Axiomatic Timed Relaxed Concurrency Model

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

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

Abstract. We introduce a new memory model, named axiomatic timed relaxed concurrency model (ATRCM), that combines conceptual time with the relaxed memory model [48] building on top of the axiomatic candidate execution model [6]. The main objectives of the memory models are that is be close to ones used for real-world imperative programming languages (C/C++/LLVM) and that it be easily understood and used to design and verify compilers. Five properties are associated with ATRCM: clarity, operability, adaptability, suitability, and comprehensibility. It also includes two different scheduling orders: coherence and FIFO. The coherence ordered version is weaker than the previous C++ axiomatic memory models [24,22,8,38], while the FIFO ordered one is stronger. Both scheduling ordered ATRCM models have been proved sound in the theorem prover Isabelle [36,34] with respect to the previous models [22,24,38], while the coherence-ordered model has been proved complete with the previous. As an experiment, we use ATRCM with a compiler verification framework (Morpheus [31]) to verify some simple compiler optimizations on a subset of LLVM.

1 Introduction

A memory model provides the semantics of concurrency in a programming language, particularly one with imperative features, delimiting the allowed execution sequences, particularly regarding the values passed between different portions of the program execution via variables whose values are stored in memory. A memory model may be inaccurate in two main ways: it may allow undesirable executions, as for example, ones that could never occur on real machines, and it may fail to allow executions that are expected for common language implementations. Problems faced by memory machines include the distinction between atomic actions and non-atomic actions, and what reordering of memory events (reads and writes and some other things) are allowable.

ATRCM is based on a conceptual virtual machine that is an abstraction of modern real-world computers.

ATRCM is parametrized scheduling orders. Is this what you mean by "adaptability"?

I should have written this introduction a long time ago. It should include three or four main points. (1) the first point in the introduction, easy to understand. (2) to prove compiler correctness, need to take care additional data dependency so that we need to grab more information from the program. The way to do it is not through relation sets, but through real action definitions because it is more stright-forward with compiler proof. (3) a new po relation to substitute sb relation in a lot of previous proofs beacuse that is the real representation of program order not the sb relation. (4) more coherent memory model based on the bottom-up view, because compiler is building from bottom up, top down relation will cause some strange comfusion like the (d) and (e) examples in fig.3

A concurrency model defines the concurrency behaviors for a programming language. In the previous relaxed memory models, named candidate execution models [6,24,22,8,38], for big imperative languages like C/C++/LLVM, the concurrency model has been introduced in an axiomatic fashion. By giving a set of candidate executions, this kind of model defines relations and predicates for selecting valid executions. We present the axiomatic timed relaxed concurrency model (ATRCM) in a similar fashion. By introducing conceptual time, which corresponds to sequence numbers, to the traditional relaxed memory models, we are able to establish the border for out-of-thin-air behaviors clearly (Sec. 2). Additionally, we are able to use the candidate execution style with ATRCM to directly prove the correctness of compiler optimizations without redesigning an operational model. More importantly we achieve five different design goals of ATRCM that distinguish it from the previous models and allow people to understand and use it. These five properties are clarity, operability, adaptability, suitability and comprehensibility.

Clarity means users can understand the model easily. Eventually, any designed model needs to be mastered by programmers so that they can use it to design programming languages, program code and verify compilers. Prior memory models for C++ demonstrate the need for this clarify. For example, Lahav et al. [24] made a great achievement in repairing the C++ memory model, and the C++ documentation cites his work in their standard as the formal C++17 memory model [14]. However, the C++ documentation makes a mistake what sequentially-consistent ordered (sc) reads and writes should be, even though it cites Lahav et al.'s work. The C++ documentation states that an sc read or write is an acquire (acq) read or release (rel) write plus a global memory location value consistency. This allows the event structure (WWMerge) (see below) in Lahav et al.'s paper, but also mistakenly allows the (SB) structure that was disallowed in the same paper. The problem is that some mathematical structures involve so many complicated structures that people cannot precisely understand the meaning without a long fight with the math. The C++ example proves that even some language designers cannot precisely understand the meaning. Even if they understand the structure, language and compiler designs are usually constructed in a bottom-up fashion, so a top-down math structure can be hard to translate into real computer code. ATRCM operates in a bottom-up fashion so that users can easily use it.

initially, x = 0 and y = 0

$$\begin{array}{l} \texttt{write}_{\texttt{sc}}\ 1\ x \\ a := \texttt{read}_{\texttt{sc}}\, y //0 \end{array} \Bigg\|\ \begin{array}{l} \texttt{write}_{\texttt{sc}}\ 1\ y \\ b := \texttt{read}_{\texttt{sc}}\, x //0 \end{array} \ \texttt{(SB)} \quad \begin{array}{l} a := \texttt{read}_{\texttt{acq}}\, x //2 \\ b := \texttt{read}_{\texttt{sc}}\, y //0 \end{array} \Bigg\|\ \begin{array}{l} \texttt{write}_{\texttt{sc}}\ 1\ x \\ \texttt{write}_{\texttt{sc}}\ 2\ x \end{array} \Bigg\|\ \begin{array}{l} \texttt{write}_{\texttt{sc}}\ 1\ y \\ c := \texttt{read}_{\texttt{sc}}\, x //0 \end{array} \ \texttt{(WWMerge)}$$

Operability means a model can be mapped to real world implementation easily. The bottom-up design of ATRCM is one part of the story. ATRCM assumes a setting for a conceptual computer (Sec. 2) and every piece of the model is designed accordingly.

Adaptability is the ability to use a model in different situations. Many previous models [24,22,8] imprecisely defined the scope of the out-of-thin-air behaviors in an execution because a great effort was required to upgrade them to usable models in a different situation. The IMM model [38] was one such upgrade accomplished by including very large groups of relations from a program, and it is still not suitable for many situations. For example, it cannot tell the full story if an execution is generated from a program containing function calls, like the example in Fig. 2 ($b3$). ATRCM has a redesigned memory action set and is adaptable to different situations when dealing with out-of-thin-air behaviors.

Suitability keeps the right ideas in the right spots. Many previous models [21,38] have complicated input and system assumptions because of combining syntactic and semantic dependencies of programs. For example, they have complicated predicates to suggest that the executions from the program examples in Fig. 12 must allow memory actions from block `A` to execute before some actions sequenced-before the branching operator, because there are no harmful out-of-thin-air executions for any possible executions generated from the examples. However, the models still do not identify all possible syntactic dependencies. The focus should be to distinguish what is definable versus what is provable. ATRCM provides a definition with clear borders of what is an allowed execution (Sec. 3); then it uses technology (Sec. 4) like bisimulation to prove that additional executions are allowed because they are bisimilar to one of the allowed ones, which we call **behavioral expansion**.

Comprehensibility means that a model is general enough to be used in a real-world situation. For example, the sequential-consistency memory model is well-known, easy to understand, and adaptable to different situations. However, it is still not commonly used in modern hardware and programming languages because it is not comprehensible, and too restrictive and costly for modern computers. The soundness and completeness of ATRCM with respect to the previous memory models (Sec. 3.4) shows that ATRCM is a comprehensible model that can describe the behaviors of modern hardware and programming languages.

With these five principles in place – clarity, operability, adaptability, suitability, and comprehensibility – ATRCM is defined to be relaxed and axiomatic, and proved to be sound and complete compared with the previous models. In Sec. 4, we plug ATRCM into the Morpheus compiler verification framework.

2 Background and ATRCM Interesting Features

Here we provide an introduction of the key aspects of ATRCM with examples.

Basic Elements. Here, we describe the basic elements and give some examples of ATRCM. The model uses candidate executions. We first assume there is a type of thread IDs (TID), parameters ($NAME$), function names ($FNAME$), memory values (V), lock keys (L), time points (T) that form discrete total order without an infinite descending chain (natural numbers), action-IDs (AID), and memory locations (LOC). These are the element types that can appear in ATRCM. For a candidate execution, there is an underlying, black-boxed, control-flow-graph-based program executed by its own operational semantics that produces traces that can interact with the main memory. There is also an algorithm to produce the corresponding sequenced-before relation (`sb`) for a candidate execution. The `sb` relation in a thread is computed based on the order of evaluation of the program instructions to be executed on that thread, which simulates the order of an in-order execution for the program piece. The only assumption about the program is that its evaluation order is fixed without undefinedness. For a C/C++ program, we assume that a fixed evaluation order is given.

$$
\begin{aligned}
A = {} & \texttt{NRead}\ \mathit{bool}\ LOC\ AID\ \mathit{nat}\ \mathit{nat}\ (AID\ \mathit{set})\ [\mathbf{R}^{none}_{(LOC,\ldots)}] \\
& \texttt{|}\ \texttt{ARead}\ \mathit{bool}\ LOC\ O\ (AID\ \mathit{set})\ [\mathbf{R}^{O}_{(LOC,\ldots)}] \\
& \texttt{|}\ \texttt{NWrite}\ \mathit{bool}\ V\ LOC\ AID\ \mathit{nat}\ \mathit{nat}\ (AID\ \mathit{set})\ [\mathbf{W}^{none}_{(LOC,V,\ldots)}] \\
& \texttt{|}\ \texttt{AWrite}\ \mathit{bool}\ V\ LOC\ O\ (AID\ \mathit{set})\ [\mathbf{W}^{O}_{(LOC,V,\ldots)}] \\
& \texttt{|}\ \texttt{RMW}\ \mathit{bool}\ V\ V\ LOC\ O\ (AID\ \mathit{set})\ [\mathbf{RMW}^{O}_{(LOC,V,V,\ldots)}]\ \texttt{|}\ \texttt{Fence}\ O\ [\mathbf{F}^{O}] \\
& \texttt{|}\ \texttt{ControlFence}\ [\mathbf{CF}]\ \texttt{|}\ \texttt{CallFence}\ FNAME\ NAME\ [\mathbf{CAF}^{FNAME}_{NAME}]\ \texttt{|}\ \texttt{Lock}\ L\ \texttt{|}\ \texttt{UnLock}\ L \\
O = {} & \texttt{rlx}\ \texttt{|}\ \texttt{rel}\ \texttt{|}\ \texttt{acq}\ \texttt{|}\ \texttt{acqrel}\ \texttt{|}\ \texttt{sc}
\end{aligned}
$$

Fig. 1: Some Basic Elements in the Axiomatic Model

ATRCM describes the concurrent behaviors between different CPUs (only the memory units handling memory actions), caches and the memory bus. We call the combination of these devices the **memory machine**. The memory actions (A) in Fig. 1 are the atomic operations that the memory machine uses. Actions are generated in a trace of a program execution when it tries to perform certain instructions that might affect the concurrent behaviors. Throughout the paper, we use the term "lifting" to mean that the information of a candidate execution (`sb`, time point set and memory action happened at every time point, etc.) is generated from a trace of a program execution. The functionality of each action is described in Section 3.

The element in brackets at the end of some action grammars is its syntactic sugar that might be used in some examples in this paper with some information neglected. The ... operation inside a syntactic sugar means that we might show necessary information there (e.g. (AID *set*)) in examples later in the paper. We might also use the constructor of an action to mean the memory action or event. The `NRead` / `NWrite` actions are non-atomic memory read/write actions, while `ARead` / `AWrite` are atomic ones. read-modify-write (`RMW`) is an atomic operation similar to an LLVM `cmpxchg` operation. These five actions are referred to as **memory operations** (Some predicates use `is_mem_op` to classify an event to be one of memory operations). The other actions are not viewed as memory operations here, but they might affect the concurrent behaviors somewhat. `Lock` / `UnLock` represent an mutex lock instruction in a program. `ControlFence` represents a fence-like structure generated by a binary branching operation in a program (assuming that programs only have binary or unconditional branching operations) to preserve control dependency, while `CallFence` represents a fence-like structure generated by a function call or return instruction in a program. In Fig. 1, O represents the atomic memory operation orders relaxed (`rlx`), release (`rel`), acquire (`acq`), release-and-acquire (`acqrel`), and sequential consistency (`sc`).

Some useful relations based on a relation (R) in this paper are reflexive ($R^?$), transitive (R^+) and reflexive-transitive closures (R^*). $R|_x$ selects all relations in R accessing the memory location x. $\sqcup(\mathtt{E})$ calculates the supremum of a total order $\mathtt{E}$. $(\mathtt{E})|_{\rho,tid}$ means to form a new set with all time points whose memory event in ρ has the thread ID tid.

Candidate Executions and Examples. Here we define the structure of a candidate execution. A candidate execution contains six components including three different sets of threads (`Tids`), memory locations (`Locs`) and time points (`E`) representing a downward closed subset excluding `0` of natural numbers for the execution; a family (`sbs`) of sequenced-before relations (`sb`), one for each thread in `Tids`, defining the evaluation order in a thread based on the program execution trace generated from the in-order execution machine executing the underlying program of the thread; a reads-from relation (`rf`) defining the relation between a memory write and a read if they are in two different threads; and finally a bijective function ρ from `E` to memory events giving the interpretation of a time point with a memory event.

The conceptual time points represent all events for an execution line up in a sequence. We assume that all time point sequences start at time `1`, and that nothing happens at time `0`, the initial state for all locations and threads. Each `sb` relation in the family of `sbs` and the `rf` relation are pairs of `E`. A memory event is a tuple of a thread-ID, an action-ID and a memory action. The action-IDs in the co-domain of a ρ uniquely determine a memory event. This property is very important because we will see in Section 4.1 that the real sequenced-before relation generated only and directly by a program execution is an $\overline{\mathtt{sbs}}$ relation involving no time points. `sbs` is generated by providing an $\overline{\mathtt{sbs}}$, an `E` and a ρ.

