Portability Analysis for Axiomatic Memory Models

PORTHOS: One Tool for all Models

Hernán Ponce-de-León*
fortiss GmbH, Germany
ponce@fortiss.org

Florian Furbach
TU Kaiserslautern, Germany
furbach@cs.uni-kl.de

Keijo Heljanko
Aalto University and HIIT, Finland
keijo.heljanko@aalto.fi

Roland Meyer†
TU Braunschweig, Germany and Aalto University, Finland
roland.meyer@tu-braunschweig.de

Abstract

We present PORTHOS, the first tool that discovers porting bugs in performance-critical code. PORTHOS takes as input a program, the memory model of the source architecture for which the program has been developed, and the memory model of the targeted architecture. If the code is not portable, PORTHOS finds a porting bug in the form of an unexpected execution — an execution that is consistent with the target but inconsistent with the source memory model.

Technically, PORTHOS implements a bounded model checking method that reduces portability analysis to the satisfiability modulo theories (SMT) problem with integer difference logic. There are two problems in the reduction that are unique to portability and that we present novel and efficient solutions for. First, the formulation of portability contains a quantifier alternation (consistent + inconsistent). We encode inconsistency as an existential query. Second, the memory models may contain recursive definitions. We compute the corresponding least fixed points efficiently in SMT.

Interestingly, we are able to prove that our execution-based notion of portability is the most liberal one that admits an efficient algorithmic analysis: for state-based portability, a polynomial SAT encoding cannot exist.

Experimentally, we applied PORTHOS to a number of case studies. It is able to check portability of non-trivial programs between interesting architectures. Notably, we present the first algorithmic analysis of portability from TSO to Power.

1. Introduction

Porting code from one architecture to another is a routine task in system development. Given that no functionality has to be added, porting is rarely considered interesting from a programming point of view. At the same time, porting is non-trivial as the hardware can influence the semantics of the code in subtle ways. The unfortunate combination of being routine and yet subtle makes porting prone to mistakes. This is particularly true for performance-critical code that interacts closely with the execution environment. Such code often has data races and thus exposes the programmer to the details of the underlying hardware. Concurrency libraries, for example, contain data races to avoid expensive synchronization constructs (barriers, fences, syncs, or cas). When the architecture is changed, the code may have to be adapted to the primitives of the target hardware.

We tackle the problem of porting performance-critical code among hardware architectures. Our contribution is the new (and to the best of our knowledge first) tool PORTHOS to fight porting bugs. It takes as input a piece of code, a model of the source architecture for which the code has been developed, and a model of the target architecture to which the code is to be ported. PORTHOS automatically checks whether every behavior of the code on the target architecture is also allowed on the source platform. This guarantees that correctness of the program in terms of safety properties (in particular synchronization properties like mutual exclusion) carry over to the targeted hardware, and thus the program remains correct after porting.

Semantics and verification under weak memory models have been the subject of study at least since 2007. Initially, the behavior of x86 and TSO has been clarified [15, 44], then the Power architecture has been addressed [40, 45], now
ARM is being tackled [30], and the study also looks beyond hardware, in particular C++11 received considerable attention [11, 12]. Research in semantics goes hand in hand with the development of verification methods. They come in two flavors: program logics [49, 50] and algorithmic approaches [1, 2, 6, 8, 10, 13, 14, 23, 24]. Notably, each of these methods and tools is designed for a specific memory model and hence is not directly able to handle porting tasks.

Portability requires an analysis method that is hardware-architecture-aware in the sense that a description of the memory models of source and target platforms has to be part of the input. A language for memory models, called CAT, has been developed only recently. In CAT, memory models are defined in terms of relations between memory operations of a program. There are three base relations (program order, marks and compare the results against state portability. We have used it to find porting bugs of multiple platforms. We have analyzed [17, 28, 29]. A prominent approach is testing where an execution is (partially) given and consistency is tested for a specified model [31, 33]. In this line and concerning (a), we show that state portability (formulated as a bounded analysis for cyclic programs) is \(\Pi_2^P\)-complete. This means there is no hope for a polynomial encoding into SAT (unless the polynomial hierarchy collapses). In contrast, our execution-based notion of portability is co-NP-complete (we look for a violation to portability), which in particular means that our portability analysis is optimal in the complexity sense.

Concerning (b), we evaluate our tool on a set of benchmarks and compare the results against state portability. We run HERD7 twice (once for each model) and compare the number of computable states to test state portability. This however will be infeasible for programs with a very large number of computable states. Our experiments show that in most of the cases (especially when testing portability to Power) both notions of portability coincide.

PORTHOS allows us to analyze not only litmus tests, but full programs. We have used it to find portability bugs of mutual exclusion algorithms and the first results evidence the applicability of our method. Interestingly, even if PORTHOS is not complete (it is a bounded model checking approach), bugs are found by unrolling each cycle only once. Our proto-

Concerning the first problem, we implement in SAT/SMT the operations that CAT defines on relations. Notably, we propose an encoding for derived relations that are defined as least fixed points. Such least fixed points are prominently used in the Power memory model [8]. A naive encoding would implement the Kleene iteration in SAT/SMT by introducing copies of the variables for each iteration step, resulting in a very large encoding. We show how to employ SAT + integer difference logic (IDL) [21] to compactly encode the Kleene iteration process. The idea is that every element in the domain is given an integer variable indicating when it enters the least fixed point solution. It is worth noting that the satisfiability problem for SAT + IDL stays within NP [21].

The second problem is to encode the quantifier alternation underlying the definition of portability. A porting bug is an execution that is consistent with the target but inconsistent with the source memory model. We show how to capture this alternation with a single existential query. Consistency is specified in terms of acyclicity (and irreflexivity) of relations. Hence, an execution is inconsistent if a derived relation of the (source) memory model contains a cycle (or is not irreflexive). The naive idea would be to model cyclicity by unsatisfiability. Instead, we reduce cyclicity to satisfiability by introducing auxiliary variables that guess the cycle.

The reader may criticize our definition of portability: one could claim that all that matters is whether safety is preserved, even if the executions differ. To be precise, a state-based notion of portability requires that every state computable under the target architecture is already computable on the source platform. We study state portability and come up with two results.

(a) Algorithmically, state portability is beyond SAT.
(b) Empirically, there is little difference between state portability and our notion.
type cannot yet compete in terms of performance with tools that have been under development for years. Yet, our results open the door to integrate portability analysis into tools such as CBMC which have already been used to verify programs under axiomatic memory models [6].

A problem less general than portability is solved in [13] where non-portable traces from SC to TSO are characterized. The problem is reduced to state reachability under the SC semantics in an instrumented program and a minimal synthesized. The problem is reduced to state reachability under the axiomatic memory models [6].

The remainder of the paper is organized as follows. In Section 2, we present the portability problem and illustrate the reduction to SAT/SMT on an example. Section 3 defines the programming model and introduces the language CAT for memory models. Portability and the reduction to SAT/SMT is formally treated in Section 4. Section 5 studies the state-based notion of portability. The experiments can be found in Section 6. Section 7 recapitulates the results.

2. Portability Analysis on an Example

Consider program IRIW in Figure 1. Threads perform computations on local registers ($r_1 ← 1$) and access the memory locations $x$ and $y$ to either read their values into registers ($r_1 ← x$) or write their register values into the memory ($y := r_1$). This program behaves differently when run on x86 than on IBM’s Power or ARM. On TSO, the memory model implemented by x86, each thread has a store buffer of pending writes. A thread can see its own writes before they become visible to other threads (by reading them from its buffer), but once a write hits the memory it becomes visible to all other threads simultaneously: TSO is a multi-copy-atomic model [20]. Power and ARM on the other hand do not guarantee that writes become visible to all threads at the same point in time. Think of each thread as having its own copy of the memory. With these two architectures in mind, consider the execution in Figure 1 (it only contains the memory accesses but omits thread-local computations). Thread $t_1$ reads $x = 1, y = 0$ and $t_2$ reads $x = 0, y = 1$, indicated by the solid edges $rfe$ and $rf$. Since under TSO any execution has a unique global view of all operations, no interleaving allows both threads to read different values of the variables. Under Power, this is possible. Our goal is to automatically detect such differences when porting a program from one architecture to another, here from TSO to Power.

When porting, the developer is confronted with a series of architectures. Our tool PORTHOS applies to various architectures, and we not only have a language for programs but also a language for memory models. The semantics of a program on a memory model is defined axiomatically, following two steps [8, 51]. We first associate with the program (and independent of the memory model) a set of executions which are

\[
\begin{align*}
\text{thread } t_0 & \quad \text{thread } t_1 & \quad \text{thread } t_2 & \quad \text{thread } t_3 \\
\text{Variables: } & r_1 & y & x & r_1 & y & x & r_1 & y & x \\
\text{Transition: } & r_1 & ← 1 & r_1 & ← x & r_1 & ← y & r_1 & ← 1 \\
y & := r_1 & r_2 & ← y & r_2 & ← x & x & := r_1 \\
Iy0 & \xrightarrow{co} Wx1 & Rx1 & \xleftarrow{po} Ry0 & Rxy1 & \xrightarrow{co} Ix0 \\
rfe & \xleftarrow{fr} Rfxy & Rfxy & \xrightarrow{fr} Rfxy & \xrightarrow{rf} Ry0 & \xrightarrow{rf} Rfxy & \xrightarrow{rf} Rfxy & \xrightarrow{rf} Ry0 & \xrightarrow{rf} Rfxy
\end{align*}
\]

Figure 1. Portability of program IRIW from TSO to Power.
candidates for the semantics. An execution is a graph (Figure 1) whose nodes are occurrences of instructions (events) and whose edges are basic dependencies: the program order $po$, the reads-from relation $rf$ (giving the write that a load reads from), and the coherence order $co$ (stating the order in which writes take effect). The memory model then defines which executions are consistent and thus form the semantics of the program on that model.

We describe memory models in the recently proposed language CAT [9]. Besides the three base relations, a model may define so-called derived relations. The consistency requirements are stated in terms of acyclicity and irreflexivity axioms over (base and derived) relations. The CAT formalization of TSO is given in Figure 2. It forbids executions forming a cycle over $rfe$ or $fr$ or $(po \backslash (W \times R))$. The red edges in Figure 1 yield such a cycle, and therefore the execution is not consistent with TSO. Power further relaxes the program order (dotted lines, the formalization is given in Figure 6) and considers the execution consistent. Hence, IRIW has more executions on Power than on TSO.

Our contribution is a bounded analysis for portability bugs. It is implemented in the tool PORTHOS available at

http://github.com/porthoss/PLDI-2017.

First, the program is unrolled up to a user-specified bound. Within this bound, PORTHOS is guaranteed to find all portability bugs. It will neither see bugs beyond the bound nor will it be able to prove a cyclic program portable. Then the unrolled (acyclic) program, together with the CAT models, is transformed into an SMT formula where satisfying assignments correspond to bugs. Encoding the computations is standard bounded model checking, we use a variant of [19]. Portability comes with two problems for which we propose efficient solutions for: (i) the alternation of quantifiers underlying the problem and (ii) the encoding of user-defined memory models.

Concerning (i), a bug is an execution consistent with the target memory model $\mathcal{M}_T$ but inconsistent with the source $\mathcal{M}_S$. We express this combination of consistency and inconsistency with only an existential quantification. The key observation is that the derived relations, which may differ in $\mathcal{M}_T$ and $\mathcal{M}_S$, are fully defined by the execution. Hence, by guessing an execution we also obtain the derived relations (there is nothing more to guess). Checking consistency for $\mathcal{M}_T$ is then an acyclicity (or irreflexivity) constraint on the derived relations that immediately yields an SMT query.

Inconsistency for $\mathcal{M}_S$ requires cyclicity. The trick is to explicitly guess the cycle. We introduce Boolean variables for every event and every edge that could be part of the cycle. In Figure 1, if $Rx1$ is on the cycle, indicated by the variable $C(Rx1)$ being set, then there should be at least one incoming and one outgoing edge also belonging to the cycle. Besides the incoming edge shown in the graph, $Rx1$ could read from the initial value $Ix0$. Since there are two possible incoming edges but only one outgoing edge, we obtain

$$C(Rx1) \Rightarrow ((C_{rfe}(Wx1,Rx1) \lor C_{fr}(Ix0,Rx1)) \land C_{po}(Rx1, Ry0)).$$

If a relation is on the cycle, then also both end-points should be part of the cycle and the relation should belong to the execution:

$$C_{po}(Rx1, Ry0) \Rightarrow (C(Rx1) \land C(Ry0) \land po(Rx1, Ry0)).$$

Finally, at least one event has to be part of the cycle:

$$C(Ix0) \lor C(Wx1) \lor C(Rx1) \lor C(Rx0) \lor C(Iy0) \lor C(Wy1) \lor C(Ry1) \lor C(Ry0).$$

The execution in Figure 1 sets every red relation, it yields a cycle and thus a violation of Axiom $\bullet$ in TSO (Figure 2). The assignment respects the axioms of Power (Figure 6).

