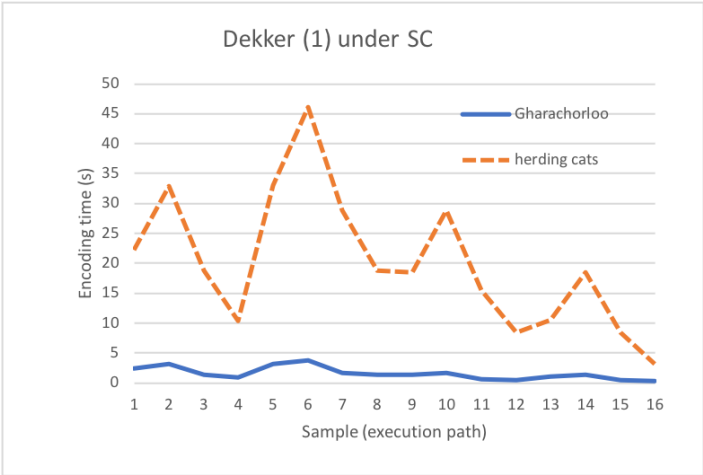


Figure 5-8: Encoding time of Dekker's algorithm

the encoding method for herding cats framework analyzes the behaviors of programs in detail than the encoding method for Gharachorloo framework. Because of the difference of formalization, the encoding time (Encoding (s)) in Table 5.6 is quite different.

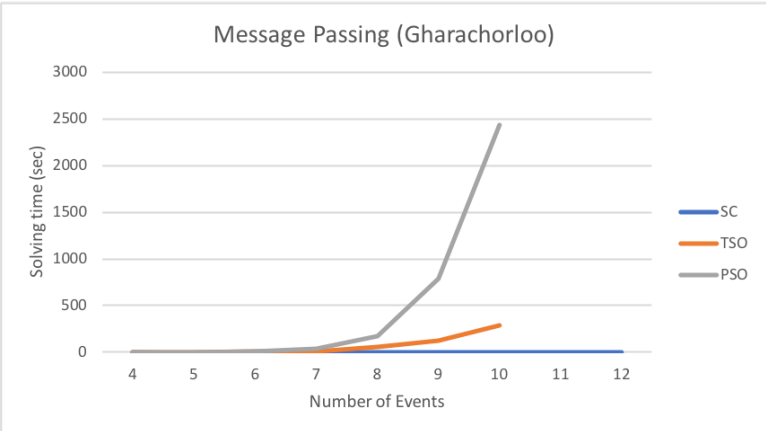


Figure 5-9: Solving time of Message passing on Gharachorloo framework

Moreover, Figures 5-9 and 5-10 show the graphs of solving time depending on the number of events. In the setting, message passing algorithm is considered with bounded loop unwinding approach to generate the number of events for the experiment. In both graphs, the solving times is growing differently in which the time of weaker memory models is growing faster than the stronger models. We presume that the state space to be considered in weaker memory models is more significant because the execution behavior is flexible than the stronger models.

According to Figure 5-9, the solving time of each relaxed memory model seemed to be growing too fast even if the number of events is too small. On the other hand, according to Figure 5-10, the solving time on Herding cats framework seems to be slower than