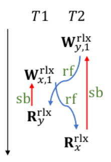
The general strategy for defining valid executions based on a set of candidate executions is to define predicates that select valid executions. In ATRCM, the process is divided into three parts. First, as in the previous axiomatic models [24,22,8,38], there are some assumptions to fulfill about the candidate executions. Mainly, they are well-formedness of candidate executions, and can be found in our Isabelle implementation (`https://github.com/liyili2/timed-relaxed-memory-model`), and some of them are in the appendix. Second, for each individual action type feature, in Sec. 3, we define an inductive set parameterized by an execution for that feature, to give the **restrictions** regarding which memory events should follow others in terms of time points. We call these sets **always-predicates**. One of the conditions that a valid execution should

$$(a)\ \begin{array}{l} a =_{\mathtt{rlx}} x \\ b = a \mathtt{\ +\ 1} \\ y =_{\mathtt{rlx}} \mathtt{b} \end{array} \Bigg\| \, x =_{\mathtt{rlx}} \mathtt{1} \ \Rightarrow\ (a1)\ \begin{array}{l} a = \mathbf{R}^{\mathtt{rlx}}_{x} //1 \\ \mathbf{W}^{\mathtt{rlx}}_{y,\mathtt{b}} //1 \end{array} \Bigg\| \, \mathbf{W}^{\mathtt{rlx}}_{x,1} \quad (a2)\ \begin{array}{l} 1.a = \mathbf{R}^{\mathtt{rlx}}_{x} //1 \\ 3.\mathbf{W}^{\mathtt{rlx}}_{y,\mathtt{b},\{1\}} //2 \end{array} \Bigg\| \, 2.\mathbf{W}^{\mathtt{rlx}}_{x,1}$$

$$(b1)\ \begin{array}{l} a =_{\mathtt{rlx}} y \\ x =_{\mathtt{rlx}} \mathtt{1} \end{array} \Bigg\| \begin{array}{l} b =_{\mathtt{rlx}} x \\ y =_{\mathtt{rlx}} \mathtt{1} \end{array} \quad (b2)\ \begin{array}{l} x =_{\mathtt{rlx}} \mathtt{1} \\ \mathtt{if\ (\ } a \mathtt{\ !=0\ )} \\ \qquad a =_{\mathtt{rlx}} y \end{array} \Bigg\| \begin{array}{l} y =_{\mathtt{rlx}} \mathtt{1} \\ \mathtt{if\ (\ } c \mathtt{\ !=0\ )} \\ \qquad b =_{\mathtt{rlx}} x \end{array} \quad (b3)\ \begin{array}{l} x =_{\mathtt{rlx}} \mathtt{1} \\ c = f\mathtt{()} \\ a =_{\mathtt{rlx}} y \end{array} \Bigg\| \begin{array}{l} y =_{\mathtt{rlx}} \mathtt{1} \\ d = g\mathtt{()} \\ b =_{\mathtt{rlx}} x \end{array}$$

$$(b4)\ \begin{array}{l} 1.a = \mathbf{R}^{\mathtt{rlx}}_{y} //1 \\ 3.\mathbf{W}^{\mathtt{rlx}}_{x,1} \end{array} \Bigg\| \begin{array}{l} 2.b = \mathbf{R}^{\mathtt{rlx}}_{x} //1 \\ 4.\mathbf{W}^{\mathtt{rlx}}_{y,1} \end{array} \quad (b5)\ \begin{array}{l} 1.\mathbf{W}^{\mathtt{rlx}}_{x,1} \\ 3.\mathbf{CF} \\ 5.a = \mathbf{R}^{\mathtt{rlx}}_{y,\{3\}} \end{array} \Bigg\| \begin{array}{l} 2.\mathbf{W}^{\mathtt{rlx}}_{y,1} \\ 4.\mathbf{CF} \\ 6.b = \mathbf{R}^{\mathtt{rlx}}_{x,\{\}} \end{array} \quad (b6)\ \begin{array}{l} \mathbf{W}^{\mathtt{rlx}}_{x,1} \\ \mathbf{CAF}^{f}_{5} \\ \cdots //f-code \\ \mathbf{CAF}^{f}_{5} \\ a = \mathbf{R}^{\mathtt{rlx}}_{y} \end{array} \Bigg\| \begin{array}{l} \mathbf{W}^{\mathtt{rlx}}_{y,1} \\ \mathbf{CAF}^{g}_{6} \\ \cdots //g-code \\ \mathbf{CAF}^{g}_{6} \\ b = \mathbf{R}^{\mathtt{rlx}}_{x} \end{array}$$

$$(c)\ \begin{array}{l} (T1, 1, a = \mathbf{R}^{\mathtt{rlx}}_{y}) \\ (T2, 2, b = \mathbf{R}^{\mathtt{rlx}}_{x}) \\ (T1, 3, \mathbf{W}^{\mathtt{rlx}}_{x,1}) \\ (T2, 4, \mathbf{W}^{\mathtt{rlx}}_{y,1}) \end{array} \quad \begin{array}{l} (T2, 4, \mathbf{W}^{\mathtt{rlx}}_{y,1}) \\ (T1, 3, \mathbf{W}^{\mathtt{rlx}}_{x,1}) \\ (T1, 1, a = \mathbf{R}^{\mathtt{rlx}}_{y}) \\ (T2, 2, b = \mathbf{R}^{\mathtt{rlx}}_{x}) \end{array}$$

(d)

T1
T2
$\mathbf{W}^{\mathtt{rlx}}_{y,1}$
$\mathbf{W}^{\mathtt{rlx}}_{x,1}$
rf
sb
rf
sb
$\mathbf{R}^{\mathtt{rlx}}_{y}$
$\mathbf{R}^{\mathtt{rlx}}_{x}$

$$(e)\ \begin{array}{l} T1 \\ \mathbf{W}^{\mathtt{rlx}}_{x,1} \\ \mathbf{W}^{\mathtt{rel}}_{z,1} \end{array} \Bigg\| \begin{array}{l} \quad T2 \\ a = \mathbf{R}^{\mathtt{acq}}_{z} //1 \\ \mathbf{F}^{\mathtt{sc}} \\ b = \mathbf{R}^{\mathtt{rlx}}_{y} //0 \end{array} \Bigg\| \begin{array}{l} \quad T3 \\ \mathbf{W}^{\mathtt{rlx}}_{y,1} \\ \mathbf{F}^{\mathtt{sc}} \\ c = \mathbf{R}^{\mathtt{rlx}}_{x} //0 \end{array}$$

Fig. 2: Programs, Event Structures and Diagrams

satisfy is that the union of all always-predicates (including `rf` but no `sbs`) has no edges from a later time point pointing to an earlier time point. Third, some features cannot be described easily by always-predicates alone. In such cases, we define predicates, named **never-predicates**, to rule out bad executions. It is satisfied if the behavior it describes never happens.

Here, we show some examples of ATRCM candidate executions. Throughout the paper, the variables a,b and c range over local variables in a program, while the parameters x,y and z range over memory locations or global variables representing locations. The variables s, t and r range over time points, and the parameters $T1$ and $T2$ range over thread IDs. Figure 2 provides some examples of **programs**, **event structures**, **ATRCM memory executions**, and **ATRCM execution diagrams**. (a), $(b1)$, $(b2)$ and $(b3)$ in Fig. 2 are C-like multi-threaded program pieces that represent a programmer's precise expectations regarding a piece of code related to memory. They are lifted to event structures $(a1)$, $(a2)$, and $(b4) - (b6)$. In them, $(a1)$ and $(b4)$ are the event structures lifted from programs (a), $(b1)$, $(b2)$ and $(b3)$ through traditional models [24,22,8,38], while the others are lifted through ATRCM. Again, "**Lifting**" in this paper means that an operational semantics executes a program and generates a list of memory events, named a **memory execution**, to communicate with the memory machine. During the process, all of the events in an execution are in a sequence (in terms of time points), and the operational semantics also provides (lifts) extra information to the memory machine (e.g. `sb`). The examples $(a2)$ and $(b4) - (b6)$ are examples of event structures representing sets of candidate executions in ATRCM. An **event structure** is a syntactic sugar that represents a set of memory executions. It does so by placing memory actions, representing memory events, in a program fashion with "||" operators that show the parallelism among the different threads. The value after the blue "//" operator is the value the read reads in a line, and it essentially puts a restriction on the event structure by requiring the value for a read in the structure. Sometimes, we show the action-ID as a number on the action in ATRCM event structures like the ($a2$), ($b4$) and ($b5$) (Fig. 2). As we have said, an action-ID can uniquely identify an event.

Event structures are just a way of presentation. What really happens in ATRCM is ATRCM memory executions, either candidate or valid ones, like the ones in (c) (Fig. 2). We show two out of the four candidate executions in (c) that are represented by ($b5$). In an execution, the "$\downarrow$" represents the flow of time. A lot of the time, the linear listing of events in an execution like (c) does not tell the whole story. We then use an ATRCM execution diagram to discuss the relations among the different events. (d) is an event diagram representing the second execution in (c). An execution diagram also has a time flow ($\downarrow$), and it lists all threads and their events in the right time points when they actually happen. In addition, we also list all restrictions by using pointed edges with names indicating which kinds of restrictions they are.

The Real-World Computer Setting. Before we discuss more details of ATRCM, we first describe the conceptual real-world computer setting that forms the basis for the the design and spirit of ATRCM. As mentioned, ATRCM executions try to describe the behaviors of a memory machine. We assume that the memory machine executes a memory event at a global time. All time points for an execution are described by the set `E` and the memory event in each time point is described by ρ. Each CPU represents the behavior of a thread, and the memory unit sees a list of memory events in sequenced-before order; but it might switch the order when executing them, so the time point order for a thread in an execution does not necessarily follow the sequenced-before order, as shown in Fig. 2 (e). Each thread only sees the local memory in its associated cache, so different threads might see different values for a memory location (see observable memory in Sec. 3.4). `sc` atomics ensure that different values for a location are merged into one value at the time they happen.

Finally, when memory events are moved to the memory bus, a scheduling order is maintained for all memory events from different threads. The most common one is **FIFO** order, where all write-read pairs happening in different threads are put in linear order. This disallows the (d) and (e) cases in Fig. 3. We also provide a scheduling order named ATRCM **coherence** order, which is weaker than FIFO order and the RC11/IMM coherence order [24,38], because it enables the (d) and (e) cases. The ATRCM coherence order has two requirements: (α) any thread views writes from other threads to the same location in the same order as the time order of the writes; (β) if a thread sends two writes, then any other thread sees the them in the sending order. α can be understood by Fig. 3 (a). The write in $T1$ happens before in time the write in $T3$ with the same location, so $T2$ sees $T1$'s write before $T2$'s. β can be understood by the (c) diagram, which is disallowed in ATRCM. $T2$ sees the write to y first, but the write to x must happen before in time the write to y, and it is invalid for the `rf` edges to cross. A direct implication here about a write-read relation in `rf` is that a read operation never reads from a write happening after it.

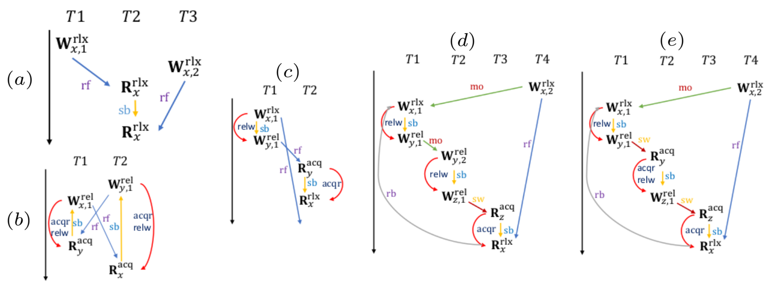


Fig. 3: ATRCM Example Diagrams

One of the problems in the traditional models [24,38] is that it is hard to justify the reason for some executions happening while others do not. For example, the executions in Fig. 3 (d) and (e) are borderline examples, meaning that they are not specifically disallowed in the hardware memory models (ARM/POWER models [39,29]), but no current compilers have enabled them. Allowing or disallowing them together is not a problem. The problem is that the traditional models allow (d) and disallow the other on the basis of the difference between a synchronized-with (`sw`) relation and a modification order relation (`mo`, similar to the α rule in ATRCM). An `sw` edge creates a happens-before relation between two events, but a `mo` edge just indicates the order of two writes for the same location. However, this is not the case here. There is an additional `rel` order on the write to y in $T1$ (d). A `rel` ordered write ensures that any other write does not go across it. In this case, the `rel` write ensures that the write to x must happen before the write to y, which creates a relation between the `mo` edges of x and y. According to the compilation scheme from their models [24,38] to POWER, cases (d) and (e) both generate light weight fences between the two elements in $T1$, $T2$ and $T3$. Essentially, from their own perspective, the two executions have no difference in terms of constraints when they are translated to POWER; so there is no reason for a model to accept one and disallow the other. In ATRCM, the FIFO ordered model disallows both, while the coherence ordered model allows them.

Properly Classifying Out-Of-Thin-Air Behaviors in Executions. One of the main targets of the paper is to lift enough dependency relations from programs for memory executions to properly identify out-of-thin-air behaviors, and so to allow proofs of compiler optimizations in an elegant framework. The similar programs $(b1)$, $(b2)$, and $(b3)$ in Fig. 2 illustrate one of the motivations. When they are lifted to memory executions in the previous models [24,22,8], all of them are lifted to $(b4)$ (which assumes that the functions f and g have no memory operations in ($b3$)). An execution diagram ((d) in Fig. 2) generated from the event structure ($b4$) is improperly classified as an out-of-thin-air behavior by these models. By creating more elegant action datatypes in ATRCM, we are able to insert more information in executions. For example, programs ($b2$) and ($b3$) are lifted to structures ($b5$) and ($b6$), respectively. With the control and call fences in between, if ($b5$) and ($b6$) appear to have the value `1` in both reads, an out-of-order behavior is definitely observed; while the (d) execution is just an outcome of a weak memory model.

3 The Axiomatic Model

In this section, we introduce different components of ATRCM, and then introduce the whole system and theorems.

3.1 The `ControlFence` and `CallFence` Actions

$$\begin{array}{l}\texttt{control\_dep(E,}\rho\texttt{,sb)} \equiv \\ \quad \{(s,t) | (s,t) \in \texttt{E}^2 \cap \texttt{sb}^+ \\ \quad \wedge (\texttt{is\_control\_fence}(\rho(s)) \wedge \neg\texttt{is\_read}(\rho(t)) \\ \qquad \vee \texttt{is\_control\_fence}(\rho(t)) \wedge \neg\texttt{is\_read}(\rho(s)))\}\end{array}
\qquad
\begin{array}{l}\texttt{call\_dep(E,}\rho\texttt{,sb)} \equiv \\ \quad \{(s,t) | (s,t) \in \texttt{E}^2 \cap \texttt{sb}^+ \\ \quad \wedge (\texttt{is\_call\_fence}(\rho(s)) \vee \texttt{is\_call\_fence}(\rho(t)))\}\end{array}$$

$$(a)\ \begin{array}{r}x =_{\texttt{rlx}} \texttt{1} \\ c = f\texttt{()} \\ a =_{\texttt{rlx}}\ y\end{array} \left\| \begin{array}{l} y =_{\texttt{rlx}} \texttt{1} \\ \\ b =_{\texttt{rlx}}\ x\end{array}\right. \Rightarrow (b)\ \begin{array}{l}\mathbf{W}^{\texttt{rlx}}_{x,1} \\ \mathbf{CAF}^{f}_{5} \\ \mathbf{W}^{\texttt{rlx}}_{z,1} \\ \mathbf{CAF}^{f}_{5} \\ a = \mathbf{R}^{\texttt{rlx}}_{y}\end{array} \left\| \begin{array}{l}\mathbf{W}^{\texttt{rlx}}_{y,1} \\ \\ \\ b = \mathbf{R}^{\texttt{rlx}}_{x}\end{array}\right. \xRightarrow{\texttt{Inlining}} (c)\ \begin{array}{l}\mathbf{W}^{\texttt{rlx}}_{x,1} \\ \mathbf{W}^{\texttt{rlx}}_{z,1} \\ a = \mathbf{R}^{\texttt{rlx}}_{y}\end{array} \left\| \begin{array}{l}\mathbf{W}^{\texttt{rlx}}_{y,1} \\ \\ b = \mathbf{R}^{\texttt{rlx}}_{x}\end{array}\right.$$

Fig. 4: The Properties of New Actions and An Inlining Optimization Example

Here, we focus on the first set of memory actions that did not commonly appear in the previous axiomatic models. The main purpose of ATRCM is to allow direct reasoning for the model with respect to compiler correctness. The way we solve this is to lift more information from the program that a candidate execution is lifted from. Lack of such information is the main reason that the previous models are not capable to be directly used by many compiler proof mechanisms. `ControlFence` and `CallFence` actions appear in ATRCM for the first time. They represent some important dependency relations in a program that the previous models neglected, and are one of the main motivations of ATRCM (see examples ($b5$) and ($b6$) in Fig. 2).

