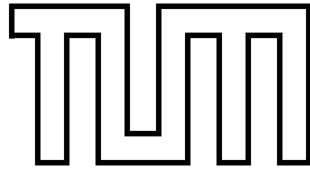


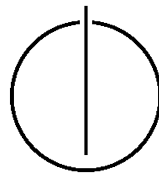
FAKULTÄT FÜR INFORMATIK

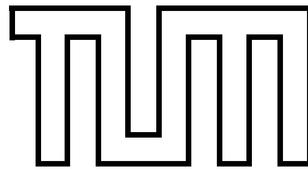
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Efficient Verification of Multi-Threaded Programs

Andreas Johannes Wilhelm





FAKULTÄT FÜR INFORMATIK

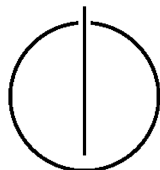
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Efficient Verification of Multi-Threaded Programs

Effiziente Verifikation von Programmen mit mehreren Threads

Author:	Andreas Johannes Wilhelm
Supervisor:	Prof. Dr. Andrey Rybalchenko
Advisor:	Dr. Corneliu Popeea
Advisor:	Dr. Tobias Schüle
Submission Date:	November 29, 2013



I assure the single handed composition of this master's thesis only supported by declared resources.

Munich, February 6, 2014 Andreas Johannes Wilhelm

Abstract

Given its pervasiveness, it is unfortunate that writing multi-threaded software is an intricate and tedious task due to a multitude of thread interactions. Model checking approaches that are based on formal specifications can significantly improve programmers productivity by allowing systematically and exhaustively exploring program behaviours and checking their correctness. However, efficient model checking is inhibited by the so-called state explosion problem. In this thesis, we propose two promising approaches to cope with this problem by reducing the number of program states that have to be explored - transaction summarization and may-happen-in-parallel information. We show that especially a priori identified transactions can significantly improve the efficiency of compositional verification. Our experimental evaluation indicate that the approaches compare favourably with state-of-the-art verifiers and can lead to two orders of magnitude reduction of verification time.

Contents

Abstract vii

Outline of the Thesis xi

I. Introduction and Background Theory 1

1. Introduction 2

2. Foundations 5
- 2.1. Model Checking . . . 5
- 2.2. Logical Foundations . . . 5
  - 2.2.1. Linear Arithmetic . . . 6
  - 2.2.2. Horn-like Clauses . . . 7
- 2.3. Solving Horn-like Clauses . . . 8
- 2.4. Multi-threaded Programs . . . 10
- 2.5. Proof Rules for Verification of Multi-threaded Programs . . . 13
  - 2.5.1. Monolithic Proof Rule . . . 13
  - 2.5.2. Owicki-Gries Proof Rule . . . 13
  - 2.5.3. Rely-Guarantee Proof Rule . . . 13

II. Verification Methods 15

3. Model Checking with Transaction Summarization 16
- 3.1. Lipton's Theory of Reduction . . . 16
- 3.2. Illustration . . . 18
- 3.3. Transaction Inference . . . 25
  - 3.3.1. Locks-Held Information . . . 25
  - 3.3.2. Mover Information . . . 27
  - 3.3.3. Phase Information . . . 29
  - 3.3.4. Transaction Boundaries . . . 30
- 3.4. Proof Rule . . . 30
- 3.5. Soundness Proof (Sketch) . . . 32

4. Model Checking with May-Happen-in-Parallel Information 34
- 4.1. Illustration . . . 34
- 4.2. Dataflow Analysis for MHP . . . 35
  - 4.2.1. Parallel Execution Graph . . . 35

4.2.2. Analysis 36
4.3. Proof Rules 40
4.3.1. Monolithic Proof Rule (MHP) 40
4.3.2. Owicki-Gries Proof Rule (MHP) 40

III. Results and Conclusion **43**

5. Experimental Results **44**

6. Summary and Conclusion **49**

Bibliography **51**

Outline

Part I: Introduction and Background Theory

CHAPTER 1: INTRODUCTION

The introduction emphasizes the importance of software verification for multi-threaded programs and refers to the used state-space reduction.

CHAPTER 2: VERIFICATION OF MULTI-THREADED SOFTWARE

We present the necessary foundations of logical reasoning, the solving method for arising constraints, and the respective representation of multi-threaded software.

Part II: Verification Methods

CHAPTER 3: TRANSACTION SUMMARIZATION

We describe our verification approach based on transaction summarization. Besides an illustration of the overall process, each step is formulated using logical reasoning.

CHAPTER 4: MAY-HAPPEN-IN-PARALLEL INFORMATION

This chapter introduces MHP information, our second approach, by giving an illustration, an effective algorithm to compute this information, and corresponding proof rules.

Part III: Results and Conclusion

CHAPTER 5: EXPERIMENTS

We show experimental results for both approaches when applied on multi-threaded software.

CHAPTER 6: CONCLUSION

Finally, the last chapter concludes with the main contributions and possible future work.

Part I.

Introduction and Background Theory

1. Introduction

The advent of multi-core architectures brought parallelism into nearly every system that affects us; from small gadgets like smartphones to huge interconnected power plants. Since multiple cores require concurrent software to exploit their full potential, we increasingly rely on applications that are subtle and error prone. Such software consists of several threads that execute several tasks in parallel and interact with each other during operation. Given its pervasiveness, it is unfortunate that writing multi-threaded software is notoriously difficult to write and to debug as programmers need to keep track of all possible thread interactions. Sophisticated tools and methods are needed to ensure correctness and to support the programmers on this challenge.

In practice, the most common used techniques for correctness checking are based on testing and peer reviews [4]. Although a notable fraction of errors may be detected when applied to sequential programs, this might not be the case for multi-threaded software. Concurrency brings an additional dimension of complexity (e.g., by interference, synchronization, or communication) that makes both techniques unpractical. For example, interference causes an exponential growth of potential execution paths that exceeds the ability of available testing methods. The resulting challenge belongs to the most severe in computer science: to provide proper formalisms, techniques, and tools that enable correct multi-threaded programs despite their complexity.

Model checking approaches, which are based on formal specifications, allow systematically and exhaustively exploring program behaviours and checking their correctness. As opposed to theorem proving, the other class of formal verification methods, model checking is easily applicable and returns meaningful counterexamples when encountering erroneous program states. Unfortunately, efficient model checking is inhibited by the so-called *state-explosion problem*, especially when applied to multi-threaded software: threads introduce a combinatorial explosion of possible interleavings between threads that may lead to a vast number of distinct program states. However, there are promising approaches to cope with the state-explosion problem by reducing these multitudes of states. Two of them are partial order reduction and may-happen-in-parallel information.

Partial Order Reduction (POR) While one can easily construct a contrived program in which every interleaving leads to a different outcome, different interleavings often produce equal outcome and, hence, can be considered equivalent. Figure 1.1 shows two execution paths ($\alpha - \beta$ and $\beta - \alpha$) that both start from the program state 1 and lead to the same program state 4. Such an equivalence between interleavings suggests that only representatives of each equivalence class need to be considered when verifying a multi-threaded program. One way to exploit equivalent paths is called partial order reduction (POR) [15]. This technique is used in combination with model checking and amounts to restricting the successor computation to representative interleavings, which can be performed on-the-fly during the exploration of the model. Explicit-state [33, 19, 16] as well as symbolic [2, 20]

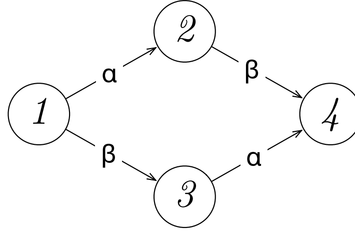


Figure 1.1.: Equivalent interleavings (α, β) and (β, α).

model checking algorithms can be effectively combined with POR. Furthermore, recent work shows that POR can also boost interpolation based verification [34], which makes it well-suited for the verification of programs with infinite-state spaces.

One way to implement partial order reduction is by using summary transformations. Such transformations summarize and replace certain sequences of statements within occurring threads by their composition into so-called *transactions*. The reduction views a transaction as a sequence of transitions $a_1, \ldots a_n, [c], b_1, \ldots, b_m$ where each a_i is a right mover, c is an optional committing action, and each b_i is a left mover. A right (left) mover is a transition that commutes to the right (left) regarding every transition of another thread. Executing such transactions atomically, i.e., without any preemption, produces representative interleavings due to the commuting behaviour [22]. Transactions can simplify deductive verification of multi-threaded programs using proof assistants, see e.g., [11, 10]. For finite state systems, transactions can be effectively identified and created on-the-fly during model checking [13, 3, 30]. Unfortunately, this does not hold for automatic verification tools for multi-threaded programs with infinite-state spaces.

May-Happen-in-Parallel Information (MHP) A second opportunity to tackle the state-explosion problem (which is also based on reduction) is the usage of may-happen-in-parallel information. As opposed to transactional reasoning, transition sequences are not reduced by exploring representative interleavings but because of unreachable states regarding included control locations. For each pair of threads in a program, only those locations are considered that may actually happen in parallel to each other. Such MHP information may incorporate several semantic aspects of multi-threaded programs like dynamic thread creation, locks, or signal handling. For example, two program locations from different threads may never happen in parallel when both are protected by the same lock. MHP information is highly valuable for techniques like program optimization, debugging, program understanding tools, and detecting synchronization anomalies as race conditions or deadlock situations, see e.g., [24].

Most recent methods to compute MHP information mainly rely on data flow algorithms that are based on parallel execution graphs (PEGs), i.e., combined control flow graphs (CFGs) for every program thread [25, 21, 5].

Contributions In this thesis, we explore the applicability of both transactional reasoning and MHP information to the verification of multi-threaded programs with infinite-state spaces. These techniques are incorporated in a constraint-based setting and solved by

an algorithm in the existing HSF tool [17]. HSF is based on symbolic reasoning and predicate abstraction following the counterexample-guided abstraction refinement scheme (CEGAR) [7]. The algorithm is designed for synthesizing software verifiers from proof rules to overcome the burden of developing every new verification tool from ground up in a complex manual effort. What remains by using HSF is a two-folded process. Firstly, the formulation of suitable verification methods for desired properties like safety or termination. This is a creative activity that usually leverages existing methods and adapts them to new application domains. Secondly, the creation of proper interpreters that transform a given system representation (e.g., formal programming languages) into Horn-like clauses. HSF allows us to elegantly use a declarative formulation of our verification approaches that can efficiently deal with thread interleaving.

Technically, we make the following contributions:

1. a Horn constraint-based method for identifying commutativity (mover annotations) of program statements (see Section 3.3);

2. a proof rule that composes transition sequences to transaction summaries, uses them for the verification of safety properties, and utilize compositional reasoning outside transactions based on the Owicki-Gries [28] proof rule (see Section 3.4);

3. two proof rules for the verification of safety properties that combine may-happen-in-parallel information with monolithic reasoning and the Owicki-Gries proof rule (see Section 4.3);

4. a demonstration of the proof rules by application to several example programs utilizing HSF among a comparison to state-of-the-art verifiers (see Chapter 5).

In summary, this thesis aims to show that a priori identified transactions can significantly improve the efficiency of compositional verification of multi-threaded programs, without requiring deep and intricate modifications of the underlying solving techniques. Since transactions rarely contain all statements of a thread, i.e., there are multiple transactions in each thread as well as some statements do no belong to any transaction, we integrate compositional reasoning into our exploration as a complementary technique for avoiding the explicit exploration of all interleavings. That is, our method relies on transaction whenever possible, while statements outside of transactions are subject to compositional reasoning, along with the transactions them self.

Our preliminary experimental evaluation shows that the conceptual separation of concerns, i.e., treatment of equivalence between interleavings via transactions and keeping track of interleavings using compositional proof system, compares favourably with state-of-the-art approaches and can lead to two orders of magnitude reduction of verification time on selected benchmarks.

2. Foundations

In this chapter, we introduce model checking, a promising approach for formal software verification. We then define the logical foundations for our methods followed by an illustration of the used tool to solve generated constraints. The distinguishing feature of this solving process is the ability to compute infinite state spaces and unbounded data ranges by means of a counterexample-guided (predicate) abstraction refinement. Finally, we present the necessary representation of multi-threaded software and three proof rules for the verification of safety properties.

2.1. Model Checking

Model checking is an automated verification technique that is based on system models describing possible system behaviour. Since the systems itself may be too complex to be verified, such models abstract from irrelevant aspects. They can be derived either manually or automatically from system representations in a programming language (e.g., C or Java) or a hardware description language (e.g., VHDL or Verilog). A model checker (tool) checks whether the model satisfies some desired properties (e.g., termination, invariants, etc.) from the system specification by exhaustively iterating through relevant system states. If a system state violates a desired property, the model checker generates a counterexample (execution path from an initial state to the violating state) which can be used to reproduce the error.

In this thesis, we focus on verification of the safety property, i.e., checking whether an error state is reachable from an initial state. Unfortunately, the set of reachable states is not generally computable since the state space may grow infinitely. Cases in which this state-space explosion problem occur limit the applicability of enumerative techniques like explicit-state model checking [6]. However, there have been some efforts made to overcome the burden of exhaustively check every single state. Symbolic model checking [20] represents sets of states by predicates that can be solved using symbolic fixpoint computations. Abstraction interpretation [8] tries to abstract only relevant properties for the verification task. Finally, partial order reduction techniques [19] consider only single transitions (*representatives*) from a respective equivalence class where every transition leads to the same result. As we will show, our model checking methods rely on the mentioned techniques to verify safety properties of multi-threaded software.

2.2. Logical Foundations

We now define the used notation for our verification approach, i.e., syntax and semantics for both the theory of linear arithmetic ($\mathcal{T}_{LI}$) and Horn like clauses over $\mathcal{T}_{LI}$.

2.2.1. Linear Arithmetic

Let $i, j \in \mathbb{N}$ and $q \in \mathbb{Q}$ where $\mathbb{N}$ and $\mathbb{Q}$ refers to the sets of natural numbers and rational numbers, respectively. We use the standard definition of operations ($+$, $*$) and relations ($<$, $\leq$, $=$) for $\mathbb{N}$ and $\mathbb{Q}$.

Propositional logic Let AP be a set of atomic propositions containing atoms like b. A propositional logic formula $\phi \in \mathit{Formula}$ is defined by the grammar $\phi := b \mid \phi_1 \wedge \phi_2 \mid \neg\phi$. We use a model $M_0 : \mathit{Formula} \rightarrow \{\mathit{true}, \mathit{false}\}$ that assigns a boolean value to propositional formulas. The satisfaction relation $M_0 \models_{AP} \phi$ (read "M_0 satisfies ϕ") is defined by using terms of meta-logic (words from natural language like "not" or "and") as follows.

- $M_0 \models_{AP} b$ iff $M_0(b) = \mathit{true}$
- $M_0 \models_{AP} \phi_1 \wedge \phi_2$ iff $M_0 \models_{AP} \phi_1$ and $M_0 \models_{AP} \phi_2$
- $M_0 \models_{AP} \neg\phi$ iff not $M_0 \models_{AP} \phi$

Syntax Let V be a finite set of variables occurring in a linear arithmetic formula ϕ_{LI}, with $v \in V$. The grammar for the theory $\mathcal{T}_{LI}$ consists of terms, atoms and formulas:

$$
\begin{array}{lll}
\mathit{terms}_{LI} & \ni t & := q \mid v \mid q * v \mid t + t \\
\mathit{atoms}_{LI} & \ni a & := t \leq q \mid t < q \\
\mathit{formulas}_{LI} & \ni \phi_{LI} & := a \mid \phi_{LI} \wedge \phi_{LI} \mid \neg\phi_{LI} \mid \exists v : \phi_{LI}
\end{array}
$$

The shown grammar is kept concise without loss of expressivity. For example, the formula $3 \leq 5$ can be written as $0 * v \leq 2$, while the relation $v = y$ is representable by a conjunction of two inequalities ($v \leq y \wedge y \leq v$). Let $f, r \in \phi_{LI}$. For convenience, we will use additional symbols for formulas, which can be defined using the above grammar:

$$
\begin{array}{ll}
f \vee r & = \neg(\neg f \wedge \neg r) \\
f \rightarrow r & = \neg f \vee r = \neg(f \wedge \neg r) \\
f \leftrightarrow r & = f \rightarrow r \wedge r \rightarrow f \\
\bigvee_{i \in \{1,\ldots,N\}} f_i & = f_1 \vee \cdots \vee f_N \\
\bigwedge_{i \in \{1,\ldots,N\}} f_i & = f_1 \wedge \cdots \wedge f_N \\
\mathit{false} & = f \wedge \neg f \\
\mathit{true} & = \neg \mathit{false}
\end{array}
$$

Semantics We assign a truth value from $\{\mathit{true}, \mathit{false}\}$ to each linear arithmetic formula ϕ_{LI} depending on the valuation of the occurring variables. Such a valuation is a *model* $M_1 : V \rightarrow \mathbb{Q}$ that returns the assigned value of a variable $v \in V$. The values of a term can be obtained by an evaluation function $\mathit{eval}_{LI} : (\mathit{terms}_{LI}, V \rightarrow \mathbb{Q}) \rightarrow \mathbb{Q}$ using the term, a given model and the following equations.

- $\mathit{eval}_{LI}(q, M_1) = q$
- $\mathit{eval}_{LI}(v, M_1) = M_1(v)$
- $\mathit{eval}_{LI}(q * v, M_1) = q * M_1(v)$

- $eval_{LI}(t_1 + t_2) = eval_{LI}(t_1) + eval_{LI}(t_2)$

The definition of the satisfaction relation $\models_{LI}$ relies on a project function $project(\phi_{LI}, v)$ to handle quantifier elimination [1], i.e., simplification of formulas by producing equivalent ones without quantifiers as follows.

- $M_1 \models_{LI} t \leq q$, iff $eval_{LI}(t, M_1) \leq q$
- $M_1 \models_{LI} t < q$, iff $eval_{LI}(t, M_1) < q$
- $M_1 \models_{LI} \phi_1 \wedge \phi_2$, iff $M_1 \models \phi_1$ and $M_1 \models \phi_2$
- $M_1 \models_{LI} \neg\phi$, iff not $M_1 \models \phi$
- $M_1 \models_{LI} \exists v : \phi$, iff $M_1 \models project(\phi, v)$

Example Let us consider the formula $\phi = (\exists v : v = 2 \wedge v * x - 3 * y \leq 5)$ and the assignments $M_1(x) = 2$, $M_1(y) = 3$. We obtain that M_1 satisfies ϕ (written $M_1 \models_{LI} \phi$) since:

$$\begin{aligned}
project(v = 2 \wedge v * x - 3 * y \leq 5, v) &= 2 * x - 3 * y \leq 5 \text{ and} \\
eval_{LI}(2 * x - 3 * y, M_1) &= eval_{LI}(2 * x, M_1) + eval_{LI}(-3 * y, M_1) \\
&= 2 * eval_{LI}(x, M_1) + (-3) * eval_{LI}(y, M_1) \\
&= 2 * 2 + (-3) * 3 \\
&= -2
\end{aligned}$$

□

2.2.2. Horn-like Clauses

Our verification approaches are based on proof-rules in the form of Horn-like clauses HC. Such clauses are regular Horn clauses where occurring constraints may contain both disjunctions and conjunctions. From now, we write v to denote a non-empty tuple of variables with an arbitrary arity $n \in \mathbb{N}$, i.e., $v = (v_1, \ldots, v_n) \in V^+$. We refer to linear arithmetic formulas as constraints. Let $c(v)$ be a constraint over the variables v and *false* be an unsatisfiable constraint.

