Universiteit Leiden
The Netherlands

Opleiding Informatica

Dynamic reductions for more efficient software verification

Benjamin Steffens

Supervisors:
Alfons Laarman
Jetty Kleijn

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

28/08/2018

Abstract

Model checking is a method to prove that programs function as intended. To achieve this, however, model checkers have to exhaustively explore (and store) the large state space of the program. Reduction methods, such as partial-order reduction and transaction reduction, reduce the state space of parallel programs by pruning unnecessary parallel interleavings. Often these reduction methods are limited by the preciseness of apriori static analysis to identify commutativity. In this thesis, we present a transaction reduction theorem with a weaker notion of commutativity, which is less restricted and therefore more widely applicable.

Acknowledgements

I would like to thank my supervisor Alfons Laarman for his guidance. I would also like to thank my supervisor Jetty Kleijn for proofreading this bachelor thesis. I would also like to thank the Leiden Institute of Advanced Computer Science for giving me the opportunity to learn about and research this field of Computer Science.

Contents

Chapter 1

Introduction

Model checking [9, 10] is an automated verification approach that can prove the absence of bugs. It does so by exhaustively exploring the parallel interleavings of the program under verification. Compared to testing, model checkers use more resources, however, they guarantee completeness (up to termination of the procedure). Several reduction methods [1–8, 11–14] have been introduced to optimize model checking. This thesis presents a novel transaction reduction theorem that uses a weak notion of independence, in order to deliver more powerful reductions.

In Section 1.1 we will introduce the problem of state-space explosion and provide the concepts of dependency and reductions, which can be used to combat state-space explosion. In Section 1.2 we will introduce the main problem with strong conditional dependence and how we intend to avoid it. Section 1.3 will introduce our research questions, and Section 1.4 will provide an overview of this thesis.

1.1 Context

1.1.1 State-space explosion

A program state is a complete assignment of the program's variables to their values. A transition is an ordered pair of two states. An action translates to a statement of a program and is a set of transitions. We will focus on programs consisting of multiple threads, simultaneously executing processes which share resources. By checking all possible interleavings of threads, calculating all the reachable different states results in state-space explosion. For every simultaneously running thread and every variable in the program, the number of states increases exponentially. To illustrate this explosion take the following example program [1]:

program1 := `if (fork()) {a = 0; b = 2; } else { x = 1; y = 2; }`

Figure 1.1 shows the different ways in which the statements in this program can interleave. Arrows indicate statements. Nodes inbetween arrows are states. A path, or interleaving, consists of an initial state followed by

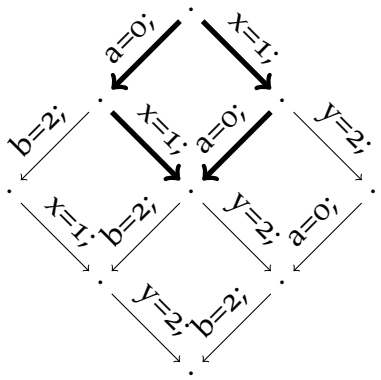

Figure 1.1: From [1]. State space of *program1*. Statements of thread 1 are represented by arrows moving downward from right-to-left, statements of thread 2 move downward left-to-right. The thick arrows indicate the interleaving `a = 0; x = 1; b = 2; y = 2;`. Note that all interleavings ultimately lead to the same final state, but may reach different states between the initial state and the final state.

a sequence of statements, ending in a final state. Each path from top to bottom is an interleaving of *program1*'s statements. The top state indicates the initial state, and the bottom state indicates the final state.

1.1.2 Independence and commutativity

There are many methods to effectively deal with state space explosion such as partial-order reduction [2, 4–6, 11, 14] and transaction reduction [1, 3, 7, 8, 12, 13]. We will detail the difference between these two methods in Section 1.1.3. Both of these methods rely on the notion of *dependence* between actions, which in turn relies on a notion of *commutativity*.