Concerning (ii), the challenge is to capture relations that are defined recursively as the least solution to a system of equations. Our contribution is an encoding of the Kleene iteration process into (quantifier-free) integer difference logic. IDL has an NP-complete satisfiability problem, and can be translated into SAT with only low-order polynomial blowup. For every recursive relation and every pair of events, we introduce an integer variable $\Phi^{cd}_{e1,e2}$ representing the iteration step in which the pair $(e1, e2)$ entered the value of relation $r$.

A Kleene iteration then corresponds to a total ordering on these integer variables. Crucially, we only have one Boolean variable $\mathbb{I}(e1, e2)$ per pair rather than one per iteration step.

We illustrate the encoding on a simplified version of the preserved program order for Power defined as $ppo := ii \cup ic$ (cf. Figure 6 for the full definition). The relation is derived from the mutually recursive relations $ii := dd \cup ic$ and $ic := cd \cup ii$, where $dd$ and $cd$ represent data and control dependencies. Call $Rx1$ and $Ry0$ respectively $e1$ and $e2$. The encoding is

$$\mathbb{I}(e1, e2) \Leftrightarrow (dd(e1, e2) \land (\Phi^{dd}_{e1,e2} > \Phi^{cd}_{e1,e2})) \lor (ic(e1, e2) \land (\Phi^{ic}_{e1,e2} > \Phi^{cd}_{e1,e2})).$$

We explain the first equivalence, the second is similar. The pair $(e1, e2)$ that belongs to relation $dd$ in step $\Phi^{dd}_{e1,e2}$ of the Kleene iteration can be added to relation $ii$ at a later step.
\[ \langle \text{prog} \rangle ::= \text{program} \langle \text{thr}\rangle^* \]
\[ \langle \text{thr} \rangle ::= \text{thread} \langle \text{tid} \rangle \langle \text{inst} \rangle \]
\[ \langle \text{inst} \rangle ::= \langle \text{com} \rangle | \langle \text{inst} \rangle ; \langle \text{inst} \rangle | \text{while} \langle \text{pred} \rangle \langle \text{inst} \rangle | \text{if} \langle \text{pred} \rangle \text{then} \langle \text{inst} \rangle \text{else} \langle \text{inst} \rangle \]
\[ \langle \text{com} \rangle ::= \langle \text{reg} \rangle \leftarrow \langle \text{expr} \rangle | \langle \text{reg} \rangle \leftarrow \langle \text{loc} \rangle \]
\[ \langle \text{loc} \rangle ::= \langle \text{reg} \rangle | \langle \text{fence} \rangle \]

**Figure 3.** Programming language.

Since \( dd \) and \( cd \) are empty for IRIW, the relations \( ii \) and \( ic \) have to be identical. Identical non-empty relations will not yield a solution: the integer variables cannot satisfy \( \Phi_{e_1, e_2}^{ii} > \Phi_{e_1, e_2}^{ic} \) and \( \Phi_{e_1, e_2}^{ic} > \Phi_{e_1, e_2}^{ii} \) at the same time. Hence, the only satisfying assignment is the one where both \( ii \) and \( ic \) are the empty relation, which implies that \( ppo \) is empty. This is consistent with the preserved program order of Power for program IRIW.

### 3. Programs and Memory Models

We introduce our language for programs and the core of the language CAT. The presentation follows \[9, 51\] and we refer the reader to those works for details.

#### 3.1 Programs

Our language for shared memory concurrent programs is given in Figure 3. Programs consist of a finite number of threads from a while-language. The threads operate on assembly level, which means they explicitly read from the shared memory into registers, write from registers into memory, and support local computations on the registers. The language has various fence instructions (sync, lwync, and isync on Power) that enforce ordering and visibility constraints among instructions. We refrain from explicitly defining the expressions and predicates used in assignments and conditionals. They will depend on the data domain. For our analysis, we only require the domain to admit an SMT encoding in a logic which has its satisfiability problem in NP. Also note that we make the locations that reads and writes operate on explicit. This eases the presentation. Our technique can be adapted to support address arithmetic.

For the rest of the paper we will assume that programs are acyclic: any while statement is removed by unrolling the program to a depth specified by the user. Since verification is generally undecidable for while-programs [43], this underapproximation is necessary for cyclic programs.

#### 3.2 Executions

The semantics of a program is given in terms of executions, partial orders where the events represent occurrences of the commands and the ordering edges represent dependencies. The definition is given in Figure 4. An execution consists of a set of executed events \( \mathbb{E} \) and so-called base and induced relations satisfying the Axioms \( \spadesuit \) \( \spadesuit \). Base relations \( po \), \( rf \) and \( co \) actually define an execution (they are the ones to be guessed by the solver). Induced relations are divided in two categories: \( sthd \) and \( scl \) are equivalences relating events belonging to the same thread \( \spadesuit \) and accessing the same location \( \spadesuit \). They can be extracted directly from the source code of the program. Relations \( ad \), \( dd \), and \( cd \) depend on the choice of \( po \) and the program. They represent address, data, and control dependencies. The axioms in Figure 4 are common to all memory models and are natively implemented by our tool. To state them, let \( R, W, F \) represent the reads, writes, and fences in the execution (with \( R \cup W \cup F = E \)). Reads and writes are memory accesses \( M := R \cup W \). By \( R_l \) and \( W_l \) we refer respectively to the reads and writes that access location \( l \). The events of thread \( t \) form the set \( E_t \).

Axiom \( \spadesuit \), which we do not make explicit, requires the events to form a path in the threads’ control flow. Axiom \( \spadesuit \) states that the program order \( po \) is an intra-thread relation which \( \spadesuit \) forms a total order when projected to events in the same thread (predicate \( total(r, A) \) holds if \( r \) is a total order on the set \( A \)). By Axioms \( \spadesuit \) and \( \spadesuit \), the reads-from relation \( rf \) gives for each read a unique write to the same location from which the read obtains its value. Here, \( r_1; r_2 := \{(x, y) | \exists z : (x, z) \in r_1 \text{ and } (z, y) \in r_2\} \) is the composition of the relations \( r_1 \) and \( r_2 \). We write \( r^{-1} := \{(y, x) | (x, y) \in r\} \) for the inverse of relation \( r \). Finally, \( id(A) \) is the identity relation on the set \( A \). By Axioms \( \spadesuit \) and \( \spadesuit \), the coherence relation \( co \) relates writes to the same location, and it forms a total order for each location. We will assume the existence of an initial write event for each location which assigns value 0 to the location. This event is first in the coherence order. Address dependencies are either read-to-read or read-to-write \( \spadesuit \). data dependen-
The model relies on the recursively defined relations \( ii \), \( ic \), \( ci \), and \( cc \). Recall that these are the names that denote the relations obtained as the least solution to the four recursive equations. The fence instructions are captured by the relations \( sync \), \( lwsync \), and \( cd-sync \) derived similar to the \( fence \) relation in Figure 2.

Given a program \( P \) and an MCM \( \mathcal{M} \), an execution \( X \) of \( P \) is consistent with \( \mathcal{M} \) if the derived relations of \( \mathcal{M} \) constructed from the base relations of \( X \) satisfy the axioms of \( \mathcal{M} \). We denote the set of consistent executions by \( \text{consistent}_{\mathcal{M}}(P) \). For example, the execution of IRIW shown in Figure 1 satisfies Axioms \( \mathfrak{a}, \mathfrak{b}, \mathfrak{c} \), but not \( \mathfrak{d} \). Thus, the execution is consistent with Power but not with TSO.

### 4. Portability Analysis

Given a program \( P \) and two MCMs \( \mathcal{M}_S \) and \( \mathcal{M}_T \), our goal is to find an execution which is consistent with \( \mathcal{M}_T \) but not with \( \mathcal{M}_S \). In such a case \( P \) has more executions on the target architecture and we say it is not portable from \( \mathcal{M}_S \) to \( \mathcal{M}_T \).

**Definition 1 (Portability).** Let \( \mathcal{M}_S \), \( \mathcal{M}_T \) be two MCMs. A program \( P \) is portable from \( \mathcal{M}_S \) to \( \mathcal{M}_T \) if

\[
\text{consistent}_{\mathcal{M}_T}(P) \subseteq \text{consistent}_{\mathcal{M}_S}(P).
\]

Note than if \( \mathcal{M}_S \) is weaker than \( \mathcal{M}_T \) in the sense that for any program, every execution allowed by \( \mathcal{M}_S \) is also allowed by \( \mathcal{M}_T \), portability boils down to checking that the consistent executions of the program for both MCMs coincide. This is the case for example between SC and TSO, but not between the incomparable RMO and Alpha.

Our method finds non-portable executions as satisfying assignments to an SMT formula. Recall from Section 3.2 that an execution is uniquely represented by the relations \( po \), \( rf \), and \( co \), which need to be guessed by the solver. All other relations are derived from these relations, the source code of the program, and the MCMs in question. Thus, we also have to encode the derived relations of the two MCMs defined in the language of Figure 5. As the last part, we encode the assertions expressed in the language of Figure 5 on these relations in such a way that the guessed execution is allowed by \( \mathcal{M}_T \) (all the assertions stated for \( \mathcal{M}_T \) hold) while the same execution is not allowed by \( \mathcal{M}_S \) (at least one of the axioms of \( \mathcal{M}_S \) is violated).

The sublogic of SMT we use is integer difference logic, which allows our formulas to be more compact than a pure Boolean encoding. The full SMT formula is of the form

\[
\phi_{\text{CF}} \land \phi_{\text{DF}} \land \phi_{\mathcal{M}_T} \land \phi_{\lnot \mathcal{M}_S}.
\]

Here, \( \phi_{\text{CF}} \) and \( \phi_{\text{DF}} \) encode the control flow and data flow of the executions, \( \phi_{\mathcal{M}_T} \) encodes the derived relations and all assertions of \( \mathcal{M}_T \), and \( \phi_{\lnot \mathcal{M}_S} \) encodes the derived relations of \( \mathcal{M}_S \) and a violation of at least one of the assertions of the source memory model.

The control-flow and data-flow encodings are standard for bounded model checking [19]. The control-flow formula captures the branching and merging in the acyclic program. The data-flow encoding relates the values of the variables according to program assignments. For this, we first transform the program into static single assignment form.
The rest of the section focuses on the parts that are new in this work: how to encode the derived relations needed for representing both the MCMs in Section 4.1, how to encode assertions for the target memory model in Section 4.2, and how to encode assertions for the source memory model in Section 4.3.

4.1 Encoding Derived Relations

For any pair of events \( e_1, e_2 \in E \) and relation \( r \subseteq E \times E \) we use a Boolean variable \( \tau(e_1, e_2) \) representing the fact that \( e_1 \rightarrow e_2 \) holds. We similarly use fresh Boolean variables to represent the derived relations, using the encoding to force its value as follows. Computing a reverse relation requires reversing the events; for the union (resp. intersection) of two relations, at least one of them (resp. both of them) should hold; set difference requires that the first relation holds and the second one does not; for the composition of relations we iterate over a third event and check if it belongs to the range of the first relation and the domain of the second.

We define the transitive closure of \( r \) recursively where the base case \( tc_0 \) holds if events are related according to \( r \) and the recursive case uses a relation composition. These are computed with the iterative squaring technique using the relation composition. Finally reflexive and transitive closure checks if the events are the same or are related by \( r^+ \).

The encodings are summarized below.

\[
\begin{align*}
\tau^{-1}(e_1, e_2) &\iff \tau(e_2, e_1) \\
\tau_1 \cup \tau_2(e_1, e_2) &\iff \tau_1(e_1, e_2) \lor \tau_2(e_1, e_2) \\
\tau_1 \cap \tau_2(e_1, e_2) &\iff \tau_1(e_1, e_2) \land \tau_2(e_1, e_2) \\
\tau_1 \setminus \tau_2(e_1, e_2) &\iff \tau_1(e_1, e_2) \land \neg \tau_2(e_1, e_2) \\
\tau_1; \tau_2(e_1, e_2) &\iff \bigvee_{e_3 \in E} \tau_1(e_1, e_3) \land \tau_2(e_3, e_2) \\
\tau^*(e_1, e_2) &\iff \tau^+(e_1, e_2) \lor (e_1 = e_2) \\
\tau^+(e_1, e_2) &\iff \tau_0(e_1, e_2) \lor \tau_0(e_2, e_1) \\
\tau_0(e_1, e_2) &\iff \tau(e_1, e_2) \lor (\tau_0(e_1, e_2) ; \tau_0(e_1, e_2)) \\
\tau_{c_0+1}(e_1, e_2) &\iff \tau_0(e_1, e_2) \lor (\tau_{c_0}(e_1, e_2) ; \tau_{c_0}(e_1, e_2)) \\
\end{align*}
\]

**Encoding Recursively-Defined Relations:** Recall that some of the relations (e.g. \( ii \) and \( ic \) of Power) can be defined mutually recursively, and that we are using the least fixed point (smallest solution) semantics for cyclic definitions. A classical algorithm for solving such equations is the Kleene fixpoint iteration. The iteration starts from the empty relations as initial approximation and on each round computes a new approximation until the (least) fixpoint is reached. Such an iterative algorithm can be easily encoded into SAT. The problem of such an encoding is the potentially large number of iterations needed, and thus the resulting formula size can grow to be large.