Syntax So far, we used so called *interpreted predicates* for symbols from linear arithmetic ($+, *, <, \leq, =$). First order theory (in particular Horn clauses) make use of *uninterpreted predicate symbols* (or *query symbols*) $\mathcal{Q}$. The arity of a query symbol is encoded in its name, i.e., in the number of formal arguments. Based on the query symbols $p \in \mathcal{Q}$ we define a language of Horn clauses:

$$\begin{array}{lll}
\mathcal{Q} & \ni u & := p(v) \\
bodies_{HC} & \ni b & := u \mid \phi_{LI} \mid b_1 \wedge b_2 \mid b_1 \vee b_2 \\
heads_{HC} & \ni h & := u \mid \phi_{LI} \mid h_1 \vee h_2 \\
clause_{HC} & \ni cl & := \forall v : b \rightarrow h \\
clauses_{HC} & \ni cls & := \{cl, \ldots, cl\}
\end{array}$$

[1] A possible implementation would be the Fourier-Motzkin elimination algorithm.

Semantics A set of clauses represents an assertion over the query symbols that occur in these clauses. We use a Model $M_2 : \mathcal{Q} \to \phi_{LI}$ that takes query symbols $p(v)$ and returns a constraint over v. The following definition of a satisfaction relation $\vDash_{HC}$ for Horn-like clauses uses a substitution function and quantifier elimination from background theory. The substitution function $\phi_{LI}[v_i/a_i]$ replaces all occurrences of variable v_i in the formula ϕ_{LI} by variable a_i.

$$M_2 \vDash_{HC} \forall v : c_0(v_0) \wedge \bigwedge_{i \in 1..n-1} p_i(v_i) \to \phi_{LI}(v_n), \text{ iff}$$
$$\forall v : c(v_0) \wedge \bigwedge_{i \in 1..n-1} M_2(p_i(v_i))[v_i/a_i] \to \begin{cases} c_1(v_n) & \text{if } \phi_{LI}(v_n) \text{ is } c_1(v_n) \\ M_2(p_n(v_n))[v_n/a_n] & \text{if } \phi_{LI}(v_n) \text{ is } p_n(v_n) \end{cases}$$

Example Let us consider the following formula presented in the language of Horn-like clauses, which is implicitly universally quantified over all free variables.

$$\begin{aligned} HC_1 = \{ \quad & x \geq 0 \to p(x), \\ & p(x) \wedge y = x + 1 \to q(y) \\ & q(x) \to x \geq 0 \quad \} \end{aligned}$$

The formulas $x \geq 0$ and $y = x + 1$ are interpreted predicates whereas p and q are query symbols of arity 1. The assignments $M_2(p(a)) = (a \geq 0)$ and $M_2(q(a)) = (a \geq 1)$ interpret these predicate symbols. We obtain that M_2 satisfies HC_1 since the following formulas are true.

$$\begin{aligned} \forall a : \quad & a \geq 0 \to a \geq 0 \\ \forall a, b : \quad & a \geq 0 \wedge b = a + 1 \to b \geq 1 \\ \forall a : \quad & a \geq 1 \to a \geq 0 \end{aligned}$$

□

2.3. Solving Horn-like Clauses

We now illustrate the HSF algorithm that we use for solving Horn-like clauses over linear inequalities from $\mathcal{T}_{LI}$.HSF uses symbolic reasoning to handle infinite domains, loop invariants, and ranking functions. The algorithm finds a solution for recursive Horn-like clauses by following an iterative, abstraction based approach that relies on (spurious) counterexample derivations to refine the abstraction in case of imprecision. It inherits the advantages and disadvantages of the existing counterexample-guided abstraction refinement schemes: a sufficiently precise abstraction can be discovered automatically although the abstraction discovery may not terminate. However, such a non-terminating behaviour is sufficiently seldom in practice.

Technically, an abstraction function α takes as input a constraint $\varphi(v)$ together with a finite set of predicates (atomic constraints) $Preds(v) := \{c_1(v), \ldots, c_n(v)\}$ over v. It returns a safe overapproximation of $\varphi(v)$ that is constructed using boolean operators as follows.

$$\alpha(\varphi(v), Preds(v)) = \bigwedge \{p(v) \in Preds(v) \mid \varphi(v) \models p(v)\}$$

The overapproximation guarantees that combining logical inference with abstraction yields solutions to inference clauses, i.e., $\varphi(v) \models \alpha(\varphi(v), Preds)$. The second important property

of α is monotonicity, i.e., if $\varphi(v) \models \psi(v)$ then $\alpha(\varphi(v), Preds) \models \alpha(\psi(v), Preds)$, which guarantees that fixpoint algorithms will finally return a result.

For example, let us consider the constraint $x \leq y \wedge y \leq z \wedge z \leq 0$ and a set of predicates $\{x \leq 0, x > 0, x \leq z\}$. When applied to the predicate abstraction function, it returns the conjunction $x \leq 0 \wedge x \leq z$.

HSF Algorithm

The input given to HSF is a model consisting of a finite set of clauses HC that is partitioned into inference clauses $\mathcal{I}$ and property clauses $\mathcal{P}$. Inference clauses contain query symbols in their heads and, thus, impose a relationship between query symbols. Property clauses contain interpreted predicates in their heads that impose absolute assertions on query symbols.

$$\begin{aligned} \mathcal{I} &= \{cl \in HC \mid cl = \ldots \rightarrow p(v)\} \\ \mathcal{P} &= HC \setminus \mathcal{I}. \end{aligned}$$

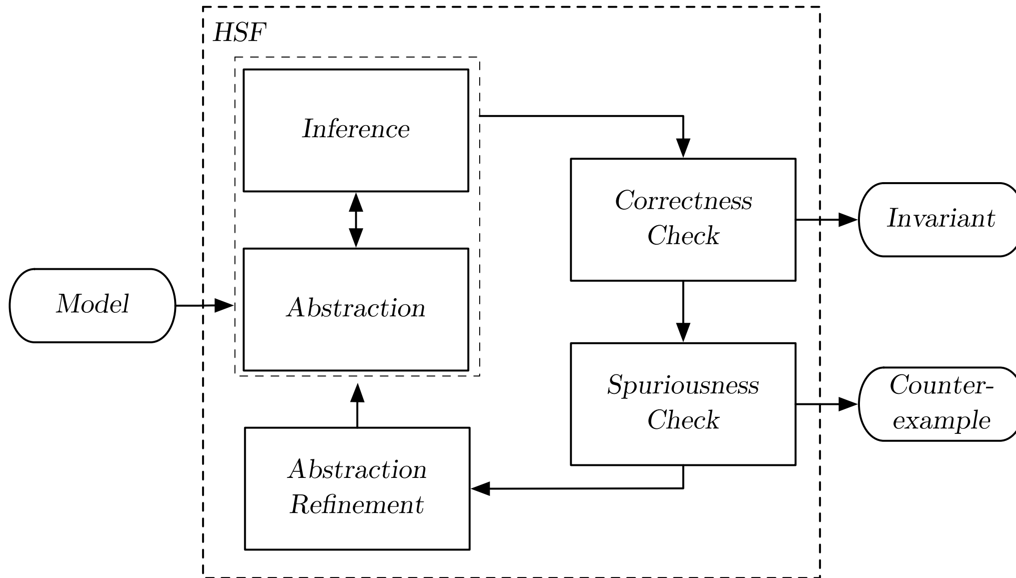


Figure 2.1.: Illustration of the HSF algorithm for solving Horn-like clauses.

The result of the algorithm is either an invariant that proves satisfiability of the property clauses or a counterexample for the opposite case. HSF uses a worklist-based approach that relies on existing off-the-shelf solvers for satisfiability modulo theories (SMT-solver) to solve arising inference problems. Every iteration step adds additional predicates that are used with the abstraction function and, hence, increase the precision of the obtained solutions. Figure 2.1 shows the distinct steps of each single iteration with the following functionality.

Inference + Abstraction Each iteration starts by performing logical inference to find an invariant for the inference clauses, i.e., it finds the least fixpoint. Here, the algorithm

relies on predicate abstraction to ensure termination and efficiency in the presence of recursion and large clause sets, respectively. At the initial setting, the respective predicates may be empty or come from background knowledge (to reduce the number of refinement steps). For each iteration step a finite number of predicates are added by abstraction refinement. The inference process terminates since the number of predicates that can be added is finite and the abstraction function is monotonic.

Correctness Check This step checks whether the computed solution from the previous inference satisfies the property clauses $\mathcal{P}$. If some property clause is not satisfied (which indicates an error state) the process continues with the spuriousness check, otherwise it returns the invariant as solution.

Spuriousness Check Due to the used abstraction, it may be that the solution proved to be incorrect is spurious. Therefore, the satisfaction checking will be repeated on the setting without any abstraction. If the violation is still present then HSF returns the corresponding execution path as witness. Otherwise, the obtained solution is forwarded to the abstraction refinement step.

Abstraction Refinement Spurious solutions may be used to make progress by refining abstraction predicates. Such a refinement avoids encountering identical solutions in following iterations. The algorithm obtains additional predicates by few actions. Firstly, it reconstructs inference steps that produced the found solution, i.e., using logical resolution to obtain recursive-free auxiliary clauses. Secondly, HSF solves these clauses by existing verification tools (e.g., some which use the lemma of Farkas). Finally, if a solution exists, the desired predicates are extracted. If no solution exists, the algorithm stops by returning the solution as witness of the property violation.

Correctness

The HSF algorithm computes a solution for the model upon termination. The proof relies on the soundness of the approach and the progress of refinement, which is standard for counterexample guided abstraction refinement schemes. Soundness can be proven by the fact that abstraction functions always return safe overapproximations. The progress of refinement property holds since the abstraction refinement method never analyzes a set of counterexample clauses twice.

2.4. Multi-threaded Programs

In this section, we describe the used transition system for multi-threaded programs, the assumed synchronization-model, and their computations.

Transition system A multi-threaded program P consists of $N \geq 1$ threads where each thread $i \in 1..N$ has a set of control locations $\mathcal{L}_i$. We assume that P is represented as abstract transition system $\langle V, \mathit{init}, \mathcal{R}, \mathit{error} \rangle$, given by the following components:

- $V = (V_G, V_1, \ldots, V_N)$ are program variables, partitioned into global Variables V_G shared by all threads and thread-local variables V_i, which are accessible only by

thread i. We distinguish two types of variables, data-variables and program counter variables. Firstly, data-variables are either global variables or thread-local variables that are exclusively manipulated by program statements. Secondly, program counter variables $\texttt{pc}_i$ are thread-local variables that keep track of current control locations $\ell_p \in \mathcal{L}_i$ for a program line with label p.
$\Rightarrow$ We represent sets of program states by valuations of the program variables.

- $\mathit{init}(V)$ is a satisfiable assertion describing the initial program states.
- $\mathcal{R} = (\mathcal{R}_1, \ldots, \mathcal{R}_N)$ describes a finite set of transition relations (transitions) for each thread, i.e., binary relations between sets of program states. Each program statement is represented as such a transition $\rho_i(V_G, V_i, V_G', V_i') \in \mathcal{R}_i$ by an assertion over variables and their primed versions. Transitions of some thread i may only access global variables $V_G^{(\prime)}$ and thread-local variables $V_i^{(\prime)}$. All thread-local variables from different threads $j \in 1..N \setminus \{i\}$ are not accessible by those transitions. We write step_i for the disjunction of transitions of a thread i, i.e., $\mathit{step}_i(V_G, V_i, V_G', V_i') = \bigvee \mathcal{R}_i$. For representation, we use the constraint $\mathit{step}_i^{=}$ requiring that the thread-local variables of other threads than i do not change, i.e., $\mathit{step}_i^{=} = \bigwedge_{j \in 1..N \setminus \{i\}} (V_j = V_j')$.
- $\mathit{error}(V)$ is used to represent assertion statements for erroneous program states.

Computations Let $\models$ denote the satisfaction relation between (pairs) of states and assertions over program variables (and their primed version). A *computation* of P is a sequence of states $(s_1, s_2, \ldots)$, such that s_1 is an initial state, i.e., $s_1 \models \mathit{init}(V)$, and for each consecutive pair of states s_i and s_{i+1} there is is a transition $\mathit{step} \in \mathcal{R}$ such that $(s_i, s_{i+1}) \models \mathit{step}$. An execution path (path) is a sequence of transitions.
A state s is *reachable* if it appears in some computation. Let φ_{reach} be the symbolic representation of all reachable states from P. We say that P is safe if no error state in $\mathit{error}(V)$ is reachable, i.e., $\varphi_{\mathit{reach}}(V) \wedge \mathit{error}(V) \rightarrow \mathit{false}$.

Synchronization We assume a thread-synchronization model based on a finite set of global lock variables. A lock can explicitly be acquired and released by means of acquire and release primitives, respectively. The acquire statement waits until the lock is released by every other thread and then acquires the lock. The release statement unconditionally releases the lock. We extend our previously shown program-representation to model lock-synchronization by further partitioning V and $\mathcal{R}$.

It is assumed that the set of used lock variables Locks are part of the global variables V_G. Each lock variable $m \in \mathit{Locks}$ is released ($m = 0$) at the initial states and can be explicitly acquired ($m = 1$) during exploration. We introduce assertions $\mathit{acq}_i(\texttt{pc}_i, \texttt{pc}_i')$ and $\mathit{rel}_i(\texttt{pc}_i, \texttt{pc}_i')$ to localize acquire and releases statements, respectively. These statements are defined over the determinant program locations, i.e., $\rho_i^a \rightarrow \mathit{acq}_i$ and $\rho_i^r \rightarrow \mathit{rel}_i$ for an acquire statement represented by transition ρ_i^a and a release statement represented by transition ρ_i^r.

Example We now use the example program from Figure 2.2 to illustrate transition systems. The program consist of two threads with thread-local variables $V_1 = (\texttt{pc}_1)$ and

```
int x=0, mx=0;

       // Thread 1                    // Thread 2
                                         int a=0;
       0: acquire(mx);               0: a=x;
       1: x=x+1;                     1: assert(a==x);
       2: release(mx);
       3:
```

Figure 2.2.: Example program to illustrate transition systems.

$V_2 = (\texttt{pc}_1, \texttt{a})$ for thread 1 and 2, respectively. The global variables accessible by both threads are $V_G = (\texttt{x}, \texttt{mx})$. By consideration of the program variables $V = (V_G, V_1, V_2)$, the initial state is represented by the following conjunction.

$$init(V) = (\texttt{x} = 0 \land \texttt{mx} = 0 \land \texttt{pc}_1 = 0 \land \texttt{pc}_2 = 0 \land \texttt{a} = 0)$$

For the sake of a short representation we use two abbreviations in transition relations. Firstly, alteration of program counter variables is represented as $mv_i(\ell_j, \ell_k) \equiv (\texttt{pc}_i = \ell_j \land \texttt{pc}_i' = \ell_k)$. Secondly, if a set of program variables $(v_1, \ldots, v_n)$ are not manipulated by a transition, we write $skp(v_1, \ldots, v_n) \equiv (v_1' = v_1 \land \cdots \land v_n' = v_n)$. Hence, the transitions for thread 1 and 2 are represented by the following disjuncts.

$$\begin{array}{rcl} step_1(V_G, V_1, V_G', V_1') & = & (mv_1(0,1) \land \texttt{mx} = 0 \land \texttt{mx}' = 1 \land skp(\texttt{x})) \lor \\ & & (mv_1(1,2) \land \texttt{x}' = \texttt{x} + 1 \land skp(\texttt{mx})) \lor \\ & & (mv_1(2,3) \land \texttt{mx}' = 0 \land skp(\texttt{x})) \\ step_2(V_G, V_2, V_G', V_2') & = & (mv_2(0,1) \land \texttt{a}' = x \land skp(\texttt{x}, \texttt{mx})) \end{array}$$

Consider the first conjunction of thread 1 representing the acquire transition. If it is satisfied, the lock is not held at the beginning state ($\texttt{mx} = 0$) and is (atomically) acquired at the resulting state ($\texttt{mx}' = 1$). Respective predicates for acquire and release are $acq_1(\texttt{pc}_1, \texttt{pc}_1') = (\texttt{pc}_1 = \ell_0 \land \texttt{pc}_1' = \ell_1)$ and $rel_1(\texttt{pc}_1, \texttt{pc}_1') = (\texttt{pc}_1 = \ell_2 \land \texttt{pc}_1' = \ell_3)$. The last component of the transition system is a definition of the erroneous states, which is obtained by the assertion statement from thread 2 as follows.

$$error(V) = \big(\texttt{pc}_2 = \ell_1 \land \neg(a = x)\big)$$

As the reader might conclude on her own, the example program is not safe according to the above definition. The following invariant for reachable states (that result of several computation steps beginning from $init(V)$) shows that the intersection with the error state is not empty (considering the second disjunct).

$$\begin{array}{rl} \varphi_{reach} = & \big(\texttt{pc}_1 = \ell_0 \land \texttt{x} = 0 \land \texttt{mx} = 0 \land \texttt{pc}_2 \in \{\ell_0, \ell_1\} \land \texttt{a} = 0\big) \lor \\ & \big(\texttt{pc}_1 = \ell_1 \land \texttt{x} = 0 \land \texttt{mx} = 1 \land \texttt{pc}_2 \in \{\ell_0, \ell_1\} \land \texttt{a} = 0\big) \lor \\ & \big(\texttt{pc}_1 = \ell_2 \land \texttt{x} = 1 \land \texttt{mx} = 1 \land (\texttt{pc}_2 = \ell_0 \land \texttt{a} = 0 \lor \texttt{pc}_2 = \ell_1 \land \texttt{a} \in \{0,1\})\big) \lor \\ & \big(\texttt{pc}_1 = \ell_3 \land \texttt{x} = 1 \land \texttt{mx} = 0 \land (\texttt{pc}_2 = \ell_0 \land \texttt{a} = 0 \lor \texttt{pc}_2 = \ell_1 \land \texttt{a} \in \{0,1\})\big) \end{array}$$

2.5. Proof Rules for Verification of Multi-threaded Programs

In this section, we show a collection of existing proof rules for the (safety-) verification of multi-threaded programs. All of them (monolithic, Owicki-Gries, and rely-guarantee) can be automated using the presented HSF algorithm in section 2.3. We consider a multi-threaded program that consists of N threads as a tuple $(V, \mathit{init}, \mathcal{R}, \mathit{error})$, with V, init, $\mathcal{R}$ and error as defined in section 2.4.

2.5.1. Monolithic Proof Rule

The monolithic proof rule lists three conditions over a single query symbol $R(V)$ that characterizes an overapproximation of the reachable states as follows.

$$\begin{array}{lll} \textbf{CM1:} & \mathit{init}(V) & \rightarrow R(V) \\ \textbf{CM2:} & R(V) \wedge \mathit{step}_i(V_G, V_i, V_G', V_i') \wedge \mathit{step}_i^{=}(V, V') & \rightarrow R(V') \\ \textbf{CM3:} & R(V) \wedge \mathit{error}(V) & \rightarrow \mathit{false} \end{array}$$

The clauses require that $R(V)$ contains all initial states (clause **CM1**) and every successive state that is reachable by a transition relation (clause **CM2**). Thread interleaving implicitly happens by the non-deterministic choice of thread i in clause **CM2**. Clause **CM3** requires that the intersection of reachable states and error states is empty.