In compiling a program to CPU assembly code, a lot of fences will be generated to prevent CPU reordering instructions. In all current compiler implementations (C, C++, LLVM, etc), without specific flags (e.g. enabling function-inlining optimization), function calls are always surrounded by fences to prevent later instructions from being executed earlier than the content in the function calls. The `CallFence` action puts these fences around the function calls. For a functional call, a pair of `CallFences` are inserted to the top of the functional call and to the return statement. A `CallFence` has two arguments: the function name that it surrounds (*FNAME*), and a unique identifier (*NAME*) for a pair of `CallFences` that surrounds a function call. This identifier gives the power to define some aggressive compiler optimizations. For example, the definition of an inlining optimization on a function call in a program can be viewed in ATRCM as removing a pair of `CallFences`. One example of generating executions on the program applied an inlining optimization is in Fig. 4 (a), (b) and (c). The left is a program piece with a function call f in one thread. Let's assume that f has a single line of code to write `1` to a location z. The original program lifts an event structure like the one in (b) (Fig. 4) with `CallFences` surrounding the events inside function f. After we apply the inlining optimization, we remove the `CallFences` so that the event structure becomes (c). In this event structure, the write to z event is free to execute before the write to x event. `ControlFence` is an action lifted from a binary branching instruction in a program (not an unconditional branching), assuming that we have only binary and unconditional branching instructions in a program. They represent the control dependency in that program. Like the comparison of programs ($b1$) and ($b2$) in Figure 2, control dependency can determine if a program has out-of-thin-air behaviors or not. The achievement of `ControlFence` actions have been stated in Sec. 2.

In Figure 4, we define two always-predicates for `CallFences` and `ControlFence` actions as `control_dep`, `call_dep`. In all these definitions, the predicates `is_`*something* determines if a memory event's action is a or a group of certain actions (like the `is_mem_op` predicate in Sec. 2). For example, the `is_control_fence` defines if a given memory event's action is a `ControlFence`, and the `is_read` checks if the given action of an event is a read (`NRead`/`ARead`). `call_dep` restricts that for a `CallFences` , any sequenced-before-or-after action cannot happen after-or-before it in time (let's name this property as "goes across" for an action and "reordering" for a pair of actions in the rest of the paper). The `ControlFence` is a little different. A `ControlFence` does not allow any action other than a read to go across it. It means that ATRCM actually allows speculative read. As we have seen in the next section, the correctness of it is guaranteed by the additional data dependency defined in the (*AID set*) field of a memory operation.

3.2 Atomic Memory Operations and Fences

In this section, we discuss the action behaviors of the atomic memory operations (`ARead`,`AWrite` or `RMW`) and memory fences (`Fence`). The previous axiomatic models [6,24,22,8] define these actions based on a top-down view. They build general relations parameterized by individual candidate execution and see if there are cycles on some combinations of these relations, which indicate a violation of the properties for a valid execution. The problems of this approach is described in Sec. 1 and 2 when we discuss clarity and the real-world setting.

$$
\begin{aligned}
&\texttt{dd}(\texttt{E},\rho,\texttt{sb}) \equiv \quad \{(s,t) \in\texttt{E}^2 \mid (s,t) \in\texttt{sb}^+ \wedge\texttt{is_mem_op}(\rho(s)) \wedge \texttt{is_mem_op}(\rho(t)) \\
&\qquad\qquad\qquad \wedge\neg\texttt{is_both_read}(\rho(s),\rho(t))\wedge \texttt{same_loc}\ (\rho(s),\rho(t))\} \\
&\texttt{vol_dep}(\texttt{E},\rho,\texttt{sb}) \equiv \{(s,t) \in\texttt{E}^2 \mid (s,t) \in\texttt{sb}^+ \wedge\texttt{is_volatile}(\rho(s)) \wedge \texttt{is_volatile}(\rho(t))\} \\
&\texttt{add_dd}(\texttt{E},\rho,\texttt{sb}) \equiv \ \{(s,t) \in\texttt{E}^2 \mid (s,t) \in\texttt{sb}^+ \wedge\texttt{is_mem_op}(\rho(s)) \wedge \texttt{is_mem_op}(\rho(t)) \\
&\qquad\qquad\qquad \wedge\texttt{get_aid}(\rho(s)) \in \texttt{get_dep_set}(\rho(t))\} \\
&\texttt{acqr}(\texttt{E},\rho,\texttt{sb}) \equiv \{(s,t) \in\texttt{E}^2 \mid (s,t) \in\texttt{sb}^+ \wedge\texttt{is_read}(\rho(s))\wedge \texttt{is_acq}\ (\rho(s)) \wedge \texttt{is_mem_op}(\rho(t))\} \\
&\texttt{seqr}(\texttt{E},\rho,\texttt{sb}) \equiv \{(s,t) \in\texttt{E}^2 \mid (s,t) \in\texttt{sb}^+ \wedge\texttt{is_read}(\rho(s))\wedge \texttt{is_sc}\ (\rho(s)) \wedge \texttt{is_mem_op}(\rho(t))\} \\
&\qquad\qquad \cup\{(s,t) \in\texttt{E}^2 \mid (s,t) \in\texttt{sb}^+ \wedge\texttt{is_write}(\rho(s))\wedge \texttt{is_sc}\ (\rho(t)) \wedge \texttt{is_read}(\rho(t))\} \\
&\texttt{seqw}(\texttt{E},\rho,\texttt{sb}) \equiv \{(s,t) \in\texttt{E}^2 \mid (s,t) \in\texttt{sb}^+ \wedge\texttt{is_mem_op}(\rho(s))\wedge \texttt{is_sc}\ (\rho(t)) \wedge \texttt{is_write}(\rho(t))\} \\
&\qquad\qquad \cup\{(s,t) \in\texttt{E}^2 \mid (s,t) \in\texttt{sb}^+ \wedge\texttt{is_write}(\rho(s))\wedge \texttt{is_sc}\ (\rho(s)) \wedge \texttt{is_write}(\rho(t))\} \\
&\texttt{sr}(\texttt{E},\rho,\texttt{rf}) \equiv \\
&\qquad (s,t) \in\texttt{E}^2 \wedge s < t \wedge \texttt{is_write}(\rho(s)) \wedge \texttt{is_sc_fence}(\rho(t)) \wedge x = \texttt{get_loc}(\rho(s)) \\
&\qquad \wedge\neg\texttt{sc_fence_in_mid}(\texttt{E},\rho,s,t) \wedge \neg\texttt{write_in_mid}(\texttt{E},\rho,s,t,x) \wedge \neg\texttt{sc_read_in_mid}(\texttt{E},\rho,s,t,x) \\
&\qquad\qquad\qquad \Rightarrow (s,x,t) \in \texttt{sr}(\texttt{E},\rho,\texttt{rf})\ (*\ \textsf{scBase}\ *) \\
&\qquad \mid\ (s,x,r) \in \texttt{sr}(\texttt{E},\rho,\texttt{rf}) \wedge r < t \wedge t \in\texttt{E} \wedge\texttt{is_sc_fence}(\rho(t)) \wedge x = \texttt{get_loc}(\rho(s)) \\
&\qquad \wedge\neg\texttt{sc_fence_in_mid}(\texttt{E},\rho,r,t) \wedge \neg\texttt{write_in_mid}(\texttt{E},\rho,r,t,x) \wedge \neg\texttt{sc_read_in_mid}(\texttt{E},\rho,r,t,x) \\
&\qquad\qquad\qquad \Rightarrow (s,x,t) \in \texttt{sr}(\texttt{E},\rho,\texttt{rf})\ (*\ \textsf{scInduct}\ *) \\
&\qquad \ldots \\
&\texttt{cw}(\texttt{E},\rho,\texttt{rf}) \equiv \{(s,x,t) \mid (s,t) \in\texttt{rf} \wedge x = \texttt{get_loc}(\rho(s))\} \cup \texttt{sr}(\texttt{E},\rho,\texttt{rf}) \cup \texttt{tr}(\texttt{E},\rho,\texttt{rf}) \\
&\texttt{full_rf}(\texttt{E},\rho,\texttt{rf}) \equiv\texttt{rf} \cup\{(s,t) \mid (s,x,t) \in \texttt{tr}(\texttt{E},\rho,\texttt{rf})\} \\
&\texttt{sc_co_cw}(\texttt{E},\rho,\texttt{rf}) \equiv \\
&\quad \{(s,t) \mid (s,t) \in \texttt{full_rf}(\texttt{E},\rho,\texttt{rf}) \wedge (\exists r \in\texttt{E}\ .\texttt{is_sc_write}(\rho(r)) \wedge \texttt{same_loc}(\rho(s),\rho(r)) \wedge s < r < t)\} \\
&\quad \cup\{(s,t) \mid (s,t) \in \texttt{full_rf}(\texttt{E},\rho,\texttt{rf})\wedge \\
&\qquad\quad (\exists r \in\texttt{E}\ .\texttt{is_sc_fence}(\rho(r)) \wedge s < r < t \wedge (s,\texttt{get_loc}(\rho(s)),r) \notin \texttt{cw}(\texttt{E},\rho,\texttt{rf}))\} \\
&\quad \cup\{(s,t) \mid (s,t) \in \texttt{full_rf}(\texttt{E},\rho,\texttt{rf})\wedge \\
&\qquad\quad (\exists r \in\texttt{E}\ .\texttt{is_sc_read}(\rho(r)) \wedge s < r < t \wedge (s,\texttt{get_loc}(\rho(s)),r) \notin \texttt{cw}(\texttt{E},\rho,\texttt{rf}))\} \\
&\quad \cup\{(s,t) \mid (s,t) \in \texttt{full_rf}(\texttt{E},\rho,\texttt{rf}) \wedge \neg(\exists r \in\texttt{E}\ .s \neq r \wedge (r,\texttt{get_loc}(\rho(s)),t) \in \texttt{cw}(\texttt{E},\rho,\texttt{rf})\} \\
&\texttt{non_fifo}(\texttt{E},\rho,\texttt{rf}) \equiv\ \{(s,t) \mid (s,t) \in \texttt{full_rf}(\texttt{E},\rho,\texttt{rf})\wedge \\
&\qquad\qquad (\exists(r,r') \in \texttt{full_rf}(\texttt{E},\rho,\texttt{rf}).(s > r \wedge r' > t) \vee (s < r \wedge r' < t))\} \\
&\texttt{non_at_co}(\texttt{E},\rho,\texttt{rf}) \equiv \\
&\quad \{(s,t) \mid (s,t) \in \texttt{full_rf}(\texttt{E},\rho,\texttt{rf}) \wedge (\exists(r,r') \in \texttt{full_rf}(\texttt{E},\rho,\texttt{rf}).r < s \wedge t < r' \\
&\qquad\qquad\qquad \wedge\texttt{same_loc}(\rho(s),\rho(r)) \wedge \texttt{same_thread}(\rho(r'),\rho(t)))\} \quad (*\ \alpha\ *) \\
&\quad \cup\{(s,t) \mid \exists r\ r'.(s,r) \in \texttt{full_rf}(\texttt{E},\rho,\texttt{rf}) \wedge (t,r') \in \texttt{full_rf}(\texttt{E},\rho,\texttt{rf}) \wedge \texttt{same_thread}(\rho(s),\rho(t)) \\
&\qquad\qquad \wedge\texttt{same_thread}(\rho(r),\rho(r')) \wedge ((s < t \wedge r > r') \vee (s > t \wedge r < r'))\} \quad (*\ \beta\ *) \\
&\texttt{co_cw}(\texttt{E},\rho,\texttt{rf}) \equiv\ \texttt{sc_co_cw}(\texttt{E},\rho,\texttt{rf}) \cup \texttt{non_at_co}(\texttt{E},\rho,\texttt{rf}) \\
&\texttt{co_cw_fifo}(\texttt{E},\rho,\texttt{rf}) \equiv\ \texttt{sc_co_cw}(\texttt{E},\rho,\texttt{rf}) \cup \texttt{non_fifo}(\texttt{E},\rho,\texttt{rf})
\end{aligned}
$$

Fig. 5: Atomic Memory Operation and Fence Behaviors

In ATRCM, based on real world settings (Sec. 2), we define these actions based on a bottom-up view. We define the properties (always-predicates and never-predicates) for an action based on the relations between it and others. The machinery for defining the properties can be divided into two groups based on the memory order components of the actions. First, if there is no `sc` ordered actions in an execution, each action only has direct effects in the thread executing it. The effects observed from other threads are in fact side-effects of the action because of the non-determinism of the whole system. Each action with a `sc` order has and only has one global effect; that is, it maintains a global consistency of memory values for certain memory locations. This is one of the foundational principles in this paper.

We first discuss the single-threaded behaviors for all these actions including the `sc` atomics but not including their global effects. To do so, we define always-predicates with three elements provided: a set of time points (`E`), an `sb` relation, and a function ρ (Sec.2). The always-predicates declare the restrictions when a memory event must follow another one. Figure 5 provides the definitions for the always-predicates for a few cases.