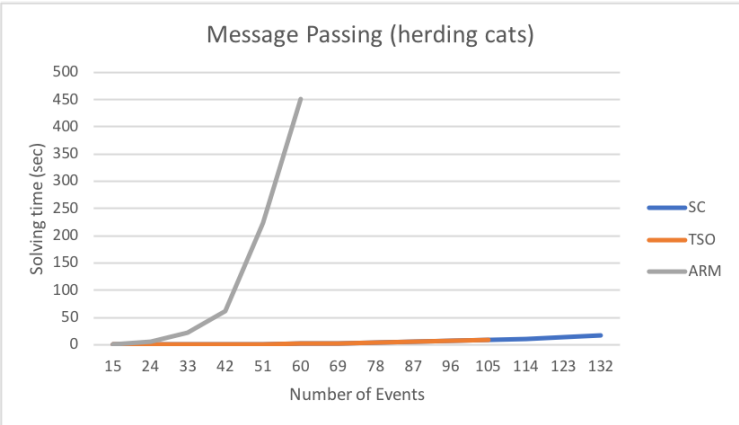


Figure 5-10: Solving time of Message passing on Herding Cats

Table 5.7: Experiment on the number of processors

	Runtime (s)			
# Processors	**SC**	**TSO**	**PSO**	**ARM**
2	14.67	19.03	18.77	88.86
3	123.47	153.18	150.56	1027.9
4	707.47	910.21	904.57	9547.81

Gharachorloo framework. However, if the programs are complicated and contain many instructions, the solving time also seemed to be increased exponentially.

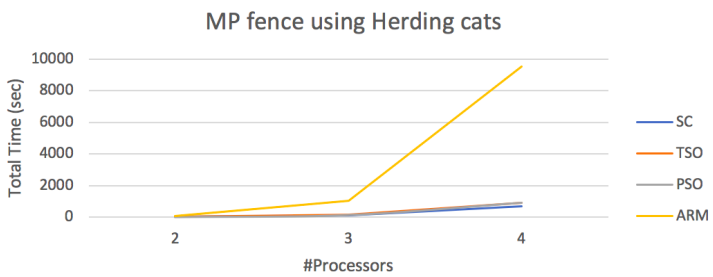


Figure 5-11: Experiment on the number of processors

Besides, there is an experiment on the number of processors to be considered. In this experiment, the message passing with the fence operations is considered by replicating the program containing a loop to consider more than two processors in the experiment. Table 5.7 and Figure 5-11 show if the number of processors is increased, the runtime is exponentially increased. The reason would be the number of processors increases the number of events to be considered, and the relations among processors would be more complicated than smaller processors.

According to the encoded formula to be solved by an SMT solver, the number of events

to be considered affects the instances in the formula for solving. Thus, the complexity of the formula depends on the number of events to be considered. To improve the scalability of the method in the future, one would provide abstraction techniques to reduce the number of events based on the fact of target memory model. For instance, the effect of the memory events occurs later than an event would not be considered in sequential consistency model.

Chapter 6

Related Work

6.1 Relaxed Memory Models

Although the effect of relaxed memory models causes the difficulty in program verification, there are several works to model the behavior of programs executed on a memory model and/or verify the program property on the behavior occurring on a relaxed memory model.

Typically, a high-level language could have its memory model, such as Java Memory Model (JMM) [MPA05], in which Java Virtual Machine (JVM) permits the write access of a thread to be delayed. By such allowance, the effect of concurrent programs on threads could be not the same as the programs executed sequentially. However, the high-level memory model is orthogonal to the memory model at the hardware level. In addition to Java, other programming languages such as C/C++ also has its standard of the memory model, called C/C++11 [BA08], to identify the behavior of reads and writes from threads.

In general, C++ and C are programming language which can be implemented as concurrent programs; however, there are standards [BA08] for the languages to specify the behavior of concurrent behaviors written by the language, such as a group of write accesses can be delayed. As the standard of the high-level memory model might not be formalized, there are works published by [BOS$^+$11] and [NMS16] to formalize the behavior of the C/C++ memory model. In addition to memory models for Java and C/C++, there is also the *weakly ordering model* proposed in [BP09] for high-level languages.

In contrast to high-level memory models, hardware memory models are provided for multiprocessor systems, in which the system would allow the effect of memory accesses to be completed out-of-order from the program order. Although the concurrent program is affected by memory model at high-level programming language and/or compiler optimization, the hardware memory models, such as ARM and POWER, do not know the behavior at the high-level. At the hardware level, the memory accesses issued by a processor in the program order are considered to realize the behavior on the system.