2.5.2. Owicki-Gries Proof Rule

We list a proof rule that is based on the Owick-Gries method [28]. The reasoning about reachable states is localized by replacing the global auxiliary assertion R from above with N query symbols $R_1(V), \ldots, R_N(V)$ for N threads.

$$\begin{array}{lll} \textbf{CO1:} & \mathit{init}(V) & \rightarrow R_i(V) \\ \textbf{CO2:} & R_i(V) \wedge \mathit{step}_i(V_G, V_i, V_G', V_i') \wedge \mathit{step}_i^{=}(V, V') & \rightarrow R_i(V') \\ \textbf{CO3:} & R_i(V) \wedge \left(\bigvee_{j \in 1..N \setminus \{i\}} R_j(V) \wedge \mathit{step}_j(V_G, V_j, V_G', V_j') \wedge \mathit{step}_j^{=}(V, V') \right) & \rightarrow R_i(V') \\ \textbf{CO4:} & R_1(V) \wedge \cdots \wedge R_N(V) \wedge \mathit{error}(V) & \rightarrow \mathit{false} \end{array}$$

Clauses **CO1** and **CO2** require that R_i contains the initial states and states resulting from successively applied transition relations of some thread i, respectively. As opposed to the monolithic proof rule, thread interleaving happens explicitly at condition **CO3**. Safety is proven by the empty intersection of error states and the conjunction of reachable states from all threads (**CO4**).

2.5.3. Rely-Guarantee Proof Rule

We now present the rely-guarantee proof rule [18] for compositional verification of program safety. It uses assertions R_i over V and E_i over V and V' that represent reachable

states and environment transitions for each thread $i \in 1..N$, respectively.

CR1: $init(V) \rightarrow R_i(V)$

CR2: $R_i(V) \wedge step_i(V_G, V_i, V'_G, V'_i) \wedge step_{\bar{i}}^{=}(V, V') \rightarrow R_i(V')$

CR3: $\left(\bigvee_{i \in 1..N \setminus \{j\}} R_i(V) \wedge step_i(V_G, V_i, V'_G, V'_i) \wedge step_{\bar{i}}^{=}(V, V')\right) \rightarrow E_j(V, V')$

CR4: $R_i(V) \wedge E_i(V, V') \rightarrow R_i(V')$

CR5: $R_1(V) \wedge \cdots \wedge R_N(V) \wedge error(V) \rightarrow false$

Each assertion $R_i(V)$ contains the initial states (clause **CR1**), states reachable from local transitions (clause **CR2**), and states reachable from environment transitions (clause **CR4**). $E_i(V)$ include compositions of transition steps from other threads $j \neq i$ (clause **CR3**). Similar to the previous rules, the emptiness of the intersection of reachable states and $error(V)$ ensures safety by clause **CR5**.

Part II.

Verification Methods

3. Model Checking with Transaction Summarization

Development of practical verification tools for multi-threaded programs requires dealing with the explosion of the number of thread interleavings.

In this chapter, we present our transaction-based approach to exploit equivalence of different interleavings. Transactions are sequences of successive program transitions that may be verified without consideration of some interference from other threads. Our verification technique handles multi-threaded programs with infinite state spaces in the constrained based setting by relying on the mentioned HSF algorithm. Our design decisions were directed by the following considerations. Commutativity inference serves as preliminary step for a constraint based verification run. We allow this inference to be more precise and data dependent in comparison with type based approaches, e.g., [12]. Even though being potentially more expensive, the ability to infer larger transactions at this step may lead to dramatic reduction in verification time.

Our summarization rule is inspired by the use of procedure summaries, see e.g., [31], however instead of being driven by calls/returns to mark start/finish points of summaries, we use transitions that enter/exit from transactions. Summarization constraints allow us to eliminate statements contributing to transactions from the program and keep track of their effect by applying the transaction summaries. Note that complex control flow constructs, including loops, can be directly supported as parts of transactions, since summarization constraints defer reasoning about complex control flow to the final solving step.

We begin with an introduction of transactions by giving some formal definitions of related terms. For illustration, we additionally present the overall approach by an extensive example. The next section is used for our transaction inference algorithms that delivers transaction boundaries from a given input program represented as Horn-like clauses. The last two sections present our proof rules for safety verification together with a sketch of the soundness proof.

3.1. Lipton's Theory of Reduction

We use the theory of right and left movers from Lipton [22] to define transactions for aiding verification of software with multiple threads. The intuition behind this theory is to simplify proofs by declaring a particular sequence of statements as atomic, i.e., these statements can not be interleaved by statements from other threads. Lipton proposes that a program P containing such a sequence of atomic statements T is equivalent to a reduced Program $P \setminus T$ where all statements from T are executed without interference. To identify T, some transition relations $\rho i \in \mathcal{R}_i$ of a thread i are defined as *right movers* and *left movers* as follows.

- A transition $\rho_i \in \mathcal{R}_i$ of thread i is a right mover if $\forall \rho_{j,\ j \in 1..N \setminus \{i\}} \in \mathcal{R}_j$:

$$\big(\rho_i(V_G, V_i, V'_G, V'_i) \wedge \rho_j(V'_G, V_j, V''_G, V'_j)\big) \\ \Longleftrightarrow \\ \big(\rho_j(V_G, V_j, V'''_G, V'_j) \wedge \rho_i(V'''_G, V_i, V''_G, V'_i)\big)$$

- Similarly, a transition $\rho_i \in \mathcal{R}_i$ of thread i is a left mover if $\forall \rho_{j,\ j \in 1..N \setminus \{i\}} \in \mathcal{R}_j$:

$$\big(\rho_j(V_G, V_j, V'_G, V'_j) \wedge \rho_i(V'_G, V_i, V''_G, V'_i)\big) \\ \Longleftrightarrow \\ \big(\rho_i(V_G, V_i, V'''_G, V'_i) \wedge \rho_j(V'''_G, V_j, V''_G, V'_j)\big)$$

The first definition asserts that if there is a right mover transition ρ_i of thread i followed by any transition ρ_j of some other thread j, the resulting state is equivalent to the one resulting from the sequence where ρ_j is executed before ρ_i, i.e., a right mover commutes to the right. The second definition of left mover transitions is symmetric, i.e. a left mover commutes to the left. Transitions that are both left movers and right movers are defined as *both movers*. All other transitions, i.e., the ones that do not commute with every other transition from other threads (in at least one direction) are *non-movers*. As we will show in Section 3.3, there are some general observations about mover information in multi-threaded programs:

- Program statements that acquire locks are right movers. After obtaining the lock there is no other thread that may have access to the protected variables. Hence, these statements commute to the right with every other statement from other threads.

- Program statements that release locks are left movers. Before the release operation takes place no other thread has access to the protected variables. Thus, the release statement commutes to the left with statements from other threads.

- Program statements that only access local variables are both movers since these statements commute to the left and to the right regarding other threads.

- Program statements that access global variables are both movers if they are exclusively enabled at states where some common locks are held. Thus, every access to this shared variable from other threads is also protected by the same locks and, hence commutes.

Let $n, m \in \mathbb{Z}^+$. We define a *transaction* as a non-empty sequence of transition relations $a_1, \ldots, a_n, [c], b_1, \ldots, b_m$ where each a_i is a right mover, c is one optional non-mover, and each b_i is a left mover. Note that this definition also includes both mover transitions since left movers and right movers are also both movers. The optional non-mover c that resides between the sequence of right movers (pre commit phase) and left movers (post commit phase) is called the *committing transition* (the term comes from the theory of database-transactions). Transactions can safely be summarized since all transitions a_i commute to the right and all transitions b_i commute to the left.

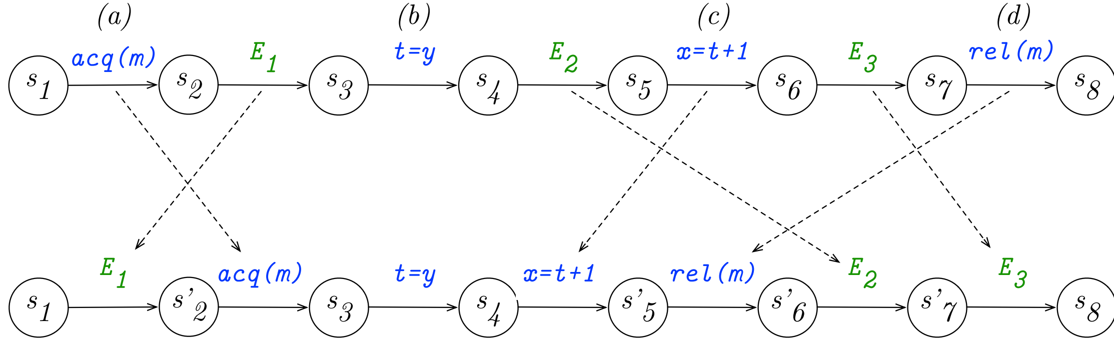


Figure 3.1.: Two equivalent transition sequences.

Example We use the transition sequences (executions) shown in Figure 3.1 to illustrate the above definitions. Let us first consider the upper transition sequence. It contains four thread-local transition relations $((a), (b), (c), (d))$ from some thread i and three environment transitions E_j from threads different than i. The execution starts at program state s_1 and ends with program state s_8 by successively applying the transitions. The statements $acq(m)$ and $rel(m)$ acquire and release the lock m, respectively. The other two statements $((b)$ and $(c))$ access the thread-local variable t and the global variables x, y. For this example, we presume that x is entirely protected by the same lock throughout the program. However, variable y is written by some other thread than i without being protected by m, which makes transition (b) to a non-mover.

According to our definition, the sequence $a-b-c-d$ can be summarized to a transaction since (a) is a right mover, (b) is a non-mover, (c) is a both mover (hence, a left mover), and (d) is a left mover. The execution below is equivalent to the one above since (a) right commutes with E_1 and both (c) and (d) left commute with the other environment transitions.

We will sometimes refer to a transaction as the set of control locations that belong to states resulting from transitions that enter a transaction, are inside a transactions, or leave a transaction.

3.2. Illustration

We illustrate our transaction-based verification approach with a multi-threaded program consisting of three threads. See Figure 3.2 for the program P1-1 which uses locks (`mx`, `my`) to protect accesses to shared variables (`x`, `y`). Its representation is shown in Figure 3.3, according to the formulation in Section 2.4.

For illustration, we aim to prove that the value of the variable `x` is not equal to 11 at the end locations (this is indeed the case as the reader may conclude on her own after inspecting possible program executions). A verification method needs to consider a large number of thread interleavings that is exponential with number of threads [1]. Our method

[1]In fact, the number of possible thread interleavings is $\frac{\left(\sum_{t=1}^{N} |\mathcal{L}_t|\right)!}{\prod_{t=1}^{N}(|\mathcal{L}_t|)!}$ where N is the number of threads and $|\mathcal{L}_t|$ is the number of program locations for a thread t.

```
int x=2, y=2, mx=0, my=0;

   // Thread-1              // Thread-2            // Thread-3
       int a;
    0: acquire(mx);         0: acquire(mx);        0: acquire(my);
    1: a = x;               1: x = x+2;            1: y = y+2;
    2: acquire(my);         2: release(mx);        2: release(my);
    3: y = y+a;             3:                     3:
    4: release(my);
    5: a = a+1;
    6: acquire(my);
    7: y = y+a;
    8: release(my);
    9: x = 2*x+a;
   10: release(mx);
   11:
```

Figure 3.2.: Program P1-1 consisting of three threads; the error assertion is given as $\mathit{error}(V) = \mathtt{x} = 11 \wedge \mathtt{pc}_1 = 11 \wedge \mathtt{pc}_2 = 3 \wedge \mathtt{pc}_3 = 3$.

is based on transaction reasoning and considers explicitly only a small number of these interleavings; for this example, 12 interleavings are considered. The approach consists of two steps, transaction inference and transaction-based verification.

Transaction Inference

The objective of transaction inference is to get transactions that are as large as possible in order to minimize the number of explored interleavings during verification. In this section, we assume the results of transaction inference are given (see Section 3.3 for a formal description of our transaction inference algorithm and its application on the P1-1 example). The following table shows all transaction boundaries (a), (b), (c) and (d) obtained for each thread. Note that these boundaries are presented in the form `thread-i`$\{\ell_b - \ell_e\}$, i.e., the lower bound is $\mathtt{pc}_i = \ell_b$ and the upper bound is $\mathtt{pc}_i''' = \ell_e$ for a transaction $\big(\mathit{step}_i(V_G, V_i, V_G', V_i') \wedge \cdots \wedge \mathit{step}_i(V_G'', V_i'', V_G''', V_i''')\big)$.

(a)	`thread-1` $\{\ell_0 - \ell_6\}$	(c)	`thread-2` $\{\ell_0 - \ell_3\}$
(b)	`thread-1` $\{\ell_6 - \ell_{11}\}$	(d)	`thread-3` $\{\ell_0 - \ell_3\}$

For `thread-2` and `thread-3`, all transitions are composed to transactions (c) and (d), respectively. For `thread-1`, two transactions are obtained. The first one spans over the program locations ℓ_0 and ℓ_6, the second one spans over the program locations ℓ_6 and ℓ_{11}.

We encode the result of transaction inference using a partitioning of statements into three categories for each thread i:

$$
\begin{array}{lcl}
V_G & = & (\mathtt{x},\mathtt{y},\mathtt{mx},\mathtt{my}),\ V_1 = (\mathtt{a},\mathtt{pc}_1),\ V_2 = (\mathtt{pc}_2),\ V_3 = (\mathtt{pc}_3) \\
init(V_G,V_1,V_2,V_3) & = & (\mathtt{pc}_1 = \mathtt{pc}_2 = \mathtt{pc}_3 = \ell_0 \wedge \mathtt{x} = 2 \wedge \mathtt{y} = 2 \wedge \mathtt{mx} = 0 \wedge \mathtt{my} = 0) \\
step_1(V_G,V_1,V_G',V_1') & = & (mv_1(\ell_0,\ell_1) \wedge \mathtt{mx} = 0 \wedge \mathtt{mx}' = 1 \wedge skp(\mathtt{x},\mathtt{y},\mathtt{my},\mathtt{a})) \vee \\
 & & (mv_1(\ell_1,\ell_2) \wedge \mathtt{a}' = \mathtt{x} \wedge skp(\mathtt{x},\mathtt{y},\mathtt{mx},\mathtt{my})) \vee \\
 & & (mv_1(\ell_2,\ell_3) \wedge \mathtt{my} = 0 \wedge \mathtt{my}' = 1 \wedge skp(\mathtt{x},\mathtt{y},\mathtt{mx},\mathtt{a})) \vee \\
 & & (mv_1(\ell_3,\ell_4) \wedge \mathtt{y}' = \mathtt{y} + \mathtt{a} \wedge skp(\mathtt{x},\mathtt{mx},\mathtt{my},\mathtt{a})) \vee \\
 & & (mv_1(\ell_4,\ell_5) \wedge \mathtt{my}' = 0 \wedge skp(\mathtt{x},\mathtt{y},\mathtt{mx},\mathtt{a})) \vee \\
 & & (mv_1(\ell_5,\ell_6) \wedge \mathtt{a}' = \mathtt{a} + 1 \wedge skp(\mathtt{x},\mathtt{y},\mathtt{mx},\mathtt{my})) \vee \\
 & & (mv_1(\ell_6,\ell_7) \wedge \mathtt{my} = 0 \wedge \mathtt{my}' = 1 \wedge skp(\mathtt{x},\mathtt{y},\mathtt{mx},\mathtt{a})) \vee \\
 & & (mv_1(\ell_7,\ell_8) \wedge \mathtt{y}' = \mathtt{y} + \mathtt{a} \wedge skp(\mathtt{x},\mathtt{mx},\mathtt{my},\mathtt{a})) \vee \\
 & & (mv_1(\ell_8,\ell_9) \wedge \mathtt{my}' = 0 \wedge skp(\mathtt{x},\mathtt{y},\mathtt{mx},\mathtt{a})) \vee \\
 & & (mv_1(\ell_9,\ell_{10}) \wedge \mathtt{x}' = 2 * \mathtt{x} + \mathtt{a} \wedge skp(\mathtt{y},\mathtt{mx},\mathtt{my},\mathtt{a})) \vee \\
 & & (mv_1(\ell_{10}\ell_{11}) \wedge \mathtt{mx}' = 0 \wedge skp(\mathtt{x},\mathtt{y},\mathtt{my},\mathtt{a})) \\
 & & \\
step_2(V_G,V_2,V_G',V_2') & = & (mv_2(\ell_0,\ell_1) \wedge \mathtt{mx} = 0 \wedge \mathtt{mx}' = 1 \wedge skp(\mathtt{x},\mathtt{y},\mathtt{my})) \vee \\
 & & (mv_2(\ell_1,\ell_2) \wedge \mathtt{x}' = \mathtt{x} + 2 \wedge skp(\mathtt{y},\mathtt{mx},\mathtt{my})) \vee \\
 & & (mv_2(\ell_2,\ell_3) \wedge \mathtt{mx}' = 0 \wedge skp(\mathtt{x},\mathtt{y},\mathtt{my})) \\
 & & \\
step_3(V_G,V_3,V_G',V_3') & = & (mv_3(\ell_0,\ell_1) \wedge \mathtt{my} = 0 \wedge \mathtt{my}' = 1 \wedge skp(\mathtt{x},\mathtt{y},\mathtt{mx})) \vee \\
 & & (mv_3(\ell_1,\ell_2) \wedge \mathtt{y}' = \mathtt{y} + 2 \wedge skp(\mathtt{x},\mathtt{mx},\mathtt{my})) \vee \\
 & & (mv_3(\ell_2,\ell_3) \wedge \mathtt{my}' = 0 \wedge skp(\mathtt{x},\mathtt{y},\mathtt{mx})) \\
 & & \\
error(V_G,V_1,V_2,V_3) & = & (\mathtt{x} = 11 \wedge \mathtt{pc}_1 = \ell_{11} \wedge \mathtt{pc}_2 = \ell_3 \wedge \mathtt{pc}_3 = \ell_3)
\end{array}
$$

Figure 3.3.: Representation of program P1-1 as transition system.

- $step_outout_i(V_G, V_i, V_G', V_i')$ describes transitions that are not contained in any transaction, i.e., both $\mathtt{pc}_i$ and $\mathtt{pc}_i'$ are control locations outside transactions.

- $step_in_i(V_G, V_i, V_G', V_i')$ describes transitions that are either the first transition of a transaction or are in the middle of a transaction (but not at the end), i.e., $\mathtt{pc}_i'$ is a control location that is inside a transaction.

- $step_inout_i(V_G, V_i, V_G', V_i')$ describes transitions that are the last transition of a transaction, i.e. $\mathtt{pc}_i$ is a control location inside a transaction and $\mathtt{pc}_i'$ is a control location outside transactions.

For our example, we obtain the following partitioning of statements.

$$
\begin{aligned}
step_in_1(V_G, V_1, V_G', V_1') \quad &= step_1(V_G, V_1, V_G', V_1') \wedge \\
&\quad (mv_1(\ell_0, \ell_1) \vee mv_1(\ell_1, \ell_2) \ \vee mv_1(\ell_2, \ell_3) \vee \\
&\quad\ mv_1(\ell_3, \ell_4) \vee mv_1(\ell_4, \ell_5) \vee mv_1(\ell_6, \ell_7) \vee \\
&\quad\ mv_1(\ell_7, \ell_8) \vee mv_1(\ell_8, \ell_9) \ \vee mv_1(\ell_9, \ell_{10})) \\
step_inout_1(V_G, V_1, V_G', V_1') &= step_1(V_G, V_1, V_G', V_1') \wedge (mv_1(\ell_5, \ell_6) \vee mv_1(\ell_{10}, \ell_{11})) \\
step_in_2(V_G, V_2, V_G', V_2') \quad &= step_2(V_G, V_2, V_G', V_2') \wedge (mv_2(\ell_0, \ell_1) \vee mv_2(\ell_1, \ell_2)) \\
step_inout_2(V_G, V_2, V_G', V_2') &= step_2(V_G, V_2, V_G', V_2') \wedge (mv_2(\ell_2, \ell_3)) \\
step_in_3(V_G, V_3, V_G', V_3') \quad &= step_3(V_G, V_3, V_G', V_3') \wedge (mv_3(\ell_0, \ell_1) \vee mv_3(\ell_1, \ell_2)) \\
step_inout_3(V_G, V_3, V_G', V_3') &= step_3(V_G, V_3, V_G', V_3') \wedge (mv_3(\ell_2, \ell_3))
\end{aligned}
$$

There are no transitions outside transactions, i.e.,

$$
\begin{aligned}
&step_outout_1(V_G, V_1, V_G', V_1') = step_outout_2(V_G, V_2, V_G', V_2') = \\
&step_outout_3(V_G, V_3, V_G', V_3') = \mathit{false}.
\end{aligned}
$$

Proof Rule

The crux of our verification approach is a proof rule for transaction-based reasoning. The proof rule lists conditions on three kinds of auxiliary assertions (program invariants):

- $R(V)$ describes reachable program states outside transactions, i.e., program states resulting from application of either transitions outside transactions or transaction summaries.
- $P_i(V_G, V_i, V_G', V_i')$ are binary path relations (paths) in a transaction, i.e., a composition of transition relations in a transaction beginning from its first transition to any other transition inside it (except the last one).
- $Summ_i(V_G, V_i, V_G', V_i')$ represents binary (summarization) relations that are compositions of all transition relations in a transaction.