The first three always-predicates we defined for the memory operations are `dd`, `vol_dep` and `add_dd`. In these sets, `is_both_read` checks if two events' actions are both reads, `get_aid` gets the action-ID of an event, and `get_dep_set` gets the additional dependency set in an event (the (*AID set*) field in memory operations in Fig. 1). The `same_loc` predicate is to test if two events have the same locations. The `dd` defines the traditional data dependency restrictions for two time points if their event's actions (in ρ) are memory operations accessing the same memory locations. The `vol_dep` set defines the restrictions on time points representing volatile memory operations. The volatility of a memory action is given by its *bool* value (see Fig. 1 and is checked by the `is_volatile` predicate). Our model adopts the volatile concept from LLVM, i.e. two volatile memory operations cannot go across each other. One of the contributions in this paper is the creation of the `add_dd` relation, which literally means "additional data dependency". It collects the data dependency relations that cannot be captured by `dd`. An example is the program (a) and its lifted execution $(a2)$ in Fig. 2. As we have said, we add the action-ID set `{1}` in the write to y in $(a2)$ to ensure the write never goes across the event with action-ID `1`, which is the read to x.

In ATRCM, we have defined an always-predicate for each of the combination of a memory order (excluding `rlx`) and a memory operation or fence (no `acq` write or `rel` read). Here, we only show the `acqr`, `seqr` and `seqw` sets to put restrictions on `acq AReads`, `sc AReads` and `sc AWrites`. In these sets, the predicate `is_write` tests if an event's action is `AWrite`/`NWrite`/`RMW`. Based on the real-world setting, the single thread behavior of these atomics is just to reorder memory events in an execution. The always-predicates for the atomics in a single thread are to put restrictions on the reordering. For example, the `acqr` set prevent any sequenced-after events for a `acq AReads` to happen before it. The `seqr` set for a `sc ARead` additionally requires all writes sequenced-before the `sc ARead` to not happen in time after it. The `seqw` set for `sc AWrites` firstly requires that the memory operations sequenced-before the write not to go across it (similar to the restrictions for `rel AWrites`), and secondly requires writes sequenced-after the `sc AWrites` to not go across it. The other atomics have always-predicates similar to the ones here.

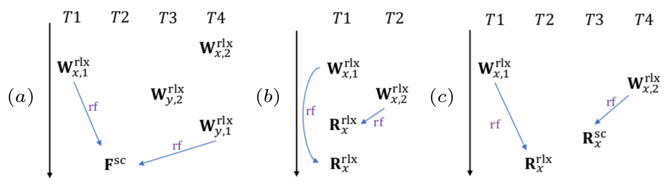

$T1$ $T2$ $T3$ $T4$
$\mathbf{W}^{\text{rlx}}_{x,2}$
(a)
$\mathbf{W}^{\text{rlx}}_{x,1}$
rf
$\mathbf{W}^{\text{rlx}}_{y,2}$
$\mathbf{W}^{\text{rlx}}_{y,1}$
rf
$\mathbf{F}^{\text{sc}}$
$T1$ $T2$
(b)
$\mathbf{W}^{\text{rlx}}_{x,1}$
rf
$\mathbf{W}^{\text{rlx}}_{x,2}$
$\mathbf{R}^{\text{rlx}}_{x}$
rf
$\mathbf{R}^{\text{rlx}}_{x}$
$T1$ $T2$ $T3$ $T4$
(c)
$\mathbf{W}^{\text{rlx}}_{x,1}$
$\mathbf{W}^{\text{rlx}}_{x,2}$
rf
rf
$\mathbf{R}^{\text{sc}}_{x}$
$\mathbf{R}^{\text{rlx}}_{x}$

Fig. 6: Diagrams For `cw` and `co_cw`

Building on the single-threaded behavior, we now define the global effects of `sc` atomics by defining the multi-threaded behaviors. We adopt the setting that the memory structure has caches, so an access from a thread to a memory location might only see or affect the value in a cache without affecting the whole memory at a time. In addition, the memory machine maintains requests from different threads in FIFO or coherence order (Sec. 2).

To achieve the global effects, we define an always-predicate `cw` and a never-predicate `co_cw` to maintain the coherence order (`co_cw_fifo` for FIFO order) based on an input time point set `E`, `rf` relation, and function ρ. The `cw` set is to add restrictions on top of `rf`, and it represents a family of time point relations: one per memory location. It assumes that all restrictions in `rf` is valid, and adds two more sets: (1) write-read pairs between a `sc` fence/read and its most recent write (as `sr`), and (2) all remaining write-read pairs in a single thread that does not in `rf` (as `tr`). The concept of `tr` is simple. For any reads not showing in `rf`, they are reads reading values from writes in the same thread, so we pick the latest writes in the same thread for them in `tr`. `sr` is to find the latest writes for `sc` fences and reads. For an `sc` fence in an execution, we view it as a read to observe and unify all values from different caches for each location in the execution, and the value stored in each location is from the latest write, as we shown in Fig. 6 (a). In this diagram, we build write-read pairs from the latest writes to x and y with the fence in $T2$. We do this for both `sc` fences and reads (also $RMWs$) in defining the `sr` always-predicate in Fig. 5, which is defined inductively. In these inductive rules. `same_thread` checks if two events happening in the same thread, `sc_fence_in_mid` checks if there is a `sc` fence happens in between the two time points in its argument, $write\_in\_mid$ checks if if there is a write to a certain location (x) happens in between the two time points in the argument, and `sc_read_in_mid` checks if there is a `sc` read from a certain location (x) happens in between the two time points in the argument. The two rules in Fig. 5 for `sr` defines the base and inductive cases for handling `sc` fences, and there are other rules omitted to handle `sc` reads.

We also define the never-predicate `co_cw` based on the `cw` and `full_rf` (`rf` + `tr`) sets of an execution. The definition of it is to collect bad behaviors with two parts. A valid execution has an empty `co_cw` set. The first part `sc_co_cw` suggests that for any write-read pair (a, b) in the `full_rf` set (defined in Fig. 5, basically `rf`+`tr`), there is no any `sc` atomics (for the same location as the event in a) in between whose write value not from a. This can be understood by the diagram in Fig. 6 (c). Performing `sc` atomics is a way to unify all values from different caches into one for a location (or for all locations in an `sc` fence). If one sees a read ($T2$) reads a value from a write ($T1$), and in the middle of the time, a `sc` read happens and it is not reading from the same write as the $T1$'s read, then there must be something wrong with the `sc` read because it does not guarantee the requirement for unifying all values. In Fig. 5, `sc_co_cw` have a definition of a union of four different sets. The first three are for `sc` write, fence and read, respectively. The fourth one checks if a `sc` read potentially reads from two different locations because `cw` might contain edges for a `sc` read different from the original write-read pair for the read.

The second part is divided into two possible never-predicates: `non_fifo` and `non_at_co`, which are to ensure different orders (FIFO and coherence orders). They are described in in Sec. 2. An example of violating the orders is in Fig. 6 (*b*). The examples in Fig. 6 represent a very important design principle of ATRCM. At first, it seems that the definitions of the `cw` and `co_cw` are too restrictive. However, all these always-predicates and never-predicates are just to eliminate specific invalid executions. In Sec. 4, we will see that what we really want to prove is about the program. Based on a program, a set of executions are generated by given all possible ways that time points and rf relations can form. Based on this set, we can check if the program produces harmful executions by analyzing invalid executions, or if an optimized program preserves its original program by comparing two sets of valid executions. Hence, if a program is designed properly, there must be valid executions. For example, if the underlying program of the execution in Fig. 6 (*b*) is designed properly, it must exist symmetric executions that does not violate the FIFO order or coherence order; e.g., an execution with the second read from x in $T1$ happens before the first read in the same thread.

The `full_rf` set defined here is essentially the `rf` relation is previous work. In Sec. 3.4, we have a lemma about the uniqueness of every read in `full_rf`. We have introduced all atomics in ATRCM here, and we will introduce non-atomic operations and locks in the next section.

3.3 Locks and Non-Atomic Memory Operations

$$
\begin{array}{l}
\texttt{termInMid}(\texttt{E}, \rho, s, t, e) \equiv \exists r. r \in \texttt{E} \wedge s < r < t \wedge (\exists tid\ aid. \rho(r) = (tid, aid, e)) \\
\texttt{termInSameTid}(\texttt{E}, \rho, s, t, e) \equiv \exists r. r \in \texttt{E} \wedge s < r < t \wedge (\exists tid\ aid. \rho(r) = (tid, aid, e)) \\
\qquad\qquad\qquad\qquad\qquad \wedge \texttt{same\_thread}(s, r) \wedge \texttt{same\_thread}(r, t)
\end{array}
$$

$$
\begin{array}{ll}
\texttt{lock\_dep(E,}\rho\texttt{,sb)} \equiv & \texttt{unlock\_dep(E,}\rho\texttt{,sb)} \equiv \\
\quad \{(s, t) | (s, t) \in \texttt{E}^2 \cap \texttt{sb} & \quad \{(s, t) | (s, t) \in \texttt{E}^2 \cap \texttt{sb} \\
\quad \wedge(\texttt{is\_lock}(\rho(s)) \vee \texttt{is\_unlock}(\rho(t)))\} & \quad \wedge(\texttt{is\_unlock}(\rho(s)) \vee \texttt{is\_unlock}(\rho(t)))\}
\end{array}
$$

$$
\begin{array}{l}
\texttt{lock\_prop}(\texttt{E}, \rho, s, l) \equiv (\forall t\ r \in \texttt{E} . t > s \wedge \texttt{same\_thread}\ (s, t) \wedge r > s \\
\qquad\qquad \Rightarrow \texttt{get\_action}(\rho(t)) \neq \texttt{UnLock}\ l \wedge \texttt{get\_action}((\rho(r)) \neq \texttt{Lock}\ l) \\
\qquad\qquad \vee(\exists t \in \texttt{E} . t > s \wedge \texttt{same\_thread}(s, t) \wedge \texttt{get\_action}(\rho(t)) = \texttt{UnLock}\ l \\
\qquad\qquad \wedge\neg\texttt{termInMid}(\texttt{E}, \rho, s, t, \texttt{Lock}\ l) \wedge \neg\texttt{termInSameTid}(\texttt{E}, \rho, s, t, \texttt{UnLock}\ l)) \\
\texttt{locks}(\texttt{E}, \rho) \equiv \forall t \in \texttt{E} . (\forall l . \texttt{get\_action}(\rho(t)) = \texttt{Lock}\ l \Rightarrow \texttt{lock\_prop}(\texttt{E}, \rho, t, l)) \\
\texttt{race\_def}(\texttt{E}, \rho) \equiv \exists s\ t\ r \in \texttt{E} . \texttt{is\_non\_atomic}(\rho(t)) \wedge \texttt{is\_non\_atomic}(\rho(r)) \\
\qquad\qquad \wedge\neg\texttt{both\_read}(\rho(s), \rho(r)) \wedge \texttt{same\_loc}(\rho(t), \rho(r)) \wedge \texttt{same\_aid}(\rho(t), \rho(r)) \\
\qquad\qquad \wedge\texttt{get\_i}(\rho(t)) = 1 \wedge \texttt{get\_i}(\rho(r)) = \texttt{get\_n}(\rho(r)) \\
\qquad\qquad \wedge t < s < r \wedge \texttt{exist\_other\_op}(\rho, t, \texttt{get\_aid}(\rho(t)), \texttt{get\_loc}(\rho(t)))
\end{array}
$$

Fig. 7: Non-Atomic Actions and Lock Consistency Predicates

Now let's bring in the non-atomic memory operations and locks, which are the `NRead`, `NWrite`, `Lock` and `UnLock` actions (Fig. 1). These actions are different from those in the previous model [8]. The previous model achieves races for non-atomic memory operations by assuming that two non-atomic operations can happen at the same time with the assumption of dividing memory locations into atomic/non-atomic memory locations. This not only complicates the model design, but it's also not based on the real-world setting. In the real-world setting, there is no distinction between atomic and non-atomic memory locations. Additionally, races are not usually caused by two actions happening at exactly the same time, since the CPU usually guarantees that such behavior never happens. The real problem is that a non-atomic memory instruction is compiled to a list of small atomic memory operations, so the CPU might execute a part of the memory operations, and then execute memory operations from other instructions, and so on. This overlapping execution process in the CPU causes races if the program is not written carefully with locks. Batty `et al.` create the idea of mutex memory locations (MML), and the keys for locks in their model are MMLs. They also create a predicate to check if all keys are MMLs. Even though this implementation has a basis in reality, it is too low-level. The only advantage of implementing lock keys as memory locations is to be able to verify that no one is doing harmful things to the lock keys. However, keys and mutex locks are managed by OS in a modern computer. The case when someone can do unsecured behaviors to locks is just too rare. It should not be a focus in axiomatic models, so we decided to implement lock keys as a separate type, This way we can keep ATRCM focusing on the big pictures without going into too many lower-level details.

The always-predicates for the non-atomic memory actions and locks are in Figure 7. The design spirit of the non-atomic memory operations in ATRCM contains two key points. First, a non-atomic memory instruction in a program is lifted by splitting into a list of memory actions (`NRead` and `NWrite`) that have the same action-ID and are sent as requests to the memory machine. Second, ATRCM requires each time point to work on exactly one action (one event), so a non-atomic memory instruction is not going to finish at one time point if it is split into a list with length more than one. In this case, for a group of non-atomic actions with the same action-ID, there is a time range from the time when the first action executes to the time when the last one does. In a candidate execution, if two groups of non-atomic memory actions are accessing the same location and their time ranges overlap, there is a race. The complicated argument structures for the syntax of `NRead` and `NWrite` are to satisfy these two key requirements. The *bool* value in a `NRead` and `NWrite` action indicates a volatile action, the LOC is the memory location, and the (AID *set*) deals with additional dependency that is defined in Fig. 5. The AID associates a group of non-atomic actions originally from a same memory instruction. There are two *nat* type fields in a `NRead` or `NWrite` action. The first one, named an i value, is the i-th action in the group happening in an execution. The second one, named an n value, is the total number of actions in a group. For a well-formed candidate execution, we assume that any i-th action in a non-atomic action group in an execution never happens in any other rank (e.g. happening as the $i-1$-th,$i+1$-th one). This assumption helps us simplify the proofs in Sec. 3.4.

We adopt the locking mechanism from the Standard C mutex library. We assume there is a possibly infinite set of keys (L) acting as mutex locks. `Lock` and `UnLock` actions have one L field representing the key to hold or unhold. When the `Lock` of key l happens in an execution, it means that starting from that time, the thread of the `Lock` event holds l. In contrast, the `UnLock` with key l means that from that time the thread stops holding l. During the time range between a `Lock` action and an `UnLock` action on the same key l in the same thread, there cannot be another `Lock` action, which is in any thread, in a valid execution.