A more clever way to encode this is an approach that has been already used in earlier work on encoding mutually recursive monotone equation systems with nested least and greatest fixpoints [34], a framework more expressive than the one used in this work. The encoding of this paper uses an extension of SAT with integer difference logic, a logic that is still NP complete and supported by the SMT solver natively. A SAT encoding is also possible but incurs an overhead in the encoding size: if the SMT encoding is of size \( O(n) \), the SAT encoding is of size \( O(n \log n) \) [34]. The encoding is also closely related to [35, 41] which encodes logic programming under the stable model semantics. They also use a least fixed point semantics for recursive definitions encoded into integer difference logic and SAT. While the basic encoding idea is not new, the approach to encoding axiomatic memory models is novel, and can be of interest to other researchers for efficiently encoding, e.g. Power memory model semantics using an SMT solver.

The basic idea of the encoding is to guess a certificate that contains the iteration number in which a tuple would be added to the result relations in the Kleene iteration. For this we use additional integer variables and enforce that they actually locally follow the propagations made by the fixed point iteration algorithm. Thus, for any pair of events \( e_1, e_2 \in E \) and relation \( r \subseteq E \times E \) we use a fresh integer
variable $\Phi_{r_1,e_2}$ representing the round in which the variable would be derived by the Kleene iteration algorithm. Using these fresh variables we guess the execution of the Kleene fixed point iteration algorithm, and then locally check that every guess that was made is also a valid propagation of the fixed point iteration algorithm. For a simple example on how the encoding for the union of relations needs to be modified to also handle recursive definitions, consider a definition where both $r_1$ and $r_2$ are and

\[ r_1(e_1, e_2) \iff (r_2(e_1, e_2) \land (\Phi_{r_1,e_1,e_2} > \Phi_{r_2,e_1,e_2})) \lor (r_3(e_1, e_2) \land (\Phi_{r_1,e_1,e_2} > \Phi_{r_3,e_1,e_2})) \]

\[ r_2(e_1, e_2) \iff (r_1(e_1, e_2) \land (\Phi_{r_2,e_1,e_2} > \Phi_{r_1,e_1,e_2})) \lor (r_4(e_1, e_2) \land (\Phi_{r_2,e_1,e_2} > \Phi_{r_4,e_1,e_2})). \]

If both $r_3$ and $r_4$ happen to be empty relations, the $r_1$ and $r_2$ relations need to become identical. Also, any candidate solution where both relations are non-empty is not a solution, as in particular the integer variables will not have any solution where both $\Phi_{r_1,e_1,e_2} > \Phi_{r_2,e_1,e_2}$ and $\Phi_{r_2,e_1,e_2} > \Phi_{r_1,e_1,e_2}$ hold at the same time. Thus, the only satisfying assignment for the encoding is the one where both $r_1$ and $r_2$ are empty, which is the correct least fixed point semantics.

### 4.2 Encoding Target Memory Model Assertions

For the target architecture we need to encode all acyclicity and irreflexivity assertions of the memory model. This can be done with a conjunction. For handling acyclicity we again use non-Boolean variables in our SMT encoding for compactness reasons. One can encode that a relation is acyclic by adding a numerical variable $\Psi_e \in \mathbb{N}$ for each event $e$ in the relation we want to be acyclic. Then acyclicity of relation $r$ is encoded as

\[ acyclic(r) \iff \bigwedge_{e_1,e_2 \in E} (r(e_1, e_2) \Rightarrow (\Psi_{e_1} < \Psi_{e_2})). \]

Notice that we can impose a total order with $\Psi_{e_1} < \Psi_{e_2}$ only if there is no cycle. Our encoding is the same as the SAT + IDL encoding in [32] where more discussion of SAT modulo acyclicity can be found.

The irreflexive constraint is simply encoded as

\[ irreflexive(r) \iff \bigwedge_{e \in E} \neg r(e, e). \]

### 4.3 Encoding Source Memory Model Assertions

For the source architecture we have to encode that one of the derived relations does not fulfill its assertions. On the top level this can be encoded as a simple disjunction over all the assertions of the source memory model, forcing at least one of the irreflexivity or acyclicity constraints to be violated.

For the irreflexivity violation, we can reuse the same encoding as for the target memory model simply as $\neg irreflexive(r)$. What remains to be encoded is $acyclic(r)$, which requires the relation $r$ to be cyclic. Here, we give an encoding that uses only Boolean variables. We add Boolean variables $C(e)$ and $C_r(e_1, e_2)$, which guess the edges and nodes constituting the cycle.\(^1\)

We ensure that for every event in the cycle, there should be at least one incoming edge and at least one outgoing edge that are also in the cycle:

\[ c_n = \bigwedge_{e_1 \in E} (C(e_1) \Rightarrow (\bigvee_{e_2 \Rightarrow e_1} C_r(e_2, e_1) \land \bigvee_{e_1 \Rightarrow e_2} C_r(e_1, e_2))). \]

If an edge is guessed to be in a cycle, the edge must belong to relation $r$, and both events must also be guessed to be on the cycle:

\[ c_e = \bigwedge_{e_1, e_2 \in E} (C_r(e_1, e_2) \Rightarrow (r(e_1, e_2) \land C(e_1) \land C(e_2))). \]

A cycle exists, if these formulas hold and there is an event in the cycle:

\[ cyclic(r) \iff (c_e \land c_n \land \bigvee_{e \in E} C(e)). \]

### 5. State Portability

Portability from $M_S$ to $M_T$ requires that there are no new executions in $M_T$ that did not occur in $M_S$. One motivation to check portability is to make sure that safety properties of $M_S$ carry over to $M_T$. Safety properties only depend on the values that can be computed, not on the actual executions. Therefore, we now study a more liberal notion of so-called state portability: $M_T$ may admit new executions as long as they do not compute new states. Admitting more executions means we require less synchronization (fences) to consider a ported program correct, and thus state portability promises more efficient code. The notion has been used in [36].

The main finding in this section is negative: a polynomial encoding of state portability to SAT does not exist (unless the polynomial hierarchy collapses). Phrased differently, state portability does not admit an efficient bounded analysis (like our method for portability). We remind the reader that we restrict our input to acyclic programs (that can be obtained from while-programs with bounded unrolling). For while-programs, verification tasks are generally undecidable [43].

Fortunately, our experiments indicate that new executions often compute new states. This means portability is not only

\(^1\)To improve performance, we actually encode that events and edges \textit{may} belong to the cycle, i.e. for every event (resp. edge) in the cycle its corresponding variable is set to true, but a variable can be set to true even if the event or the edge are not part of the cycle.
a sufficient condition for state portability but, in practice, the two are equivalent. Combined with the better algorithmics of portability, we do not see a good motivation to move to state portability.

A state is a function that assigns a value to each location and register. An execution $X$ computes the state $\text{state}(X)$ defined as follows: a location receives the value of the last write event (according to $\text{co}$) accessing it; for a register, its value depends on the last event in $\text{po}$ that writes to it.

**Definition 2** (State Portability). Let $M_S$, $M_T$ be MCMs. Program $P$ is state portable from $M_S$ to $M_T$ if

$$\text{state}(\text{consistent}_{M_S}(P)) \subseteq \text{state}(\text{consistent}_{M_T}(P)).$$

The relationship between the notions is as follows.

**Lemma 1.** (1) Portability implies state portability. (2) State portability does not imply portability.

For Lemma 1(2), consider a variant of IRIW (Figure 1) where all values are 0. The program is trivially state portable from Power to TSO, but like IRIW, not portable.

We turn to the hardness argumentation. To check state portability, every $M_T$-computable state seems to need a formula checking whether some $M_S$-consistent execution computes it. The result would be an exponential blow-up or a quantified Boolean formula, which is not practical. But can this exponential blow-up or quantification be avoided by some clever encoding trick? The answer is no! Theorem 2 shows that state portability is in a higher class of the polynomial hierarchy than portability. So state portability is indeed harder to check than portability.

The polynomial hierarchy [46] contains complexity classes between NP and PSPACE. Each class is represented by the problem of checking validity of a Boolean formula with a fixed number of quantifier alternations. We need here the classes co-NP $= \Pi^P_1 \subseteq \Pi^P_2$. The tautology problem (validity of a closed Boolean formula with a universal quantifier $\forall x_1 \ldots x_n : \psi$) is a $\Pi^P_2$-complete problem. The higher class $\Pi^P_2$ allows for a second quantifier: validity of a formula ($\forall x_1 \ldots x_n \exists y_1 \ldots y_m : \psi$) is a $\Pi^P_2$-complete problem. Theorem 2 refers to a class of common memory models that we define in a moment. Moreover, we assume that the given pair of memory models $M_S$ and $M_T$ is non-trivial in the sense that $\text{consistent}_{M_T}(P) \subseteq \text{consistent}_{M_S}(P)$ fails for some program, and similar for state portability.

**Theorem 1.** Let $M_S$ and $M_T$ be a non-trivial pair of common MCMs. (1) Portability from $M_S$ to $M_T$ is $\Pi^P_2$-complete. (2) State portability is $\Pi^P_2$-complete.

By Theorem 2(2), state portability cannot be solved efficiently. The first part says that our portability analysis is optimal. We focus on this lower bound to give a taste of the argumentation: given a non-trivial pair of memory models, we know there is a program that is not portable. Crucially, we do not know the program but give a construction that works for any program. The proof of Theorem 2.(2) is along similar lines but more involved.

**Definition 3.** We call an MCM common if

(i) the inverse operator is only used in the definition of $fr$;
(ii) the constructs sthd, sloc, and $(\ast \times \ast)$ are only used to restrict (in a conjunction) other relations,
(iii) it satisfies uniproc (Axiom 1), and
(iv) every program is portable from this MCM to SC.

Notice that all memory models considered in [8] and in this paper are common ones.

We explain the definition. When formulating a MCM, one typically forbids well-chosen cycles of base relations (and $fr$). To this end, derived relations are introduced that capture the paths of interest, and acyclicity constraints are imposed on the derived relations. The operators inverse and $(\ast \times \ast)$ may do the opposite; they add relations that do not correspond to paths of base relations (and $fr$). Besides stating what is common in MCMs, Properties (ii) and (ii) help us compose programs (cf. next paragraph). Uniproc is a fundamental property without which an MCM is hard to program. Since the purpose of an MCM is to capture SC relaxations, we can assume MCMs to be weaker than SC. Properties (iii) and (iv) guarantee that the program $P_\psi$ given below is portable between any common MCMs.

The crucial property of common MCMs is the following. For every pair of events $e_1, e_2$ in a derived relation, (1) there are (potentially several) sequences of base relations (and $fr$) that connect $e_1$ and $e_2$, and (2) the derived relation only depends on these sequences. The property ensures that if we append a program $P'$ to a location-disjoint program $P$, consistency of composed executions is preserved.

It remains to prove $\Pi^P_2$-hardness of portability. We first introduce the program $P_\psi$ that generates some assignment and checks if it satisfies the Boolean formula $\psi(x_1 \ldots x_m)$ (over the variables $x_1 \ldots x_m$). The program $P_\psi := t_1 \parallel t_2$ consists of the two threads $t_1$ and $t_2$ defined below. Note that we cannot directly write a constant $i$ to a location, so we first assign $i$ to register $r_{c,i}$.

| thread $t_1$ | thread $t_2$ |
|---|---|
| $r_{c,0} \leftarrow 0$; $r_{c,1} \leftarrow 1$; $r_{c,2} \leftarrow 2$ | $r_{c,1} \leftarrow 1$; |
| $x_1 := r_{c,0} \ldots x_m := r_{c,0}$; | $x_1 := r_{c,1} \ldots x_m := r_{c,1}$; |
| $r_1 \leftarrow x_1 \ldots r_m \leftarrow x_m$; | if $\psi(r_1 \ldots r_m)$ then |
| if $\psi(r_1 \ldots r_m)$ then |
| $y := r_{c,2}$; | $y := r_{c,1}$; |
| else | |

We reduce checking whether $\forall x_1 \ldots x_m : \psi(x_1 \ldots x_m)$ holds to portability of a program $P_\psi$. The idea for $P_\psi$ is
6. Experiments

The encoding from Section 4 has been implemented in a tool called PORTHOS which uses Z3 [25] as the back-end SMT solver. In this section we evaluate PORTHOS on benchmark programs on a wide range of well-known MCMs. For SC, TSO, PSO, RMO and Alpha (henceforth called traditional architectures) we use the formalizations from [3]. We also compare two Power models, the one in Figure 6 taken from [8] and a previous slightly different version from [5], henceforth called CAV10 and formalized in Figure 9.

We divide our results in three categories: Section 6.1 discusses portability of mutual exclusion algorithms, Section 6.2 portability of litmus tests, and Section 6.3 the performance of the tool.