6.2 Program Verification for Relaxed Memory Models

In program verification at low-level programming language, one would provide the semantics of instruction set architecture (ISA) for specific architectures. [FM10] has faithfully formalized the instruction set architecture of ARMv7 processors in a monadic style. The model is described based on ARM manual document [ARM07]. This model is quite concrete to represent the execution of each instruction in ARM processor, due to the model is already validated by using the *random testing* approach with the practical system to confirm their model do represent the real behaviors of instructions. This means this formalization is applicable for program verification at the low-level programming language.

[AFI+09, AAS03] have captured the effect of behavior on relaxed memory models by events. The related work [AAS03] have proposed the model for POWER shared-memory architecture by using in-out operations. Such operations are used to capture the behavior of information that flows in the system. In contrast, the related work [AFI+09] has proposed an *axiomatic model* to consider the *valid execution* of events. In this work, concurrent programs are mapped into the corresponding events to interact with the shared-memory location, in which the semantics is extended from instruction semantics proposed by [FM10] to capture the micro-operations inside the hardware system.

Instead of considering the behavior of programs in details on specific architectures, there are several works consider the behavior of memory models in an abstraction that is sufficient to verify the program property on a relaxed memory model. One would provide program logic to reasoning the behavior of programs on target memory model to prove the program property, such as [DL15, Rid10, LV15, AM16]. [Rid10, LV15] provide the specific program logic for x86-TS0 and C++11 memory model, respectively, while [AM16] introduce 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 program execution are not derived from a memory model specification systematically.

Program Logic and Reasoning Method The proof system in [Rid10] adopts rely-guarantee reasoning to verify concurrent programs, which focuses on weak x86-TSO memory model to reason x86-assembly programs. To reason the behavior of a branch instruction, an invariant must be provided for the instruction to the target location. Besides, there is a method in [LV15] extends a well- known Owicki-Gries method to reason concurrent programs executed on non-interference criterion not to assume sequential consistency, which is sound for reasoning programs in the release-acquire fragment of the C++11 memory model. Therefore, these works can prove the concurrent programs on specific memory models.

In contrast to the proof system [Rid10] and reasoning method [LV15], our research adopts a modeling framework such as [AMT14, Gha95] to check the behavior of concurrent programs regarding the specification of a memory model, which is flexible to change a memory model to be verified. This means the existing events that can be instantiated

from concurrent programs must be given beforehand. In particular, our research focuses on the way to realize the symbolic execution systematically. Although the proof system [Rid10] and reasoning method [LV15] do not need the instantiating step beforehand to prove the programs, those works cannot deal with a variety of memory models.

Moreover, there is an invariant proof method [AC17] of programs for various relaxed memory models, in which the programs are described in a programming syntax, named LISA [AC16], and provides a new style of semantics to describe the behavior of programs for relaxed memory models. In program verification for the loop behavior, there is a counter variable to indicate a memory access on each iteration, and a program invariant is given as the program property. The invariant is then proved by providing a condition satisfying the invariant, and providing a specification in cats language [ACM16] satisfying the condition. To ensure the program property is proved on the specification of a memory model, a proof must be provided to show the behavior of provided specification is included in the specification of the target memory model.

Model Checking To the best of our knowledge, most of the ordinary model checkers must extend the execution of programs to include the behaviors on relaxed memory models. For TSO and POWER, various works use buffers in their semantics to realize the behavior such as [HH16, AAA+15]. As for POWER and ARM multiprocessors, the mechanisms of program executions also includes speculative executions and more complicated mechanisms; thus, the behavior of POWER and ARM multiprocessors would be difficult to realize the explicitly behaviors in the ordinary model checkers. This means some abstractions would be needed to extend the model checker to support the behavior, such as Shasha and Snir trace [SS88].

Instead of verifying program property directly, SATCheck [DL15] considers whether any execution of programs on a relaxed memory model preserved the effect of execution as same as executing the program on sequential consistency model. SATCheck tool computes a concrete execution for checking, in which the execution is used to construct an event graph to capture observed control flow paths of the program.