Two actions commute if the order of execution of these actions does not influence the final state reached. In Figure 1.1 all actions of both threads commute as the diamond-shape of the state space indicates; Regardless of the order of executing the program statements, the final state is the same. (Global) dependence of two actions means that there *may* be a reachable state for which the actions do not commute. Global independence is the inverse. When two actions are globally independent, they commute in every state. If actions are independent, a reduction method can often prune states and/or transitions from the corresponding state space. Reduction methods using global independence are *static*. Estimating dependence, however, is as hard as model checking itself, so global dependence can be overestimated (hence the 'may' in its definition). This means that global independence often does not yield much reduction as it is a strong requirement. Therefore, we focus on two notions of local or conditional independence.

Conditional independence means that we have access to a predicate that determines in which states actions commute [14] instead of just true or false, as in the notion of global independence. *Dynamic* reduction methods use this notion to deliver reductions in only those states where actions are independent. We will refer to conditional independence as weak conditional independence from now on, as we will describe a stronger notion of conditional independence below.

Strong conditional independence has as added restriction that states in which actions commute as indicated

by the predicate are inductively invariant. If we consider these states as a subset of the state space then this means that there exists no execution of actions such that a state in this subset can lead to a state outside of this subset. In other words: Once the predicate holds in a state, it will need to hold in all future reachable states from that state onward; The conditional independence predicate cannot be disabled once it has been enabled.

Because weak conditional independence does not have this restriction, it holds true more often, and can therefore result in more reductions than strong conditional independence. Unfortunately, using weak conditional independence in reduction methods has been difficult thus far. Before illustrating how weak conditional independence makes reductions potentially more applicable, we will briefly elaborate on partial-order reduction and transaction reduction, now that we have explained dependency.

1.1.3 Reduction methods

Before introducing partial-order reduction [4, 5] and transaction reduction [3] we would like to refer to Figure 1.2 to indicate the differences between the two reduction methods using *program1* [1] again.

Partial-order reduction [4, 5], or 'POR', prunes interleavings in order to reduce state spaces, see Figure 1.2. The reduction preserves for each interleaving in the original program's state space, at least one interleaving in the reduced state space. A trace is representative for another if it can be transformed to the other by swapping adjacent, independent actions, as defined by Mazurkiewicz [15].

Transaction reduction [3], or 'TR', finds sequential actions in one thread and merges these into one atomic action, see Figure 1.2. All interleavings between actions belonging to other threads and merged actions will be removed in the reduced state space.

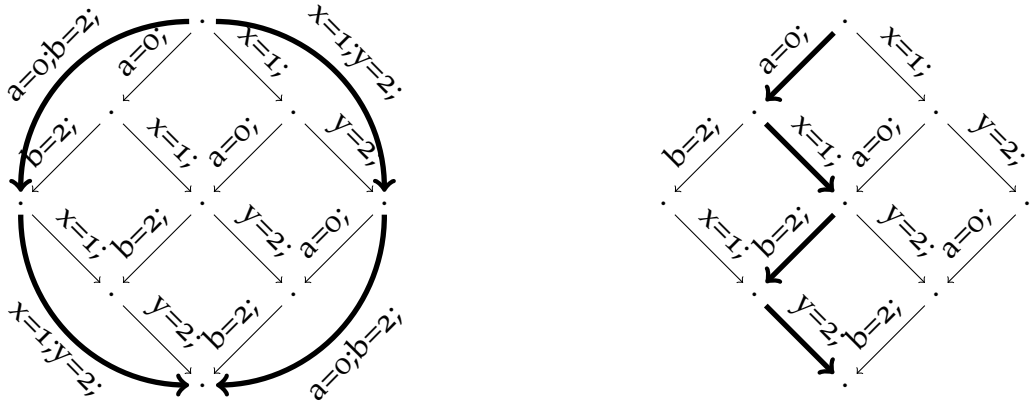

Figure 1.2: From [1]. Reductions of *program1*. Note that actions of thread 1 are represented by arrows moving downward from right-to-left, actions of thread 2 move downward left-to-right. Thick lines indicate the reduced state space after appplying reduction. Transaction reduction is applied to the left state space. Both actions of thread 1, `a=0;` and `b=2;` are merged into the single atomic action `a=0;b=2;`. Actions `x=1;` and `y=2;` of thread 2 are also merged. Thus only the two thick paths need to be checked. Partial-order reduction is applied to the right state space. All paths except for the one indicated by the thick arrows have been pruned.