The proof rule conditions are expressed as Horn-like clauses numbered from (3.1) to (3.6). The first clause states that all initial states are reachable (are contained in $R(V)$):

$$
init(V) \rightarrow R(V) \tag{3.1}
$$

Therefore, a solution of the reachable-states assertion (denoted by $\Sigma(R(V))$) will include at least the initial states; for our example, the following constraint:

$$
\Sigma(R(V)) := (\mathtt{pc}_1 = \ell_0 \wedge \mathtt{pc}_2 = \ell_0 \wedge \mathtt{pc}_3 = \ell_0 \wedge \mathtt{x} = 2 \wedge \mathtt{y} = 2 \wedge \mathtt{mx} = 0 \wedge \mathtt{my} = 0)
$$

Intra-transactional Reasoning

We now present the clauses from our proof rule for reasoning inside transactions, i.e., without interference from transitions of other threads. Note that the following rules consider

only thread-local assertions of thread i.

$$R(V) \wedge \mathit{step_in}_i(V_G, V_i, V'_G, V'_i) \rightarrow P_i(V_G, V_i, V'_G, V'_i) \tag{3.2}$$

$$P_i(V_G, V_i, V'_G, V'_i) \wedge \mathit{step_in}_i(V'_G, V'_i, V''_G, V''_i) \rightarrow P_i(V_G, V_i, V''_G, V''_i) \tag{3.3}$$

$$P_i(V_G, V_i, V'_G, V'_i) \wedge \mathit{step_inout}_i(V'_G, V'_i, V''_G, V''_i) \rightarrow \mathit{Summ}_i(V_G, V_i, V''_G, V''_i) \tag{3.4}$$

Clause (3.2) initiates P_i relations whenever a transition $\mathit{step_in}_i(V_G, V_i, V'_G, V'_i)$ at the beginning of a transaction is applicable. Once inside a transaction, clause (3.3) extends the path relation as long as this transaction contains further transitions (except the last transition). Otherwise, a summary relation Summ_i is generated for the current transaction (clause (3.4)).

We illustrate the application of these clauses with the path relations for `thread-2` by starting from the previously computed initial states in $\Sigma(R(V))$.

$$\begin{aligned}\Sigma(P_2(V_G, V_2, V'_G, V'_2)) := (&\mathit{mv}_2(\ell_0, \ell_1) \wedge \mathtt{mx} = 0 \wedge \mathtt{mx}' = 1 \wedge \mathit{skp}(\mathtt{x}, \mathtt{y}, \mathtt{my}) \vee \\ &\mathit{mv}_2(\ell_0, \ell_2) \wedge \mathtt{mx} = 0 \wedge \mathtt{mx}' = 1 \wedge \mathtt{x}' = \mathtt{x} + 2 \wedge \mathit{skp}(\mathtt{y}, \mathtt{my}))\end{aligned}$$

A summary relation for `thread-2` is generated using clause (3.4), the result from the last step and the assertion $\mathit{step_inout}_2(V_G, V_2, V'_G, V'_2)$:

$$\Sigma(\mathit{Summ}_2(V_G, V_2, V'_G, V'_2)) := (\mathit{mv}_2(\ell_0, \ell_3) \wedge \mathtt{mx} = 0 \wedge \mathtt{mx}' = 0 \wedge \mathtt{x}' = \mathtt{x} + 2 \ \wedge \mathit{skp}(\mathtt{y}, \mathtt{my}))$$

Reasoning outside Transactions

As mentioned before, reachable states $R(V)$ solely contain program states that are outside transactions. In addition to clause (3.1), we use the following rule to extend $R(V)$ by either a transaction summarization or a single transition that is outside any transaction.

$$R(V) \wedge \Big(\mathit{Summ}_i(V_G, V_i, V'_G, V'_i) \vee \mathit{step_outout}_i(V_G, V_i, V'_G, V'_i)\Big) \wedge \mathit{step}_{\bar{i}}^{=}(V, V') \rightarrow R(V') \tag{3.5}$$

For our example, we update the solution for $R(V)$ by applying clause (3.5) on the previously computed reachable states and $\mathit{Summ}_2(V_G, V_2, V'_G, V'_2)$.

$$\begin{aligned}\Sigma(R(V)) := \quad &(\mathtt{pc}_1 = \ell_0 \wedge \mathtt{pc}_2 = \ell_0 \wedge \mathtt{pc}_3 = \ell_0 \wedge \mathtt{x} = 2 \wedge \mathtt{y} = 2 \wedge \mathtt{mx} = 0 \wedge \mathtt{my} = 0) \vee \\ &(\mathtt{pc}_1 = \ell_0 \wedge \mathtt{pc}_2 = \ell_3 \wedge \mathtt{pc}_3 = \ell_0 \wedge \mathtt{x} = 4 \wedge \mathtt{y} = 2 \wedge \mathtt{mx} = 0 \wedge \mathtt{my} = 0)\end{aligned}$$

For simplicity, we showed here a monolithic style of reasoning with a single reachable-state invariant for all threads. In practice, compositional reasoning enables better scalability since individual threads can be analyzed instead of an exhaustive exploration of the the whole system monolithically. Our proof rule supports different styles of reasoning outside transactions, see Section 3.4 for a modified proof rule using Owicki-Gries reasoning outside transactions.

Proving Safety

We obtain a safety proof for a given program if there exists a solution for $R(V)$ satisfiying the above clauses together with the following clause that ensures emptiness of the intersection of reachable states and error states.

$$R(V) \wedge \mathit{error}(V) \rightarrow \mathit{false} \tag{3.6}$$

So far, we have considered an execution that only finished exploring the transaction (c) from `thread-2`. Such an execution could continue by non-deterministically exploring transactions from either `thread-1` or `thread-3`, i.e., execution ($c-a-b-d$), ($c-a-d-b$) or ($c-d-a-b$).

The benefit of transactional reasoning is that few interleavings are effectively explored due to the coarse-grained nature of transactions. Besides the shown execution sequence for P1-1, the following list contains all possible transaction-interleavings.

(I1)	$a-b-c-d$	(I7)	$c-a-b-d$
(I2)	$a-b-d-c$	(I8)	$c-a-d-b$
(I3)	$a-c-b-d$	(I9)	$c-d-a-b$
(I4)	$a-c-d-b$	(I10)	$d-a-b-c$
(I5)	$a-d-b-c$	(I11)	$d-a-c-b$
(I6)	$a-d-c-b$	(I12)	$d-c-a-b$

The effect of all these interleavings is captured by the auxiliary assertions from our proof rule. Correspondingly, our method computes the final solution $\Sigma(R(V))$ representing the invariant for reachable states as follows:

$$\begin{aligned}
&\Big(\mathtt{pc}_1 = \ell_0 \ \wedge \mathtt{mx} = 0 \wedge \mathtt{pc}_3 \in \{\ell_0, \ell_3\} \wedge (\ \mathtt{pc}_2 = \ell_0 \wedge \mathtt{x} = 2 \vee \\
&\qquad\qquad \mathtt{pc}_2 = \ell_3 \wedge \wedge 4 \leq \mathtt{x} \leq 7)\ \Big)\vee \\
&\Big(\mathtt{pc}_1 = \ell_6 \ \wedge \mathtt{mx} = 1 \wedge \mathtt{pc}_3 \in \{\ell_0, \ell_3\} \wedge (\ \mathtt{pc}_2 \in \{\ell_0, \ell_3\} \wedge \mathtt{x} = 2 \wedge 2\mathtt{x} + \mathtt{a} = 7 \vee \\
&\qquad\qquad \mathtt{pc}_2 \in \{\ell_0, \ell_3\} \wedge 4 \leq \mathtt{x} \leq 7 \wedge 2\mathtt{x} + \mathtt{a} \geq 13)\ \Big)\vee \\
&\Big(\mathtt{pc}_1 = \ell_{11} \wedge \mathtt{mx} = 0 \wedge \mathtt{pc}_3 \in \{\ell_0, \ell_3\} \wedge (\ \mathtt{pc}_2 = \ell_0 \wedge \mathtt{x} \leq 7 \vee \\
&\qquad\qquad \mathtt{pc}_2 = \ell_3 \wedge \mathtt{x} \leq 9 \vee \\
&\qquad\qquad \mathtt{pc}_2 \in \{\ell_0, \ell_3\} \wedge \mathtt{x} \geq 13 \wedge 2\mathtt{x} + \mathtt{a} \geq 13)\Big)
\end{aligned}$$

Our verification method uses the CEGAR-based HSF algorithm and, hence, computes over-approximations. Consequently, some constraints do not appear in the solution since they are not used for abstraction refinement. For example, constraints on the value of variable `y` are not present in the above reachable states due to the used error specification $error(V) = (\mathtt{x} = 11 \wedge \mathtt{pc}_1 = \ell_{11} \wedge \mathtt{pc}_2 = \ell_3 \wedge \mathtt{pc}_3 = \ell_3)$.

The shown constraints are divided into three cases (outer disjuncts) that contain either the location $\mathtt{pc}_1 = \ell_0$, $\mathtt{pc}_1 = \ell_6$, or $\mathtt{pc}_1 = \ell_{11}$ resulting from the given transaction boundaries for `thread-1`. This thread holds the lock `mx` only at location ℓ_6 due to the acquire statement at location ℓ_0 and the release statement at location ℓ_{11}. The inner disjuncts in each case result from varied execution orders of `thread-1` and `thread-2`. Statements of `thread-3` have no influence on the scheduling (and hence on the solution) since there is no access of variable `x` (used in $error(V)$). At states with $\mathtt{pc}_1 = \ell_{11}$, we observe three possible outcomes. Firstly, `thread-2` has not yet started ($\mathtt{x} \leq 7$). Secondly, `thread-2` may have been executed after `thread-1` ($\mathtt{x} \leq 9$). Finally, `thread-2` may have been executed before `thread-1` ($\mathtt{x} \geq 13$). Safety is proven by the fact that the value of `x` is always smaller or bigger than 11. Note that a safety-critical interference of `thread-1` at location ℓ_6 by `thread-2` is prohibited by the held lock `mx`. For illustration, we show below how the interference can be induced by unprotecting the access to variable `x` in `thread-2`.

Data Races

As shown above, our method benefits from the race-free nature of statements, infers coarse-grained transactions and therefore leads to effective verification. We use two modified variants of the example from Figure 3.2 to illustrate how our approach works in the presence of race conditions. The first variant induces a counterexample for the used safety assertion. The second variant remains safe but adds a non mover transition and, by that, emphasizes the effects of race conditions regarding to scalability.

Program P2-1 Consider a different implementation of `thread-2` where the statement that access the global variable `x` is not protected by a lock:

```
// Thread-2

0: x = x+2;
1:
```

As opposed to the original program, all transitions of `thread-1` that access variable `x` become potential interleaving points since `thread-2` may interfere. Consequently, transaction inference returns a different list of transaction boundaries with two additional transactions for `thread-1` as follows.

(a)	`thread-1` $\{\ell_0 - \ell_2\}$	(e)	`thread-2` $\{\ell_0 - \ell_1\}$
(b)	`thread-1` $\{\ell_2 - \ell_6\}$	(f)	`thread-3` $\{\ell_0 - \ell_3\}$
(c)	`thread-1` $\{\ell_6 - \ell_9\}$		
(d)	`thread-1` $\{\ell_9 - \ell_{11}\}$		

The HSF solver returns the execution in Figure 3.4 as counterexample for the safety assertion of clause (3.6) with $error(V) = (\mathtt{x} = 11 \wedge \mathtt{pc}_1 = 11 \wedge \mathtt{pc}_2 = 1 \wedge \mathtt{pc}_3 = 3)$. Note that

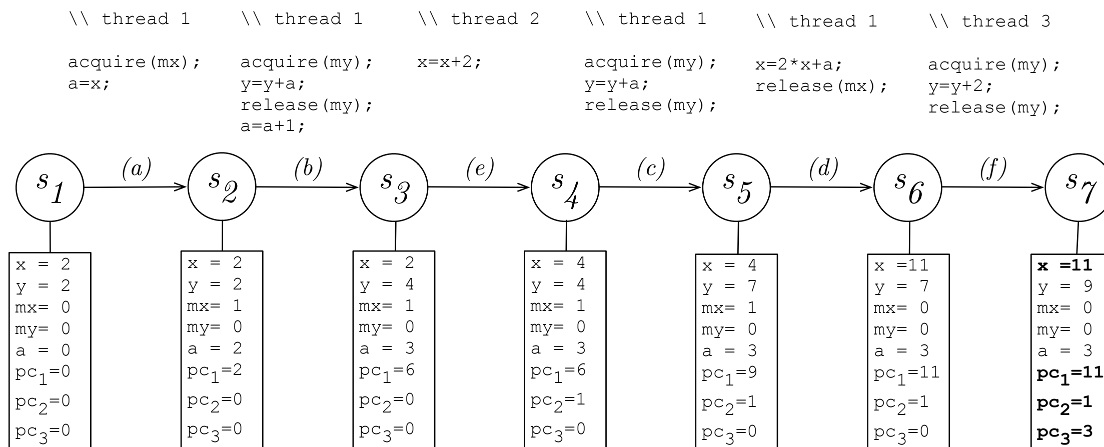


Figure 3.4.: Counterexample execution from program $P2 - 1$.

the variable x holds the value 11 at the final state s_7, i.e., the program is not safe according to the specification.

Program P3-1 Now consider a second deviation from the program P1-1 where `thread-1` and `thread-2` remain unchanged but `thread-3` is extended by the assignment `y=2`:

```
// Thread-3

0: acquire(my);
1: y = y+2;
2: release(my);
3: y = 2;
4:
```

The unprotected write access by the new statement induces race conditions at `thread-1` on every access of variable `y` ($\mathtt{pc}_1 = \ell_3$ and $\mathtt{pc}_1 = \ell_7$). However, transaction inference only returns one additional transaction for `thread-3` when compared to P1-1:

(a)	`thread-1` $\{\ell_0 - \ell_6\}$	(d)	`thread-3` $\{\ell_0 - \ell_3\}$
(b)	`thread-1` $\{\ell_6 - \ell_{11}\}$	(e)	`thread-3` $\{\ell_3 - \ell_4\}$
(c)	`thread-2` $\{\ell_0 - \ell_3\}$		

Our method detects that all statements in `thread-1` accessing variable y are surrounded by transitions that commute with transitions from other threads. Thus, the transitions are still composed to two transaction summaries.

Transaction summarization significantly reduces the number of interleavings to be explored. See chapter 5 for comparison between our implementation and state-of-the-art verifiers showing how verification is able to cope with versions of programs P1-1, P2-1, and P3-1 where we vary the number x of statements from each transaction to obtain P1-x, P2-x, and P3-x.

3.3. Transaction Inference

To minimize the number of explored interleavings and to maximize reuse of respective summaries, it is desirable to define transactions that are as large as possible. In this section, we show our approach to determine transaction boundaries encasing transition sequences that are independent from interleavings. Therefore, we use a chain of four consecutive analysis stages to split this task. Figure 3.5 illustrates the analyses to infer (1) the held locks of each program location, (2) the mover types for every transition relation, (3) the transaction phase for every program location, and the final transaction boundaries. Every stage uses the given transition relations and the result from the predecessor (if available) for its analysis.

3.3.1. Locks-Held Information

We use simple data flow analysis to compute $lh_i(\ell)$, the set of held locks for a program location ℓ of thread i. The idea is to add or remove a lock $m \in \mathit{Locks}$ whenever the program executes an $\mathit{acquire}(m)$ or $\mathit{release}(m)$ statement, respectively. For the sake of illustration, we describe the data flow problem using known terms and functions from the literature. However, the computation of $lh_i(\ell)$ information is done differently utilizing the given HSF solver, as we will describe at the end of this section.

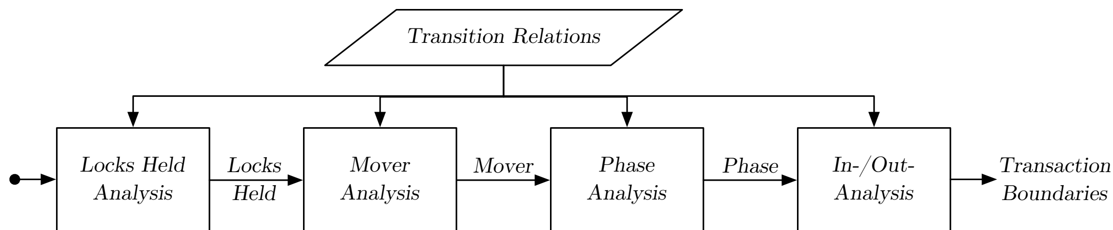


Figure 3.5.: Processing stages for transaction inference.

Data flow analysis Following the data flow approach from Nielson *et al.* [27], we define the analysis in terms of lattice theory [2]. Instead of using control flow graphs (CFGs), we rely on the described transition systems for multi-threaded programs. The data flow analysis operates in forward direction using a pair of functions that maps program locations and transition relations to the property set (*Locks*) of the complete lattice. The first function $lh_i(\ell')$ specifies the locks that are held on a program location ℓ'.

$$lh_i(\ell') = \begin{cases} \emptyset, & \text{if } init(V) \rightarrow (\mathtt{pc} = \ell') \\ \bigcap \left\{ lh_i^{exit}(step_i) \mid \exists \rho_i \in \mathcal{R}_i : \rho_i \models mv(\ell, \ell') \right\}, & \text{otherwise} \end{cases}$$

At initial program states, the function returns the empty set since no locks are held at the beginning. Otherwise, it returns the intersection of propagated locks from predecessor locations ℓ that reach location ℓ' by some transition $\rho_i \in \mathcal{R}_i$. Therefore, we use the second function $lh_i^{exit}(\rho_i)$ that returns such lock information as follows.

$$\begin{aligned} lh_i^{exit}(\rho_i) &= \left(lh_i(\ell) \setminus kill_i^{lh}(\rho_i)\right) \cup gen_i^{lh}(\rho_i) \\ &\text{where } \rho_i \rightarrow mv(\ell, \ell') \end{aligned}$$

The returned set of locks for a transition ρ_i is determined by the held locks at the beginning location ℓ and the transition semantics. If ρ_i represents a release statement of lock m, then m is subtracted from the returned set of locks. Successively, if ρ_i acquires a lock m, then m is added to the result. The used helper functions gen_i^{lh} and $kill_i^{lh}$ can be statically computed according to the following definition.

$$gen_i^{lh}(\rho_i) = \begin{cases} m, & \text{if } \rho_i \rightarrow acq_i \wedge W_i(m), \text{ for each } m \in Locks \\ \emptyset, & \text{otherwise} \end{cases}$$

$$kill_i^{lh}(\rho_i) = \begin{cases} m, & \text{if } \rho_i \rightarrow rel_i \wedge W_i(m), \text{ for each } m \in Locks \\ \emptyset, & \text{otherwise} \end{cases}$$

The used predicates $W_i(m)$ (indicating that m is written) allows us to identify acquired and released locks $m \in Locks$ at acquire and release statements, respectively.