In this research, SMT-based program verification approach is adopted to model check the behavior of program executions regarding the memory model. Instead of considering the state explicitly, the states are abstracted in the encoded formulae. Then, the valid states to be considered are determined by axioms. Among the research on relaxed memory models, there are various works such as [Bur07, Hua15, HH16, AKT13].

CheckFence [Bur07] encodes the C implementation with a test program as the set of execution traces and defines the sets of execution traces for sequential consistency and relaxed memory model. In verification, the execution traces of relaxed memory model must behave in as executing in sequential consistency model. Besides, their relaxed memory model is an over-approximation model, which is weaker than TSO/PSO/RMO [WG], Alpha [Sit92], and IBM memory models.

SMC with Maximal Causality Reduction approach in [Hua15, HH16] construct an initial concrete execution for checking whether the execution can occur under target memory model using SMT solvers and then find new concrete executions until new executions

cannot be produced. [HH16] extended MCR approach for TSO and PSO models by relaxing the must-happens-before relations.

Our proposed approach to use SMT Solver is similar to [AKT13], which finds a witness execution from a litmus test. In contrast to the approach in [AKT13], the program assertions in addition to program executions are encoded into a formula to find a violating execution. The litmus test is, however, only checked whether the specific result is allowed or not. Also, their approach relies on herding cats framework [AMT14], while our work also considers Gharachorloo framework [Gha95].

In addition to using SMT-based program verification on the symbolic executions, [AAJL16] provides a stateless model checker for POWER multiprocessors that realize the execution on-the-fly as ordinary model checkers. Due to relaxed memory models, an execution is represented by Shasha and Snir trace [SS88]. Besides, the execution step is considered by fetching an instruction as fetched events. However, the effect of events will be considered if the event is committed. For the consideration of which events can be committed, the memory model specifications provided by Herding cats are considered as a predicated function. In contrast to our research, the decision of whether a transition is valid is considered based on the specification for the transition to commit an event, while our research realizes all possible executions finding a valid execution based on the relaxed memory model.

6.3 Symbolic Execution Analysis

In program verification, there are various techniques to extract the symbolic executions of a program in either static way or dynamic way. In addition, as the program could contain loops, the behavior of the loop is usually abstracted or is bounded by a number of loop iterations. However, most of the techniques rely on a sequential execution. This means the effect of each instruction in an execution is completed step-by-step. However, the effect of concurrent programs executed on relaxed memory models could contradict the usual assumption to generate symbolic executions.

In a system using a relaxed memory model, all of the existing execution could affect the data flown in the system. We have used a bounded loop unwinding method in [MCA17], which directly unwinds the loop within a bound. In addition, the method assumes every execution of a program must eventually be terminated, even if there is an infinite loop. Obviously, this is an under-approximation approach, because we need the finite number of instances to realize the valid execution regarding a memory model. Consequently, the symbolic execution can capture the issued events from the programs and the effect of programs on a memory model can be realized using a modeling framework.

Instead of analyzing symbolic executions, one would realize a concrete execution dynamically based on a relaxed memory model by considering the value of computation on-the-fly, such as [AAA+15, AAJL16, Hua15, Hua15]. Relaxed Stateless Model Checking (RSMC) approach [AAA+15, AAJL16] derives a concrete execution model, described by transition system, regarding a memory model for program verification using stateless model checker. Note that the derived model can be considered from the axioms of a mem-

ory model specified by cats language [ACM16]. The approach constructs an execution model by simulating the behavior of fetching and executing on-the-fly.

In software verification, there are various techniques to program verification with loop behavior, such as Hoare logic, assume-guarantee, and static/dynamic executions. The proof system [Rid10] provides the semantics based on rely-guarantee reasoning to abstract the behavior of other processes in a rely-condition, while our method abstracts a loop itself to capture the write values of write events issued by the loop. Note that loop invariant is supposed to be an assumption to guarantee the side effect of the loop behavior.