While both reduction methods can use the strong notion of conditional independence (for TR see [1, 7] and for POR see [6]), it is currently unknown whether weak independence can be exploited to improve reductions (see

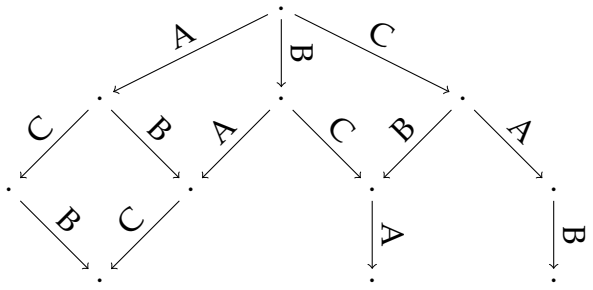

Figure 1.3: State space of *program2*. Note that as each process only executes one action, the actions have been labeled by the letter of the process they belong to, e.g. action `*p := 25;` has been labeled as A. Also note how in certain states two actions commute (under weak conditional independence), but do not in other states. For example, actions A and B in the initial state do commute, but do not after first executing action C.

Chapter 4 for more details).

1.2 Problem

To illustrate how our TR using weak conditional independence can affect more actions, we use an example program with the following three threads:

program2

```
Process A          Process B          Process C

*p := 25;          *q := 26;          p := q;
```

Where `p` and `q` are two previously defined, shared integers. We assume that in the initial state $\{p \neq q\}$ holds. If the condition $\{p \neq q\}$ is met, then actions `*p := 25;` and `*q := 26;` commute. As Process C will introduce a state where $\{p \neq q\}$ does not hold, in the initial state, before executing any of the statements in *program2*, actions `*p := 25;` and `*q := 26;` are not strongly conditionally independent, despite the fact that `p` and `q` have been initialised to hold different values.

See Figure 1.3 for the state space of *program2*.

Using a weak notion of conditional independence, we do not need to look into the future and check for the existence of actions such as *program2*'s `p := q;`. If, for the current state, the different orders of execution of two actions lead to the same state, then two actions commute. This means that `*p := 25;` and `*q := 26;` do commute under predicate $\{p \neq q\}$ in *program2*, when using weak conditional independence.

1.3 Method

It has proven difficult to combine weak conditional independence with POR or TR. Instead only a strong version of conditional independence can be used [1,6,7], where the independence condition is required to be inductively invariant as well. Günther et al. [7] and Laarman [1] have thus far implemented strong conditional

independence in their TR. Albert et al. [6] have implemented strong conditional independence in POR. We aim to combine weak conditional independence with TR. To do this we will use Gribomont's framework because it is general. Our aim is to provide a dynamic version of Gribomont's reduction hypothesis [8], based on weak conditional independence. This theorem will be more generally applicable than the reduction hypothesis [8]

Our main research question would be:

Could transaction reduction be implemented using a weak notion of conditional independence?

We would also like to answer the question:

Could a more dynamic theorem be implemented efficiently?

So the second goal is to create an algorithm that can implement a dynamic version of our theorem. Should we be able to create such an algorithm and put it to use on testprograms we may see how practical these dynamic reductions could potentially be.