One can easily prove distributivity of the shown data flow analysis. Hence, it is possible to use the reasonable meet over all paths (MOP) approach for computation. Simplified,

[2] The complete lattice of our analysis is $L = 2^{Locks}$ and it is partially ordered by superset inclusion $\supseteq$, the bottom element $\bot$ is the whole set *Locks*, and the top element $\top$ is the empty set $\emptyset$.

MOP intersects the sets of held locks at a location ℓ resulting from different execution paths of the program. However, we utilize the HSF algorithm that elegantly allows us to obtain held locks for each reachable program location. Even though the intuition behind both approaches (MOP and HSF) is the same.

Computation by HSF Technically, the analysis problem is represented as Horn clauses over the first order theory $\mathcal{T}_{LI}$ which we compute by the assumed HSF solver. The set of clauses over queries $\mathcal{Q}_1 := \{R_1(V), \ldots, R_N(V)\}$ models reachable states in computations of each thread i.

$$HC_1 := \{ \quad \begin{array}{ll} init(V) & \rightarrow R_i(V), \\ R_i(V) \wedge step_i(V_G, V_i, V'_G, V'_i) \wedge step_{\bar{i}}^{=}(V) & \rightarrow R_i(V') \quad \} \end{array}$$

To compute $lh_i(\ell)$ information for a program location ℓ we initialize the predicate function with predicates over program counter and lock variables as follows.

$$Preds_1(R_i(V)) := \{\mathtt{pc}_i = \ell \mid \ell \in \mathcal{L}_i\} \cup \{m = 0, m = 1 \mid m \in Locks\}$$

Using such an initialization leads to an invariant that only contains valuations of variables for program locations and locks. An invocation of the HSF solver gives the solution $\Sigma_1 := \text{HSF}(HC_1, \mathcal{Q}_1, Preds_1)$, which finally enables us to extract locks held information as follows.

$$lh_i(\ell) := \{m \in Locks \mid \forall V : \Sigma_1(R_i(V)) \wedge \mathtt{pc}_i = \ell \rightarrow m = 1\}$$

Note that the $\forall$-quantifier constraints that a specific location ℓ only holds a lock m if every successor state leading to ℓ satisfies $m = 1$.

Example When we apply our analysis on the first thread of example 3.2, we get the following invariant.

$$\begin{array}{rl} \Sigma_1(R_1(V)) := (& pc_1 \in \{\ell_0, \ell_{11}\} \wedge mx = 0 \wedge my = 0 \vee \\ & pc_1 \in \{\ell_1, \ell_2, \ell_5, \ell_6, \ell_9, \ell_{10}\} \wedge mx = 1 \wedge my = 0 \vee \\ & pc_1 \in \{\ell_3, \ell_4, \ell_7, \ell_8\} \wedge mx = 1 \wedge my = 1) \end{array}$$

The program locations are divided into three cases, namely when both lock variables `mx` and `my` are 0, only `mx` is 1, and both locks are 1. For example, the locks held information derived at program location ℓ_3 is $(mx = 1 \wedge my = 1)$. □

3.3.2. Mover Information

Multi-threaded programs contain transactions because of the presence of transitions that are right or left mover. For this analysis, we modify our used representation for transition relations from $step_i(V_G, V_i, V'_G, V'_i)$ to $\rho_i(\mathtt{pc}_i, \mathtt{pc}'_i, W_i, R_i)$. These assertions consider only thread-local program location variables $\mathtt{pc}_i, \mathtt{pc}'_i$ and two new sets W_i, R_i describing global variables that are written and read in the represented statement. Note that this information can efficiently be obtained by static analysis.

Our analysis partitions these new transition relations into four mover types by means of previously calculated lock held sets. Each mover type is represented by a boolean function defined over pairs of program locations: $rm_i(\texttt{pc}_i, \texttt{pc}'_i)$, $lm_i(\texttt{pc}_i, \texttt{pc}'_i)$, $bm_i(\texttt{pc}_i, \texttt{pc}'_i)$, and $nm_i(\texttt{pc}_i, \texttt{pc}'_i)$. If a function returns true for some program locations (ℓ, ℓ'), it determines a transition $\rho_i(\texttt{pc}_i, \texttt{pc}'_i, W_i, R_i)$ whenever $\texttt{pc}_i = \ell$ and $\texttt{pc}'_i = \ell'$.

Following the theory of reduction[22] and its application in type systems [14] and model checking [30], we use the following semantics for each function:

- $rm_i(\texttt{pc}_i, \texttt{pc}'_i)$ determines right mover transitions $\rho_i(\texttt{pc}_i, \texttt{pc}'_i, W_i, R_i)$.
  It returns true if the transition represents an acquire statement.

- $lm_i(\texttt{pc}_i, \texttt{pc}'_i)$ determines left mover transitions and, hence, returns true if the respective transition $\rho_i(\texttt{pc}_i, \texttt{pc}'_i, W_i, R_i)$ represents a release statement.

- $bm_i(\texttt{pc}_i, \texttt{pc}'_i)$ determines both mover transitions. It returns true if the respective transition $\rho_i(\texttt{pc}_i, \texttt{pc}'_i, W_i, R_i)$ satisfies one of the following two conditions. Firstly, the transition only accesses local variables. Secondly, if a global variable $\texttt{x}$ is accessed from ρ_i then there is no other transition $\rho_j(\texttt{pc}_j, \texttt{pc}'_j, W_j, R_j)$ of a different thread j that accesses $\texttt{x}$ when the intersection of common locks is empty and at least one access is a write operation.

- $nm_i(\texttt{pc}_i, \texttt{pc}'_i)$ determines non-mover transitions. It returns true if both the respective transition $\rho_i(\texttt{pc}_i, \texttt{pc}'_i, W_i, R_i)$ and some transition from a different thread j accesses some common global variable $\texttt{x}$ when the intersection of common locks is empty and at least one access is a write operation.

The constraint representation for the mover function follows.

$$
\begin{aligned}
rm_i(pc_i, pc'_i) \quad &:= \quad acq_i(pc_i, pc'_i), && \text{for } i \in 1..N \\
lm_i(pc_i, pc'_i) \quad &:= \quad rel_i(pc_i, pc'_i), && \text{for } i \in 1..N \\
nm_i(pc_i, pc'_i) \quad &:= \quad \rho_i(pc_i, pc'_i, W_i, R_i) \wedge \rho_j(pc_j, pc'_j, W_j, R_j) \wedge \\
&\quad\;\; \big(W_i \cap W_j \neq \emptyset \vee W_i \cap R_j \neq \emptyset \vee R_i \cap W_j \neq \emptyset\big) \wedge \\
&\quad\;\; \big(lh_i(pc_i) \cap lh_j(pc_j) = \emptyset\big), && \text{for } i \text{, exists } j \neq i \in 1..N \\
bm_i(pc_i, pc'_i) \quad &:= \quad \rho_i(pc_i, pc'_i, W_i, R_i) \wedge \Bigg(\bigwedge_{j \in 1..N \setminus \{i\}} \rho_j(pc_j, pc'_j, W_j, R_j) \wedge \\
&\quad\;\; \Big(\big(W_i \cap W_j = \emptyset \wedge W_i \cap R_j = \emptyset \wedge R_i \cap W_j = \emptyset\big) \vee \\
&\quad\;\; \big(lh_i(pc_i) \cap lh_j(pc_j) \neq \emptyset\big) \Big) \Bigg), && \text{for } i \in 1..N
\end{aligned}
$$

Although conservative, these checks may spuriously declare both mover transitions as non-mover since the computation is not flow-sensitive, e.g., program states are not explored by following transition relations (like in reachable state analysis). Consequently, our approach may introduce thread interleavings that are not possible in a real scheduling due to some unreachable control locations.

Example (cont.) The first thread of example 3.2 is again used for illustration. We assume that the program transitions are partitioned into acquire transitions $acq_1(pc_1, pc'_1)$, release transitions $rel_1(pc_1, pc'_1)$, and other transitions $\rho_1(pc_1, pc'_1, W_1, R_1)$. The resulting mover information is as follows.

$$\begin{aligned}
rm_1 &:= \{(\ell_0, \ell_1), (\ell_2, \ell_3), (\ell_6, \ell_7)\} \\
lm_1 &:= \{(\ell_4, \ell_5), (\ell_8, \ell_9), (\ell_{10}, \ell_{11})\} \\
nm_1 &:= \emptyset \\
bm_1 &:= \{(\ell_1, \ell_2), (\ell_3, \ell_4), (\ell_5, \ell_6), (\ell_7, \ell_8), (\ell_9, \ell_{10})\}
\end{aligned}$$

□

3.3.3. Phase Information

In order to determine the transaction boundaries we use a boolean phase variable p for each thread i. It states whether a transition within a transaction is in the *pre-commit* phase ($p = 1$) or *post-commit* phase ($p = 0$). The phase variable is 1 at the beginning of each thread and remains 1 as long as thread i is in the right mover part of a transaction. Note that both mover transitions are also right movers and, hence, extend the pre-commit phase as long as no (pure) left mover or non-mover occur.

Calculating the phase information is again done utilizing the HSF solver. Therefore, we define the following set of Horn clauses over queries $\mathcal{Q}_2 := \{Ph_1(V, p), \ldots, Ph_N(V, p)\}$ that represents reachable states for each thread i extended by the phase variable p.

$$\begin{aligned}
HC_2 := \{ & init(V) && \rightarrow Ph_i(V,\ 1), \\
& Ph_i(V, p) \wedge step_i(V_G, V_i, V'_G, V'_i) \wedge rm_i(pc_i, pc'_i) && \rightarrow Ph_i(V', 1), \\
& Ph_i(V, p) \wedge step_i(V_G, V_i, V'_G, V'_i) \wedge (lm_i(pc_i, pc'_i) \vee nm_i(pc_i, pc'_i)) && \rightarrow Ph_i(V', 0), \\
& Ph_i(V, p) \wedge step_i(V_G, V_i, V'_G, V'_i) \wedge bm_i(pc_i, pc'_i) && \rightarrow Ph_i(V', p)\}
\end{aligned}$$

The predicate function is initialized with predicates over program counter and phase variables as follows.

$$Preds_2(Ph_i(V, p)) := \{pc_i = \ell_i \mid \ell_i \in \mathcal{L}_i\} \cup \{p = 0, p = 1\}$$

We invoke the HSF solver to get the solutions $\Sigma_2 := \text{HSF}(HC_2, \mathcal{Q}_2, Preds_2)$.

Example (cont.) When applied to our ongoing example, the solution corresponding to the first thread indicates two pre-commit phases by the following disjunction.

$$\begin{aligned}
\Sigma_2(Ph_1(V, p)) := (& p = 1 \wedge pc_1 \in \{\ell_0, \ell_1, \ell_2, \ell_3, \ell_4\} \vee \\
& p = 0 \wedge pc_1 \in \{\ell_5\} \vee \\
& p = 1 \wedge pc_1 \in \{\ell_6, \ell_7, \ell_8\} \vee \\
& p = 0 \wedge pc_1 \in \{\ell_9, \ell_{10}, \ell_{11}\})
\end{aligned}$$

Exemplary, the state at location ℓ_3 is in a pre-commit phase and the state at location $\ell 5$ is either in a post-commit phase or outside a transaction. □

3.3.4. Transaction Boundaries

A program location is inside a transaction if it is not contained in an initial state and either the phase variable is 1 (pre-commit phase) or all enabled transitions from this state are left movers. Similarly, a program location is outside a transaction if it is either contained in an initial state or the phase variable is 0 and there is a enabled transition that is no left mover. We represent transaction boundaries for each thread i by using two boolean functions defined over program locations. The function $In_i(pc_i)$ holds when pc_i is a location inside a transaction, while $Out_i(pc_i)$ holds when pc_i is a location outside any transaction. The respective constraints extract this transaction information from the solution Σ_2 as follows.

$$\begin{aligned}
In_i(pc_i) \;&:= \Sigma_2(Ph_i(V,p)) \wedge \neg init(V) \wedge \\
&\quad\; (p = 1 \;\vee p = 0 \wedge \forall pc'_i : lm_i(pc_i, pc'_i) \vee bm_i(pc_i, pc'_i)) \\
Out_i(pc_i) &:= \neg In_i(pc_i)
\end{aligned}$$

Given the transaction information, we partition the transition relation of a thread depending on the program locations of the start state and the target state. $step_in_i$, $step_inout_i$, and $step_outout_i$ are relations that represent transitions inside, into, and outside transactions, respectively.

$$\begin{aligned}
step_in_i(V_G, V_i, V'_G, V'_i) \quad &:= \quad step_i(V_G, V_i, V'_G, V'_i) \wedge In_i(pc'_i) \;\wedge \neg error(V') \\
step_inout_i(V_G, V_i, V'_G, V'_i) \quad &:= \quad step_i(V_G, V_i, V'_G, V'_i) \wedge In_i(pc_i) \;\wedge \big(Out_i(pc'_i) \vee error(V')\big) \\
step_outout_i(V_G, V_i, V'_G, V'_i) \quad &:= \quad step_i(V_G, V_i, V'_G, V'_i) \wedge Out_i(pc_i) \wedge \big(Out_i(pc'_i) \vee error(V')\big)
\end{aligned}$$

Note that we only declare transition relations as inside transactions ($step_in_i$ transitions) if their primed variables do not intersect with the error states. This is a necessary step to ensure soundness since such error states may contain program locations within transactions from two or more threads. As transactions are summarized during verification, program locations inside multiple transactions are never explored. Hence, we have to split the transactions on potential error states to enable interleaving on the respective program locations.

Example (cont.) We obtain the following results for all threads.

$$\begin{aligned}
&Out_1 := \{\ell_0, \ell_6, \ell_{11}\} \quad && In_1 := \{\ell_1, \ell_2, \ell_3, \ell_4, \ell_5, \ell_7, \ell_8, \ell_9, \ell_{10}\} \\
&Out_2 := \{\ell_0, \ell_3\} && In_2 := \{\ell_1, \ell_2\} \\
&Out_3 := \{\ell_0, \ell_3\} && In_3 := \{\ell_1, \ell_2\}
\end{aligned}$$

□

3.4. Proof Rule

In this section, we present our proof rule that combines thread-modular reasoning inside transactions with Owicki-Gries reasoning outside transactions. The proof rule lists conditions for each thread $i \in 1..N$ over:

- N queries $P_i(V_G, V_i, V'_G, V'_i)$ representing path edge information.

- N queries $Summ_i(V_G, V_i, V'_G, V'_i)$ representing transaction summaries.
- N queries $R_i(V)$ representing state reachability information outside transaction boundaries.

As before, V denotes the tuple of variables $(V_G, V_1, \dots, V_N)$. All the clauses from (1) to (10) are replicated for each $i \in 1..N$:

$$\begin{array}{ll}
(1) & init(V) \rightarrow R_i(V) \\[1em]
(2) & R_i(V) \wedge step_in_i(V_G, V_i, V'_G, V'_i) \rightarrow P_i(V_G, V_i, V'_G, V'_i) \\
(3) & P_i(V_G, V_i, V'_G, V'_i) \wedge step_in_i(V'_G, V'_i, V''_G, V''_i) \rightarrow P_i(V_G, V_i, V''_G, V''_i) \\
(4) & P_i(V_G, V_i, V'_G, V'_i) \wedge step_inout_i(V'_G, V'_i, V''_G, V''_i) \rightarrow Summ_i(V_G, V_i, V''_G, V''_i) \\[1em]
(5) & R_i(V) \wedge step_outout_i(V_G, V_i, V'_G, V'_i) \wedge step^{=}_i(V, V') \rightarrow R_i(V') \\
(6) & R_i(V) \wedge Summ_i(V_G, V_i, V'_G, V'_i) \wedge step^{=}_i(V, V') \rightarrow R_i(V') \\
(7) & R_i(V) \wedge R_j(V) \wedge step_outout_j(V_G, V_j, V'_G, V'_j) \wedge step^{=}_j(V, V') \rightarrow R_i(V') \\
 & \qquad\qquad\qquad \text{for } j \in 1..N \setminus \{i\} \\
(8) & R_i(V) \wedge R_j(V) \wedge Summ_j(V_G, V_j, V'_G, V'_j) \wedge step^{=}_j(V, V') \rightarrow R_i(V') \\
 & \qquad\qquad\qquad \text{for } j \in 1..N \setminus \{i\} \\[1em]
(9) & R_i(V) \wedge error(V) \rightarrow false
\end{array}$$

The clause (1) considers initial states as reachable states. The clauses (2), (3) and (4) do thread-modular reasoning inside transaction boundaries. The clauses (5), (6), (7) and (8) perform Owicki-Gries reasoning outside transactions. The constraint $step^{=}_i(V, V')$ from the body of clauses (5) and (6) requires that variables local to all threads except i are unchanged. The last clause (9) checks that states reachable outside transactions do not intersect the error states.

We let HC_3 refer to a set containing the above clauses. These clauses are defined over $\mathcal{Q}_3$ representing queries for reachability, path-edges, and transaction summaries as follows.

$$\begin{array}{ll}
\mathcal{Q}_3 := \{ & R_1(V), \dots, R_N(V), \\
 & P_1(V_G, V_1, V'_G, V'_1), \dots, P_N(V_G, V_N, V'_G, V'_N) \\
 & Summ_1(V_G, V_1, V'_G, V'_1), \dots, Summ_N(V_G, V_N, V'_G, V'_N) \quad \}
\end{array}$$

The predicate function $Preds_3 := \lambda q \in \mathcal{Q}_3.\emptyset$ is initialized with the empty set to consider every state utilizing the HSF solver: $\Sigma_3 := \text{HSF}(HC_3, \mathcal{Q}_3, Preds_3)$. The existence of a solution Σ_3 guarantees that the program given in clause-form is safe (see Section 3.5).