Chapter 7

Conclusion

This thesis introduces the methods to program verification of assembly programs for multiprocessor systems using relaxed memory models, which are the bounded method and the inductive invariant method. In particular, our research focuses on the side-effect of program executions that is affected by relaxed memory models. In our methods, the behavior of assembly programs is considered in an abstract way to capture the essential behaviors that are sufficient for program verification. For the program verification, existing frameworks to model the effect of a program execution executed on a relaxed memory model are adopted. To adopting the frameworks in program verification, (1) the way to analyze the program executions for the frameworks is proposed, and (2) an SMT solver is adopted to model the effect of the program execution under a relaxed memory model, in which an encoding method regarding a modeling framework is proposed.

In the motivation of our research, the concurrent programs should be ensured on a relaxed memory model. In particular, assembly concurrent programs is considered. In practical, there are various assembly languages regarding a variety of processor architectures. Thus, to provide a method for program verification, the abstraction level of assembly language is proposed as *operation structures*. An *operation structure* is used to capture the essential behavior of assembly instructions to verify concurrent programs on relaxed memory models.

In program verification on relaxed memory models, an SMT solver is adopted to realize the effect of a symbolic execution always satisfies the program property, in which existing frameworks are adopted to realize the *encoding methods* to abstract the behavior regarding a relaxed memory model into a first-order formula to be used in the solver.

Moreover, as the operation structure cannot be used in program verification directly, this research proposed two static analysis methods, which systematically considers the operation structures. Especially, the proposed analysis methods are used to realize the corresponding abstract executions to be considered in program verification on relaxed memory models.

To summarize, our program verification method is applicable to verify concurrent assembly programs represented as *a sequence of operation structures*. There are two methods to extract the behavior of program execution symbolically. The first method, bounded method, extracts the behavior to be considered directly and bound the number of in-

stances provided by loops for program verification. On the other hand, the inductive invariant method was introduced to deal with the loop behavior which is bounded by the previous method. In latter method, the loop behavior is abstracted by loop invariant to provide sufficient abstraction of program executions for program verification.

7.1 Advantages

According to the definition of our operation structure, the essential behaviors of assembly instructions are taken into account, in which the behaviors that could cause the flaw due to relaxed memory models are considered. As the target property to be verified is safety property, in which the read value of a read access is affected by relaxed memory models, the behaviors of assembly instructions are computation behavior and memory-related behavior. Thus, the operation structure seemed to cover the most of behaviors that sufficient for program verification on relaxed memory models.

For the bounded method, although the loop behavior is bounded to be an under-approximation approach, a valuation founded by this method can disprove the correctness of the concurrent assembly programs. Besides, if the programs contain no loop, the partial correctness can be ensured if there is no valuation founded by the SMT solver.

On the other hand, the inductive invariant approach proposed by [DHKR11] is adopted to analyze symbolic executions from a program. Although the original approach is proposed to a system using sequential consistency models, the approach seemed to be usable for relaxed memory models that do not allow a read access be affected by the following memory accesses, such as total store ordering (TSO) and partial store ordering (PSO). Intuitively, the read accesses cannot be delayed on the memory model. Consequently, if the corresponding symbolic execution states provided by the inductive invariant method do not violate the program property, the program property is proved for some memory models that do not allow delayed reads.

In SMT-based program verification, a valuation can be found automatically if there is any. Besides, both of our static analysis methods of symbolic executions is also considered automatically in a systematic way. Thus, the process of program verification can be done automatically.

7.2 Limitations

In program verification, the proposed assertion language can ensure the value of temporal registers at each program point, in which the safety property on the scope of possible values of a variable can be verified. However, in practice, the safety property could consider the value of a global variable to prevent some cooperate behaviors among programs must be ensured, such as mutual exclusion property. For instance, in general, the safety property of mutual exclusion requires two or more processes cannot enter the critical section at the same time, in which a global variable is usually used to ensure this property. According to our experiment, the correctness of mutual exclusion is not ensured completely, while

an assumption is made to program verification.