1.4 Overview

In this thesis we perform TR with a weaker notion of conditional independence by generalizing the reduction hypothesis [8]. Chapter 2 includes some background information such as notation. We will also discuss how our theorem generalizes the reduction hypothesis [8]. Chapter 3 details the theorem and its proof. This theorem would answer our first research question. Chapter 4 discusses related work; Chapter 5 concludes.

Chapter 2

Background

In this chapter we will reintroduce the same abstract transition systems Gribomont [8] used. We will apply transaction reduction on these systems. We will also briefly introduce Hoare's logic, as we will be using it as a notation format. Furthermore, we will define right-commutativity in three different contexts. Finally, we will elaborate on a static reduction theorem using abstract transition systems.

2.1 Abstract transition systems

Definition 1. *An abstract transition system,* (Γ, R)*, consists of a state space* Γ *and a set of actions* $R = \bigcup_{i=1}^{n} R_i$*. Every* R_i*, where* $i \in [1, n]$*, is a set of ordered pairs, also known as transitions, over* Γ*. Note that each* R_i *is different from the other actions as each* R_i *has its own linenumber affiliated with it. This number is stored in the program counter variable. Thus, only states,* γ*, where the program counter variable holds the value of the linenumber of a certain* R_i *can appear in a transition of this* R_i*:* $(\gamma, \delta) \in R_i$*, for some state* δ*.*

If we express *program1* mentioned in Chapter 1 and drawn as a graph in Figure 1.1 as an abstract transition system, then each statement in *program1* corresponds to an R_i. Each node in Figure 1.1 is a state and corresponds to a $\gamma \in \Gamma$. Each arrow in Figure 1.1 corresponds to a $(\gamma, \delta) \in R_i$.

Note that Model checking calculates reachable states from an initial state $\gamma \in \Gamma$ or initial states $\gamma_i \in \Gamma$. Abstract transition systems do not specify initial states in Γ.

Along with abstract transition systems, we will make use of the invariant method [16, 17]. This means that for a given current state, γ, we will estimate a set of states, $I \subseteq \Gamma$, such that no states $\delta \notin I$ can be reached from γ by executing any action R_i.

2.2 Hoare's logic notation

We use Hoare's logic as a notation because this reduces the problem we want to solve to the validity problem of logic formulae. A hoare triplet is a tuple $\{A\}R_i\{B\}$ which can be read as: "If A holds before executing R_i, then B holds after executing R_i." In this triplet $\{A\}$ is the precondition and $\{B\}$ is the postcondition. R_i is an action as defined in Definition 1. A and B will often indicate sets of states throughout this thesis. Both pre- and postconditions can be written as propositional logic formulae, which can be interpreted as sets.

Let x be a program variable, and let a, b be propositional logic formulae. Then:

$[[x = 1]] == \{\gamma \in \Gamma | \gamma[x = 1]\}$

$[[a \vee b]] == [[a]] \cup [[b]]$

$[[a \wedge b]] == [[a]] \cap [[b]]$

$[[a \implies b]] == [[a]] \subseteq [[b]]$

We can interpret the logic formulae on the left as the set operators on the right. We will be using propositional logic as notation.

Definition 2. *Composition of set and relation*

Let $A \implies \Gamma$ be a set of states. Let R_i be a relation over Γ. The composition of A and R_i is $A.R_i = \{\delta | \gamma \in A \wedge (\gamma, \delta) \in R_i\}$

The operator indicating composition of set and relation, ., will mostly be used implicitly, meaning we do not write it down.

Definition 3. *Composition of two relations*

Let R_i and R_j be two relations over Γ. The composition of R_i and R_j is $R_i; R_j = \{(\gamma, \delta) | (\gamma, \sigma) \in R_i \wedge (\sigma, \delta) \in R_j\}$.

The notation $\{A\}R_i\{B\}$ is also used to symbolize $A.R_i \implies B$, where $A, B \implies \Gamma$ and $R_i \implies \Gamma^2$. Because of this we can rewrite a formula such as $\{A\}R_i; R_j\{B\}$ to $\{A.R_i\}R_j\{B\}$, where $R_i, R_j \implies R$ as defined in Definition 1, as both notations still symbolize $A.R_i.R_j \implies B$. Operators . and ; are as defined in Definitions 2 and 3.