6.1 Portability of Mutual Exclusion Algorithms

The other tools that are MCM-aware [8, 51] accept only litmus tests as inputs. PORTHOS, however, can analyze cyclic programs with control flow branching and merging by unrolling them into acyclic form. In order to show the broad applicability of our method, we tested portability of several mutual exclusion algorithms: Lamport’s bakery [37], Burns’ protocol [16], Dekker’s [27], Lamport’s fast mutex [38], Parker’s [26], Peterson’s [42] and Szymanski’s [47]. The benchmarks also include previously known fenced versions for TSO (marked as x86) and new versions we introduced using Power fences (marked as POWER). The mutual exclusion program loops were unrolled once in all the experiments to obtain an acyclic program, and the discussion in what follows is for the portability analysis of this acyclic program.

While these algorithms have been proven correct for SC, it is well known that they do not guarantee mutual exclusion when ported to weaker architectures. The effects of relaxing the program order have been widely studied; there are techniques that even place fences automatically to guarantee portability, but they assume SC as the source architecture [7, 13]. In this section we do not only confirm that fenceless versions of the benchmarks are not portable from SC to TSO and fenced versions of them are (1st column of Table 1), we also show that those fences are not enough to guarantee mutual exclusion when ported from TSO to Power (5th column of the table).

Table 1 shows that new behaviors are introduced in several of the fenceless benchmarks for most of the architectures combinations. The few cases where no behaviors are added occur when porting from TSO to PSO or between RMO and Alpha (in both directions). Even though such

| Benchmark      | SC-TSO | SC-PSO | SC-RMO | TSO-PSO | TSO-RMO | RMO-Alpha | RMO-Power | PSO-Alpha | PSO-Power | Alpha-RMO | Alpha-Power |
|----------------|--------|--------|--------|---------|---------|-----------|-----------|-----------|-----------|-----------|-------------|
| BAKERY         | ✓      | ✓      | ✓      | ✓       | ✓       | ✓         | ✓         | ✓         | ✓         | ✓         | ✓           |
| BAKERY X86     | ✓      | ✓      | ✓      | ✓       | ✓       | ✓         | ✓         | ✓         | ✓         | ✓         | ✓           |
| BAKERY Power   | -      | -      | -      | -       | -       | -         | -         | -         | -         | -         | -           |
| Burns          | x      | x      | ✓      | ✓       | ✓       | ✓         | ✓         | ✓         | ✓         | ✓         | ✓           |
| Burns X86      | x      | x      | x      | ✓       | ✓       | ✓         | ✓         | ✓         | ✓         | ✓         | ✓           |
| Burns Power    | -      | -      | -      | -       | -       | -         | -         | -         | -         | -         | -           |
| Dekker         | ✓      | x      | x      | x       | x       | ✓         | ✓         | ✓         | ✓         | ✓         | ✓           |
| Dekker X86     | ✓      | x      | x      | x       | x       | ✓         | ✓         | ✓         | ✓         | ✓         | ✓           |
| Dekker Power   | -      | -      | -      | -       | -       | -         | -         | -         | -         | -         | -           |
| Lamport        | ✓      | ✓      | ✓      | ✓       | ✓       | ✓         | ✓         | ✓         | ✓         | ✓         | ✓           |
| Lamport X86    | ✓      | ✓      | ✓      | ✓       | ✓       | ✓         | ✓         | ✓         | ✓         | ✓         | ✓           |
| Lamport Power  | -      | -      | -      | -       | -       | -         | -         | -         | -         | -         | -           |
| Parker         | x      | x      | ✓      | ✓       | ✓       | ✓         | ✓         | ✓         | ✓         | ✓         | ✓           |
| Parker X86     | x      | x      | x      | ✓       | ✓       | ✓         | ✓         | ✓         | ✓         | ✓         | ✓           |
| Peterson       | x      | x      | x      | ✓       | ✓       | ✓         | ✓         | ✓         | ✓         | ✓         | ✓           |
| Peterson X86   | -      | -      | -      | -       | -       | -         | -         | -         | -         | -         | -           |
| Peterson Power | -      | -      | -      | -       | -       | -         | -         | -         | -         | -         | -           |
| Szymanski      | ✓      | x      | x      | x       | x       | ✓         | ✓         | ✓         | ✓         | ✓         | ✓           |
| Szymanski X86  | ✓      | x      | x      | x       | x       | ✓         | ✓         | ✓         | ✓         | ✓         | ✓           |
| Szymanski Power| -      | -      | -      | -       | -       | -         | -         | -         | -         | -         | -           |

Table 1. Bounded portability analysis of mutual exclusion: portable (✓), non-portable (X), does not apply (-).
models are different, such differences are subtle enough not to be exhibited for those particular programs at the analyzed bounded depth.

Even though previously known fenced versions of the benchmarks guarantee mutual exclusion on TSO (and even on PSO for most programs), that guarantee is lost when porting to Power (except for PARKER x86). We have used PORTHOS to find portability bugs when porting from TSO to Power and manually added fences to forbid such executions (see benchmarks marked as POWER). To the best of our knowledge these are the first results about portability of mutual exclusion algorithms from memory models weaker than SC to the Power architecture.

6.2 Checking Portability on Litmus Tests

We compare the results of PORTHOS (which implements portability) against HERD7 (http://diy.inria.fr/herd) which reasons about state reachability and can be used to test state portability. HERD7 systematically constructs all consistent executions of the program and exhaustively enumerates all possible computable states. Such enumeration can be very expensive for programs with lots of computable states, e.g., for many programs with a very large level of concurrency. Since HERD7 only allows to reason about one memory model at a time, for each test we run the tool twice (one for each MCM) and compare the set of computable states. The program is not state portable if the target MCM generates computable states that are not computable states of the source MCM.

Our experiments contain two test suites: TS1 contains 1000 randomly generated litmus tests in x86 assembly (to test traditional architectures) and TS2 contains 2427 litmus tests in Power assembly taken from [40]. Each test contains between 2 and 4 threads and between 4 and 20 instructions.

Portability vs State Portability. For very small programs, such as the litmus tests, sometimes the new executions allowed by the target architecture do not create new computable states of the program. Table 2 reports the number of non-portable (w.r.t. both definitions) litmus tests (XX), the number of portable and state-portable litmus tests (✓✓) and the number of litmus tests that are not portable but are still state portable (✓✓). In the last case the new executions allowed by the target memory model do not result in new computable states of the program. We will show that in many cases both notions of portability coincide. For TS1 on traditional architectures, the amount of non state-portable tests is very low (1.38%), while the non portability of the program does not generate a new computable state in 13.97% of the cases. For TS2 from traditional architectures to Power, the number of non state-portable litmus tests rises to 31.93%, while only in 6.16% of cases the two notions of portability do not match because the new executions do not result in a new computable state for the program.

In order to remove some executions that do not lead to new computable states, PORTHOS optionally supports the use of dead executions which has been recently proposed in [51]. Dead executions are either consistent or lead to non computable states. Formally an execution X is dead if X \notin consistentMCA(P) implies that state(X) \neq state(Y) for all \ Y \in consistentMCA(P).

Instead of looking for any execution which is not consistent for the source architecture, we restrict the search to non-consistent and dead executions of M\mathcal{S}. Dead executions can be approximated with constraints \#1 and \#3 from Figure 7 where r^+ represents the reflexive closure of r, and imm(r) = r \setminus (r;r^+). These constraints can be easily encoded into SAT. Our tool has an implementation which rules out quite a few executions not computing new states. The last two columns of Table 2 show that by restricting the search to dead executions, the ratio of litmus tests the tool reports as non portable, but are actually state portable is reduced to 10.77% for traditional architectures and to 4.44% for Power.

Multi-copy Atomicity. Multi-copy atomicity (MCA) is a property of MCMs that ensures that different threads cannot observe writes in conflicting order in the absence of thread-local reordering, i.e., there is an unique copy of memory that serializes all writes [20]. It has been axiomatically defined for the first time in [51] where the authors systematically show there exists no program containing at most 6 events which differentiates between TSO and MCA. This is consistent with TSO satisfying the MCA property. The model is given in Figure 8 and comprises write/write coherence \#1 and the acyclicity of the write order relation wo which is parametrized by the preserved program order relation ppo \#1

Here fr_{init} is the projection of fr over the events that read from an initial write.

In order to test their axiomatic definition of MCA we performed an experiment which consists of 1000 randomly generated litmus tests with up to 30 events and we have not found any counterexample. We also tested MCA for Power on TS2 and found several counter-examples (e.g., a variation of IRIW) showing that Power is not an MCA model.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure7.png}
\caption{Figure 7. Deadness [51].}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure8.png}
\caption{Figure 8. Multi-copy atomicity [51].}
\end{figure}
Comparing Power models. The CAV10 model in Figure 9 is a slightly semantically different formalization of Power that has a simpler definition than the most recent version shown in Figure 6. It only considers that events in the program order related by data/control dependencies or isync instructions are guaranteed to be preserved. The model uses recursive definitions and composition of relations to model cumulativity of fences (absync and ablsyn relations) and is formalized by the uniproc Axiom 1 and constraint 2 in Figure 9. We have run PORTHOS to test portability both from Power to CAV10 and from CAV10 to Power on TS2. As explained in [8], both Power models are not comparable since there exist tests that violates portability in both directions. Our tool found such examples on TS2. However, both models seem to be sound w.r.t. hardware.

6.3 Performance

For small litmus test, the running times of HERD7 outperform PORTHOS. However, as soon as the programs become bigger, HERD7 does not perform as well as PORTHOS. We believe this is due to the use of efficient search techniques in the Z3 SMT solver. The impact on efficiency is manifested as the number of executions HERD7 has to explicitly simulate by enumeration grows. We evaluate the solving times of our tool on the mutual exclusion benchmarks. Our prototype encoding implementation is done in Python; the encoding generations times are on the average 31.3 s with a minimum of 1.1 s and a maximum of 160.6 s. The encodings involving Power are usually more time consuming than traditional models since Power has both transitive closures and least fixed points in its encoding. We expect that the encoding times could be vastly improved by a careful C/C++ implementation of the encoding.

Figure 10 presents the solving times of PORTHOS (using Z3) for the mutual exclusion algorithms, which are actually much lower than the encoding times for our prototype implementation. For most of the cases it takes less than 10 secs; exceptions are LAMPORT POWER and all SZYMANSKI versions when testing portability to Power. For most of such cases the programs are portable which correspond to the encoding being UNSAT, which typically is harder for SAT/SMT solvers than finding a satisfying truth assignment. See for example SZYMANSKI x86 for RMO-Power and Alpha-Power which takes more time than the other models combinations. Solving times for traditional models are faster than those of Power by one order of magnitude.

7. Conclusions

We introduce the first method that tests portability between any two axiomatic memory models defined in the CAT language. The method reduces portability analysis to satisfiability of an SMT formula in SAT + integer difference logic. We propose efficient solutions for two crucial tasks: reasoning about two user-defined MCMs at the same time and encoding recursively defined relations (needed for Power) into SMT. Our complexity analysis and experimental results both suggest that our definition of portability is preferable over the state-based notion of portability. If state-based portability is required, the complexity results show that it can not be done with a single SMT solver query, unlike the approach to portability analysis suggested in this paper. We also show that our method is not restricted just to litmus tests that have no branching structure but actually for the first time report on automated tool-based portability analysis of mutual exclusions algorithms from several axiomatic memory models to Power.

Acknowledgments

This work was partially supported by the Academy of Finland project 277522. Florian Furbach was supported by the DFG project R2M2: Robustness against Relaxed Memory Models.
References

[1] P. A. Abdulla, S. Aronis, M. F. Atig, B. Jonsson, C. Leonardsson, and K. F. Sagoras. Stateless model checking for TSO and PISO. In TACAS, volume 9035 of LNCS, pages 353–367. Springer, 2015.

[2] P. A. Abdulla, M. F. Atig, B. Jonsson, and C. Leonardsson. Stateless model checking for POWER. In CAV, volume 9780 of LNCS, pages 134–156. Springer, 2016.

[3] J. Alglave. *A Shared Memory Poetics*. Thèse de doctorat, L’université Paris Denis Diderot, 2010.

[4] J. Alglave and L. Maranget. Stability in weak memory models. In CAV, volume 6806 of LNCS, pages 50–66. Springer, 2011.

[5] J. Alglave, L. Maranget, S. Sarkar, and P. Sewell. Fences in weak memory models. In CAV, volume 6174 of LNCS, pages 258–272. Springer, 2010.

[6] J. Alglave, D. Kroening, and M. Tautschnig. Partial orders for efficient bounded model checking of concurrent software. In CAV, volume 8044 of LNCS, pages 141–157. Springer, 2013.

[7] J. Alglave, D. Kroening, V. Nimal, and D. Poetzl. Don’t sit on the fence - A static analysis approach to automatic fence insertion. In CAV, volume 8559 of LNCS, pages 508–524. Springer, 2014.

[8] J. Alglave, L. Maranget, and M. Tautschnig. Herding cats: Modelling, simulation, testing, and data mining for weak memory. *ACM Trans. Program. Lang. Syst.*, 36(2):7:1–7:74, 2014.