Although the operation structure proposed to abstract the assembly program, the operation structure used for inductive invariant method restrict the behavior to be in a structured programming style. This limitation is caused by the need that the inductive invariant approach needs the loop invariant to be specified directly to the loop. Consequently, the manual transformation of an assembly program into an operation structure is needed.

In the inductive invariant method, a loop invariant must be provided manually by users. Besides, as the approach has modified the original approach from [DHKR11], it has seemed not applicable to any relaxed memory models that are weaker than PSO.

7.3 Future Directions

As the limitation of assertion language is restricted to consider only the values of local variables, some program property might not be able to be verified by our approach. 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; This could extend the expressiveness to consider the situation of other programs. On the other hand, it would be possible to capture the changing value of a local variable as an automaton. Consequently, the program property could be able to define an LTL property on local variables, in which the expressiveness is improved, and the liveness property might be able to be ensured. However, all extension must be encoded in a proper first-order formula for SMT-based program verification. Also, to ensure the liveness property, the infinite execution should be considered in program verification in the future.

For the concurrent programs contain a loop, the inductive invariant approach is adopted to abstract the loop behavior for program verification. However, the approach requires the programs be described in a structured programming style, which could not present an assembly program directly. In software verification, there is a various approach to dealing with loops for SAT/SMT solvers, such as *k-induction methods* [DKR11, DKR10], and *combine-case k-induction* [DHKR11]. Those approaches would apply to explore the finite set of symbolic execution states systematically, in which those approaches might be able to cover more behaviors than the current methods.

Currently, a loop invariant must be proposed manually by users; there might be better if there is a suggestion mechanism for loop invariant. According to our experiment, the loop invariant used in program verification considers on the possible values of flag registers, which is either 1 or 0; However, it has seemed to be sufficient for program verification for total store ordering (TSO) and partial store ordering (PSO). Thus, the heuristic suggestion for our assertion language would be possible.

Publication

International Journal

[1] Pattaravut Maleehuan, Yuki Chiba, and Toshiaki Aoki, "A Verification Framework for Assembly Programs under Relaxed Memory Model using SMT Solver", IEICE Trans. Inf. & Syst., Dec. 2018.

International Conference

[2] Pattaravut Maleehuan, Yuki Chiba, and Toshiaki Aoki, "Assembly Program Verification for Multiprocessors with Relaxed Memory Model using SMT Solver", in International Symposium on Theoretical Aspects of Software Engineering, 2017, pp. 1-8.

[3] Pattaravut Maleehuan, Takashi Tomita, and Toshiaki Aoki, "Indictive Invariant Method for SMT-based Program Verification under Relaxed Memory Model". (to be submitted).

Bibliography

[AAA+15] Parosh Aziz Abdulla, Stavros Aronis, Mohamed Faouzi Atig, Bengt Jonsson, Carl Leonardsson, and Konstantinos Sagonas. Stateless model checking for tso and pso. In *Proceedings of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems - Volume 9035*, pages 353–367, New York, NY, USA, 2015. Springer-Verlag New York, Inc.

[AAJL16] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, and Carl Leonardsson. *Stateless Model Checking for POWER*, pages 134–156. Springer International Publishing, Cham, 2016.

[AAS03] Allon Adir, Hagit Attiya, and Gil Shurek. Information-flow models for shared memory with an application to the PowerPC architecture. *IEEE Transactions on Parallel and Distributed Systems*, 14(5):502–515, 2003.

[AC16] Jade Alglave and Patrick Cousot. Syntax and analytic semantics of LISA. *CoRR*, abs/1608.06583, 2016.

[AC17] Jade Alglave and Patrick Cousot. Ogre and Pythia: An Invariance Proof Method for Weak Consistency Models. *SIGPLAN Not.*, 52(1):3–18, jan 2017.