Hoare's logic also assumes a set of axioms to hold. We will make use of the following axioms to prove the theorem in Chapter 3:

Axiom 1. *Rule of Consequence:*

$$\frac{p_1 \rightarrow p_2, \{p_2\}S\{q_2\}, q_2 \rightarrow q_1}{\{p_1\}S\{q_1\}}$$

Axiom 1 allows us to strengthen the precondition and weaken the postcondition. This means one can remove states from the precondition and add states to the postcondition using Axiom 1.

Axiom 2. *Disjunction:*

$$\frac{\{p_1\}S\{q_1\}, \{p_2\}S\{q_2\}}{\{p_1 \vee p_2\}S\{q_1 \vee q_2\}}$$

Axiom 2 allows us to create a new triplet out of two previous triplets if their actions are the same. The preconditions and postconditions become disjuncts of the merged triplet's precondition and postcondition.

Axiom 3. *Conjunction:*

$$\frac{\{p_1\}S\{q_1\},\{p_2\}S\{q_2\}}{\{p_1 \wedge p_2\}S\{q_1 \wedge q_2\}}$$

Axiom 3 is similar to Axiom 2. The difference being that merged preconditions and postconditions become conjuncts of the merged triplet's precondition and postcondition.

Axiom 4. *Composition:*

$$\frac{\{p\}S_1\{q\},\{q\}S_2\{r\}}{\{p\}S_1;S_2\{r\}}$$

Axiom 4 allows us to composit sequential actions if the postcondition of one matches the precondition of the other.

2.3 Defining global and conditional dependence

Now that we can express programs in abstract transition systems, see Definition 1, we can also define when two actions are globally or conditionally independent. Before that, we will have to define what it means for a condition or predicate to be inductively invariant, and what a left-restriction of a relation to a set is.

Definition 4. *Inductive invariance.*

Given an abstract transition system (Γ, R)*, a condition or predicate,* $C \Longrightarrow \Gamma$*, is inductively invariant in a state* $\gamma \in \Gamma$ *if* $\forall(\gamma, \delta) \in R : \delta \in C$*. In other words: C is invariant if* $\{C\}R\{C\}$*.*

Definition 5. *Left-restricting a relation to a set.*

Given a relation R_i *over state space* Γ*, and a set of states* $C \Longrightarrow \Gamma$*,* R_i *left-restricted to set C is* $C//R_i = \{(\gamma, \delta) \in R_i | \gamma \in C\}$*.*

Commutativity can be defined in global independence, weak conditional independence or strong conditional independence. We will be using a restricted form of commutativity called right-commutativity. Like commutativity, right-commutativity comes in the aforementioned three flavours: global, strongly and weakly conditional. These are defined in Definitions 6, 7 and 8.

Definition 6. *Global right-commutativity.* $R_i \overrightarrow{\bowtie} R_j$

Given an abstract transition system (Γ, R)*, in global independence* R_i *right-commutes with* R_j*,* $R_i \overrightarrow{\bowtie} R_j$*, if* $R_i; R_j \Longrightarrow R_j; R_i$*.*

Definition 7. *Conditional right-commutativity.* $R_i \overrightarrow{\bowtie}_C R_j$

Given an abstract transition system (Γ, R)*,* R_i *conditionally right-commutes with* R_j*,* $R_i \overrightarrow{\bowtie}_C R_j$*, for some condition* $C \Longrightarrow \Gamma$ *in a state* $\gamma \in \Gamma$ *if* $C//R_i; R_j \Longrightarrow C//R_j; R_i$*.*

Definition 8. *Strong conditional right-commutativity.* $R_i \overrightarrow{\bowtie}_C R_j$

Given an abstract transition system (Γ, R) *and two actions* R_i, R_j *such that* $R_i \overrightarrow{\bowtie}_C R_j$ *for some condition* C*, if* C *is inductively invariant, then this right-commutativity relation is strongly conditional.*

When two actions, R_i, R_j, are globally dependent, they do not commute globally. R_i, R_j could still be conditionally independent for some condition C. This means R_i, R_j could still commute under conditional independence in some states, despite being globally dependent.