Example (cont.) We now show the obtained invariant for reachable states of our ongoing example. Remember the used error constraint $(\mathtt{x} = 11 \wedge \mathtt{pc}_1 = \ell_{11} \wedge \mathtt{pc}_2 = \ell_3 \wedge \mathtt{pc}_3 = \ell_3)$ for safety checking. Since this constraint only checks for the last location of each thread,

we do not present results for the path edges.

$$
\begin{aligned}
\Sigma_3(R_1(V)) := \quad & \big(\mathtt{pc}_1 = \ell_0 \ \wedge \mathtt{mx} = 0 \wedge \mathtt{pc}_3 \in \{\ell_0, \ell_3\} \wedge (\ \mathtt{pc}_2 = \ell_0 \wedge \mathtt{x} = 2 \ \vee \\
& \qquad \mathtt{pc}_2 = \ell_3 \wedge \mathtt{x} = 4\big) \vee \\
& \big(\mathtt{pc}_1 = \ell_6 \ \wedge \mathtt{mx} = 1 \wedge \mathtt{pc}_3 \in \{\ell_0, \ell_3\} \wedge (\ \mathtt{pc}_2 \in \{\ell_0, \ell_3\} \wedge \mathtt{x} \leq 7 \wedge 2\mathtt{x} + \mathtt{a} \leq 7 \ \vee \\
& \qquad \mathtt{pc}_2 \in \{\ell_0, \ell_3\} \wedge \mathtt{x} \leq 7 \wedge 2\mathtt{x} + \mathtt{a} \geq 13)\ \big) \vee \\
& \big(\mathtt{pc}_1 = \ell_{11} \wedge \mathtt{mx} = 0 \wedge \mathtt{pc}_3 \in \{\ell_0, \ell_3\} \wedge (\ \mathtt{pc}_2 \in \{\ell_0, \ell_3\} \wedge \mathtt{x} \leq 7 \ \vee \\
& \qquad \mathtt{pc}_2 = \ell_3 \wedge \mathtt{x} \leq 9 \ \vee \\
& \qquad \mathtt{pc}_2 \in \{\ell_0, \ell_3\} \wedge \mathtt{x} \geq 13 \wedge 2\mathtt{x} + \mathtt{a} \geq 13)\big)
\end{aligned}
$$

The solution for $R_1(V)$ is divided into three outer disjuncts representing locations outside transactions ($\{\ell_0, \ell_6, \ell_{11}\}$). Apparently, the location of the third thread has no influence on the result since this thread does not manipulate variable $\mathtt{x}$, which is contained in $error(V)$. Regarding the cases for $\mathtt{pc}_1 = \ell_0$ and $\mathtt{pc}_1 = \ell_6$, the two inner disjuncts represent executions where either thread 2 has not yet started or has executes before thread 1, respectively. The case for $\mathtt{pc}_1 = \ell_{11}$ has three inner disjuncts for executions where thread 2 has not yet started, finished execution after thread 1, or executes before thread 1, respectively.

$$
\begin{aligned}
\Sigma_3(R_2(V)) := \quad & \big(\mathtt{pc}_2 = \ell_0 \ \wedge \mathtt{pc}_3 \in \{\ell_0, \ell_3\} \wedge (\ \mathtt{mx} = 0 \wedge \mathtt{x} \leq 7 \ \vee \\
& \qquad \mathtt{mx} = 0 \wedge \mathtt{x} \geq 13 \wedge 2\mathtt{x} + \mathtt{a} \geq 13 \ \vee \\
& \qquad \mathtt{mx} = 1 \wedge \mathtt{pc}_1 = 6 \wedge \mathtt{x} \leq 7\ \big) \vee \\
& \big(\mathtt{pc}_2 = \ell_3 \ \wedge \mathtt{pc}_3 \in \{\ell_0, \ell_3\} \wedge (\ \mathtt{mx} = 0 \wedge \mathtt{x} \leq 9 \ \vee \\
& \qquad \mathtt{mx} = 0 \wedge \mathtt{x} \geq 13 \wedge 2\mathtt{x} + \mathtt{a} \geq 13 \ \vee \\
& \qquad \mathtt{mx} = 1 \wedge \mathtt{pc}_1 = 6 \wedge \mathtt{x} \leq 7\ )\big)
\end{aligned}
$$

As before, the invariant of $R_2(V)$ is divided by outer disjuncts representing locations outside transactions ($\{\ell_0, \ell_3\}$). The inner disjuncts separate cases where the first thread has already finished its execution or is at location ℓ_6.

$$
\begin{aligned}
\Sigma_3(R_3(V)) := \quad \mathtt{pc}_3 \in \{\ell_0, \ell_3\} \ \wedge (\ & \mathtt{pc}_1 = \ell_0 \wedge \mathtt{pc}_2 = \ell_0 \wedge \mathtt{mx} = 0 \wedge \mathtt{x} = 2 \ \vee \\
& \mathtt{pc}_1 = \ell_0 \wedge \mathtt{pc}_2 = \ell_3 \wedge \mathtt{mx} = 0 \wedge \mathtt{x} = 4 \ \vee \\
& \mathtt{pc}_1 = \ell_6 \wedge \mathtt{pc}_2 \in \{\ell_0, \ell_3\} \wedge \mathtt{mx} = 1 \wedge \mathtt{x} \leq 7 \ \vee \\
& \mathtt{pc}_1 = \ell_{11} \wedge \mathtt{pc}_2 = \ell_0 \wedge \mathtt{mx} = 0 \wedge \mathtt{x} \leq 7 \ \vee \\
& \mathtt{pc}_1 = \ell_{11} \wedge \mathtt{pc}_2 = \ell_3 \wedge \mathtt{mx} = 0 \wedge \mathtt{x} \leq 9 \ \vee \\
& \mathtt{pc}_1 = \ell_{11} \wedge \mathtt{pc}_2 \in \{\ell_0, \ell_3\} \wedge \mathtt{mx} = 0 \wedge 2\mathtt{x} + \mathtt{a} \geq 13 \wedge \mathtt{x} \geq 13\ )
\end{aligned}
$$

Our proof rule induces solutions for $R_3(V)$ that do not depend on the location of thred 3 there is no access of variable $\mathtt{x}$. The disjuncts represent already mentioned interleaving orders of thread 1 and thread 2.

3.5. Soundness Proof (Sketch)

We now show the correctness of our proof rules, i.e., if the program may reach a state that intersects with $error(V)$, our verification approach explores this state. We rely on the fact that erroneous states are strictly located outside transactions since transition relations are disjunctively partitioned into $step_in$, $step_inout$, and $step_outout$, and according to their definition only the last two transition types may intersect with error states. Additionally we assume that every transaction will finish its computation in a finite number of transition steps n.

Theorem 3.1 (Summarization)**.** *Let $s_0, \ldots, s$ be a sequence of states that result from some transition applications with $s_0 \models init(V)$ and $s \models \varphi_{reach} \wedge error(V)$. Then, there is also a state $s' \models R_i \wedge error(V)$ that is reachable by a sequence of transactions starting from s_0.*

Proof. All occurring states up to s are disjunctively either states outside transactions, states in a pre-commit phase of a transaction, or states in a post-commit phase of a transaction. This can be proven by the fact that states in the pre-commit phase have phase value 1 and states in the post-commit phase have phase value 0. States outside transactions are by definition disjunct.

Additionally it is not possible to have a sequence of two consecutive states where the first state is in a post-commit phase and the second state is in a pre-commit phase. This can be shown by contradiction because this situation needs a phase variable that goes from 0 to 1 and, hence, needs a right mover transition in the post-commit phase, which is not possible.

Now we can show our assertion. Since all states are disjunctively partitioned and states in a pre-commit phase precede states in the post-commit phase, we can transform the sequence of transitions that lead to s in an equivalent sequence that leads to s'. We do this by appropriately right commuting transitions leading to states in the pre-commit phase and left commuting transitions leading to states in the post-commit phase so that: (1) for every thread i no transition from a different thread j occurs in the middle of a transaction from thread i, and (2) the (optional) committing transitions in the resulting transactions are executed in the same order as in the original transition sequence. From the properties of right and left movers we get that the new sequence also leads to a state s' that intersects with the error states. □

Theorem 3.2 (Proof rule)**.** *Let $R_1, \ldots, R_N$, $P_1, \ldots, P_N$, and $Summ_1, \ldots, Summ_N$ satisfy the clauses from (1) to (9). We prove safety for each thread i by showing that $\mathcal{A} := s \models R_i$ for each reachable error state $s \models \varphi_{reach} \wedge error(V)$.*

Proof. We use induction over the length k of a shortest computation segment $s_1, \ldots, s_k$ such that $s_1 \models init$ and $s_k = s$. For the base case $k = 1$, $\mathcal{A}$ holds because of clause (1). For the induction step we assume that $\mathcal{A}$ holds for states reachable in $k \geq 1$ and prove this for their successor states. Regarding to theorem 3.1 we only have to consider transitions outside transactions and summary relations. Clauses (3) and (4) do transition composition to get $Summ_i$ relations.

Let $s_k \models R_i$ for each thread i. If s_k does not have any successor state, i.e., $\neg(\exists s_{k+1} : (s_k, s_{k+1}) \models step_i)$, we do not have to consider any further state. Otherwise, we choose a successor state s_{k+1} using one of the available transitions.

- A transition that is outside any transaction, i.e., $(s_k, s_{k+1}) \models step_outout_i$. From clause (5) follows that $s_{k+1} \models R_i$ so that $\mathcal{A}$ holds.
- A transition that represents a transaction summarization, i.e., $(s_k, s_{k+1}) \models Summ_i$. From clause (6) follows that $s_{k+1} \models R_i$ so that $\mathcal{A}$ holds.

To show that $s_{k+1} \models R_j$ for each thread $j \in 1..N \setminus \{i\}$ we rely on the clauses (7) and (8) in the same manner as we did with clauses (5) and (6).

□

4. Model Checking with May-Happen-in-Parallel Information

In this chapter, we introduce may-happen-in-parallel (MHP) information for facilitating software verification of multi-threaded programs. MHP returns for every pair of statements whether it may happen in parallel during some execution of the program. Such information is valuable for techniques like program optimization, debugging, program understanding tools, and detecting synchronization anomalies as race conditions or deadlock situations. Regarding verification, MHP information can be elegantly used to reduce the number of program states that have to be inspected and, thus, tackles the mentioned state-explosion problem.

Formally, an MHP relation $mhp : \mathcal{L}_a \times \mathcal{L}_b$ contains all pairs of program locations from threads a and b, respectively, that may happen in parallel, i.e., if a pair of control locations $\ell_p \in \mathcal{L}_a$ and $\ell_q \in \mathcal{L}_b$ may happen in parallel, then there is a state $s \models \varphi_{reach}$ such that $s \rightarrow (\texttt{pc}_a = \ell_p) \wedge (\texttt{pc}_b = \ell_q)$.

The relation $mhp_{a,b}(\texttt{pc}_a, \texttt{pc}_b) = \{(\ell_p, \ell_q), (\ell_r, *)\}$ states that ℓ_p of thread a may execute in parallel to ℓ_q of thread b whereas location ℓ_r may execute in parallel to any location from thread b.

In the following section we illustrate the intuition behind MHP information by a small example. We describe an existing data flow analysis that is able to calculate MHP information in a worst-case-time bound of $\mathcal{O}(S^3)$ where S is the number of program statements. Finally, we present our approach for combining MHP with existing proof rules for the verification of multi-threaded software.

4.1. Illustration

We again assume a programming model that is based on shared-memory and locks for thread-synchronization that are accessed only with respective acquire and release statements. Additionally, we consider dynamic creation and joining of threads by *start* and *join* statements, respectively. Other primitives that may influence MHP (e.g., signals, barriers, atomic operations, etc.) are not considered for the sake of simplicity (our examples are lacking such features).

Example Let us consider the example program MHP-1 from Figure 4.1. It consists of two threads where `t1` is meant as the main thread and `t2` represents a worker thread. Thus, `t1` starts and joins `t2` by using explicit statements at program locations ℓ_0 and ℓ_4, respectively. Both threads access the global variable `x` while being protected by the lock `mx`. We assume the following MHP information (in Section 4.2, we show an algorithm to

```
int x=0, mx=0;

          // Thread t1                    // Thread t2

          0: start(t2);                   6: acquire(mx);
          1: acquire(mx);                 7: x = x+x;
          2: x=1;                         8: release(mx);
          3: release(mx);                 9:
          4: join(t2);
          5:
```

Figure 4.1.: Program MHP-1 consisting of two threads.

obtain this information):

$$mhp_{1,2}(\texttt{pc}_1, \texttt{pc}_2) = \{(\ell_1, *), (*, \ell_6), (*, \ell_9), (\ell_4, *)\}$$

$mhp_{1,2}(\texttt{pc}_1, \texttt{pc}_2)$ is a relation describing MHP information of threads `t1` and `t2`, respectively. The symbol $*$ describes "any" control location of the corresponding thread. Since `t1` starts `t2` at location ℓ_0, the statements of `t2` may not execute before ℓ_1, i.e., there is no $(\ell_0, *)$ in the set. Program locations ℓ_1 and ℓ_4 are not protected by the lock `mx` and, hence, may execute in parallel with any statement of `t2` (since `t2` is created but not joined). The locations ℓ_2 and ℓ_3 are protected by `mx`, which encapsulates them from `t2` at locations ℓ_7 and ℓ_8 since these are also protected by `mx`. Finally, there is no $(\ell_5, *)$ in the MHP set because `t2` is joined from `t1` after ℓ_4. Note that the MHP relation is symmetric, i.e., $\forall \ell_a \in \mathcal{L}_1, \forall \ell_b \in \mathcal{L}_2 : (\ell_a, \ell_b) \in mhp \iff (\ell_b, \ell_a) \in mhp$.

4.2. Dataflow Analysis for MHP

Now we describe a slightly modified version of a state-of-the-art data flow analysis proposed by Naumovich *et al.* [26], without considering signal handling. In general, calculating precise MHP information for recursive programs with dynamic thread creation is undecidable [32]. However, some papers show that practical MHP analysis is possible by conservative overapproximations [9, 21, 25, 26, 5, 1]. Even if the original method is constructed for the Java concurrency model, it is easily adaptable to our used programming model. The worst case time bound of the applied algorithm remains $\mathcal{O}(S^3)$ with S representing the number of program statements.

4.2.1. Parallel Execution Graph

The dataflow algorithm is based on a so-called *parallel execution graph* (PEG) to represent concurrent programs. This graph is constructed on inter-threaded control flow graphs (CFGs) combined with special edges for thread start and join. We use inlining for all procedures except communication methods (thread start, thread join, acquire lock, and release lock) resulting in a single PEG for all threads. Contained nodes are partitioned into two types, calls to communication methods and other statements. Call to communication

methods are labeled in the form $(object, name, caller)$ where $object$ is the object owning the method, $name$ is the label of the method, and $caller$ is the identifier of the calling thread. For convenience, we use the symbol $*$ for "any" value in the object or caller field. The labels for all other nodes simply contain the statements.

Example Figure 4.2 shows the parallel execution graph corresponding to the example from Figure 4.1. The solid edges represents local control flow within a thread whereas the dashed edges are used for thread creation/joining. Nodes with round edges are calls to communication methods, rectangles are nodes for other statements. The shaded regions include nodes that are protected by a lock.

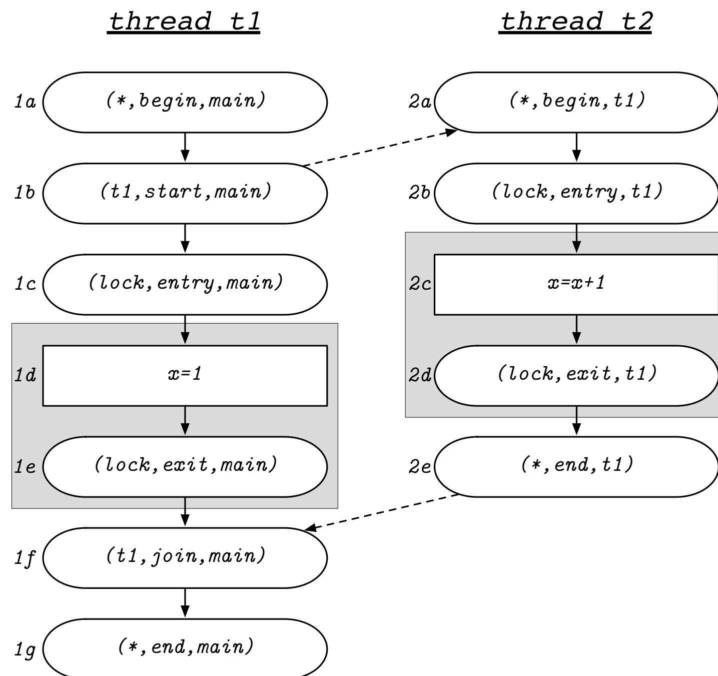


Figure 4.2.: PEG for the example from Figure 4.1.

4.2.2. Analysis

The idea behind the dataflow analysis is to infer that some nodes in the PEG may happen in parallel with others, propagate this information to successor nodes, and iterate until a fixpoint is reached. The approach relies on two sets for each node n. The first set $MHP(n)$ are indeed all nodes that may happen in parallel to n. The second set $OUT(n)$ are all nodes that may happen in parallel to successors of n. Although the analysis strictly operates in forward-flow direction, there are two subtle deviation from standard data flow algorithms. Firstly, a symmetric-step is required after each iteration that inserts a node m to the set $MHP(n)$ whenever n is in $MHP(m)$. Secondly, soundness requires that the first and the last node of a thread t (i.e., $(*, begin, t)$ and $(*, end, t)$ nodes, respectively) may happen

in parallel to every node from a different thread. Otherwise non-enabled threads lead to deadlock situations.

Preliminaries Let $G = (V, E)$ be the PEG consisting of nodes V and edges E. Additionally, let $N_T(t)$ be a function that returns all nodes of a thread t; let $N_L(obj)$ be a function that returns all nodes that are protected by lock obj; let $Thread(n)$ be a function that returns the corresponding thread of node n; let $StartPred(n)$ be a function that returns respective start nodes $(t, start, *) \subseteq V$ if n is a thread begin node $(*, begin, t) \in V$, otherwise it returns the empty set $\emptyset$; and finally, let $LocalPred(n)$ be a function that returns all predecessor nodes of n. Note that all of the previous functions can be determined statically.

Computing MHP **sets** The analysis computes $MHP(n)$ sets for a node n utilizing propagated information from predecessor nodes as follows.

$$MHP(n) = MHP(n) \cup \begin{cases} \bigcup_{p \in StartPred(n)} OUT(p) \setminus N_T(Thread(n)), & \text{if } n \in (*, begin, *) \\ \bigcup_{p \in LocalPred(n)} OUT(p), & \text{otherwise} \end{cases} \tag{4.1}$$

Let us assume that n is the first node of a thread t and m is a node that starts t. Then all nodes that may run in parallel to m may also run in parallel to n (except nodes that are in t) since thread creation is non-blocking. For all other nodes than thread begin nodes the $MHP(n)$ set is the union of propagated information from all predecessor nodes.

Computing OUT **sets** The $OUT(n)$ set represents MHP information that has to be passed to successor nodes. For every node n it contains the set of nodes that currently run in parallel together with some nodes that run in parallel to the successors and without some nodes that do not run in parallel with successor nodes.

$$OUT(n) = (MHP(n) \cup gen_{mhp}(n)) \setminus kill_{mhp}(n) \tag{4.2}$$

The equation requires a $gen_{mhp}(n)$ set and a $kill_{mhp}(n)$ set. The former set is defined by the following equation.

$$gen_{mhp}(n) = \begin{cases} (*, begin, t), & \text{if } n \in (t, start, *) \\ \emptyset, & otherwise \end{cases} \tag{4.3}$$

For start nodes, $gen_{mhp}(n)$ contains the corresponding start node of the thread that is started by n. For all other nodes the set is empty. Intuitively, parallelism is only introduced by thread creation.

$$kill_{mhp}(n) = \begin{cases} N_T(t), & \text{if } n \in (t, join, *) \\ N_L(m), & \text{if } n \in (m, acquire, *) \\ \emptyset, & otherwise \end{cases} \tag{4.4}$$

If the currently inspected node n joins another thread t, then n will block until t terminates. Thus, successor nodes of n may not happen in parallel to any node of thread t, i.e., $N_T(t)$. If node n acquires a lock m, successor nodes of n may not run in parallel to other nodes that also hold m.