The non-atomic memory operations are no different from atomic memory operations in terms of the behavior definitions in ATRCM, if they are not weaker than the atomic memory operations. The key point for defining their behaviors is to respect the assumptions of non-atomic memory operations for a candidate execution in Section 2. Other than the one defining the appearance of the i value in a thread for a list of non-atomic actions with the same AID, another assumption that only the last action, whose i and n values are the same, in a group of non-atomic actions with the same action-ID can join the restrictions in a `rf` or `full_rf` set. The `tr` set defined in Fig. 5 avoids the story about the non-atomic actions. To complete the `tr` set, we need another set of restrictions. The interesting thing is interactions between non-atomic memory actions with mutex locks. The predicate `race_def` (Fig. 7) defines when a race can happen. An execution results in race if, between a non-atomic operation's first action time (when $i = 1$, time $t1$) and last action time (when $i = n$, time $t2$), there exists a time t whose action has different action-ID as the actions in $t1$ and $t2$, and they are accessing the same memory location.

We now define the behaviors of locks. The predicate `locks` defines all lock behaviors in a valid execution. Specifically, it looks for all `Lock` actions in an execution, and makes sure that every such action satisfies the `lock_prop`, which states that after a time t that has a `Lock` action with key l, there is either not a time with an `UnLock` action in the same thread as t's thread, so that there is not any time with a `Lock` with key l, or there is a time t' with an `UnLock` action in the same thread on l; furthermore, during the time t to t', there is no time with a `Lock` with key l.

We have discussed the behaviors of mutex locks and non-atomic memory operations in this section. In the next section, we will put these together and show some theorems.

3.4 Putting it All Together

We have defined the predicates to capture the behaviors of individual memory events, Here we merge them together in a single predicate and investigate the equivalence between ATRCM and other models. Mainly, the proofs of the equivalence are based on the SRA model [22], the RC11 model [24,22] and a little bit from the IMM model [38]. To keep uniformity, we use the basic actions and other element syntax defined by Batty et al. [8].

We first connect all pieces of ATRCM in the previous sections together in Figure 8. The always-predicate `single_order` combines all of the restrictions of different atomic memory operations with different orders (stricter than `rlx`), such as `acqr`, `seqr` and `seqw`, which we define in Section 3.2. The available-after relation (`af`) is the combination of all of the always-predicates that we defined in previous sections for single-threaded behaviors in an execution. `init_state` defines the initial time point for each thread to be the one without pointed to by other points. `sat_thread` is the predicate for each thread to satisfy. In a valid execution, the order of time points in a thread needs to follow one of the paths defined in the complete set of `af` for the thread. `sat_threads` pushes the predicate to all threads. Finally, the predicate `sat` is the predicate for a valid execution to satisfy under coherence order assumption, while `sat_fifo` is the predicate for a valid execution under FIFO order assumption.

$$
\begin{aligned}
&\texttt{single\_order}(\texttt{E},\rho,\texttt{sb}) \equiv \texttt{acqr}(\texttt{E},\rho,\texttt{sb}) \cup \texttt{seqr}(\texttt{E},\rho,\texttt{sb}) \cup \texttt{seqw}(\texttt{E},\rho,\texttt{sb}) \cup \ldots\\
&\texttt{af}(\texttt{E},\rho,\texttt{sb}) \equiv \texttt{call\_dep}(\texttt{E},\rho,\texttt{sb}) \cup \texttt{call\_dep}(\texttt{E},\rho,\texttt{sb}) \cup \texttt{single\_order}(\texttt{E},\rho,\texttt{sb})\\
&\qquad\qquad\qquad \cup\texttt{lock\_dep}(\texttt{E},\rho,\texttt{sb}) \cup \texttt{unlock\_dep}(\texttt{E},\rho,\texttt{sb})\\
&\qquad\qquad\qquad \cup\texttt{dd}(\texttt{E},\rho,\texttt{sb}) \cup \texttt{vol\_dep}(\texttt{E},\rho,\texttt{sb}) \cup \texttt{add\_dd}(\texttt{E},\rho,\texttt{sb})\\
&\texttt{always\_prop}(\texttt{Tid},\texttt{E},\rho,\texttt{sbs},\texttt{rf}) \equiv \forall (s,t) \in (\texttt{rf} \cup \bigcup_{tid \in \texttt{Tid}} \texttt{af}(\texttt{E},\rho,\texttt{sbs}(tid))).s < t\\
&\texttt{sat}(\texttt{Tid},\texttt{E},\rho,\texttt{sbs},\texttt{rf}) \equiv \ \texttt{well\_formed}(\texttt{Tid},\texttt{E},\rho,\texttt{sbs},\texttt{rf})\\
&\qquad \wedge\texttt{always\_prop}(\texttt{Tid},\texttt{E},\rho,\texttt{sbs},\texttt{rf}) \wedge \texttt{co\_cw}(\texttt{E},\rho,\texttt{rf}) \neq \emptyset \wedge \texttt{locks}(\texttt{E},\rho)\\
&\texttt{sat\_fifo}(\texttt{Tid},\texttt{E},\rho,\texttt{sbs},\texttt{rf}) \equiv \ \texttt{well\_formed}(\texttt{Tid},\texttt{E},\rho,\texttt{sbs},\texttt{rf})\\
&\qquad \wedge\texttt{always\_prop}(\texttt{Tid},\texttt{E},\rho,\texttt{sbs},\texttt{rf}) \wedge \texttt{co\_cw\_fifo}(\texttt{E},\rho,\texttt{rf}) \neq \emptyset \wedge \texttt{locks}(\texttt{E},\rho)\\
&\texttt{observedMem}(\texttt{Tid},\texttt{E},\rho,\texttt{sbs},\texttt{rf},\texttt{Locs}) \equiv\\
&\quad tid \in\texttt{Tid} \wedge t \in\texttt{E} \wedge x \in\texttt{Locs} \wedge\neg\texttt{is\_mem\_op\_fence}(\rho(t))\\
&\quad \wedge\texttt{same\_thread}(tid,\rho(t)) \wedge \neg(\exists t' \in\texttt{E} .(t',t) \in \texttt{sbs}^{+}(tid))\\
&\qquad\quad \Rightarrow (tid,t,x,0) \in \texttt{observeMem}(\texttt{Tid},\texttt{E},\rho,\texttt{sbs},\texttt{rf},\texttt{Locs})\ (*\ \textsf{moBase}\ *)\\
&\ \mid\ tid \in\texttt{Tid} \wedge t \in\texttt{E} \wedge x \in\texttt{Locs} \wedge\texttt{is\_read}(\rho(t)) \wedge \texttt{same\_thread}(tid,\rho(t))\\
&\quad \wedge\texttt{same\_loc}(tid,\rho(t)) \wedge \neg(\exists t' \in\texttt{E} .(t',x,t) \in \texttt{cw}^{+})\\
&\qquad\quad \Rightarrow (tid,t,x,0) \in \texttt{observeMem}(\texttt{Tid},\texttt{E},\rho,\texttt{sbs},\texttt{rf},\texttt{Locs})\ (*\ \textsf{moReadBase}\ *)\\
&\ldots\\
&\ \mid\ tid \in\texttt{Tid} \wedge t \in\texttt{E} \wedge x \in\texttt{Locs} \wedge\texttt{is\_seq\_fence}(\rho(t)) \wedge \texttt{same\_thread}(tid,\rho(t)) \wedge (t',x,t) \in \texttt{cw}\\
&\qquad\quad \Rightarrow (tid,t,x,t') \in \texttt{observeMem}(\texttt{Tid},\texttt{E},\rho,\texttt{sbs},\texttt{rf},\texttt{Locs})\ (*\ \textsf{moSeqFence}\ *)
\end{aligned}
$$

Fig. 8: The Predicates For Connecting Different Parts of ATRCM

Now we have discussed all aspects of ATRCM. We have constructed all of these aspects of ATRCM in Isabelle/HOL, and we also construct the RC11, SRA and IMM models based on the memory actions from Batty et al.'s models in Isabelle, and prove the equivalence of our ATRCM model with previous models by proving the soundness and completeness between ATRCM and RC11/SRA with a little from IMM. The construction of the soundness proof is based on proving that ATRCM has the same properties as these models, while the completeness proof is based on constructing a transformation from the memory actions of Batty et al.'s model to those of ATRCM, and proving that what a valid execution in RC11/IMM can be observed in ATRCM. The completeness is a relative one and we put extra conditions on top of the valid executions in RC11/IMM to make the proof go through.

In Sec. 3.2, we questioned the well-formedness of the `full_rf` always-predicate, because any read in a write-read pair in the set might not read from a unique write. Here, we prove a lemma suggesting that all reads are uniquely reading a write in the `full_rf` always-predicate if the execution is valid. This lemma is proved in Isabelle and it allows us to prove later theorems in this section.

Lemma 1. For any valid execution $\mathtt{sat}(\mathtt{Tid}, \mathtt{E}, \rho, \mathtt{sbs}, \mathtt{rf})$ in ATRCM, for any write-read pair $(t1, t2)$ in $\mathtt{full\_rf}(\mathtt{E}, \rho, \mathtt{rf})$, there does not exist any $t3$, such that $t1 \neq t3$ and $(t3, t2)$ in $\mathtt{full\_rf}(\mathtt{E}, \rho, \mathtt{rf})$.

Before we show any other proofs, we first utilize the final element set ($Locs$) we defined in Sec. 2, to define an observable memory (`observedMem` in Fig. 8), whose concept is from Schrodinger's cat. An `observedMem` is a function $((TID \times T \times LOC) \rightharpoonup T)$ representing the concept of different threads seeing different memory values for a same location in the real-world setting (Sec. 2). `observedMem` is defined as a relation in Fig. 8, and one can easily prove that the `observedMem` is actually functional. Its meaning is that for a thread, at a time, and the value of a memory location is either from previous event, a result of a write at the time, or an observation from the read event at the time.

$$
\begin{array}{l}
\mathtt{mo}_x(\mathtt{E}, \rho) \equiv \{(s,t) \in \mathtt{E}^2 \mid a < b \wedge \mathtt{same\_loc}(\rho(s), \rho(t)) \wedge \mathtt{is\_write}(\rho(s)) \wedge \mathtt{is\_write}(\rho(s))\} \\
\mathtt{po}(\mathtt{Tid}, \mathtt{E}, \rho, \mathtt{sbs}) \equiv \bigcup_{tid \in \mathtt{Tid}} \mathtt{af}(\mathtt{E}, \rho, \mathtt{sbs}(tid)) \\
\mathtt{lock\_sw}(\mathtt{E}, \rho) \equiv \{(s,t) \in \mathtt{E}^2 \mid s < t \wedge \neg(\mathtt{same\_thread}(\rho(s), \rho(t))) \\
\qquad \wedge \mathtt{same\_key}(\rho(s), \rho(t)) \wedge \mathtt{is\_unlock}(\rho(s)) \wedge \mathtt{is\_lock}(\rho(t))\} \\
\mathtt{mo} \equiv \bigcup_{x \in \mathtt{Locs}} \mathtt{mo}_x \quad \mathtt{rb} \equiv \mathtt{full\_rf}^{-1}; \mathtt{mo} \quad \mathtt{eco} \equiv (\mathtt{full\_rf} \cup \mathtt{mo} \cup \mathtt{rb})^{+} \quad \mathtt{hb} \equiv (\mathtt{po} \cup \mathtt{sw})^{+} \\
\mathtt{rs} \equiv [\mathtt{W}]; \mathtt{sb}|^{?}_{\mathtt{loc}}; [\mathtt{W}^{\sqsupseteq \mathtt{rlx}}]; (\mathtt{full\_rf}; [\mathtt{RMW}])^{*} \quad \mathtt{psc} \equiv ([\mathtt{F}^{\mathtt{sc}}]; (\mathtt{hb} \cup \mathtt{hb}; \mathtt{eco}; \mathtt{hb}); [\mathtt{F}^{\mathtt{sc}}]) \cup \ldots \\
\mathtt{sw} \equiv \mathtt{lock\_sw} \cup [\mathtt{Q}^{\sqsupseteq \mathtt{rel}}]; ([\mathtt{F}]; \mathtt{po})^{?}; \mathtt{rs}; \mathtt{full\_rf}; [\mathtt{R}^{\sqsupseteq \mathtt{rlx}}]; (\mathtt{po}; [\mathtt{F}])^{?}; [\mathtt{Q}^{\sqsupseteq \mathtt{acq}}]
\end{array}
$$

Fig. 9: Parts of the Restriction Sets in Batty's Model and SRA/RC11/IMM

We also define the modification order on a specific location x as $\mathtt{mo}_x$ in Figure 9, which respects the modification order definition in RC11/SRA/IMM. Based on `observedMem` and `mo`, the first lemma we want to prove is that ATRCM satisfies the modification order printed in the C++11 memory model documentation [14]. We only show the one under coherence assumption. The FIFO ordered executions trivially satisfy this lemma. The proof of lemma 2 is a direct consequence of the requirement for coherence order in the memory machine defined by the α part of the `cross_co_cw` never-predicate, and it has been formalized and proved in Isabelle.

Lemma 2. For any valid execution $\mathtt{sat}(\mathtt{Tid}, \mathtt{E}, \rho, \mathtt{sbs}, \mathtt{rf})$ in ATRCM, and given a location set (`Locs`) collecting all location creation actions in ρ, then, for every thread ID (tid) and location (x), if time points $t1$,$t2$, $s1$ and $s2$ exist, such that $(tid, t1, x, s1) \in \mathtt{observedMem}(\mathtt{Tid}, \mathtt{E}, \rho, \mathtt{sbs}, \mathtt{rf}, \mathtt{Locs})$ and $(tid, t2, x, s2) \in$

$\texttt{observedMem}(\texttt{Tid}, \texttt{E}, \rho, \texttt{sbs}, \texttt{rf}, \texttt{Locs})$, and $t1 < t2$, then $s1$ is either equal to $s2$ or $(s1, s2) \in \texttt{mo}_x(\texttt{E}, \rho)$.

The second theorem is to build the relation between ATRCM and SRA. Essentially, the SRA model is to define the relations of `acq` and `rel` atomics. To show the soundness of ATRCM with respect to SRA, we prove that ATRCM satisfies the main property about SRA (the SRA-coherent property in [22]). Before getting into the details of the proof, we first introduce several always-predicates/relations, mostly from SRA/RC11/IMM, and we use them in our proofs in Fig. 9. Except $\texttt{mo}_x$, `po` and `lock_sw`, we do not list the arguments for each of the set for simplicity. To proper use these sets, proper arguments for each one are necessary. In RC11/SRA/IMM, the meaning of these sets are relations over memory events, since they do not have the concept of time points. Here, the sets are built over relations of time points mapped to memory events. In these relation definitions, `[X]` means to create an identity relations over the set `X`, and the ; operator is the left function composition. In an execution, `W` refers to the set of all writes, `R` refers to the set of all reads, `RMW` refers to the set of all read-modify-writes, `F` means the set of all memory fences, `Q` is the sum of all above. $\sqsupseteq O$ means that the atomics having at least the order of O, so $\sqsupseteq$ `rlx` means all atomics, $\sqsupseteq$ `rel` means the atomics whose order is at least `rel` (`rel`, `acqrel`, and `sc`).