2.4 A static reduction theorem

Gribomont [8] uses global independence to define atomicity refinement. Atomicity refinement is essentially the opposite to transaction reduction: it splits actions in smaller parts instead of conjoining them into larger sequential transactions. This is useful in theorem proving, where shorter actions can be handled more effeciently, as opposed to model checking, where larger transaction yield fewer states.

Definition 9 will introduce two abstract transition systems, **Old** and **New**, which will be used to illustrate atomicity refinement and transaction reduction in Theorem 1. **Old** is the abstract transition system representing the state space of a program before atomicity refinement. **New** is the abstract transition system representing the state space of this program after atomicity refinement. In this thesis we are more interested in transaction reduction, which treats **Old** and **New** the opposite way around. **Old** is the abstract transition system representing the state space of a program after transaction reduction. **New** is the abstract transition system representing the state space of this program before transaction reduction.

Definition 9. *Let there be two abstract transition systems,* $\boldsymbol{Old} = (\Gamma, \{S, R\})$ *and* $\boldsymbol{New} = (\Gamma, \{S_1, S_2, R\})$*, where* S_1 *and* S_2 *are consecutive statements in the same thread. Further, there are two conditions or predicates,* $A, B \Longrightarrow \Gamma$*. Note that* S_1 *and* S_2 *are actions which would be incorporated into* R*. We note them down separately as we wish to apply transaction reduction on these two actions. The following conditions hold:*

1. $\models (A \Longrightarrow B)$;

2. $\{B\}S_1\{\overline{B}\}$;

3. $\{B\}S_2\{false\}$;

4. $\{\overline{B}\}S_2\{B\}$;

5. $\{\overline{B}\}S_1\{false\}$;

6. $\{B\}R\{B\}$;

7. $\{\overline{B}\}R\{\overline{B}\}$;

8. $S = S_1; S_2$ *(sequential composition).*

States satisfying B are relevant states, and states not satisfying B (these satisfy $\overline{B}$) are transient states. As S_2 directly follows S_1, the program counter variable in transient states point to S_2. The dichotomy between the

relevant states and transient states is something we will be trying to avoid with our theorem. This way we can relax condition 2 and condition 4. $\overline{B}$ is the complement of B. The notation used above is Hoare's logic notation which we elaborated on in Section 2.2

Gribomont's reduction hypothesis [8] uses Definition 9 and is as follows:
Theorem 1. *For any $I \Longrightarrow B$, if $S_1 \overrightarrow{\bowtie} R$, as defined in Definition 6, then $I \vee IS_1$ is a **New**-invariant if and only if I is an **Old**-invariant. In other words: $\{I \vee IS_1\}S_1\{I \vee IS_1\}$, $\{I \vee IS_1\}S_2\{I \vee IS_1\}$, $\{I \vee IS_1\}R\{I \vee IS_1\} \Longleftrightarrow \{I\}S\{I\}$, $\{I\}R\{I\}$. [8]*

Note that as the implication works in both directions, we can use this theorem not just to refine **Old** into **New**, but also to reduce **New** into **Old**.

Also note that the condition $S_1 \overrightarrow{\bowtie} R$ indicates that action S_1 right-commutes with all other actions R in **Old**. This essentially enforces global independence and therefore restricts the application of Theorem 1.

Chapter 3

A fully dynamic reduction theorem

In this chapter we will provide an alternative theorem to Theorem 1. Definition 10 will function much like Definition 9 in that it introduces two abstract transition systems, **Unreduced** and **Reduced**. **Unreduced** is the abstract transition system representing the state space of a program before transaction reduction. **Reduced** is the abstract transition system representing the state space of this program after transaction reduction.