Fixpoint algorithm The algorithm is based on a worklist implementation, i.e., existing nodes in the worklist will be sequentially processed until the list is empty. Figure 4.3 shows the algorithm, which basically consists of three stages. Firstly, the initial stage pre-

input
  V - set of all PEG nodes
output
  $MHP(n)$ - set of nodes that may happen in parallel to n
vars
  WL - worklist with nodes
  $OUT(n)$ - set of nodes that may happen in parallel to successors of n
begin
1 $\forall n \in V$:
2    $kill_{mhp}(n) = gen_{mhp}(n) = MHP(n) = OUT(n) = \emptyset$
3   **case** $n \in (t, join, *) \Rightarrow kill_{mhp}(n) = N_T(t)$
4   **case** $n \in (m, acquire, *) \Rightarrow kill_{mhp}(n) = N_L(m)$
5   **case** $n \in (t, start, *) \Rightarrow gen_{mhp}(n) = (*, begin, t)$
6 $WL := \{n \in (t, start, *) \mid$ t is the *main* thread$\}$
7 **while** $WL \neq \emptyset$ **do**
8    $n :=$ take from WL
9    $MHP_{old}(n) = MHP(n)$
10    $OUT_{old}(n) = OUT(n)$
11   **if** $n \in (*, begin, *)$
12      $MHP(n) = MHP(n) \cup \bigcup_{p \in StartPred(n)} OUT(p) \setminus N_T(Thread(n))$
13   **else**
14      $MHP(n) = MHP(n) \cup \bigcup_{p \in LocalPred(n)} OUT(p)$
15    $OUT(n) = (MHP(n) \cup gen_{mhp}(n)) \setminus kill_{mhp}(n)$
16   **if** $MHP_{old}(n) \neq MHP(n)$ **then**
17      $\forall m \in (MHP(n) \setminus MHP_{old}(n))$:
18        $MHP(m) = MHP(m) \cup \{n\}$
19        $WL = WL \cup \{m\}$
20   **if** $OUT_{old}(n) \neq OUT(n)$:
21      $WL = WL \cup (LocalSucc(n) \cup StartSucc(n))$
22 $\forall n \in V$:
23    $MHP_{old}(n) = MHP(n)$
24   **case** $n \in (*, begin, t) \Rightarrow MHP(n) = \bigcup_{j \in 1..N \setminus \{t\}} N_T(j)$
25   **case** $n \in (*, end, *) \Rightarrow MHP(n) = \bigcup_{j \in 1..N \setminus \{t\}} N_T(j)$
26    $\forall m \in (MHP(n) \setminus MHP_{old}(n))$:
27      $MHP(m) = MHP(m) \cup \{n\}$
end

Figure 4.3.: May-happen-in-parallel algorithm.

pares gen_{mhp} and $kill_{mhp}$ sets according to equations (4.3) and (4.4), respectively. Additionally, the worklist WL is initialized with nodes from the main thread that may create other threads (see lines 1-6). These are the only nodes that enable parallelism.

# Iteration	1	2	3	4	5	6	...
WL	(1b)	(1c,2a)	(2a,1d)	(1d,2b)	(2b,2a,1e)	(2a,1e,1c,1d,2c)	...
n	1b	1c	2a	1d	2b	...	...
$MHP(n)$	∅	{2a}	{1c}	{2a}	{1c,1d}	...	...
$OUT(n)$	{2a}	{2a}	{1c}	{2a}	{1c}	...	...
Sym. Step	×	✓	×	✓	✓	...	...

Table 4.1.: Iteration sequence for example program MHP-1.

Secondly, the iteration stage computes MHP and OUT using the worklist containing all nodes that have to be investigated. Therefore, it uses the equations (4.1) and (4.2) for computation (see lines 11-15) and compares the new results with the old ones. If the $MHP(n)$ set of node n changed by a set of additional nodes M, the mentioned symmetric step adds n to the $MHP(m)$ set for all nodes $m \in M$. Also, these new nodes M are added to the worklist in order to consider successive nodes (see lines 16-19). If the $OUT(n)$ set changed, all successor nodes (from the local thread and from created threads) are added to WL (see lines 20-21).

Finally, after the algorithm finishes its iterations and the worklist is empty, the MHP sets for begin and end nodes get initialized with all nodes from different threads (see lines 22-27). Together with a consecutive symmetric step this is necessary for the correctness of the algorithm. The proofs for soundness and termination can be found in [26].

Example Table 4.1 illustrates the first six iteration steps of the MHP algorithm when applied to the program MHP-1. Each iteration step is represented as separate column beginning from the left. The rows show the content of the respective worklist WL, the current node n, its $MHP(n)$ and $OUT(n)$ sets, and a boolean flag indicating whether the current iteration performs a symmetric step.

Initially, the worklist contains the only start node from the main thread ($1b$). The algorithm picks this entry as current node and adds node $2a$ to the $OUT(1b)$ set and to the worklist (and the local predecessor node $1c$) since it is contained in $gen_{mhp}(n)$. The next iteration leads to a new entry in $MHP(1c)$ due to the OUT of its predecessor. This results in a symmetric step so that also $1c$ is added to $MHP(2a)$. The third iteration takes $2a$ as current node, but has no new node that may happen in parallel to it. Consider iteration 5 where the $MHP(2b)$ set differs from $OUT(2b)$ because the kill set ranges over all protected nodes by the lock, i.e., $kill_{mhp}(2b) = \{1d, 1e, 2c, 2d\}$.

The algorithm continues this computation until a fixpoint is reached, i.e., the worklist is empty, and the final step adds begin and end nodes to every node from a different thread. The following map presents the results where bold entries are obtained by the final step.

$$
\begin{array}{llll}
MHP = (& 1a, 1g \rightarrow \{\mathbf{2a}, \mathbf{2b}, \mathbf{2c}, \mathbf{2d}, \mathbf{2e}\}, & 1b \rightarrow \{\mathbf{2a}, \mathbf{2e}\}, \\
& 1c, 1f \rightarrow \{2a, 2b, 2c, 2d, 2e\}, & 1d, 1e \rightarrow \{2a, 2b, 2e\}, \\
& 2a, 2e \rightarrow \{\mathbf{1a}, \mathbf{1b}, 1c, 1d, 1e, 1f, \mathbf{1g}\}, & 2b \rightarrow \{\mathbf{1a}, 1c, 1d, 1e, 1f, \mathbf{1g}, \} \\
& 2c, 2d \rightarrow \{\mathbf{1a}, 1c, 1f, \mathbf{1g}\} \quad)
\end{array}
$$

4.3. Proof Rules

In this section, we present our combination of the known proof rules from section 2.5 with given MHP information. Like the original versions of monolithic and Owicki-Gries reasoning, our rules can be automated by means of the presented HSF algorithm in section 2.3. We still consider a multi-threaded program that consists of N threads as a tuple $(V, \mathit{init}, \mathcal{R}, \mathit{error})$, with V, init, and error as defined in Section 2.4.

4.3.1. Monolithic Proof Rule (MHP)

The extended monolithic proof rules list the same three conditions over a single query symbol as the clauses **CMx** from Section 2.5. However, the second clause contains additional MHP constraints in the body as follows.

$$\begin{array}{lll}
\textbf{CM1-MHP:} & \mathit{init}(V) & \rightarrow R(V) \\
\textbf{CM2-MHP:} & R(V) \wedge \mathit{step}_i(V_G, V_i, V_G', V_i') \wedge \mathit{step}_i^{=}(V, V') \wedge & \\
 & \bigwedge_{j \in 1..N \setminus \{i\}} \left(\mathit{mhp}_{i,j}(\mathtt{pc}_i, \mathtt{pc}_j) \wedge \mathit{mhp}_{i,j}(\mathtt{pc}_i', \mathtt{pc}_j) \right) & \rightarrow R(V') \\
\textbf{CM3-MHP:} & R_i(V) \wedge \mathit{error}(V) & \rightarrow \mathit{false}
\end{array}$$

As opposed to clause **CM2**, which implicitly allows interleaving of all applicable threads, our clause **CM2-MHP** constraints the possible transition relations to those related in MHP (with current and successive program location). Obviously, utilizing MHP information is straight-forward and allows a preselection of transitions.

4.3.2. Owicki-Gries Proof Rule (MHP)

We follow the same idea as above for combining the Owicki-Gries proof rule with MHP information. The query symbols $R_1(V), \ldots, R_N(V)$ are auxiliary assertions for reachable states of each thread $i \in 1..N$.

$$\begin{array}{lll}
\textbf{CO1-MHP:} & \mathit{init}(V) & \rightarrow R_i(V) \\
\textbf{CO2-MHP:} & R_i(V) \wedge \mathit{step}_i(V_G, V_i, V_G', V_i') \wedge \mathit{step}_i^{=}(V, V') \wedge & \\
 & \bigwedge_{j \in 1..N \setminus \{i\}} \left(\mathit{mhp}_{i,j}(\mathtt{pc}_i, \mathtt{pc}_j) \wedge \mathit{mhp}_{i,j}(\mathtt{pc}_i', \mathtt{pc}_j) \right) & \rightarrow R_i(V') \\
\textbf{CO3-MHP:} & R_i(V) \wedge \left(\bigvee_{j \in 1..N \setminus \{i\}} R_{j,l}(V) \wedge \mathit{step}_j(V_G, V_j, V_G', V_j') \wedge \right. & \\
 & \left. \mathit{step}_j^{=}(V, V') \ \wedge \bigwedge_{k \in 1..N \setminus \{j\}} \left(\mathit{mhp}_{j,k}(\mathtt{pc}_j, \mathtt{pc}_k) \wedge \mathit{mhp}_{j,k}(\mathtt{pc}_j', \mathtt{pc}_k) \right) \right) & \rightarrow R_i(V') \\
\textbf{CO4-MHP:} & R_i(V) \wedge \mathit{error}(V) & \rightarrow \mathit{false}
\end{array}$$

Clause **CO3-MHP** allows explicit thread interleaving and, thus, is the proper condition to utilize MHP. Like in the monolithic proof rules, we constraint possible transition relations from other threads to those related in MHP (with current and successive program locations).

Example We illustrate our monolithic proof rule by applying it on the example program from Figure 4.1 (the Owicki-Gries rule computes similar results). The transition system for MHP-1 is shown in Figure 4.4 where transition relations are labeled ρ_1 to $\rho 8$. We use an

$$
\begin{array}{lcll}
V_G & = & (\mathtt{x}, \mathtt{t}, \mathtt{mx}),\ V_1 = (\mathtt{pc}_1),\ V_2 = (\mathtt{pc}_2) & \\
init(V_G, V_1, V_2) & = & (\mathtt{pc}_1 = \ell_0 \wedge \mathtt{pc}_2 = \ell_6 \wedge \mathtt{x} = 0 \wedge \mathtt{t} = 0 \wedge \mathtt{mx} = 0) & \\
step_1(V_G, V_1, V_G', V_1') & = & (mv_1(\ell_0, \ell_1) \wedge \mathtt{t}' = 1 \wedge skp(\mathtt{x}, \mathtt{mx})) \vee & (\rho_1) \\
 & & (mv_1(\ell_1, \ell_2) \wedge \mathtt{mx} = 0 \wedge \mathtt{mx}' = 1 \wedge skp(\mathtt{x}, \mathtt{t})) \vee & (\rho_2) \\
 & & (mv_1(\ell_2, \ell_3) \wedge \mathtt{x}' = 1 \wedge skp(\mathtt{t}, \mathtt{mx})) \vee & (\rho_3) \\
 & & (mv_1(\ell_3, \ell_4) \wedge \mathtt{mx}' = 0 \wedge skp(\mathtt{x}, \mathtt{t})) \vee & (\rho_4) \\
 & & (mv_1(\ell_4, \ell_5) \wedge \mathtt{t} = 2 \wedge skp(\mathtt{x}, \mathtt{t}, \mathtt{mx})) & (\rho_5) \\
 & & & \\
step_2(V_G, V_2, V_G', V_2') & = & (mv_2(\ell_6, \ell_7) \wedge \mathtt{t} = 1 \wedge \mathtt{mx} = 0 \wedge \mathtt{mx}' = 1 \wedge skp(\mathtt{x}, \mathtt{t})) \vee & (\rho_6) \\
 & & (mv_2(\ell_7, \ell_8) \wedge \mathtt{x}' = \mathtt{x} + \mathtt{x} \wedge skp(\mathtt{t}, \mathtt{mx})) \vee & (\rho_7) \\
 & & (mv_2(\ell_8, \ell_9) \wedge \mathtt{mx}' = 0 \wedge \mathtt{t}' = 2 \wedge skp(\mathtt{x})) & (\rho_8)
\end{array}
$$

Figure 4.4.: Representation of program MHP-1 as transition system.

additional global variable `t` to represent the semantics of `start` and `join`, i.e., thread `t2` may not start before thread `t1` assigns the value 1 to variable `t` (in transition ρ_1) and the join operation of thread `t1` waits until thread `t2` assigns the value 2 to the variable `t` (in transition ρ_8). We aim to prove that the program ends with a value for variable `x` that is lower than 3, i.e., $error(V) = (\mathtt{pc}_1 = \ell_5 \wedge \mathtt{pc}_2 = \ell_9 \wedge \mathtt{x} \geq 3)$.

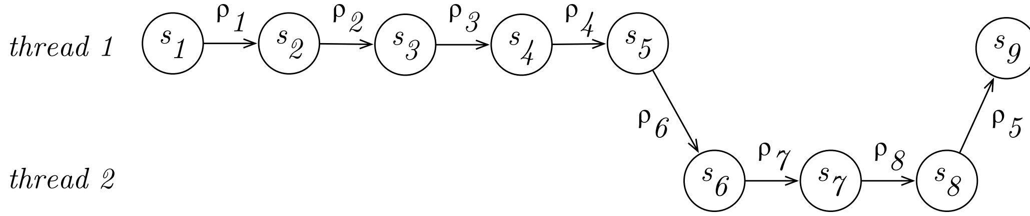


Figure 4.5.: Execution for program MHP-1.

We verify the example again utilizing the HSF solver and give the the transition system and the MHP-based proof rule as input. As mentioned in section 2.3, the algorithm computes a sequence of abstract reachability trees for every possible execution. Let us consider the execution in Figure 4.5 where thread `t2` is scheduled at the join statement of thread `t1` in program location ℓ_4. The program states that are returned by abstract computation are labeled s_1 to s_9.

In this example, we assume that the abstraction function initially only takes track of the program locations, i.e., $Preds = \{\mathtt{pc}_1 = \ell_1, \ldots, \mathtt{pc}_1 = \ell_5, \mathtt{pc}_2 = \ell_6, \ldots, \mathtt{pc}_2 = \ell_9\}$. We obtain the initial abstract state s_1 as follows.

$$s_1 := \alpha(init(V), Preds) = (\mathtt{pc}_1 = \ell_0 \wedge \mathtt{pc}_2 = \ell_6)$$

Note that formulas that do not entail any predicate from $Preds$ are abstracted, e.g., $\alpha(\mathtt{x} = 0, Preds) = true$. For the next step only transition relation ρ_1 is applicable due to the MHP constraints. The transaction ρ_6 is not applicable because $mhp_{1,2}(\mathtt{pc}_1, \mathtt{pc}_2) \rightarrow (\mathtt{pc}_1 =$

$\ell_1 \wedge \mathtt{pc}_2 = \ell_6)$ but not $mhp_{1,2}(\mathtt{pc}_1, \mathtt{pc}_2) \rightarrow (\mathtt{pc}_1 = \ell_1 \wedge \mathtt{pc}_2 = \ell_7)$. The abstract successor of s_1 is computed using the abstraction function and rule **CM2-MHP** as follows.

$$s_2 := \alpha(\rho_2, Preds) = (\mathtt{pc}_1 = \ell_1 \wedge \mathtt{pc}_2 = \ell_6)$$

The following computations up to s_9 are done similarly.

$$\begin{array}{llll} s_3 := & (\mathtt{pc}_1 = \ell_2 \wedge \mathtt{pc}_2 = \ell_6) & s_7 := & (\mathtt{pc}_1 = \ell_4 \wedge \mathtt{pc}_2 = \ell_8) \\ s_4 := & (\mathtt{pc}_1 = \ell_3 \wedge \mathtt{pc}_2 = \ell_6) & s_8 := & (\mathtt{pc}_1 = \ell_4 \wedge \mathtt{pc}_2 = \ell_9) \\ s_5 := & (\mathtt{pc}_1 = \ell_4 \wedge \mathtt{pc}_2 = \ell_6) & s_9 := & (\mathtt{pc}_1 = \ell_5 \wedge \mathtt{pc}_2 = \ell_9) \\ s_6 := & (\mathtt{pc}_1 = \ell_4 \wedge \mathtt{pc}_2 = \ell_7) & & \end{array}$$

The HSF algorithm stops the reachability computation on s_9 since it intersects with the error state. Yet, we cannot assert that the program is incorrect because abstraction was involved. Consequently, HSF checks whether the logical inference yields a solution that still violates the property clause (the one that contains the error assertion) when used without any abstraction. For our example the algorithm returns a solution and, by that, shows the spuriousness of the counterexample.

As a result, we obtain two new predicates that are used to refine the abstraction functions. This allows us to eliminate the source of spuriousness so that the same counterexample will not appear during subsequent abstract reachability computations. The new predicates are as follows.

$$Preds := Preds \cup \{(\mathrm{x} \leq 2), (\mathrm{x} + \mathrm{x} \leq 2)\}$$

When performing another iteration with the new predicates, it turns out that no abstract state intersects with the error state. Due to the used MHP information, no assertions about variables `t` or `mx` are necessary to prove safety of the program. This can be demonstrated by the following solution for reachable states that only contains states with program locations conform with the MHP set.

$$\begin{array}{ll} \big((\mathrm{x} \leq 2) \wedge (\mathrm{x}+\mathrm{x} \leq 2) \wedge & (\ \mathtt{pc}_1 = \ell_0 \wedge \mathtt{pc}_2 = \ell_6 \vee \\ & \ \ \mathtt{pc}_1 = \ell_1 \wedge \mathtt{pc}_2 = \ell_6 \vee \\ & \ \ \mathtt{pc}_1 = \ell_2 \wedge \mathtt{pc}_2 = \ell_6 \vee \\ & \ \ \mathtt{pc}_1 = \ell_3 \wedge \mathtt{pc}_2 = \ell_6 \vee \\ & \ \ \mathtt{pc}_1 = \ell_4 \wedge \mathtt{pc}_2 = \ell_6 \vee \\ & \ \ \mathtt{pc}_1 = \ell_1 \wedge \mathtt{pc}_2 = \ell_7 \vee \\ & \ \ \mathtt{pc}_1 = \ell_3 \wedge \mathtt{pc}_2 = \ell_9 \vee \\ & \ \ \mathtt{pc}_1 = \ell_4 \wedge \mathtt{pc}_2 = \ell_7) \quad \big) \vee \\ \big((\mathrm{x} \leq 2) \wedge & (\ \mathtt{pc}_1 = \ell_1 \wedge \mathtt{pc}_2 = \ell_8 \vee \\ & \ \ \mathtt{pc}_1 = \ell_1 \wedge \mathtt{pc}_2 = \ell_9 \vee \\ & \ \ \mathtt{pc}_1 = \ell_2 \wedge \mathtt{pc}_2 = \ell_9 \vee \\ & \ \ \mathtt{pc}_1 = \ell_4 \wedge \mathtt{pc}_2 = \ell_8 \vee \\ & \ \ \mathtt{pc}_1 = \ell_4 \wedge \mathtt{pc}_2 = \ell_9 \vee \\ & \ \ \mathtt{pc}_1 = \ell_5 \wedge \mathtt{pc}_2 = \ell_9) \quad \big) \end{array}$$

Part III.

Results and Conclusion

5. Experimental Results

This section gives details on the implementation of our verification methods and experimental results for our example programs to which we applied the presented proof rules. Both transaction-based reasoning and MHP-based reasoning aims to reduce interleavings rather than improving precision of the analysis results. Hence, we are mainly interested in how our tools behave regarding efficiency and scalability. To evaluate efficiency, we compare the observed runtimes with those of two state-of-the-art verifiers for multi-threaded programs. The first is Impara [34], a verifier that combines partial-order-reduction and interpolation-based reasoning [23]. The second is Threader [29], the winner of SV-COMP 2013 in the concurrency category. Checking scalability is done by means of different versions for each example program with linearly increasing program sizes. Note that this leads to an exponential increased problem size due to the additional number of interleavings.

Our methods for transaction inference and verification are implemented utilizing the HSF approach from section 2.3. Hence, each verification step is given as input declarative specification, i.e., as proof rule written in the form of Horn-like clauses. The same holds for examined example programs, which we manually translate from given C code. For the MHP-based proof rule we assume that MHP information is given and, hence, add this information by hand. Thus, the shown runtimes for our MHP approach does not contain the needed timings for obtaining this information. However, the worst-case-time bound for reasonable algorithms (cubic to the number of program statements) indicates that the complexity is rather low compared to the verification task.