We introduce a key item that distinguishes ATRCM from the previous axiomatic models – the definition of a program order relation (`po`). In the previous models [24,22,8,38], this was a critical matter in different theorems. However, they either did not specify what a program order [8,22] exactly was, or claimed that `po` was just the `sb` relation [6,24,38]. Since ATRCM is used as a model for proving compiler correctness, we cannot just leave the program order without an answer. The latter definition is too restrictive and rules out too many good cases. For example, the $(b4)$ event structure (Fig. 2) can never happen according to RC11's NO-THIN-AIR property because the definition uses `sb` as its key part. The essential usage of `po` is to take into account all kinds of program dependencies. In a traditional memory model setting, the memory actions are those reads and writes that directly interact with the memory, so other than assuming that program instructions relate to each other line by line (`sb`), there is no good way to enforce the program dependencies. In ATRCM, since the memory actions lifted from many program dependencies are designed to be explicit, we are able to compute the actual program dependencies by combining different kinds of dependencies together, like the `po` definition in Figure 9.

With the `po` relation in mind, we prove the SRA-coherent property [22] about coherence ordered ATRCM executions, listed in Theorems 1 and 2. In them, we use `full_rf` in ATRCM to substitute for the `rf` relations in SRA, because the `full_rf` always-predicate is precisely the `rf` relation in SRA. The first version (Theorem 1) uses the `sb` relation, as it is part of the precise SRA-coherent property listed in SRA. This version only works if there are no non-atomics nor `rlx` atomics in an execution. (d) in Fig. 2 is the diagram of $(b4)$ and is an example that does not satisfy SRA-coherence (having a cycle $\mathbf{W}_y \rightarrow \mathbf{R}_y \rightarrow \mathbf{W}_x \rightarrow \mathbf{R}_x \rightarrow$

$\mathbf{W}_y$) but C++ allows. The second version (Theorem 1) uses `po` in ATRCM to substitute for `sb`, and we do not need to rule out non-atomics and `rlx` atomics in this case, because some `sb` edges do not occur in `po` (e.g. the `sb` edges in (d)). With a precise definition of program dependencies instead of the blurry `sb` relation in SRA, the SRA-coherence can be used to describe the behaviors of ($b4$) in Fig. 2, and ($b4$) can be distinguished with ($b5$) if the two reads in ($b5$) both read value `1` (not-allowed in SRA). This is because there are edges between the reads and the writes in ($b5$) in its `po`, but there are no such edges in ($b4$). In all later proofs about ATRCM on top of properties from other models, we will use `po` in ATRCM instead of blurry program order or sequenced-before relations.

Theorem 1. For any valid execution $\mathtt{sat}(\mathtt{Tid}, \mathtt{E}, \rho, \mathtt{sbs}, \mathtt{rf})$ in ATRCM, a location set ($\mathtt{Locs}$) is given as collecting all location creation actions in ρ, and every memory operation action in ρ has at least `acq` and `rel` order (no non-atomic actions nor `rlx` atomics), then $\bigcup_{tid \in \mathtt{Tid}} \mathtt{sbs}^{+}(tid) \cup \bigcup_{x \in \mathtt{Locs}} \mathtt{mo}_x(\mathtt{E}, \rho) \cup \mathtt{full\_rf}(\mathtt{Tid}, \mathtt{E}, \rho, \mathtt{sbs})$ is acyclic.

Theorem 2. For any valid execution $\mathtt{sat}(\mathtt{Tid}, \mathtt{E}, \rho, \mathtt{sbs}, \mathtt{rf})$ in ATRCM, and a location set ($\mathtt{Locs}$) given as collecting all location creation actions in ρ, then $\mathtt{po}^{+}(\mathtt{Tid}, \mathtt{E}, \rho, \mathtt{sbs}) \cup \bigcup_{x \in \mathtt{Locs}} \mathtt{mo}_x(\mathtt{E}, \rho) \cup \mathtt{full\_rf}(\mathtt{Tid}, \mathtt{E}, \rho, \mathtt{sbs})$ is acyclic.

The next set of theorem proofs show that ATRCM satisfies the properties in RC11. There are four main properties about RC11-consistency: COHERENCE, ATOMICITY, SC and NO-THIN-AIR. In these properties, we do not need to show ATOMICITY, because the `RMW` atomics in ATRCM are itself atomic. ATRCM executions with coherence order do not satisfy the COHERENCE property in RC11/IMM, because they do not allow the behavior in (e) of Fig. 3. COHERENCE in RC11 is defined as $\mathtt{hb}; \mathtt{eco}^{?}$ (`eco` and `hb` in Fig.9). The extended-coherence order is the transitive closure of the `rf`, `mo` and reads-before (`rb` in Fig.9) restrictions sets, while the happens-before (`hb`) is the transitive closure of the `sb` and synchronized-with (`sw`) sets. In (e), even if we substitute `sb` with `po`, it still has a reflexive edge between the write to x at thread $T1$ and the read from x in thread $T3$. This shows that ATRCM executions with the coherence order is weaker than RC11/IMM. Essentially, RC11/IMM requires COHERENCE to ensure that programs with only one shared location are sequentially consistent (The SC-per-location property), so we can just prove the SC-per-location property about the coherence ordered ATRCM model. We have shown that the ATRCM executions with coherence order satisfy SC-per-location in Theorem 3, and ATRCM executions with FIFO order satisfy the RC11/IMM COHERENCE property in Theorem 4. The proofs are in Isabelle and are a directly results of the `po` and the `co_cw_fifo`/`co_cw` definitions.

Theorem 3. (ATRCM-COHERENCE). For a valid execution $\mathtt{sat}(\mathtt{Tid}, \mathtt{E}, \rho, \mathtt{sbs}, \mathtt{rf})$ in ATRCM, for every location x in the execution, $\mathtt{po}|_x \cup \mathtt{rf} \cup \mathtt{mo} \cup \mathtt{rb}$ is acyclic.

Theorem 4. For a valid execution $\texttt{sat\_fifo}(\texttt{Tid}, \texttt{E}, \rho, \texttt{sbs}, \texttt{rf})$ in ATRCM, then $\texttt{hb}; \texttt{eco}^?$ is irreflexive.

The SC property in RC11/IMM is to ensure the `sc` atomics perform properly in a valid execution. To prove the correctness, RC11/IMM defines a very complicated relation `psc`, and a part of the definition is in Fig. 9, which only restricts the `sc` fence behavior. One example is the (e) in Fig. 2 where the two reads after the `sc` fence in threads $T2$ and $T3$ must happen before each other in time, and has a happening time conflict. The last property is NO-THIN-AIR in RC11/IMM to prevent out-of-thin-air behaviors. In RC11, the property is defined by the predicate: $\texttt{sb} \cup \texttt{rf}$ is acyclic, so RC11 actually disallows the behavior in Fig. 2 (b) (the event structure $(b4)$). This program is commonly allowed in a lot of hardware or compiler implementations. IMM makes a weaker version by replacing `sb` with a relation, named `ar`, defining a relation of unions of all program restrictions. This idea is similar to `po`. However, `ar` is not complete because they lack of the definitions for locks, and their definition of control dependencies is stronger than the ATRCM one because they disallow speculative read. For example, they disallow the action 6 in $(b5)$ in Fig. 2 to execute before action 2. ATRCM's version is to use *po* to replace `sb`. We have formalized and proved the two properties about the coherence ordered ATRCM model in Isabelle, as shown in Theorem 5.

Theorem 5. Any valid execution $\texttt{sat}(\texttt{Tid}, \texttt{E}, \rho, \texttt{sbs}, \texttt{rf})$ in ATRCM satisfies that `psc` is acyclic (SC), and $\texttt{po} \cup \texttt{rf}$ is acyclic (NO-THIN-AIR).

The final theorem in this section is the relative completeness theorem of ATRCM with respect to RC11/IMM. It is relative because we need some modifications to RC11/IMM to be successful. The first modification is to use the memory events in Batty `et al.`'s model [8], because RC11 splits `RMW` actions into two different atomics. Since we do not support this feature, we use Batty `et al.`'s set of events, and we redefine `sw` by adding lock synchronization (`lock_sw` in Fig. 9). Second, a candidate execution in RC11 is a tuple as ($\texttt{Acts}_r$, $\texttt{Tid}_r$, $\texttt{Locs}_r$, $\texttt{sb}_r$, $\texttt{dd}_r$, $\texttt{rf}_r$, $\texttt{mo}_r$, $\texttt{sc}_r$). $\texttt{Tid}_r$, $\texttt{Locs}_r$, $\texttt{sb}_r$, $\texttt{dd}_r$, $\texttt{rf}_r$ and $\texttt{mo}_r$ have similar entities described in this paper. $\texttt{Acts}_r$ is a set of memory events in RC11, and $\texttt{sc}_r$ is a relation defining the *sc* atomics with respect to other events happening in an execution. Since RC11/SRA does not have the concept of time points, its candidate executions need these relations to describe the execution behaviors. The problem is that the definition of $\texttt{mo}_r$, $\texttt{sc}_r$ and $\texttt{dd}_r$ can be absolutely anything. Even though the implementation of RC11 has well-formed checks, it is not enough. For example, there is no rule to prevent $\texttt{mo}_r$ from being defined as an empty set in a candidate execution, while the execution can still be RC11-consistent. To prevent this, we require all events in $\texttt{Acts}_r$ to appear once in $\texttt{sb}_r$, all write events in $\texttt{Acts}_r$ to appear once in $\texttt{mo}_r$, all `seq` events to appear once in $\texttt{sc}_r$, and the translation of $\texttt{dd}_r$ in ATRCM is the `dd` relation generated from the translation of $\texttt{sb}_r$ through the `dd` definition in Fig. 8. We describe the above properties as well-formedness in Theorem 6. In addition, after the translation of an RC11 execution to a set of executions in ATRCM, the memory locations and lock keys will be the same type, so we need to have a predicate (`no_key_be_loc`) to indicate that locations (used in memory operations) and keys (used in lock actions) are never the same fields. The `trans` function translates an RC11 execution to a set of ATRCM ones. For any execution in RC11, `trans` generates a set of executions in ATRCM, since RC11 is descriptive and every valid execution defined in RC11 describes a group of valid executions. It is hard to make an execution in RC11 strictly unique. We show Theorem 6 below, as we have proved the theorem.

Theorem 6. For a valid and well-formed execution ($\texttt{Acts}_r$, $\texttt{Tid}_r$, $\texttt{Locs}_r$, $\texttt{sb}_r$, $\texttt{dd}_r$, $\texttt{rf}_r$, $\texttt{mo}_r$, $\texttt{sc}_r$) that is RC11-consistent, with no keys and locations in conflict, and ($\texttt{Tids}$,$\texttt{Locs}$, $\texttt{E}$, ρ, $\texttt{sb}$, $\texttt{rf}$) $\in$ $\texttt{trans}$($\texttt{Acts}_r$, $\texttt{Tid}_r$, $\texttt{Locs}_r$, $\texttt{sb}_r$, $\texttt{dd}_r$, $\texttt{rf}_r$, $\texttt{mo}_r$, $\texttt{sc}_r$), and $\texttt{trans}_{\texttt{dd}}(\texttt{dd}_r) = \texttt{dd}(\texttt{E}, \rho, \texttt{sb})$, then $\texttt{sat}(\texttt{Tid}, \texttt{E}, \rho, \texttt{sbs}, \texttt{rf})$.

4 Equivalence Relations and Usage of ATRCM

We show a usage of ATRCM by plugging it into the Morpheus framework [31]. In their work, Mansky `et al.` [30] explain the process of proving compiler optimization correctness by combining an axiomatic memory model, Morpheus framework and a underlying operational instruction level semantics for a programming language. Here we substitute ATRCM for the axiomatic model in Mansky `et al.`'s proof to prove the correctness of some simple compiler optimizations in a small programming language (MiniLLVM). The correctness here means to preserve the program meaning after applying optimizations based on the optimization quality measure, and it also means to expand the observable valid execution set from executing a program based on the program semantic perspective. For example, applying the simple redundant elimination optimization to examples in Fig. 13 is a way of expanding their valid execution sets in ATRCM, because the `ControlFences` generated for the branching operators between the `S` and `A` blocks in the examples disappear after applying the optimizations, so memory actions in block `A` are potentially free to execute before some actions in `S`; thus, the optimizations expand the set of valid executions.

4.1 Equivalence

Here we introduce the equivalence relation for a set of valid executions of ATRCM. First, we introduce a very important concept $\overline{\texttt{sb}}$ having type $AID \mapsto (TID \times A)$, which represents the second form of a sequenced-before relation in executing a program (control flow graph) in an in-order operational machine for a thread. The order on an $\overline{\texttt{sb}}$ is described by the action-IDs that are instantiated as natural numbers here. Therefore, action-IDs now can be used to determine the order of a given action in a CFG. The difference between $\overline{\texttt{sb}}$ and $\texttt{sb}$ is that the former can be directly generated from the operational semantics of a program, while the latter needs additional time point information (a set $\texttt{E}$ and a function ρ). For a multi-threaded program, we need a family of $\overline{\texttt{sbs}}$ as one $\overline{\texttt{sb}}$ per thread.

$$
\begin{aligned}
&\texttt{equal\_sb}(\texttt{E}, \rho, \texttt{sb}, \overline{\texttt{sb}}) \equiv \\
&\quad (\forall (a, b) \in \texttt{sb} \,.\texttt{same\_thread}(\rho(a), \texttt{get\_aid}(\overline{\texttt{sb}})) \wedge \texttt{same\_action}(\rho(a), \overline{\texttt{sb}}(\texttt{get\_aid}(a))) \\
&\quad \wedge \texttt{same\_thread}(\rho(b), \texttt{get\_aid}(\overline{\texttt{sb}})) \wedge \texttt{same\_action}(\rho(b), \overline{\texttt{sb}}(\texttt{get\_aid}(b)))) \\
&\quad \wedge (\forall (a, b) \in \texttt{dom}^2(\overline{\texttt{sb}}).a \neq b \Rightarrow (a, b) \in \texttt{sb}^{+} \\
&\qquad \wedge \texttt{same\_thread}(\rho(a), \texttt{get\_aid}(\overline{\texttt{sb}})) \wedge \texttt{same\_action}(\rho(a), \overline{\texttt{sb}}(\texttt{get\_aid}(a))) \\
&\qquad \wedge \texttt{same\_thread}(\rho(b), \texttt{get\_aid}(\overline{\texttt{sb}})) \wedge \texttt{same\_action}(\rho(b), \overline{\texttt{sb}}(\texttt{get\_aid}(b)))) \\
&\texttt{gen}(\overline{\texttt{sbs}}) \equiv \{(\texttt{Tid}, \texttt{E}, \rho, \texttt{sbs}, \texttt{rf}, \texttt{Locs}) | \texttt{Tid} = \texttt{collect\_tids}(\overline{\texttt{sbs}}) \wedge \texttt{Locs} = \texttt{collect\_locs}(\overline{\texttt{sbs}}) \\
&\qquad\qquad \wedge (\forall tid \in \texttt{Tid}.\texttt{equal\_sb}(\texttt{E}, \rho, \texttt{sbs}(tid), \overline{\texttt{sb}}(tid))) \wedge \texttt{sat}(\texttt{Tid}, \texttt{E}, \rho, \texttt{sbs}, \texttt{rf}))\} \\
&Label = \texttt{read}\ LOC\ \ V \mid \texttt{write}\ LOC\ \ V \mid \tau
\end{aligned}
$$

Fig. 10: Equivalence Relation Predicates

The $\overline{\texttt{sbs}}$ relation is a graph representation of a multi-threaded program. We can compute the set of all possible valid executions from the $\overline{\texttt{sbs}}$ relation by the `gen` function whose property is defined in Fig. 10. If two programs have the same $\overline{\texttt{sbs}}$, their sets of possible valid executions should be the same. This is because all executions in ATRCM are assumed to have an underlying multi-threaded program, and $\overline{\texttt{sbs}}$ is its representation. The first form of the equivalence relation between two sets of valid executions based on two different programs is as follows:

Definition 1. (Strong Equivalence) Two sets of valid executions ($\texttt{gen}(\overline{\texttt{sbs}})$ and $\texttt{gen}(\overline{\texttt{sbs'}})$) are the same if $\overline{\texttt{sbs}} = \overline{\texttt{sbs'}}$.