[9] J. Alglave, P. Cousot, and L. Maranget. Syntax and semantics of the weak consistency model specification language CAT. *CoRR*, abs/1608.07531, 2016.

[10] M. F. Atig, A. Bouajjani, S. Burckhardt, and M. Musuvathi. On the verification problem for weak memory models. In POPL, pages 7–18. ACM, 2010.

[11] M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber. Mathematizing C++ concurrency. In POPL, pages 55–66. ACM, 2011.

[12] M. Batty, A. F. Donaldson, and J. Wickerson. Overhauling SC atomics in C11 and OpenCL. In POPL, pages 634–648. ACM, 2016.

[13] A. Bouajjani, E. Derezene, and R. Meyer. Checking and enforcing robustness against TSO. In ESOP, volume 7792 of LNCS, pages 533–553. Springer, 2013.

[14] S. Burckhardt and M. Musuvathi. Effective program verification for relaxed memory models. In CAV, volume 5123 of LNCS, pages 107–120. Springer, 2008.

[15] S. Burckhardt, R. Alur, and M. M. K. Martin. CheckFence: Checking consistency of concurrent data types on relaxed memory models. In PLDI, pages 12–21. ACM, 2007.

[16] J. Burns and N. Lynch. Bounds on shared memory for mutual exclusion. *Information and Computation*, 107(2):171 – 184, 1993.

[17] J. F. Cantin, M. H. Lipasti, and J. E. Smith. The complexity of verifying memory coherence and consistency. *IEEE Trans. Parallel Distrib. Syst.*, 16(7):663–671, 2005.

[18] E. M. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, 2001.

[19] H. Collavizza and M. Rueher. Exploration of the capabilities of constraint programming for software verification. In TACAS, volume 3920 of LNCS, pages 182–196. Springer, 2006.

[20] W. W. Collier. *Reasoning about parallel architectures*. Prentice Hall, 1992.

[21] S. Cotton, E. Asarin, O. Maler, and P. Niebert. Some progress in satisfiability checking for difference logic. In FORMATS, volume 3253 of LNCS, pages 263–276. Springer, 2004.

[22] D. C. da Cruz, M. J. Frade, and J. S. Pinto. Verification conditions for single-assignment programs. In SAC, pages 1264–1270. ACM, 2012.

[23] A. M. Dan, Y. Meshman, M. T. Vechev, and E. Yahav. Predictive abstraction for relaxed memory models. In SAS, volume 7935 of LNCS, pages 84–104. Springer, 2013.

[24] A. M. Dan, Y. Meshman, M. T. Vechev, and E. Yahav. Effective abstractions for verification under relaxed memory models. In VMCAI, volume 8931 of LNCS, pages 449–466. Springer, 2015.

[25] L. De Moura and N. Björner. Z3: An efficient smt solver. In TACAS, volume 4963 of LNCS, pages 337–340. Springer, 2008.

[26] D. Dice. A race in locksupport park() arising from weak memory models, 2009. URL https://blogs.oracle.com/dave/entry/a_race_in_locksupport_park.

[27] E. W. Dijkstra. Cooperating sequential processes. In P. B. Hansen, editor, *The Origin of Concurrent Programming*, pages 65–138. Springer-Verlag New York, Inc., 2002.

[28] C. Enea and A. Farzan. On atomicity in presence of non-atomic writes. In TACAS, volume 9636 of LNCS, pages 497–514. Springer, 2016.

[29] A. Farzan and P. Madhusudan. Monitoring atomicity in concurrent programs. In CAV, volume 5123 of LNCS, pages 52–65. Springer, 2008.

[30] S. Flur, K. E. Gray, C. Pulte, S. Sarkar, A. Sezgin, L. Maranget, W. Deacon, and P. Sewell. Modelling the ARMv8 architecture, operationally: concurrency and ISA. In POPL, pages 608–621. ACM, 2016.

[31] F. Furbach, R. Meyer, K. Schneider, and M. Sentfleben. Memory-model-aware testing: A unified complexity analysis. *ACM Trans. Embedded Comput. Syst.*, 14(4):63, 2015.

[32] M. Gebser, T. Janhunen, and J. Rintanen. SAT modulo graphs to propositional clauses. In *JELIA*, pages 137–151, 2014.

[33] P. B. Gibbons and E. Korach. Testing shared memories. *SIAM Journal on Computing*, 26:1208–1244, 1997.

[34] K. Heljanko, M. Kiehnänen, M. Lange, and I. Niemelä. Solving parity games by a reduction to SAT. *J. Comput. Syst. Sci.*, 78(2):430–440, 2012.

[35] T. Janhunen and I. Niemelä. Compact translations of non-disjunctive answer set programs to propositional clauses. In *Logic Programming, Knowledge Representation, and Non-
[36] M. Kuperstein, M. T. Vechev, and E. Yahav. Automatic inference of memory fences. *SIGACT News*, 43(2):108–123, 2012.

[37] Leslie Lamport. A new solution of Dijkstra’s concurrent programming problem. *Commun. ACM*, 17(8):453–455, 1974.

[38] Leslie Lamport. A fast mutual exclusion algorithm. *ACM Trans. Comput. Syst.*, 5(1):1–11, 1987.

[39] F. Liu, N. Nedev, N. Prisadnikov, M. T. Vechev, and E. Yahav. Dynamic synthesis for relaxed memory models. In *PLDI*, pages 429–440. ACM, 2012.

[40] S. Mador-Haim, L. Maranget, S. Sarkar, K. Memarian, J. Alglave, S. Owens, R. Alur, M. K. Martin, P. Sewell, and D. Williams. An axiomatic memory model for POWER multiprocessors. In *CAV*, volume 7358 of *LNCS*, pages 495–512. Springer, 2012.

[41] I. Niemelä. Stable models and difference logic. *Ann. Math. Artif. Intell.*, 53(1-4):313–329, 2008.

[42] G. L. Peterson. Myths about the mutual exclusion problem. *Inf. Process. Lett.*, 12(3):115–116, 1981.

[43] H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):358–366, 1953.

[44] S. Sarkar, P. Sewell, F. Z. Nardelli, S. Owens, T. Ridge, T. Braibant, M. O. Myreen, and J. Alglave. The semantics of x86-CC multiprocessor machine code. In *POPL*, pages 379–391. ACM, 2009.

[45] S. Sarkar, P. Sewell, J. Alglave, L. Maranget, and D. Williams. Understanding POWER multiprocessors. In *PLDI*, pages 175–186. ACM, 2011.

[46] L. J. Stockmeyer. The polynomial-time hierarchy. *Theor. Comput. Sci.*, 3(1):1–22, 1976.

[47] B. K. Szymanski. A simple solution to Lamport’s concurrent programming problem with linear wait. In *ICS*, pages 621–626. ACM, 1988.

[48] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific journal of mathematics*, 5(2):285–309, 1955.

[49] A. Turon, V. Vafeiadis, and D. Dreyer. GPS: navigating weak memory with ghosts, protocols, and separation. In *OOPSLA*, pages 691–707. ACM, 2014.

[50] V. Vafeiadis and C. Narayan. Relaxed separation logic: a program logic for C11 concurrency. In *OOPSLA*, pages 867–884. ACM, 2013.

[51] J. Wickerson, M. Batty, T. Sorensen, and G. A. Constantinides. Automatically comparing memory consistency models. In *POPL*, 2017.
A. Rest of the encoding

This section details the remaining two sub formulas for the portability encoding, i.e. the control-flow and the data-flow.

A.1 Control-flow

Instead of representing the branching of the program with a tree [22], we use a direct acyclic graph (DAG) capturing the branches of the program and how those merge again. This allows to keep the size of the control-flow formula linear w.r.t the (unfolded) program. The tree representation can be exponential if the program has several if statements. We rewrite the (unfolded) program. The tree representation can

For each instruction $i$, we use a Boolean variable $cf_i$ representing the fact that the instruction is actually executed by the execution. For a sequence $i_1 := i_2; i_3$, instruction $i_1$ belongs to the execution iff both $i_2$ and $i_3$ belong too (1). Assignments (local computations, loads and stores) and fences do not impose any restriction in the control-flow encoding (2)-(5): belonging or not to the execution depends on them being part of the body of some if statement at a higher level of the recursive definition. Given an instruction $i_1 := \text{if } b \text{ then } i_2 \text{ else } i_3$, we use three control-flow variables $cf_{i_1}, cf_{i_2}, cf_{i_3}$: then $i_1$ is executed iff one of $i_2, i_3$ is performed (6), which one actually depends on the value of $b$ and this is encoded in the data-flow formula $\phi_{DF}$. These restrictions are encoded recursively by the following constraints:

\[
\phi_{CF}(i_2; i_3) = cf_{i_2} \Rightarrow (cf_{i_2} \land cf_{i_3}) \quad (1)
\]
\[
\phi_{CF}(r \leftarrow e) = cf_{r=e} \quad (2)
\]
\[
\phi_{CF}(r \leftarrow l) = cf_{r=l} \quad (3)
\]
\[
\phi_{CF}(l := r) = cf_{l=r} \quad (4)
\]
\[
\phi_{CF}(\text{fence}) = cf_{\text{fence}} \quad (5)
\]
\[
\phi_{CF}(\text{if } b \text{ then } i_2 \text{ else } i_3) = cf_{i_1} \Rightarrow (cf_{i_2} \lor cf_{i_3}) \land \phi_{CF}(i_2) \land \phi_{CF}(i_3) \quad (6)
\]

A.2 Data-flow

We encode the data flow with single static assignments using the method of [19]. Formula $\phi_{DF}$ represents how the data flows between locations and registers; we first focus on how the data-flow of the local thread behavior is encoded (sub-formula $\phi_{DF_{thrd}})$. For each location of the program (resp. register) we use several integer variables (one for each variable in the SSA form of the program) representing the value carried by that location (resp. register) in the execution. For loads, stores and local computations, if the instruction is part of the execution (i.e. its control-flow variable is True) then both sides of the assignment should coincide (7)-(9). For a sequence, the formula is the conjunction of the encoding of the corresponding instructions (10).

Suppose register $r$ and location $l$ have been already assigned $p$ and $q$ times respectively, then:

\[
\phi_{DF_{thrd}}(r \leftarrow e) = cf_{r=e} \Rightarrow (r_{p+1} = e) \quad (7)
\]
\[
\phi_{DF_{thrd}}(r \leftarrow l) = cf_{r=l} \Rightarrow (r_{p+1} = l_{q+1}) \quad (8)
\]
\[
\phi_{DF_{thrd}}(l := r) = cf_{l=r} \Rightarrow (l_{q+1} = r_p) \quad (9)
\]
\[
\phi_{DF_{thrd}}(i_1; i_2) = \phi_{DF_{thrd}}(i_1) \land \phi_{DF_{thrd}}(i_2) \quad (10)
\]

Following the SSA form, the left hand side of each assignment introduces new variables; for registers in the right hand side, indexes are not updated so they match with the last value which can only be modified by the same thread (9). However for locations in the right hand side, the index is also updated (8) to allow variables to match not only with the last assignment done by that thread, but also from other threads (see the sub-formula $\phi_{DF_{mem}}$ below).

If statements may have a different number of assignments in their branches for certain variables. The idea here is to insert dummy assignments to ensure that both branches have the same number of assignments. We show the encoding for the simple case where each branch consists only of local computations to a register $r$. The same process is applied individually for each register and location assigned in a branch. If the branches contain if statements, the procedure must be applied recursively to each of them. Consider the if statement $\text{if } b \text{ then } i_1 \text{ else } i_2$ where the first branch has less assignments to $r$ than the second one, i.e. $i_1 := r \leftarrow e_1, \ldots; r \leftarrow e_{1,p}$ and $i_2 := r \leftarrow e_{2,1}, \ldots; r \leftarrow e_{2,q}$ with $p < q$ (the encoding is symmetric for $q < p$). Assume $r$ has been already assigned $x$ times, then the encoding of the instruction contains the following constraint:

\[
\phi_{DF_{thrd}}(\text{if } b \text{ then } i_1 \text{ else } i_2) = (b \Rightarrow cf_{i_1}) \land (\neg b \Rightarrow cf_{i_2}) \land cf_{i_1} \Rightarrow (r_{x+1} = e_{1,1}) \land (r_{x+p} = e_{1,p}) \land (r_{x+q} = e_{2,q}) \quad (11)
\]

Constraint (11) imposes that which branch is followed depends on the value of the predicate. Next, we specify how the value of $r$ is updated depending on the branch: if the first branch is taken, then the value of $r$ is updated according to the expressions the first $p$ times (12)-(13) and it remains unchanged for the remaining $q - p$ assignments (14); if the second branch is taken, the value of $r$ is updated according to the corresponding expressions in that branch (15)-(16). By adding constraints which keep assignments unchanged, we can easily model how branches merge again since any variable assigned after the if statement would be matched with the last value assigned to that variable.