Definition 10. *Let* $\textbf{\textit{Unreduced}} = (\Gamma, \{R_1, ..., R_n, S_1, S_2\})$ *and* $\textbf{\textit{Reduced}} = (\Gamma, \{R_1, ..., R_n, S\})$ *be two abstract transition systems. Let* S_1 *and* S_2 *be two consecutive actions in the same program, and let* $S = S_1; S_2$. *Let* $R = \bigcup_{i=1}^{n} R_i$. *Let* $X \Longrightarrow \Gamma$ *be a predicate. We will use* X *as the condition to which* $S_1 \overrightarrow{\bowtie}_X R'$. *We define:*

$S'_1 =_{def} X//S_1$ *(See also Definition 5)*

$S'_2 =_{def} (XS_1)//S_2$ *(See also Definition 2)*

$R' =_{def} R \cup \{(x,y) \mid (x,y) \in S_1, (x,y) \notin S'_1\} \cup \{(x,y) \mid (x,y) \in S_2, (x,y) \notin S'_2\}$.

The following conditions hold:

(1) $S' = S'_1; S'_2$

(2) $\{\overline{X}\}S'_1\{false\}$

(3) $\{\overline{XS'_1}\}S'_2\{false\}$

(4) $S'_1; R' \Longrightarrow R'; R'$

(5) $\{X\}S'_1; S'_2\{\overline{X}\}$

(6) $\{X\}R'\{\overline{XS'_1}\}$

(*) $\{XS'_1\}S'_2\{\overline{XS'_1}\}$

To elaborate a little on the conditions: Conditions (1) through (3) have actually already been defined. Condition (4) is deliberately defined differently from Gribomont's [8] $S_1; R \Longrightarrow R; S_1$ in order to avoid relying on global

independence. Condition (4) enforces $S_1 \overrightarrow{\bowtie}_X R$, as defined in Definition 7, where the condition is X, ensuring that all the states that could be reached by executing $S_1'; R'$ can still be reached by the other actions in R' without having to ensure that S_1' right-commutes with all the other actions in R'. Condition (5) makes sure that $S_1'; S_2'$ makes total progress. Condition (6) ensures that R' never reaches XS_1'. Without condition (*) we would not have managed to prove Theorem 2.

Theorem 2. *There exists an invariant I such that*
*(7) $X \Longrightarrow I \Longrightarrow \Gamma$, and I is a **Reduced**-invariant $\Longleftrightarrow$ $I \vee IS_1'$ is an **Unreduced**-invariant:*
$\{I\}R'\{I\}, \{I\}S'\{I\} \Longleftrightarrow \{I \vee IS_1'\}R'\{I \vee IS_1'\}, \{I \vee IS_1'\}S_1'\{I \vee IS_1'\}, \{I \vee IS_1'\}S_2'\{I \vee IS_1'\}$

Proof. We will first prove $\{I\}R'\{I\}, \{I\}S'\{I\} \Longrightarrow \{I \vee IS_1'\}R'\{I \vee IS_1'\}, \{I \vee IS_1'\}S_1'\{I \vee IS_1'\}, \{I \vee IS_1'\}S_2'\{I \vee IS_1'\}$. After, we will prove $\{I\}R'\{I\}, \{I\}S'\{I\} \Longleftarrow \{I \vee IS_1'\}R'\{I \vee IS_1'\}, \{I \vee IS_1'\}S_1'\{I \vee IS_1'\}, \{I \vee IS_1'\}S_2'\{I \vee IS_1'\}$.