[ACM16] Jade Alglave, Patrick Cousot, and Luc Maranget. Syntax and semantics of the weak consistency model specification language cat. *CoRR*, abs/1608.07531, 2016.

[AFI+09] Jade Alglave, Anthony Fox, Samin Ishtiaq, Magnus O. Myreen, Susmit Sarkar, Peter Sewell, and Francesco Zappa Nardelli. The semantics of power and ARM multiprocessor machine code. *ACM SIGPLAN Notices*, 44(5):8, 2009.

[AKT13] Jade Alglave, Daniel Kroening, and Michael Tautschnig. Partial orders for efficient bounded model checking of concurrent software. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 8044 LNCS:141–157, 2013.

[AM16] Tatsuya Abe and Toshiyuki Maeda. *Observation-Based Concurrent Program Logic for Relaxed Memory Consistency Models*, pages 63–84. Springer International Publishing, Cham, 2016.

[AMP06] Alessandro Armando, Jacopo Mantovani, and Lorenzo Platania. Bounded Model Checking of Software Using SMT Solvers Instead of SAT Solvers. pages 146–162. Springer Berlin Heidelberg, 2006.

[AMP09] Alessandro Armando, Jacopo Mantovani, and Lorenzo Platania. Bounded model checking of software using SMT solvers instead of SAT solvers. *International Journal on Software Tools for Technology Transfer*, 11(1):69–83, 2009.

[AMT14] Jade Alglave, Luc Maranget, and Michael Tautschnig. Herding cats: Modelling, simulation, testing, and data mining for weak memory. *ACM Trans. Program. Lang. Syst.*, 36(2):7:1–7:74, July 2014.

[ARM07] ARM. *ARM Architecture Reference Manual (ARMv7-A and ARMv7-R edition)*. ARM, 2007.

[BA08] Hans-J. Boehm and Sarita V. Adve. Foundations of the C++ concurrency memory model. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation - PLDI '08*, page 68, 2008.

[BOS+11] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. Mathematizing C ++ Concurrency. *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 55–66, 2011.

[BP09] Gérard Boudol and Gustavo Petri. Relaxed memory models: an operational approach. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, volume 44, pages 392–403, 2009.

[Bur07] CheckFence : Checking Consistency of Concurrent Data Types on Relaxed Memory Models. *Memory*, 42(6):12–21, 2007.

[DHKR11] Alastair F. Donaldson, Leopold Haller, Daniel Kroening, and Philipp Rümmer. Software verification using k-induction. In *Proceedings of the 18th International Conference on Static Analysis*, SAS'11, pages 351–368, Berlin, Heidelberg, 2011. Springer-Verlag.

[DKR10] Alastair F. Donaldson, Daniel Kroening, and Philipp Rümmer. Automatic analysis of scratch-pad memory code for heterogeneous multicore processors. In *Proceedings of the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'10, pages 280–295, Berlin, Heidelberg, 2010. Springer-Verlag.

[DKR11] Alastair F Donaldson, Daniel Kroening, and Philipp Rümmer. Automatic analysis of DMA races using model checking and k-induction. *Formal Methods in System Design*, 39(1):83–113, aug 2011.

[DL15] Brian Demsky and Patrick Lam. SATCheck: SAT-directed Stateless Model Checking for SC and TSO. *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 20–36, 2015.

[DMB08] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[FM10] Anthony Fox and Magnus O. Myreen. A trustworthy monadic formalization of the ARMv7 Instruction set architecture. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 6172 LNCS, pages 243–258, 2010.

[Gha95] Kourosh Gharachorloo. Memory consistency models for shared-memory multiprocessors. Technical report, Stanford University, Stanford, CA, USA, 1995.

[HH16] Shiyou Huang and Jeff Huang. Maximal causality reduction for TSO and PSO. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications - OOPSLA 2016*, pages 447–461, New York, New York, USA, 2016. ACM Press.