Since fresh variables are added for locations in both sides of the assignments (8), their values are not yet constrained. We now specify how the data flows between instructions that access locations in the shared memory (possibly in different threads). This depends on where the values are read-from (i.e. the $rt$ relation) and is encoded
by constraints $DF_{mem}(i_1, i_2)$. A write instruction $l := r$ generates data-flow constraint $cf_{l := r} \Rightarrow (l = r)$ and a read $r \leftarrow l$ is encoded by $rf_{l \rightarrow e} \Rightarrow (r = l_j)$. The variables $l_i, l_j$ remain unconstrained. If both instructions (call them $t_1$ and $i_2$) are related over the $rf$ relation, then their values need to match:

$$\phi_{DF_{mem}}(i_1, i_2) = rf(i_1, i_2) \Rightarrow l_i = l_j$$

Finally, the data-flow between register and location either within or between threads is encoded as:

$$\phi_{DF} = \bigwedge_{(i_1, i_2) \in \{W \times R\} \cap sloc} \phi_{DF_{loc}}(i_1, i_2) \wedge \bigwedge_{t \in T} \phi_{DF_{thrd}}(i_t)$$

where $i_t$ represent the instruction at the highest recursive level of the thread.

### B. Complexity Proofs

We recall the main theorem and the program $P_t := t_1 \parallel t_2$ from the paper:

**Theorem 2.** Let $M_S$ and $M_T$ be a non-trivial pair of common MCMs. (1) Portability from $M_S$ to $M_T$ is $\Pi^1_2$-complete. (2) State portability is $\Pi^1_2$-complete.

#### $t_1$

1. $r_{c,0} \leftarrow 0; r_{c,1} \leftarrow 1; r_{c,2} \leftarrow 2;$
2. $x_1 := r_{c,0}; \ldots; x_m := r_{c,0};$ \hspace{1cm} // Writes $w_{1,0} \ldots w_{m,0}$
3. $r_1 \leftarrow x_1; \ldots; r_m \leftarrow x_m;$ \hspace{1cm} // Reads $r_1 \ldots r_m$
4. if $\psi(r_1, \ldots, r_m)$ then \hspace{1cm} // If $A$ satisfies $\psi$,
5. \hspace{1cm} $y := r_{c,2};$ \hspace{1cm} // return 2.
6. else
7. \hspace{1cm} $y := r_{c,1};$ \hspace{1cm} // If it doesn’t, return 1.

#### $t_2$

1. $r_{c,1} \leftarrow 1;$
2. $x_1 := r_{c,1}; \ldots; x_m := r_{c,1};$ \hspace{1cm} // Writes $w_{1,1} \ldots w_{m,1}$

We give some technical results and show $\Pi^1_2$-Completeness of Portability for common MCMs.

**Lemma 2.** $P_t$ is portable from every common MCM to another common MCM.

**Proof.** According to property (iv), any common MCM is portable to SC. In SC, an execution corresponds to an interleaving of the two thread executions. First, the threads create some variable assignment $A$ which is read by $t_1$. Then, $t_1$ checks whether the assignment satisfies $\psi$. If it does, $y$ is set to 2, otherwise $y$ is set to 1.

We show that any consistent execution of some common MCM is SC-consistent by examining possible executions:

- If $w_{i,0} \overset{co}{\rightarrow} w_{i,1}$ and $w_{i,1} \overset{cf}{\rightarrow} r_i$, then this corresponds to an interleaving where $w_{i,1}$ occurs first, then $t_1$ writes $w_{i,0}$ and reads 0 ($r_i$).
- If $w_{i,0} \overset{co}{\rightarrow} w_{i,1}$ and $w_{i,0} \overset{cf}{\rightarrow} r_i$, then $w_{i,0}$ and $r_i$ in $t_1$ occur first and then $w_{i,1}$.
- If $w_{i,1} \overset{co}{\rightarrow} w_{i,0}$ and $w_{i,0} \overset{cf}{\rightarrow} r_i$, then $t_1$ writes $w_{i,0}$, $t_2$ over-writes this with $w_{i,1}$ and afterward $t_1$ reads 1 with $r_i$.
- If $w_{i,1} \overset{co}{\rightarrow} w_{i,0}$ and $w_{i,0} \overset{cf}{\rightarrow} r_i$, then the derived relation $fr := rf^{-1}; co$ satisfies $r_i \overset{co}{\rightarrow} w_{i,0}$. Since $w_{i,0}$ and $r_i$ are related by po and access the same location there is a cycle $w_{i,0} \overset{po}{\rightarrow} \overset{loc}{\rightarrow} w_{i,0}$. This is a violation of uniproc, the situation can not occur in a common MCM.

So we can construct a corresponding interleaving for any execution of a common MCM. It follows that every execution of a common MCM is SC-consistent and according to property (iv) consistent with any common MCM.

We use the following technical lemmas to show hardness. We call the relations po, rf, co, ad, dd, cd and fr basic. Given a common MCM, we define the violating cycles as follows: For an assertion acyclic(r), any cycle of $r$ is violating. For an assertion irreflexive(r), any cycle of the form $e \rightarrow e$ is violating.

The following lemma shows that an execution is not consistent if it contains a violating cycle.

**Lemma 3.** Let $M$ be common. An execution $X$ is consistent with $M$ if $X$ contains no violating cycle of $M$.

This follows directly from the definition of violating cycles.

We say a relation $r$ satisfies the path condition if $e_1 \rightarrow e_2$ implies $e_1 \rightarrow^* e_2$ with $b := po \cup rf \cup co \cup ad \cup dd \cup cd \cup fr$.

**Lemma 4.** Any relation $r$ of a common MCM satisfies the path condition.

**Proof.** Note that the recursively defined relations of a common MCM can be obtained with a Kleene iteration. We use a structural induction over the Kleene iteration.

Induction Basis: Any named relation is initially the empty relation and trivially satisfies the path condition.

Induction Step: Assume all named relations satisfy the path condition. A named relation is updated according to its defining equation using the current assignments of the named relations and basic relations. To simplify this proof, we can assume that any equation only contains one operator or a relation in base. Any basic relation trivially satisfies the condition. Let $r_1$ and $r_2$ satisfy the path condition. We examine the operations that can be applied to them according to properties (i) and (ii) of common MCMs. We see that $r_1 \cup r_2$, $r_1 \cap r_2$, $sloc$, $sval$, sthd, $\langle \{} \rangle \times \langle \{} \rangle$, $r_1 \setminus r_2$ all satisfy the path condition. The relation $r_1 \setminus r_2$ requires an event $e_3$ with $e_3 \in r_1$ and $e_3 \in r_2$ and since $r_1$ and $r_2$ satisfy the path condition there is a path from $e_1$ to $e_2$ and from $e_3$ to $e_2$ and thus $r_1 \setminus r_2$ also satisfies the path condition. Similarly, $r_1^+$ adds a relation only where there already is a path and satisfies the path condition. Relation $r_1^*$ consists of $r_1^+$ and the identity relation, which also satisfies the path condition (there is always a path of length 0 from some $e_1$ to $e_1$). A named relation still satisfies the path condition after a Kleene iteration step.

Note that according to Lemma 4, a violating cycle implies a cycle of basic relations (called a basic path) that contains all events of the violating cycle. We can argue about consistency of an execution by examining the paths of basic relations.
Lemma 5. Given a relation $r$ of a common MCM and events $e_1, e_2$ of an execution, whether $r(e_1, e_2)$ holds is determined by the basic paths from $e_1$ to $e_2$.

Proof. We use a structural induction over the Kleene iteration.

Induction Basis: Any named relation is initially the empty relation and trivially satisfies the condition. Any basic relation trivially satisfies the condition.

Induction Step: Assume all current assignments of relations are determined by the basic paths between related events. Two events $e_1$ and $e_2$ are related by some relation $r$ iff the basic paths from $e_1$ to $e_2$ satisfy some property. A named relation is updated according to its defining equation using the current assignments of the named relations and basic relations. Let $r_1, r_2$ be either named or basic relations. Events $e_1, e_2$ are related by $r_1 \cup r_2$ (or $r_1 \cap r_2$) if the basic paths from $e_1$ to $e_2$ satisfy the property of $r_1$ or $r_2$ (resp. $r_1$ and $r_2$). Similar, events are related by $r_1 \setminus r_2$ if the basic paths satisfy the property of $r_1$ but not $r_2$. For $r_1 \cap \text{sloc}, r_1 \cap \text{std}, r_1 \cap \{p\} \times \{q\}$, two events $e_1, e_2$ are related if the property of $r_1$ is satisfied and $e_1$ and $e_2$ satisfy an additional condition. The condition for the events $e_1, e_2$ can be expressed as a conditional path relation for paths from $e_1$ to $e_2$.

The path $r_1; r_2$ relates $e_1$ to $e_2$ if there is an $e_3$ such that the basic paths from $e_1$ to $e_3$ ensure $r_1(e_1, e_3)$ and the basic paths from $e_3$ to $e_2$ ensure $r_2(e_3, e_2)$. It follows that $r_1; r_2(e_1, e_3)$ depends on the basic paths from $e_1$ to $e_2$ over some $e_3$. The relations $r_1'$ and $r_1''$ are derived using previously examined operators over relations and thus they satisfy the condition.

A named relation still satisfies the path condition after a Kleene iteration step.

Proof of Theorem 1.1. We show that $P_{\psi_0}$ is not portable iff $\psi_0$ has an unsatisfying assignment.

($\Rightarrow$): We assume an execution $X$ exists that is $M_T$-consistent but not $M_S$-consistent. According to Lemma 3, the execution is a violating cycle for $M_S$. We assume towards contradiction that no event of $P_{\psi_0}$ or $P_{\psi_0}^c$ is executed. The read $r \leftarrow y$ in $t^{\psi_0}_i$ has to read from the write in $t_1$ (in $P_{\psi_0}$) according to unipro (the execution is $M_T$-consistent). It occurs after $t_1$ in the program order. The read has incoming basic relations from events in $P_{\psi_0}$ but no outgoing relations to some event in $P_{\psi_0}$. Any read $r \leftarrow y$ in another thread can either read from the write in $P_{\psi_0}$ (it has an incoming $rf$ relation from $P_{\psi_0}$) or it reads the initial value which results in an outgoing-from-read relation to $P_{\psi_0}$. There are no other basic relations between a read $r \leftarrow y$ and some event in $P_{\psi_0}$. Any read $r \leftarrow y$ has either basic incoming relations from $P_{\psi_0}$ $(rf, po)$ or outgoing to $P_{\psi_0}^c (fr)$, not both.

It follows from Lemma 4, that no read $r \leftarrow y$ has both incoming and outgoing derived relations and so no read $r \leftarrow y$ is in a violating cycle. Any violating cycle is in $P_{\psi_0}$. Further, there is no basic path from some event $e_1$ in $P_{\psi_0}$ to one of the reads and back to some $e_2$ in $P_{\psi_0}$. The reads do not affect the basic paths between events in $P_{\psi_0}$. So the violating cycle for $M_S$ is still present if we remove the reads and thus restrict the execution to events of $P_{\psi_0}$. Since removing the reads does not affect the basic paths in $P_{\psi_0}$, no violating cycle for $M_T$ has been added. It follows that the execution of $P_{\psi_0}$ is consistent with $M_T$ but not $M_S$. This is a contradiction to $P_{\psi_0}$ being always portable for common MCMs (Lemma 2).

It follows that any violating cycle requires events from $P_{np}$ to be executed. Since an event of $P_{np}$ can only be executed if $y = 1$ was read in the thread (and thus written by $t_1$), the if-condition in Line 8 was not satisfied. It follows that there is an assignment that does not satisfy $\psi_0$.

($\Leftarrow$): We now assume that there is an assignment that doesn’t satisfy $\psi_0$. There is an SC-consistent execution $Z$ of $P_\psi$ that executes the write $y := r_{e_1}$. We extend this execution of $P_\psi$ to an execution $X$ of $P_{\psi_0}$. We ensure that all reads $r \leftarrow y$ read from $y := r_{e_1}$ and thus $P_{np}$ is executed. Let $Y$ be an execution of $P_{np}$ that is $M_T$-consistent but not $M_S$-consistent. This results in an execution $X \cup Y \cup Z$ of $P_{\psi_0}$ that contains the executions of $P_\psi$ and $P_{np}$, the read from relation for the reads of $y$ and the required program order additions. Since $P_{np}$ has no registers or locations in common with the rest of the program and occurs last in $po$, no basic relation of $X$ leaves $Y$. It follows that $X$ contains the same basic paths between events of $Y$ as $Y$. Thus the violating cycle for $M_S$ in $Y$ is also in $X$. It follows that $X$ is not $M_S$-consistent. We show that $X$ is still $M_T$-consistent. We assume towards contradiction that there is a violating cycle for $M_T$: As before, it holds that a violating cycle must contain an event from $Y$. Since $Y$ is never left, any basic cycle of $X$ with events in $Y$ must be contained entirely in $Y$. It follows from (Lemma 4) that a violating cycle for $M_{T}$ must be entirely in $Y$. This is a contradiction to the execution of $P_{np}$ being $M_T$-consistent.