Synthetic Examples We use the shown example programs from section 3.2 to demonstrate the potential of transaction summarization. Remember that accesses to shared variables are entirely protected by common locks in Program P1-1, whereas race conditions may occur in both Program P2-1 and P3-1. Such a distinction allows us to identify implications of various transaction sizes since the race conditions lead to additional non-mover transitions. Every synthetic example program contains four increment statements ($\mathtt{pc}_1 = \ell_3$, $\mathtt{pc}_1 = \ell_7$, $\mathtt{pc}_2 = \ell_1$, and $\mathtt{pc}_3 = \ell_1$). We repeat each of these statements x times to get the programs P1-x, P2-x, and P3-x for $x \in \{5, 10, 50\}$, respectively. As mentioned, this duplication allows us to quantify the scalability of our approach.

To underline the strength of MHP based verification, we present the program P4-1 that is shown in Figure 5.5. It induces a race condition due to the unprotected access of global variable `x` in `thread-3`. However, variable `x` is correctly protected by a common lock in `thread-1` and `thread-2`. This allows the threads to execute atomically relative to each other. Only transitinos from `thread-3` have to be scheduled in any possible interleaving.

Competition Example The program stack-safe-5 is a benchmark from the SV-COMP 2013 competition. It belongs to the most challenging examples for the partial-order-reduction

method implemented in Impara. During execution, two threads are used to push 5 items onto a stack and pop the same items from the stack, respectively. Each operation is protected by the same mutex m resulting in a program without any race condition. The safety assertion checks whether an underflow (< 0 elements) or an overflow (> 5 elements) occur. Similar to the synthetic examples, we vary the number of elements stored in the stack, either 5 or 10. Additionally we use the slightly modified examples stack-unsafe-5 and stack-unsafe-10 where an underflow of the stack may occur.

Results The experiments were performed on an Intel Xeon E5-2680 system with 2.8 GHz and 8GB main memory. Table 5.1 shows the runtimes for each benchmark. We report the expected verification result in Column 2 and a measure of the size of its control location domain in Column 3. The timings of our new verification approaches are shown in Columns 4 to 8 where the first three columns present partial results for executing mover-analysis, transaction-in/out-analysis, and safety-checking, respectively. Column 7 sums these partial results (TOG-HSF stands for the Transaction-Owicki-Gries proof rule implemented using HSF). In Column 8, we show the results that are obtained utilizing MHP-based reasoning by the monolithic proof rules. Column 9 presents results for Owicki-Gries reasoning based on the HSF implementation for direct comparison with the TOG-HSF column. (OG-HSF stands for Owicki-Gries proof rule implemented using HSF.) The last three columns show runtimes that are obtained with the two mentioned state-of-the-art verifiers. The results in column 10 are obtained with Impara [34] whereas the last two columns show runtimes of two proof rules implemented in Threader (the first is based on the Owicki-Gries rule, the second on the rely-guarantee rule).

For the sake of a better overview, we repeat the programs P1-1, P2-1, and P3-1 in the Figures 5.2, 5.3, and 5.4, respectively. We mark specific program statements with "o" indicating the ones which are repeated x times to get programs P1-x, P2-x, and P3-x.

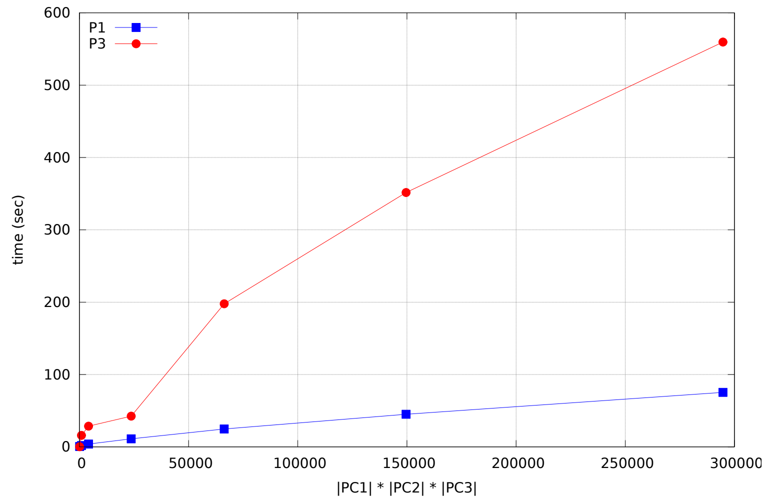


Figure 5.1.: Verification time for P1-x and P3-x with different program sizes.

The runtimes illustrate that reduction has a big impact on verification time for multi-threaded programs. Especially transaction-summarization benefits from large atomic trans-

action sequences. The most evident example is program P1-x where our approach outperforms existing state-of-the-art verifiers by several orders of magnitude. Transaction inference obtains here only four transactions for the whole program, which dramatically reduces the number of possible thread interleavings. Note that the time for transaction inference is negligible compared to the model checking time. The MHP-based method is less efficient since it has to perform satisfiability computations for each interleaving step (even if MHP information constraints most interleavings).

Figure 5.1 illustrates the verification time for transactional reasoning of program P1-x and P3-x relatively to the product of the program size. In both examples the model checking approach scales linearly with the input size (for $\mathtt{x} \in [0, 3 * 10^6]$) since the number of transactions remains the same. However, when comparing P1-x and P3-x it gets obvious that the efficiency of the transactional approach reduces with an increasing number of race conditions. Due to the unprotected access to variable `y` in the latter example program (in `thread-3`), `thread-1` gets x additional transactions for each program statement that accesses `y`. Consequently, there are $2 * x$ additional interleaving points leading to the illustrated performance difference.

As opposed to P3-x, the program P2-x induces race conditions on variable `x`. This variable is contained in the error assertion and, hence, is not abstracted by the refinement procedure of HSF. This explains the slow runtimes compared to the other examples. Nevertheless, both transactional reasoning and the MHP-based approach is faster than the other verification methods.

As mentioned, program P4-x highlights the strength of may-happen-in-parallel information. As opposed to program-wide transactions, assertions can be made about pairs of threads. The example contains a race condition because of the read access to variable `x` in `thread-3`. However, this access has no influence on the error assertion and on the interplay between `thread-1` and `thread-2`. MHP information provides hints to the verifier that constraints interleaving during the locked regions. Transactional reasoning does not benefit since the race condition make every statement that accesses `x` to a non-mover.

Even though our approaches show the best times for the unsafe stack example, the safe version underlines again that small atomic regions lead to less efficiency.

Benchmark Availability For reproducibility, the benchmark files for programs P1-x, P2-x, P3-x, and P4-x are online available. You can also find further information regarding this thesis among necessary binaries and a script file that automates the test runs at `http://www7.in.tum.de/thesis/`.

```
int x=2, y=2, mx=0, my=0;

// Thread-1
   int a;
  0: acquire(mx);
  1: a = x;
  2: acquire(my);
o 3: y = y+a;
  4: release(my);
  5: a = a+1;
  6: acquire(my);
o 7: y = y+a;
  8: release(my);
  9: x = 2*x+a;
 10: release(mx);
 11:

// Thread-2

  0: acquire(mx);
o 1: x = x+2;
  2: release(mx);
  3:

// Thread-3

  0: acquire(my);
o 1: y = y+2;
  2: release(my);
  3:
```

Figure 5.2.: Program P1-x

```
// Thread-1
   int a;
  0: acquire(mx);
  1: a = x;
  2: acquire(my);
o 3: y = y+a;
  4: release(my);
  5: a = a+1;
  6: acquire(my);
o 7: y = y+a;
  8: release(my);
  9: x = 2*x+a;
 10: release(mx);
 11:

// Thread-2

o 0: x = x+2;
  1:

// Thread-3

  0: acquire(my);
o 1: y = y+2;
  2: release(my);
  3:
```

Figure 5.3.: Program P2-x

```
// Thread-1
   int a;
  0: acquire(mx);
  1: a = x;
  2: acquire(my);
o 3: y = y+a;
  4: release(my);
  5: a = a+1;
  6: acquire(my);
o 7: y = y+a;
  8: release(my);
  9: x = 2*x+a;
 10: release(mx);
 11:

// Thread-2

  0: acquire(mx);
o 1: x = x+2;
  2: release(mx);
  3:

// Thread-3

  0: acquire(my);
o 1: y = y+2;
  2: release(my);
  3: y = 2;
  4:
```

Figure 5.4.: Program P3-x

```
int x=2, mx=0;

// Thread-1

  0: acquire(mx);
o 1: x = x+1;
  2: release(mx);
  3:

// Thread-2

  0: acquire(mx);
o 1: x = x-1;
  2: release(mx);
  3:

// Thread-3
   int a;
  0: a=x;
  1:
```

Figure 5.5.: Program P4-x with the given error assertion $\mathit{error}(V) = \mathrm{x} = 1 \wedge \mathrm{pc}_1 = \ell_3 \wedge \mathrm{pc}_2 = \ell_3 \wedge \mathrm{pc}_3 = \ell_1$

Program	Exp. Result	$\approx *\lvert PC_i \rvert$	Mover	In-Out	TOG-HSF	Total	Mon-MHP	OG-HSF	Impara	OG-Threader	RG-Threader
P1-1	✓	$1 * 10^2$	0.4s	0.2s	0.4s	1.0s	21.6s	31.3s	**0.9s**	6.3s	1m22s
P1-5	✓	$1 * 10^3$	0.8s	0.4s	1.8s	**3.0s**	1m51s	13m4s	1m50s	4m41s	T/O
P1-10	✓	$4 * 10^3$	1.5s	0.7s	3.8s	**6.0s**	6m23s	78m16s	T/O	T/O	T/O
P1-50	✓	$3 * 10^5$	19.3s	7.1s	1m15s	**1m41s**	T/O	T/O	T/O	T/O	T/O
P2-5	×	$1 * 10^3$	0.9s	0.4s	1m21s	1m22s	**1m2s**	33m33s	58s	17m12s	T/O
P2-10	×	$4 * 10^3$	1.7s	0.6s	7m23s	**7m25s**	8m24s	T/O	T/O	T/O	T/O
P2-50	×	$3 * 10^5$	24.1s	6.7s	T/O	T/O	T/O	T/O	T/O	T/O	T/O
P3-5	✓	$1 * 10^3$	0.9s	0.4s	15.8s	**17.1s**	2m42s	12m19s	1m23s	10m17s	T/O
P3-10	✓	$4 * 10^3$	1.6s	0.6s	28.5s	**30.7s**	10m9s	T/O	T/O	T/O	T/O
P3-50	✓	$3 * 10^5$	19.3s	5.8s	9m14s	**9m39s**	T/O	T/O	T/O	T/O	T/O
P4-5	✓	$2.5 * 10^1$	0.4s	0.2s	1.6s	2.2s	**0.7s**	13.3s	1m37s	46.7s	30m19s
P4-10	✓	$1 * 10^2$	0.5s	0.3s	3.7s	4.5s	**1.6s**	2m44s	T/O	3m3s	T/O
P4-50	✓	$2.5 * 10^3$	2.8s	2.6s	3m31s	3m37s	**50.5s**	T/O	T/O	66m22s	T/O
stack-safe-5	✓	$1 * 10^2$	0.1s	0.1s	1.9s	2.1s	6.8s	8.7s	1m45s	14.6s	**1.4s**
stack-safe-10	✓	$1 * 10^2$	0.1s	0.1s	25.4s	25.6s	34.7s	47.4s	10m20s	2m43s	**4.7s**
stack-unsafe-5	×	$1 * 10^2$	0.1s	0.1s	0.3s	**0.5s**	0.9s	5.0s	2.7s	21.9s	0.8s
stack-unsafe-10	×	$1 * 10^2$	0.1s	0.1s	0.3s	**0.5s**	0.9s	4.9s	1m7s	4m28s	1.2s

Table 5.1.: The columns “TOG-HSF” and “MHP” represents model checking time using our new proof rules. The column “Total” gives the timings added from transaction inference (Columns 4 and 5) and model checking (Column 6). The fastest verification result among those displayed in columns 7,8,9,10,11,12 is shown in bold font. T/O stands for time out after 90 minutes.

6. Summary and Conclusion

We conclude the thesis with a short overview of the main contributions and their implications on the verification of multi-threaded software. Additionally, we present some areas of possible future work.

Summary of contributions Most importantly, the state-space-explosion problem remains one of the biggest challenges in computer science. However, we have shown that there are two promising reduction techniques for efficient verification of concurrent programs (at least when applied to our example programs). Both MHP information and transaction summarization can dramatically reduce the number of thread interleavings that have to be explored. The two approaches operate on different reduction schemes, i.e., MHP information reduces states by exclusion whereas transaction reasoning reduces transition sequences that execute atomically. This difference leads to some valuable insights when comparing experimental results.

Precise knowledge about program locations that may happen in parallel allows fine-grained interleaving decisions for verification. As opposed to transaction summarization, assertions can be made about pairs of threads instead of considering thread-local states individually. We showed by example P4-x that there are cases in which MHP-based reasoning outperforms our method based on transactions. Simplified, these cases happen when race conditions only occur between specific pairs of threads. Transactional reasoning manages a thread-local view on these race conditions and, hence, can not benefit from such interaction behavior.

Although our MHP-based approach compares favourably with state-of-the-art verifiers, we showed that transaction summarization is in most cases even more advanced. A priori identified transactions can significantly improve the efficiency of compositional verification of multi-threaded programs, yet without requiring deep and intricate modification of the underlying solving techniques. Our experimental evaluation shows that the conceptual separation of concerns, i.e., treatment of equivalence between interleavings via transactions and keep track of interleavings using compositional proof system, can lead to two orders of magnitude reduction of verification time.

Future Work The shown examples indicate encouraging efficiency and scalability results, however there are more comprehensive test cases necessary to emphasize on the practical benefit. Additional real-world examples would also boost further improvements of both MHP-based and transactional reasoning.

In order to enhance practicality, the approaches must support additional features of existing threading libraries or concurrency APIs. Some examples are dynamic thread creation/joining, barriers, condition variables, lock-free data structures, tasks, or transactional memory. All of these features enable further reduction potential by more precise

information about concurrency behaviour of different threads. For MHP-based reasoning the information can be directly used to reduce control locations that may happen in parallel, e.g, statements from a started thread may not happen before the respective start statement of another thread. Regarding our other approach, transactions can be extended for regions that are free from interference of other threads, e.g., by utilizing lock-free algorithms and corresponding data types.

Currently, our proof rules do not support procedure handling. Thus, we rely on inlining which may highly reduce the efficiency of model checking. As opposed to sequential software, interference of other threads may prevent simple procedure summarization (by composition of all procedure transitions). There are efforts that incorporate procedures of concurrent software into transaction summaries [30]. Unfortunately, unbounded stacks are necessary for the implementation which would require considerable changes in the solving process. A meaningful idea for future work is to summarize procedures in cases without interleaving and inline procedures in all other cases.

Bibliography

[1] Shivali Agarwal, Rajkishore Barik, Vivek Sarkar, and Rudrapatna K. Shyamasundar. May-happen-in-parallel analysis of x10 programs. In *PPoPP*, pages 183–193, 2007.

[2] Rajeev Alur, Robert K. Brayton, Thomas A. Henzinger, Shaz Qadeer, and Sriram K. Rajamani. Partial-order reduction in symbolic state space exploration. In *CAV*, pages 340–351, 1997.

[3] Tony Andrews, Shaz Qadeer, Sriram K. Rajamani, Jakob Rehof, and Yichen Xie. Zing: A model checker for concurrent software. In *CAV*, pages 484–487, 2004.

[4] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press, 2008.

[5] Rajkishore Barik. Efficient computation of may-happen-in-parallel information for concurrent java programs. In *LCPC*, pages 152–169. Springer-Verlag, 2006.

[6] Dirk Beyer and Stefan Löwe. Explicit-state software model checking based on CEGAR and interpolation. In *FASE*, 2013.

[7] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *CAV*, 2000.

[8] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, 1977.

[9] Evelyn Duesterwald and Mary Lou Soffa. Concurrency analysis in the presence of procedures using a data-flow framework. In *TAV4*, pages 36–48, 1991.

[10] Tayfun Elmas, Shaz Qadeer, Ali Sezgin, Omer Subasi, and Serdar Tasiran. Simplifying linearizability proofs with reduction and abstraction. In *TACAS*, pages 296–311, 2010.

[11] Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. A calculus of atomic actions. In *POPL*, pages 2–15, 2009.

[12] Cormac Flanagan, Stephen N. Freund, Marina Lifshin, and Shaz Qadeer. Types for atomicity: Static checking and inference for java. *ACM Trans. Program. Lang. Syst.*, 30(4), 2008.

[13] Cormac Flanagan and Shaz Qadeer. Transactions for software model checking. *Electr. Notes Theor. Comput. Sci.*, 89(3):518–539, 2003.

[14] Cormac Flanagan and Shaz Qadeer. Types for atomicity. In *TLDI*, pages 1–12, 2003.

[15] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*. PhD thesis, University of Liege, Computer Science Department, 1994.

[16] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*. Springer, 1996.

[17] Sergey Grebenshchikov, Nuno P. Lopes, Corneliu Popeea, and Andrey Rybalchenko. Synthesizing software verifiers from proof rules. In *PLDI*, 2012.

[18] Cliff B. Jones. Specification and design of (parallel) programs. In *IFIP Congress*, 1983.

[19] Shmuel Katz and Doron Peled. Verification of distributed programs using representative interleaving sequences. *Distributed Computing*, 6(2):107–120, 1992.

[20] Flavio Lerda, Nishant Sinha, and Michael Theobald. Symbolic model checking of software. *Electr. Notes Theor. Comput. Sci.*, 89(3):480–498, 2003.

[21] Lin Li and Clark Verbrugge. A practical mhp information analysis for concurrent java programs. In *LCPC*, pages 194–208, 2005.

[22] Richard J. Lipton. Reduction: A method of proving properties of parallel programs. *Commun. ACM*, 18(12):717–721, 1975.

[23] Kenneth L. McMillan. Lazy abstraction with interpolants. In *CAV*, 2006.

[24] Mayur Naik, Chang-Seo Park, Koushik Sen, and David Gay. Effective static deadlock detection. In *ICSE*, pages 386–396, 2009.

[25] G. Naumovich and G. S. Avrunin. A conservative data flow algorithm for detecting all pairs of statements that may happen in parallel. Technical report, Amherst, MA, USA, 1998.

[26] Gleb Naumovich, George S. Avrunin, and Lori A. Clarke. An efficient algorithm for computing mhp information for concurrent java programs. In *ESEC/FSE-7*, pages 338–354, 1999.

[27] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[28] Susan S. Owicki and David Gries. An axiomatic proof technique for parallel programs I. *Acta Inf.*, 6, 1976.

[29] Corneliu Popeea and Andrey Rybalchenko. Threader: A verifier for multi-threaded programs - (competition contribution). In *TACAS*, pages 633–636, 2013.

[30] Shaz Qadeer, Sriram K. Rajamani, and Jakob Rehof. Summarizing procedures in concurrent programs. In *POPL*, pages 245–255, 2004.

[31] Thomas W. Reps, Susan Horwitz, and Shmuel Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL*, 1995.

[32] Richard N. Taylor. Complexity of analyzing the synchronization structure of concurrent programs. *Acta Inf.*, 19:57–84, 1983.

[33] Antti Valmari. A stubborn attack on state explosion. *Formal Methods in System Design*, 1(4):297–322, 1992.

[34] Björn Wachter, Daniel Kroening, and Joel Ouaknine. Verifying multithreaded software with Impact. In *FMCAD*, 2013.