[Hua15] Jeff Huang. Stateless Model Checking Concurrent Programs with Maximal Causality Reduction. In *Pldi*, pages 165–174, 2015.

[ISS12] Luc Maranget Inria, Susmit Sarkar, and Peter Sewell. A Tutorial Introduction to the ARM and POWER Relaxed Memory Models. https://www.cl.cam.ac.uk/ pes20/ppc-supplemental/test7.pdf, 2012.

[kno] A real-world bug caused by relaxed consistency. https://stackoverflow.com/questions/16159203/why-does-this-java-program-terminate-despite-that-apparently-it-shouldnt-and-d.

[Lam97] Leslie Lamport. How to make a correct multiprocess program execute correctly on a multiprocessor. *IEEE Transactions on Computers*, 46(7):779–782, 1997.

[LMP98] Jaejin Lee, Samuel P Midkiff, and David A Padua. *Concurrent static single assignment form and constant propagation for explicitly parallel programs*, pages 114–130. Springer Berlin Heidelberg, Berlin, Heidelberg, 1998.

[LV15] Ori Lahav and Viktor Vafeiadis. Owicki-Gries Reasoning for Weak Memory Models. In *Proceedings, Part II, of the 42Nd International Colloquium on*

Automata, Languages, and Programming - Volume 9135, ICALP 2015, pages 311–323, New York, NY, USA, 2015. Springer-Verlag New York, Inc.

[MCA17] Pattaravut Maleehuan, Yuki Chiba, and Toshiaki Aoki. Assembly program verification for multiprocessors with relaxed memory model using smt solver. In *2017 International Symposium on Theoretical Aspects of Software Engineering (TASE)*, pages 1–8, Sept 2017.

[MHMS+12] Sela Mador-Haim, Luc Maranget, Susmit Sarkar, Kayvan Memarian, Jade Alglave, Scott Owens, Rajeev Alur, Milo M. K. Martin, Peter Sewell, and Derek Williams. An axiomatic memory model for power multiprocessors. In *Proceedings of the 24th International Conference on Computer Aided Verification*, CAV'12, pages 495–512, Berlin, Heidelberg, 2012. Springer-Verlag.

[MPA05] Jeremy Manson, William Pugh, and Sarita V. Adve. The Java memory model. *ACM SIGPLAN Notices*, 40(1):378–391, 2005.

[NMS16] Kyndylan Nienhuis, Kayvan Memarian, and Peter Sewell. An operational semantics for C/C++11 concurrency. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications - OOPSLA 2016*, pages 111–128, 2016.

[Rid10] Tom Ridge. *A Rely-guarantee Proof System for x86-TSO*, pages 55–70. VSTTE'10. Springer-Verlag, Berlin, Heidelberg, 2010.

[RN07] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications: an Appetizer*. 2007.

[Sit92] Richard L. Sites, editor. *Alpha Architecture Reference Manual*. Digital Press, Newton, MA, USA, 1992.

[SMO+12] Susmit Sarkar, Kayvan Memarian, Scott Owens, Mark Batty, Peter Sewell, Luc Maranget, Jade Alglave, and Derek Williams. Synchronising C/C++ and POWER. *SIGPLAN Not.*, 47(6):311–322, jun 2012.

[SS88] Dennis Shasha and Marc Snir. Efficient and correct execution of parallel programs that share memory. *ACM Trans. Program. Lang. Syst.*, 10(2):282–312, April 1988.

[SSA+11] Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. Understanding POWER multiprocessors. *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation - PLDI '11*, 46(6):175, 2011.

[TOP10] TOPPERS Project Inc. TOPPERS Project. https://www.toppers.jp/, 2010.

[WG] David L Weaver and Tom Germond. The SPARC Architecture Manual. https://cr.yp.to/2005-590/sparcv9.pdf.

[Wik18] Wikipedia contributors. Satisfiability modulo theories — Wikipedia, the free encyclopedia, 2018. [Online; accessed 30-April-2018].