This form of equivalence is too strong to be useful in proving the correctness of compiler transformations. To facilitate the proofs, we introduce a form of bisimulation on top of time points. Given an execution (Tid,E,ρ,sbs, rf, Locs), we construct a labeled transitions system on $(\texttt{E}, \rho)$ as $(\texttt{E}, \rho, \Lambda, \rightarrow_{\bigcirc(\texttt{E})})$, where Λ gives a label (syntax in Fig. 10) for each transition in $\rightarrow_{\bigcirc(\texttt{E})}$ based on ρ. In $t \rightarrow_{\bigcirc(\texttt{E})} t'$. If $\rho(t)$ is a memory operation event, then the transition has a label `read` or `write` with the location and value as the memory operation. An `RMW` event is divided into two consecutive transitions, and a non-atomic operation has a `read` or `write` label only if the i and n values are the same. If $\rho(t)$ is not a memory operation event, it has the label τ. In the transition $t \rightarrow_{\bigcirc(\texttt{E})} t'$, t' is the next time point following the time point t. The purpose $t \xrightarrow{\tau}{}^{*}_{\bigcirc(\texttt{E})} \xrightarrow{l}_{\bigcirc(\texttt{E})} t'$ in the following definition is to find the next time point t' whose label l is not τ.

Definition 2. (Time Point Bisimulation) In two tradition systems such as $(\texttt{E}, \rho, \Lambda, \rightarrow_{\bigcirc(\texttt{E})})$ and $(\texttt{E'}, \rho', \Lambda', \rightarrow_{\bigcirc(\texttt{E'})})$, a relation $R \subseteq \texttt{E} \times \texttt{E'}$ is bisimilar if and only if:

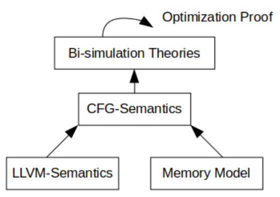
- $\forall s\ t.(s, t) \in R \wedge s \rightarrow_{\bigcirc(\texttt{E})} s' \Rightarrow (\exists t'.t \xrightarrow{\tau}{}^{*}_{\bigcirc(\texttt{E'})} \xrightarrow{l}_{\bigcirc(\texttt{E'})} t' \wedge (s', t') \in R)$
- $\forall s\ t.(s, t) \in R^{-1} \wedge s \rightarrow_{\bigcirc(\texttt{E'})} s' \Rightarrow (\exists t'.t \xrightarrow{\tau}{}^{*}_{\bigcirc(\texttt{E})} \xrightarrow{l}_{\bigcirc(\texttt{E})} t' \wedge (s', t') \in R^{-1})$

In a compiler transformation from program p to p', we care about if p' preserves the meaning of p. This means that all behaviors that can be observed by executing p' are allowed in executing p. Based on the fact that an $\overline{\texttt{sbs}}$ relation is the representation of a program, and given the bisimulation definition above, we will formulate the second form of an equivalence relation below to suggest that a program p' is represented by another program p.

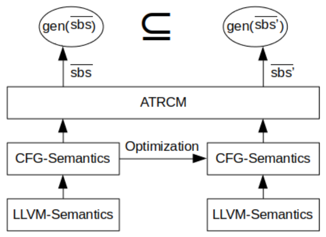
Definition 3. (Program Representation) An $\overline{\texttt{sbs'}}$ is represented by an $\overline{\texttt{sbs}}$, if and only if they can both generate a set of candidate executions: $\texttt{gen}(\overline{\texttt{sbs'}})$ and $\texttt{gen}(\overline{\texttt{sbs}})$, and there exists at least one valid execution in $\texttt{gen}(\overline{\texttt{sbs'}})$; and for every $(\texttt{Tid'}, \texttt{E'}, \rho', \texttt{sbs'}, \texttt{rf'}, \texttt{Locs'})$ in $\texttt{gen}(\overline{\texttt{sbs'}})$ there is an execution $(\texttt{Tid}, \texttt{E}, \rho, \texttt{sbs}, \texttt{rf}, \texttt{Locs})$ in $\texttt{gen}(\overline{\texttt{sbs}})$ such that they are bi-similar to each other.

In the following section, we will show an example of using this form of equivalence relation to prove a compiler optimization preserving program meaning.

4.2 Program Behavioral Expansion and Meaning Preservation



(a)



(b)

Fig. 11: Optimization Proof With Sequential Memory Model vs. ATRCM

In previous papers about Morpheus [30,31], it was shown how to combine a sequential memory model, the Morpheus framework and an underlying instruction semantics for a programming language to prove the correctness of a traditional compiler optimization (PRE). Here, we sketch how they achieved the proof in Fig. 11a. What they did was to define the instruction level semantics for a target language (a subset of LLVM), and pack the semantics with Morpheus' control flow graph semantics and a set of sequential memory model axioms. Then, based on their bisimulation theories on top of the Morpheus, they proved the optimization correctness.

We prove optimization correctness (program behavioral expansion and meaning preservation) based on ATRCM (Fig. 11b) a little differently. We keep the packing of the instruction semantics and the control flow graph semantics, because it gives us the semantics for the underlying in-order operational memory for the target programming language. For an optimization, we define the transformation in Morpheus from a CFG to the transformed CFG (indicated by the arrow between the two CFG semantics in Fig. 11b). Based on their semantics, we can easily lift them and represent them by $\overline{\texttt{sbs}}$ relations over memory events in

ATRCM ($\overline{\texttt{sbs}}$) and $\overline{\texttt{sbs'}}$)), and then generate two valid execution sets for them. The $\subseteq$ operation in the figure represents the proof of optimization correctness by using Definition 3.

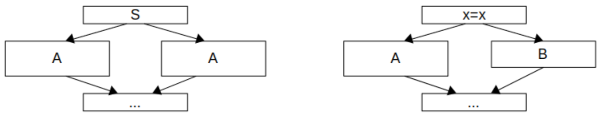


Fig. 12: Simple Redundant Elimination Optimization Examples

In Figure 12, we show two simple redundant elimination optimization examples. The left one is to change the conditional branching operation to a non-conditional one because the two target basic blocks of the conditional branching operation are the same. The right one is also to change the conditional branching operation to a non-conditional one by keeping only the `true`'s branch in the non-conditional one, because the condition in the conditional branching operation is always `true`. To express this set of optimizations here, we use the Morpheus tool [31] to help us. Morpheus is used to describe program transformations as rewrites on control flow graphs with temporal logic side conditions. The greatest advantage of using the Morpheus tool on ATRCM is that its temporal logic side conditions have semantics in terms of first order logic formulas, and the transformation can also easily be turned into a one in the program order of a given CFG; thus, the side conditions can be merged with the ATRCM.

$$\begin{aligned}
&\texttt{sameOutEdge}(a,b) \equiv \texttt{stmt}(a) = \texttt{stmt}\ (b) \wedge \texttt{sameEdges}(a,b)\\
&\qquad\qquad \vee \texttt{stmt}(a) = \texttt{stmt}\ (b) \wedge \neg\texttt{sameEdges}(a,b) \wedge \texttt{sameOutEdge}(\texttt{next}(a), \texttt{next}(b))\\
&\texttt{leftOpt}(n) \equiv \exists x\ m_1\ m_2.\texttt{SATISFIED\_AT}n\texttt{sameOutEdge}(\texttt{next}(\texttt{true}, n), \texttt{next}(\texttt{false}, n));\\
&\qquad\qquad \texttt{relabel\_node}(n, \texttt{skip}); \texttt{move\_edge}((n, \texttt{false}, m_2), m_1)\\
&\texttt{rightOpt}(n) \equiv \exists x\ m_1\ m_2.\texttt{SATISFIED\_AT}n\texttt{stmt}(x = x)\\
&\qquad\qquad ; \texttt{relabel\_node}(n, \texttt{skip}); \texttt{move\_edge}((n, \texttt{false}, m_2), m_1)
\end{aligned}$$

Fig. 13: Transformations Through Morpheus and Effects on Program Orders

In Figure 13, the Morpheus formulas `leftOpt` and `rightOpt` define the left and right compiler optimizations in Figure 12. The `sameOutEdge` formula defines the predicate for checking if two statements are the same and their children have the same outgoing edges or statements. The `leftOpt` and `rightOpt` formulas merge the two outgoing edges and put the new edge in the `true` branch edge of the branching statement at a given CFG statement labeled n. Certainly, we need other sample compiler optimizations, such as `skip` elimination and dead-code elimination, to tell the whole story of the left and right compiler optimizations in Figure 12. Interested readers can learn about them in the Morpheus framework paper [31]. We have proved the following theorems about the two optimizations in Fig. 13 in Morpheus plus ATRCM.

Theorem 7. For any program p in Morpheus (with a target language), for any n, let $p' = \texttt{leftOpt}(n)(p)$ (or $p' = \texttt{rightOpt}(n)(p)$), if $\overline{\texttt{sbs}}$ is the lifted sequenced-before relation for p and $\overline{\texttt{sbs'}}$ is the one for p', then $\overline{\texttt{sbs'}}$ is represented by an $\overline{\texttt{sbs}}$.

In this subsection, we have briefly described an equivalence relation defined on two set of valid executions from two programs, how we can define a compiler optimization in Morpheus, linked it to ATRCM, and used the program representation definition to prove the optimization correctness.

5 Related Work

Lamport probably was the first to define a memory model weaker than sequential consistency for multi-threaded programs [25]. Adve and Hill [3] started defining weak memory orders for memory operations. Collier [13] and Neiger [17], as well as Adve and Gharachorloo [2], provided general overviews of weak memory, but in a less formal style than one might prefer. During the 1990s and early 2000s, a lot of weak memory models for different languages were written informally [44,26,42,19,32] were used for improving system efficiency [28]. Focusing just on hardware models: Ahamad et al. [4] axiomatized causal memory and proved some important theorems. Higham et al. [18] formalized SPARC and a number of simpler memory models in both axiomatic and operational styles. Sevcik et al. created a formal verification framework for a small C-like language [48]. The same group [49] later developed the CompCertTSO to verify a compiler from CLight to X86 based on a relaxed memory model.

In the SPARC documentation [43], an axiomatic style similar to the candidate execution model was used. Alglave et al. [6] specified in great detail how to use a candidate execution model to define relaxed memory models and provided several verification tools. The C11 memory model was designed by the C++ standards committee based on a paper by Boehm and Adve [9]. Batty et al. formalized the C11 model with some improvements and proved the soundness of its compilation to X86-TSO [8]. A number of papers [15,46,35,10] found that Batty et al.'s model enabled thin-air behaviors. Vafeiadis et al. [45] found many other problems in Batty et al.'s model and proposed fixes. In 2016, Batty et al. proposed a more concise model for `sc` atomics [7], but the model is stronger than C11; and the `sc` fences there are too weak. Much previous work [47,33,23,22] focused on a fragment of C++ concurrency. From this corpus, we select Lahav et al.'s SRA model [22] to show the soundness of our `acq`/`rel` atomics. In 2017, Lahav et al. [24] defined a comprehensive C++ model (RC11) based on all previous models, with extra fixes on Batty et al.'s model. In Sec. 1, 2, and 3.4, we discussed this model multiple times. The main problems with it are clarity (Sec. 1), as well as its OUT-OF-THIN-AIR condition is too strong and rules out too many good executions. Many previous papers [37,20,21] also proposed solutions for out-of-thin-air problems. These models were not in the axiomatic candidate execution fashion, and one of them (the promising memory model [21]) has been proved to be represented by the IMM model [38]. Chakraborty and Vafeiadis [11] provided a concurrent abstracted memory model for LLVM IR. Not of the axiomatic candidate-execution genre, it provided the semantics for a fragment of LLVM IR memory operations while keeping the model stronger than Lahav et al.'s. Finally, the IMM model by Podkopaev et al. [38], based on RC11 and the promising memory model, fixed its OUT-OF-THIN-AIR condition with the requirement of giving many different relations (ar). We merge these relations with the given action set as we showed in Sec. 2 and 3.4. The coherence ordered ATRCM model is weaker then IMM, while the FIFO ordered ATRCM model is stronger. Our definition of the OUT-OF-THIN-AIR condition can allow speculative reads and we have a more precise definition of po than IMM. The essential difference is that IMM is designed to provide a spiritual sample for people to understand how to compile C++ to hardware code, while ATRCM is designed to be used by a compiler proof framework to prove properties about a compiler.

There are also some recent developments about hardware weak memory models [1,12,41,40,5,16], of which the most important two are the one defining the POWER memory model [29] and the new development of the ARMv8 relaxed memory model [39]. These new developments act as the target compiler source languages and their new features on weak fences are the motivations for ATRCM to have a weak coherence order that produces a memory model weaker than RC11/IMM.