Since $P_{\psi_0}$ is a polynomial-time reduction, portability is $\Pi_1^P$-hard.

B.1 $\Pi_2^P$-Completeness of State Portability

We introduce Lemma 6 and Theorem 3 in order to show that state portability is both in $\Pi_2^P$ and is $\Pi_2^P$-hard. It follows, that state portability is $\Pi_2^P$-complete for common MCMs and thus Theorem 2.2 is correct.

Lemma 6. State-based portability is in $\Pi_2$ for all MCMs.

Proof. We encode the state portability property in a closed formula (i.e. all variables are quantified) of the form $\forall \exists \psi$. We have already shown how to encode consistency of an execution $X$ with an MCM $M$ as a formula $(X \in \text{consistent}_M(P))$ in Section 4. Again we encode numbers as sequences of Boolean variables. Let $\text{val}(e)$ be the value that is read/written by a read or write event $e$ and $\text{loc}(e)$ the location it accesses. We can encode the property $\text{state}(X) = \sigma$ in a Boolean formula as follows. If a write has no outgoing $co$ relation, then it must have the same value as the location in the state:

$$\bigwedge_{w \in \text{W}} \left( \bigwedge_{w' \in \text{W}} \neg(w \xrightarrow{co} w') \Rightarrow \text{val}(w) = \sigma(\text{loc}(w)) \right).$$

In a similar way we can ensure that the last operation in the program order on a register $r$ has the value $\sigma(r)$. This means we can construct Boolean formulas for properties of the form $X \in \text{consistent}_M(P)$ and $\text{state}(X) = \sigma$. With this, we can construct the following formula:

$$\forall X \exists Y : (X \in \text{consistent}_{M_T}(P)) \Rightarrow (Y \in \text{consistent}_{M_S}(P) \land \text{state}(X) = \text{state}(Y)).$$

This is equivalent to state portability (see Definition 2). The state portability problem from $M_S$ to $M_T$ can be expressed as
a closed quantified formula of the form \( \forall \exists \psi \) and thus state based portability is in \( \Pi_2^P \).

We now introduce the program \( P_\psi \) and examine its behavior. We will then use \( P_\psi \) in order to prove \( \Pi_2^P \)-hardness.

Let \( \psi(x_1 \ldots x_m) \) be a be a Boolean formula over variables \( x_1 \ldots x_m \). The concurrent program \( P_\psi \) := \( t_1 \parallel t_2 \) with two threads \( t_1 \) and \( t_2 \) is defined below. The program is similar to the program in the previous section. It contains additional synchronization in order to ensure that the formula assignment and the computed state match. The program either computes a satisfying assignment of \( \psi \) (\( y = 1 \)), an unsatisfying assignment (\( y = 0 \)) or it ends with an error (\( y = 2 \)).

We use the value 0 to encode the Boolean value false. To avoid confusion, we assume that the variables are initialized with some other unused value, e.g. 3. This does not interfere with the validity of the proofs since our program only assigns constants and thus 0 and 3 are interchangeable.

We will see that it is sufficient to examine the program under SC (the strongest common MCMC) where an execution is an interleaving of the two thread executions. The threads first create some variable assignment \( A \); thread \( t_1 \) assigns 0 to the variables and \( t_2 \) assigns 1. The assignment \( A \) is determined by the interleaving of those writes. If the write \( x_i \) := 0 of \( t_1 \) is followed by \( x_i \) := 1 of \( t_2 \) (\( w_{i,0} \rightleftarrows w_{i,1} \)), then \( x_i \) is set to 1. Then \( t_2 \) ensures that \( t_2 \) has executed all its writes (so that the assignment doesn’t change anymore) by using the synchronization variables \( x'_1 \ldots x'_m \) to check that \( t_1 \) and \( t_2 \) reads the same assignment. If that is not the case, then some writes of \( t_2 \) have not occurred yet and \( y \) is set to 2 in Line 6. If all the writes from \( t_2 \) have occurred, \( t_1 \) checks whether \( A \) satisfies some \( \psi \) (\( y \) is set to 1) or not (\( y \) is set to 0).

We can use our simpler notion of states without registers on the input program with the added instructions to solve the original state portability problem with registers.

An assignment \( A \) of a set of Boolean variables \( V \) is a function \( A \subset V \times \{0, 1\} \) that assigns either 0 or 1 to a variable.

**Definition 4.** Given an Assignment \( A \) of \( x_1 \ldots x_m \), locations \( y_1 \ldots y_m \) and a number \( a \in \mathbb{N} \), let \( \sigma[A; y_1 \ldots y_n \leftarrow a] \) denote the state with
\[
\begin{align*}
\sigma[A; y_1 \ldots y_n \leftarrow a](x_i) &= A(x_i) \text{ for } i \leq n \\
\sigma[A; y_1 \ldots y_n \leftarrow a](y_j) &= a \text{ for } j \leq m, \\
\sigma[A; y_1 \ldots y_n \leftarrow a](z) &= v \text{ for } z \text{ any other location and } v \text{ the initial value (we use 3)}.
\end{align*}
\]

We lift the definition accordingly to
\[
\sigma[A; y_1 \ldots y_n \leftarrow a; z_1 \ldots z_l \leftarrow b].
\]

The program \( P_\psi \) can compute some assignment \( A \) with \( y = 1 \) or \( y = 0 \) depending on whether \( A \) satisfies \( \psi \).

The following lemmas show that \( P_\psi \) behaves similar for all common MCMCs.

**Lemma 7.** Let \( M \) contain uniproc and \( A \) be an assignment of \( \psi \). It holds \( A \models \psi \) (resp. \( A \not\models \psi \)) only if
\[
\exists X \in \text{consistent}_M(P_\psi) : \text{state}(X) = \sigma[A; y \leftarrow 1] \quad \text{(resp. state}(X) = \sigma[A; y \leftarrow 0])\).
\]

**Proof.** We show that any \( M \)-consistent execution with \( y = 0 \) or \( y = 1 \) computes a desired state. Let \( X \) be an execution that satisfies uniproc and computes some \( \sigma \) with \( \sigma(y) = 1 \) (\( \sigma(y) = 0 \) is analogue). Since \( y = 1 \) is executed in \( t_1 \), it follows that \( \psi \) is satisfied by the values of \( x_1 \ldots x_m \) read by \( r_1 \ldots r_m \) in Line 4 and also \( r'_1 \ldots r'_m \) in Line 3 read the same values. We call this assignment \( A \). Since the reads of Line 5 of \( t_2 \) occur after the writes \( \bar{w}_1 \ldots \bar{w}_m \) they are ordered last in \( co \) according to uniproc. The execution computes 3 for \( x'_1 \ldots x'_m \).

It remains to show that the writes accessed by \( r_1 \ldots r_m \) are indeed computed by \( X \), meaning they are ordered last in \( co \). Towards contradiction, we assume this is not the case. Then, there is a write \( w_{i,0} \) or \( w_{i,1} \) that is accessed by a read but its value is not computed (it is not last in \( co \)).

**Case 1:** Assume that this write is \( w_{i,1} \). The write is accessed by a read \( (w_{i,1} \rightleftarrows r_i) \) and it is not last in the coherence order \( (w_{i,1} \rightleftarrows w_{i,0}) \). According to \( fr := tf^{-1} \circ co \), it holds \( r_i \leftarrow w_{i,0} \). Since \( w_{i,0} \) occurs before \( r_i \) in \( t_1 \) and they access the same location, they are related w.r.t \( po \) and \( sloc \) and thus \( w_{i,0} \xrightarrow{po\circ sloc} r_i \xrightarrow{fr} w_{i,0} \). This cycle is a contradiction to \( X \) satisfying uniproc which is property (iii) of common MCMCs.

**Case 2:** Assume that there is a write \( w_{i,0} \) that is read \( (w_{i,0} \rightleftarrows r_i) \) and its value is not computed \( (w_{i,0} \not\rightarrow w_{i,1}) \). It follows that \( r_i \) reads the value 0. Since \( y = 1 \) is computed, the condition in line 8 is not satisfied and since \( val(r_i) = val(r'_i) \), we know that \( r'_i \) also reads 0. This means \( r'_i \) reads not the initial value 3 so it must read from \( \bar{w}_i \). For the write \( \bar{w}_i \) that \( r'_i \) reads from \( (\bar{w}_i \rightleftarrows r'_i) \) follows that \( val(\bar{w}_i) = val(r'_i) = 0 \). According to the data-flow, \( \bar{w}_i \) writes the value that was obtained by the previous read \( r_i \) which must be 0.

To simplify our study we will define a state only over its locations, not the registers. This does not change the complexity of the state portability problem: We could simply add instructions to our input programs that write all registers to locations in the end.
This means, that under $y$ is to construct a program that uses $P$ from Lemma 7.

Then $t_2$ reads those values and writes them to $x'_1 \ldots x'_n$. Now $t_1$ reads $x'_1 \ldots x'_n$, and the if-condition in line 5 is not satisfied, we go in the else-branch. So $t_1$ sets $y$ to 0 or 1 depending on whether $A \models \psi$ and $t_2$ sets $x'_1 \ldots x'_n$ back to the initial value 3. It follows that for every assignment $A$, there is a SC-consistent execution $X$ that computes $\sigma[A; y \leftarrow 0]$ or $\sigma[A; y \leftarrow 1]$ depending on whether $A$ satisfies $\psi$. Since any SC-consistent execution is consistent with all common MCMs, we can compute the desired state for any assignment of $\psi$. The other direction follows directly from Lemma 7.

In order to show $\Pi^2_5$-hardness, we reduce validity of a closed formula $\forall x_1 \ldots x_n \exists y_1 \ldots y_m : \psi$ to state portability. The idea is to construct a program that uses $P_\psi$ in order to check if some assignment satisfies $\psi$ and then overwrite $y_1 \ldots y_m$ with 1 so that the assignment of $y_1 \ldots y_m$ is not given by the computed state. If $\psi$ was not satisfied, the non-portable component $P_{op}$ is executed. If the execution of $P_{op}$ is $M_\mathcal{T}$-consistent but not $M_\mathcal{S}$-consistent, then it pretends that the formula was satisfied by setting all $y$ to 1. This means, that under $M_\mathcal{T}$, any assignment of $x_1 \ldots x_n$ with $y = 1$ can be computed. It follows that the program is portable if any assignment of $x_1 \ldots x_n$ with $y = 1$ can be computed under $M_\mathcal{S}$. Under $M_\mathcal{S}$ however, pretending is not possible. Here $P_\psi$ can only be set to 1 by $P_\psi$. So under $M_\mathcal{S}$, an assignment of $x_1 \ldots x_n$ and $y = 1$ can only be computed if there is some assignment of $y_1 \ldots y_m$ so that $\psi$ is satisfied. The program is portable if

$$\forall x_1 \ldots x_n \exists y_1 \ldots y_m : \psi$$

holds.

We want a simple non-portable program that always computes the initial state except under $M_\mathcal{T}$, where it can set a location $z$ to 1. We use a program $P_{op} = t'_1 \parallel \cdots \parallel t'_k$ with the following properties: Any execution consistent with $M_\mathcal{S}$ computes 3 for all its locations and contains no write that sets $z$ to 1. The $M_\mathcal{T}$-consistent executions compute either $z = 1$ or $z = 3$ and for all other locations. The program contains only one write to $z$ which is in $t'_1$.

We assume the state portability problem from $M_\mathcal{S}$ to $M_\mathcal{T}$ is not trivial and a program exists that has an $M_\mathcal{T}$-consistent execution such that no $M_\mathcal{S}$-consistent execution computes the same state $\sigma$. We can assume that the program only assigns constant values and has no write on $z$.

Similarly to the synchronization of $P_\psi$, we can add a mechanism at the end of all threads that does the following: all threads check if they read $\sigma$; if so, they communicate that to $t_1$ which sets $z$ accordingly and then all threads set all other locations back to 3. It follows that a program $P_{op}$ with the required properties exists.

Given a formula $\forall x_1 \ldots x_n \exists y_1 \ldots y_m : \psi$, we use $P_\psi = t_1 \parallel t_2$ and $P_{op} = t'_1 \parallel \cdots \parallel t'_k$ to construct a program $P_o := t'_1 \parallel \cdots \parallel t'_k$. We define the threads below.

Let $t_i := \text{skip}$ for $i \leq 3 \leq k$ ($t_1$ and $t_2$ are from $P_\psi$). The threads are defined for $2 \leq i \leq k$ as

$$t'_i := t_i; \quad y \leftarrow y; \quad \text{if } (y = 0) \text{ then } t'_i.$$  

In general terms, $P_o$ does the following: First, it executes $P_\psi$. If the state computed by $P_\psi$ did not satisfy $\psi(y = 0)$, then it executes $P_{op}$, which is not portable. If the execution of $P_{op}$ is $M_\mathcal{T}$-consistent but not $M_\mathcal{S}$-consistent ($z = 1$), then $P_\psi$ pretends that the formula was satisfied by setting $y$ to 1. Afterward, $y_1 \ldots y_m$ are set to 1, so that their former assignment checked in $P_\psi$ is no longer given by the computed state.