6 Conclusion and Future Work

We have achieved a relaxed memory model in an axiomatic candidate execution style with the properties of clarity, operability, adaptability, suitability, and comprehensibility. In ATRCM, we provide two scheduling orders: FIFO and coherence. We show that they are sound for SRA/RC11/IMM [22,24,38] with some scaling, and the coherence ordered ATRCM model is complete with respect to the RC11 model. We also define the time point bisimulation and program representation relations for answering if an optimized program preserves the meaning of its original program. For future work, we plan to prove that the **K-LLVM** semantics [27] is sound for the ATRCM model and create a new framework that not only assures the meaning preservation of a compiler but also that the compiler does not generate harmful behaviors.

References

1. Adir, A., Attiya, H., Shurek, G.: Information-Flow Models for Shared Memory with an Application to The PowerPC Architecture. IEEE Transactions on Parallel and Distributed Systems **14**(5), 502–515 (May 2003)

2. Adve, S.V., Gharachorloo, K.: Shared Memory Consistency Models: A Tutorial. Computer **29**(12), 66–76 (Dec 1996)
3. Adve, S.V., Hill, M.D.: Weak Ordering – A New Definition. SIGARCH Comput. Archit. News **18**(2SI), 2–14 (May 1990)
4. Ahamad, M., Neiger, G., Burns, J.E., Kohli, P., Hutto, P.W.: Causal Memory: Definitions, Implementation, and Programming. Distributed Computing **9**(1), 37–49 (Mar 1995)
5. Alglave, J., Fox, A., Ishtiaq, S., Myreen, M.O., Sarkar, S., Sewell, P., Nardelli, F.Z.: The Semantics of Power and ARM Multiprocessor Machine Code. In: Proceedings of the 4th Workshop on Declarative Aspects of Multicore Programming. pp. 13–24. DAMP '09, ACM, New York, NY, USA (2008)
6. Alglave, J., Maranget, L., Tautschnig, M.: Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory. ACM Trans. Program. Lang. Syst. **36**(2), 7:1–7:74 (Jul 2014). https://doi.org/10.1145/2627752, http://doi.acm.org/10.1145/2627752
7. Batty, M., Donaldson, A.F., Wickerson, J.: Overhauling SC Atomics in C11 and OpenCL. SIGPLAN Not. **51**(1), 634–648 (Jan 2016)
8. Batty, M., Owens, S., Sarkar, S., Sewell, P., Weber, T.: Mathematizing c++ concurrency. SIGPLAN Not. **46**(1), 55–66 (Jan 2011). https://doi.org/10.1145/1925844.1926394, http://doi.acm.org/10.1145/1925844.1926394
9. Boehm, H.J., Adve, S.V.: Foundations of the C++ Concurrency Memory Model. SIGPLAN Not. **43**(6), 68–78 (Jun 2008)
10. Boehm, H.J., Demsky, B.: Outlawing Ghosts: Avoiding Out-of-thin-air Results. In: Proceedings of the Workshop on Memory Systems Performance and Correctness. pp. 7:1–7:6. MSPC '14, ACM, New York, NY, USA (2014)
11. Chakraborty, S., Vafeiadis, V.: Formalizing the Concurrency Semantics of an LLVM Fragment. In: Proceedings of the 2017 International Symposium on Code Generation and Optimization. pp. 100–110. CGO '17, IEEE Press, Piscataway, NJ, USA (2017), http://dl.acm.org/citation.cfm?id=3049832.3049844
12. Chong, N., Ishtiaq, S.: Reasoning About the ARM Weakly Consistent Memory Model. In: Proceedings of the 2008 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness: Held in Conjunction with the Thirteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '08). pp. 16–19. MSPC '08, ACM, New York, NY, USA (2008)
13. Collier, W.W.: Reasoning About Parallel Architectures. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1992)
14. cppreference.com: The C++11 Standard (2018), https://www.cppreference.com
15. Dodds, M., Batty, M., Gotsman, A.: C/C++ Causal Cycles Confound Compositionality. TinyToCS **2** (2013), http://tinytocs.org/vol2/papers/tinytocs2-dodds.pdf
16. Flur, S., Gray, K.E., Pulte, C., Sarkar, S., Sezgin, A., Maranget, L., Deacon, W., Sewell, P.: Modelling the ARMv8 Architecture, Operationally: Concurrency and ISA. SIGPLAN Not. **51**(1), 608–621 (Jan 2016)
17. Gharachorloo, K., Lenoski, D., Laudon, J., Gibbons, P., Gupta, A., Hennessy, J.: Memory Consistency and Event Ordering in Scalable Shared-memory Multiprocessors. SIGARCH Comput. Archit. News **18**(2SI), 15–26 (May 1990)
18. Higham, L., Kawash, J., Verwaal, N.: Weak Memory Consistency Models. Part I: Definitions and Comparisons. Tech. rep., Department of Computer Science, The University of Calgary (1998)

19. Intel: Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 3B. Intel Corporation (August 2007)
20. Jeffrey, A., Riely, J.: On Thin Air Reads Towards an Event Structures Model of Relaxed Memory. In: Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science. pp. 759–767. LICS '16, ACM, New York, NY, USA (2016). https://doi.org/10.1145/2933575.2934536, `http://doi.acm.org/10.1145/2933575.2934536`
21. Kang, J., Hur, C.K., Lahav, O., Vafeiadis, V., Dreyer, D.: A Promising Semantics for Relaxed-memory Concurrency. SIGPLAN Not. **52**(1), 175–189 (Jan 2017). https://doi.org/10.1145/3093333.3009850, `http://doi.acm.org/10.1145/3093333.3009850`
22. Lahav, O., Giannarakis, N., Vafeiadis, V.: Taming release-acquire consistency. SIGPLAN Not. **51**(1), 649–662 (Jan 2016). https://doi.org/10.1145/2914770.2837643, `http://doi.acm.org/10.1145/2914770.2837643`
23. Lahav, O., Vafeiadis, V.: Owicki-gries reasoning for weak memory models. In: Proceedings, Part II, of the 42Nd International Colloquium on Automata, Languages, and Programming - Volume 9135. pp. 311–323. ICALP 2015, Springer-Verlag, Berlin, Heidelberg (2015)
24. Lahav, O., Vafeiadis, V., Kang, J., Hur, C.K., Dreyer, D.: Repairing sequential consistency in c/c++11. SIGPLAN Not. **52**(6), 618–632 (Jun 2017). https://doi.org/10.1145/3140587.3062352, `http://doi.acm.org/10.1145/3140587.3062352`
25. Lamport, L.: How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. IEEE Trans. Comput. **28**(9), 690–691 (Sep 1979)
26. Lea, D.: Concurrent Programming in Java. Second Edition: Design Principles and Patterns. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edn. (1999)
27. Li, L., Gunter, E.: Llvm semantics (2016), `https://github.com/kframework/llvm-semantics`
28. mailing list, L.K.: thread "spin unlockoptimization(i386)" (1999), `http://www.gossamer-threads.com/lists/engine?post=105365;list=linux.Accessed2009/11/18`
29. Mador-Haim, S., Maranget, L., Sarkar, S., Memarian, K., Alglave, J., Owens, S., Alur, R., Martin, M.M.K., Sewell, P., Williams, D.: An Axiomatic Memory Model for POWER Multiprocessors. In: Madhusudan, P., Seshia, S.A. (eds.) Computer Aided Verification. pp. 495–512. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)
30. Mansky, W., Garbuzov, D., Zdancewic, S.: An Axiomatic Specification for Sequential Memory Models. In: Kroening, D., Păsăreanu, C.S. (eds.) Computer Aided Verification. pp. 413–428. Springer International Publishing, Cham (2015)
31. Mansky, W., Gunter, E.L., Griffith, D., Adams, M.D.: Specifying and executing optimizations for generalized control flow graphs. Science of Computer Programming **130**, 2–23 (Nov 2016). https://doi.org/10.1016/j.scico.2016.06.003
32. May, C., Silha, E., Simpson, R., Warren, H., International Business Machines, Inc., C. (eds.): The PowerPC Architecture: A Specification for a New Family of RISC Processors. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1994)
33. Meshman, Y., Rinetzky, N., Yahav, E.: Pattern-based Synthesis of Synchronization for the C++ Memory Model. In: Proceedings of the 15th Conference on Formal Methods in Computer-Aided Design. pp. 120–127. FMCAD '15, FMCAD Inc, Austin, TX (2015)

34. Nipkow, T., Wenzel, M., Paulson, L.C.: Isabelle/HOL: A Proof Assistant for Higher-order Logic. Springer-Verlag, Berlin, Heidelberg (2002)
35. Norris, B., Demsky, B.: Cdschecker: Checking concurrent data structures written with c/c++ atomics. SIGPLAN Not. **48**(10), 131–150 (Oct 2013)
36. Paulson, L.C.: Isabelle: The Next 700 Theorem Provers. In: Odifreddi, P. (ed.) Logic and Computer Science, pp. 361–386. Academic Press (1990)
37. Pichon-Pharabod, J., Sewell, P.: A Concurrency Semantics for Relaxed Atomics That Permits Optimisation and Avoids Thin-air Executions. SIGPLAN Not. **51**(1), 622–633 (Jan 2016)
38. Podkopaev, A., Lahav, O., Vafeiadis, V.: Bridging the Gap Between Programming Languages and Hardware Weak Memory Models. Proc. ACM Program. Lang. **3**(POPL), 69:1–69:31 (Jan 2019). https://doi.org/10.1145/3290382, `http://doi.acm.org/10.1145/3290382`
39. Pulte, C., Flur, S., Deacon, W., French, J., Sarkar, S., Sewell, P.: Simplifying ARM Concurrency: Multicopy-atomic Axiomatic and Operational Models for ARMv8. Proc. ACM Program. Lang. **2**(POPL), 19:1–19:29 (Dec 2017)
40. Sarkar, S., Sewell, P., Alglave, J., Maranget, L., Williams, D.: Understanding POWER Multiprocessors. SIGPLAN Not. **46**(6), 175–186 (Jun 2011)
41. Sarkar, S., Sewell, P., Nardelli, F.Z., Owens, S., Ridge, T., Braibant, T., Myreen, M.O., Alglave, J.: The Semantics of x86-CC Multiprocessor Machine Code. SIGPLAN Not. **44**(1), 379–391 (Jan 2009)
42. Sites, R.L. (ed.): Alpha Architecture Reference Manual. Digital Press, Newton, MA, USA (1992)
43. SPARC International, Inc., C.: The SPARC Architecture Manual: Version 8. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1992)
44. SPARC International, Inc., C.: The SPARC Architecture Manual (Version 9). Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1994)
45. Vafeiadis, V., Balabonski, T., Chakraborty, S., Morisset, R., Zappa Nardelli, F.: Common Compiler Optimisations Are Invalid in the C11 Memory Model and What We Can Do About It. SIGPLAN Not. **50**(1), 209–220 (Jan 2015)
46. Vafeiadis, V., Narayan, C.: Relaxed Separation Logic: A Program Logic for C11 Concurrency. SIGPLAN Not. **48**(10), 867–884 (Oct 2013)
47. Vafeiadis, V., Narayan, C.: Relaxed Separation Logic: A Program Logic for C11 Concurrency. SIGPLAN Not. **48**(10), 867–884 (Oct 2013). https://doi.org/10.1145/2544173.2509532, `http://doi.acm.org/10.1145/2544173.2509532`
48. Ševčík, J., Vafeiadis, V., Zappa Nardelli, F., Jagannathan, S., Sewell, P.: Relaxed-memory Concurrency and Verified Compilation. SIGPLAN Not. **46**(1), 43–54 (Jan 2011)
49. Ševčík, J., Vafeiadis, V., Zappa Nardelli, F., Jagannathan, S., Sewell, P.: CompCertTSO: A Verified Compiler for Relaxed-Memory Concurrency. J. ACM **60**(3), 22:1–22:50 (Jun 2013). https://doi.org/10.1145/2487241.2487248, `http://doi.acm.org/10.1145/2487241.2487248`

7 Appendix

7.1 Candatate Execution Assumptions

To be a valid candidate execution, there are some assumptions about the six components of it that needs to fulfill. The unique `aid` assumption in a ρ function is one such assumption. Other assumption includes that all time points, thread IDs and memory locations appeared in `sbs`, `rf` and ρ are in the sets `E`, `Tids` and `Locs`, respectively. Also, all `sb` relations described in this paper does not account for transitive closure, so if $(a, b) \in$`sb`, it means that the execution of b follows the execution of a according to the in-order execution machine, and there is no other instruction in-between. We will use $\texttt{sb}^+$ when we need to mention the transitive closure of a `sb` relation. We also have several important assumptions about a `rf` relation, some of which have been stated in previous papers. One obvious assumption about our `rf` relations is that they cannot contain a write-read pair whose write and read operations are from the same thread, because those pairs can be generated by `sb` relations. Finally, there are assumptions about the `NRead` and `NWrite` operations. Remember that there are two *nat* (i and n) fields in the syntax of the operations. The second *nat* n represents the total number of non-atomic actions for the same `aid` operation, while the first one i represents that the non-atomic action is the i-th piece of the non-atomic operation. For simplicity, we assume that the all non-atomic actions have i range from 1 to n. The i-th non-atomic action for an `aid` happens at the i-th rank with that `aid` in an candidate execution (defined by the `non_atomic_well_order` predicate in Fig. 14). This assumption is valid by only talking about `sb` relations because we also have assumption that non-atomic actions with the same `aid` only happen in a thread, as an observation on the real world situation that there is no use of two different threads dealing with a single memory operation. The assumption eases our proofs and allows to abstract away the mechanism of distributing correct memory actions into correct memory locations. For now, we have the lemma about non-atomic actions to suggest that if an action happens at a time and its i and n values are the same, then the non-atomic memory operation is finalized at that time; otherwise, at a time, the non-atomic memory operation is still waiting for more actions to finish. The assumption is also not restrictive because all the users of our axiomatic model need to do is just to come up with a method to associate pieces of a non-atomic memory operation with the order of time points.

$$\begin{aligned}\texttt{non\_atomic\_well\_order}\ (\texttt{Tids}, \texttt{sbs}, \rho) \equiv\ & \forall t \in \texttt{Tids}\ .\forall a\ b \in (\texttt{sbs}\ (t))^+ .\\ & \texttt{is\_non\_atomic}\ (\rho(a))\\ & \wedge\ \texttt{is\_non\_atomic}\ (\rho(b))\\ & \wedge\ \texttt{same\_aid}\ (\rho(a), \rho(b))\\ & \Rightarrow \texttt{get\_i}\ (\rho(a)) < \texttt{get\_i}\ (\rho(b))\end{aligned}$$

Fig. 14: Some Basic Elements in the Axiomatic Model