Lemma 9. For every assignment $A$ of $x_1 \ldots x_n$ holds

$$\exists X \in \text{consistent}_{M_\mathcal{T}}(P_\psi) : \text{state}(X) = \sigma[A; y_1 \ldots y_m \leftarrow 1].$$

Proof. According to Lemma 8, the following holds: For every assignment $A$ of $x_1 \ldots x_n$ and $A'$ of $y_1 \ldots y_m$, there is an $M_\mathcal{T}$-consistent execution $X$ of $P_\psi$ such that either $\text{state}(X) = \sigma[A \cup A'; y \leftarrow 1]$ or $\text{state}(X) = \sigma[A \cup A'; y \leftarrow 0]$. We examine both cases:

**Case 1:** If $\text{state}(X) = \sigma[A \cup A'; y \leftarrow 1]$, then $A \cup A' \models \psi$ and according to Lemma 8 there is an SC-consistent execution of $P_\psi$ that computes $\sigma[A \cup A'; y \leftarrow 1]$. We can easily extend this to an SC-consistent execution $X'$ (represented by an interleaving) of $P_o$. After $P_\psi$ is executed, we read 1 with reads $r_y \leftarrow y$ of all threads. So the subsequence if conditions are not satisfied. Then we execute the writes in Line 9. So $y := r_{c,0} \ldots y_m := r_{c,1}$ are the only writes executed outside of $P_\psi$. These writes are ordered after the assignments of 0 to $y_1 \ldots y_m$ in co and thus $\text{state}(X') = 1$ for $i \leq m$. Since there are no further executed writes outside of $P_\psi$, the computed state otherwise coincides with $\sigma[A \cup A'; y \leftarrow 1]$ computed by $X$. The extension of $X$ computes $\sigma[A; y_1 \ldots y_m \leftarrow 1]$.

**Case 2:** If $\text{state}(X) = \sigma[A \cup A'; y \leftarrow 0]$, then write $y := r_{c,0}$ is executed in $t_1$. We construct an execution $X' \supseteq X$ of $P_o$ in the following way: We ensure all reads $r_y \leftarrow y$ of threads $t'_i$ with $i \leq k$ read from $y := r_{c,0}$ and thus all threads $t'_i$ are executed. This means $P_{op}$ is executed. According to the definition of $P_{op}$, there is
an $\mathcal{M}_T$-consistent execution $X$ of $P_{np}$ such that $\text{state}(Y)(z) = 1$ is written by some write $w_t$ in $t_1$ and $Y$ computes the initial value for all other locations of $P_{np}$. We enforce $X' \supseteq Y$ and ensure the read $r_z \leftarrow z$ reads from the write in $P_{np}$ ($z = 1$) and thus the following if condition is satisfied and the writes $z := 3$ and $y := 1$ are executed. We order them last in $co$. It follows that $X'$ computes $\sigma[A; y, y_1 \ldots y_m \leftarrow 1]$

We show that $X'$ is still $\mathcal{M}_T$-consistent. We partition the events of $X'$ into sets $E_1$, $E_2$ and $E_3$ and show that there is no violating cycle of $\mathcal{M}_T$ inside or between these sets.

Set $E_1$ consists of events in $X$ and the subsequent reads $r_y \leftarrow y$ of all threads. Set $E_2$ consists of the events of $Y$. Set $E_3$ contains the events $r_z \leftarrow z$ in $t_1$ and the following writes $z := r_c,3, y := r_c,1$ and $y_1 := r_c,1, \ldots y_m := r_c,1$.

The following two conditions hold: (i) $E_2$ is only left by basic relations leading to $E_3$. This is the case since $E_1$ precedes $E_2$ in the program order and has no common registers. (ii) There are no basic relations leading from $E_3$ to some other set: The events in $E_3$ are ordered last in the program order and the writes are ordered last in the coherence order. The read $r_z \leftarrow z$ has no outgoing fr relation, since there is only one write to $z$.

From (i) and (ii) follows that $X'$ contains no basic cycle that contains events from more than one of the sets. According to the path property (Lemma 4) exists no violating cycle for $\mathcal{M}_T$ in $X'$ that contains events from more than one of the sets. So any violating cycle must be contained entirely in one of the sets.

From (i) and (ii) follows (iii): If two events $e_1, e_2$ are in the same set, then there is no basic path from $e_1$ to $e_2$ that leaves the set.

We consider $E_1$: There are read from relations from $y := r_c,0$ in $t_1$ to all reads $r_y \leftarrow y$ in $t'_i$ with $i \leq k$. However, there are no outgoing basic relations from any of those reads to other events in $E_1$. It follows from (iii) that there is no basic path from a read $y := r_c,0$ to another event in $E_1$ and according to the path property, there is no relation from a read $y := r_c,0$ to some other element in $E_1$. It follows that a read $r_y \leftarrow y$ cannot occur in a violating cycle. From (iii) follows that the basic paths in $X'$ between events of $E_1$ (and thus the violating cycles for $\mathcal{M}_T$) are the same as in $X$. Since $X$ is $\mathcal{M}_T$-consistent, it has no violating cycle for $\mathcal{M}_T$ and thus $X'$ has no violating cycle for $\mathcal{M}_T$ in $E_1$.

We consider $E_2$: From (iii) follows that the basic paths in $X'$ between events in $E_2$ are the same as in $Y$. According to Lemma 5, $X'$ has a violating cycle for $\mathcal{M}_T$ in $E_2$ iff $Y$ has a violating cycle for $\mathcal{M}_T$. The execution $Y$ is defined to be $\mathcal{M}_T$-consistent so there is no violating cycle in $E_2$.

We consider $E_3$: The set contains no basic cycle and no basic relation leaves $E_3$ according to (ii). It follows that there is no basic cycle that contains events of $E_3$. According to the path property, there is no violating cycle in $E_3$. It follows that $X'$ is $\mathcal{M}_T$-consistent.

$\square$

**Lemma 10.** There is an $\mathcal{M}_S$-consistent execution of $P_s$ that computes $\sigma[A; y, y_1 \ldots y_m \leftarrow 1]$ with some assignment $A$ of $x_1 \ldots x_n$ iff there is an assignment $A'$ of $y_1 \ldots y_m$ such that $A \cup A' \models \psi$.

**Proof.** If a program $P'$ is contained in a program $P$, we can restrict an execution $X$ of $P$ to an execution of $P'$. We simply remove all events and relation that are not in $P'$. We denote $X$ restricted to $P'$ as $X[P']$.

$(\Rightarrow)$: Let $X$ be an $\mathcal{M}_S$-consistent execution of $P_s$ that computes $\text{state}(X)(y) = 1$. We assume towards contradiction that $X$ contains a write that sets $z$ to 1. According to the definition of $P_{np}$ the execution $X[P_{np}]$ contains a violating cycle for $\mathcal{M}_S$. As with the proof of Lemma 9, we can show that no basic path in $X$ contains events of $X[P_{np}]$ leaves $X[P_{np}]$. It follows that the basic paths between events in $X[P_{np}]$ are the same in $X$ and $X[P_{np}]$ and thus $X$ also contains the violating cycle for $\mathcal{M}_S$. This is a contradiction to $X$ being $\mathcal{M}_S$-consistent.

It follows that $X$ cannot read $z = 1$ in Line 5 and the write in Line 8 is not executed. So $y := r_{c,1}$ must have been executed in $P_0$ for $X$ to compute $y = 1$.

The execution $X$ satisfies uniproc since $\mathcal{M}_S$ is common. It has no basic cycle that violates uniproc. It follows that $X[P_{np}]$ has no such basic cycle either and thus satisfies uniproc. It follows from Lemma 7, that $X[P_s]$ computes a satisfying assignment of $y_1 \ldots y_m$. Since nothing is written to $x_1 \ldots x_n$ outside of $P_0$, $X$ computes the same values for $x_1 \ldots x_n$ as $X[P_s]$. A write $y_i := r_{c,0}$, $r_{c,1}$ is related to write $y_i := r_{c,1}$ in line 9 of $t'_i$ in $\text{po} \land \text{sloc}$ and according to uniproc also in $co$. This implies that $X$ computes 1 for $y_1 \ldots y_m$. For the synchronization variables of $P_0$, uniproc ensures that $X$ computes the initial values.

It follows, that if there is an $\mathcal{M}_S$-consistent execution that computes $\sigma[A; y, y_1 \ldots y_m \leftarrow 1]$ for some assignment $A$ of $x_1 \ldots x_n$, then there is an assignment $A'$ of $y_1 \ldots y_m$ such that $A \cup A' \models \psi$.

$(\Leftarrow)$: According to Lemma 8, for each assignment $A$ of $x_1 \ldots x_n$ and $A'$ of $y_1 \ldots y_m$ where $A \cup A'$ satisfies $\psi$, there is an SC-consistent execution $X$ of $P_s$ such that $\text{state}(X) = \sigma[A \cup A'; y \leftarrow 1]$. This SC execution of $P_s$ can be represented as an interleaving of the local executions. To this interleaving, we append the subsequent reads $r_y \leftarrow y$. They read the value 1 and the next if conditions are not satisfied. Then we append the remaining writes in line 9. The resulting interleaving represents an SC-consistent execution $X'$ of $P_s$.

The only writes not in $P_0$ that are executed by $X'$ are in Line 9 (they are last in the program order and thus also last in $\text{po} \land \text{sloc}$) and according to uniproc, they must be ordered after any write $y := 0$ in $co$. It follows that $\text{state}(X') = \sigma[A; y, y_1 \ldots y_m \leftarrow 1]$.

**Theorem 3.** State-based portability is $\Pi_2$-hard for common MCMs.

**Proof.** Given common MCMs $\mathcal{M}_S$ and $\mathcal{M}_T$, we argue that $P_s$ is a reduction of validity of a formula $\forall x_1 \ldots x_n \exists y_1 \ldots y_m : \psi(x_1 \ldots x_n)$ to state portability from $\mathcal{M}_S$ to $\mathcal{M}_T$.

Note that any execution of $P_s$ satisfying uniproc computes 1 for $y_1 \ldots y_m$ and the initial values for the synchronization variables. We show that the following properties (i)-(iii) hold:

(i) Any state with $y = 2$ that is computable with an $\mathcal{M}_T$-consistent execution of $P_s$ can be computed by an $\mathcal{M}_S$-consistent computation of $P_s$. It is easy to see that any state with $y = 2$ that can be computed by an execution satisfying uniproc, can be computed by an SC-consistent execution.

(ii) Any state with $y = 0$ that is computable with an $\mathcal{M}_T$-consistent execution $X$ of $P_s$ can be computed by an $\mathcal{M}_S$-consistent computation. No event of $P_{np}$ is executed, the initial value is computed for locations of $P_{np}$ according to uniproc, any such state has the form $\sigma[A; y \leftarrow 0; y_1 \ldots y_m \leftarrow 1]$ for some assignment $A$ of $x_1 \ldots x_n$. It is easy to see that $A$ and $y = 0$ is
computed by $X[P_e]$ and $X[P_v]$ satisfies $\text{uniproc}$. According to Lemma 7, there is an assignment $A'$ of $y_1 \ldots y_m$ such that $A \cup A' \not= \psi$. It follows from Lemma 8 that there is an SC-consistent execution of $P_v$ that computes $\sigma[A \cup A'; y \leftarrow 0]$. This can be easily extended to an SC-consistent execution of $P_e$ that computes $\sigma[A; y \leftarrow 0; y_1 \ldots y_m \leftarrow 1]$. Since $M_S$ is common, $\sigma[A; y \leftarrow 0; y_1 \ldots y_m \leftarrow 1]$ is also computable under $M_S$.

(iii) Any state with $y = 1$ computed by an $M_T$-consistent execution $X$ has the form $\sigma[A; y_1 \ldots y_m \leftarrow 1]$ for some assignment $A$ of $x_1 \ldots x_n$. This holds since $X[P_{np}]$ is also $M_T$-consistent (the basic paths between its events are the same as in $X$) and according to the definition of $P_{np}$, it computes the initial value for all its locations except maybe 1 for $z$. If that is the case, then the write $z \leftarrow r_{c,3}$ in $t_1$ is executed and according to $\text{uniproc}$ ordered later in $co$.

From (i)-(iii) follows that $P_s$ is state portable from $M_S$ to $M_T$ iff any $M_T$-computable state $\sigma[A; y_1 \ldots y_m \leftarrow 1]$ can be computed under $M_S$. Lemma 9 implies that $P_s$ is state portable from $M_S$ to $M_T$ iff any state $\sigma[A; y_1 \ldots y_m \leftarrow 1]$ can be computed under $M_S$. According to Lemma 10, this is the case iff for every assignment $A$ of $x_1 \ldots x_n$ there is an assignment $A'$ of $y_1 \ldots y_m$ with $A \cup A' \models \psi$. This is the case iff $\forall x_1 \ldots x_n \exists y_1 \ldots y_m : \psi(x_1 \ldots y_m)$ is valid. \qed