Lemma 1. $\{I\}R'\{I\}, \{I\}S'\{I\} \Longrightarrow \{I \vee IS_1'\}R'\{I \vee IS_1'\}, \{I \vee IS_1'\}S_1'\{I \vee IS_1'\}, \{I \vee IS_1'\}S_2'\{I \vee IS_1'\}$

Proof. We assume that I is a **Reduced**-invariant:
(8) $\{I\}R'\{I\}$.
(9) $\{I\}S'\{I\}$.

(10) Definition: $\{IS_1'\}R'\{IS_1'R'\}$.

(11) Consequence (4), (10): $\{IS_1'\}R'\{IR'R'\}$.

(12) Consequence (8), (11): $\{IS_1'\}R'\{IR'\}$.

(13) Consequence (8), (12): $\{IS_1'\}R'\{I\}$.

(14) Disjunction (8), (13): $\{I \vee IS_1'\}R'\{I\}$.

(15) Consequence (14): $\{I \vee IS_1'\}R'\{I \vee IS_1'\}$.

(16) Substitution (1), (9): $\{I\}S_1'; S_2'\{I\}$.

(17) Rewriting (16): $\{IS_1'\}S_2'\{I\}$.

(18) Consequence (7), (17): $\{XS_1'\}S_2'\{I\}$.

(19) Disjunction (17), (3): $\{true\}S_2'\{I\}$.

(20) Consequence (19): $\{I \vee IS_1'\}S_2'\{I \vee IS_1'\}$.

(21) Definition: $\{X\}S_1'\{XS_1'\}$.

(22) Consequence (7), (21): $\{X\}S_1'\{IS_1'\}$.

(23) Disjunction (2), (22): $\{true\}S_1'\{IS_1'\}$.

(24) Consequence (23) $\{I \vee IS_1'\}S_1'\{I \vee IS_1'\}$.

(25) Because of (15), (20) and (24) it follows that: $I \vee IS_1'$ is an **Unreduced**-invariant. □

Lemma 2. $\{I\}R'\{I\}, \{I\}S'\{I\} \Longleftarrow \{I \vee IS_1'\}R'\{I \vee IS_1'\}, \{I \vee IS_1'\}S_1'\{I \vee IS_1'\}, \{I \vee IS_1'\}S_2'\{I \vee IS_1'\}$

Proof. Starting over at (8).

We assume that $I \vee IS_1'$ is an **Unreduced**-invariant:
(8) $\{I \vee IS_1'\}R'\{I \vee IS_1'\}$.
(9) $\{I \vee IS_1'\}S_1'\{I \vee IS_1'\}$.
(10) $\{I \vee IS_1'\}S_2'\{I \vee IS_1'\}$.

(11) Consequence (7), (2): $\{\overline{I}\}S_1'\{false\}$.

(12) From (11) and (2) it follows that $\overline{I}S_1' = \overline{X}S_1'$.

(13) From (12) it follows that $IS_1' = XS_1'$.

(14) From (13) it follows that $\overline{IS_1'} = \overline{XS_1'}$.

(15) Consequence (14), (6): $\{X\}R'\{\overline{IS_1'}\}$.

(16) Conjunction (15), (8): $\{(I \wedge X) \vee (IS_1' \wedge X)\}R'\{(I \wedge \overline{I_S 1'}) \vee (IS_1' \wedge \overline{IS_1'})\}$.

(17) Consequence (7), (16): $\{I\}R'\{I\}$.

(18) Composition (2), (3): $\{\overline{X}\}S'\{false\}$.

(19) Consequence (7), (2): $\{\overline{I}\}S'\{false\}$.

(20) From (18) and (19) it follows that $\overline{I}S' = \overline{X}S'$.

(21) From (20) it follows that $IS' = XS'$.

(22) From (21) it follows that $\overline{IS'} = \overline{XS'}$.

(23) Definition: $\{I\}S_1'\{IS_1'\}$.

(24) Consequence (13), (23): $\{I\}S_1'\{XS_1'\}$.

(25) Composition (24), (*): $\{I\}S'\{\overline{XS_1'}\}$.

(26) Consequence (22), (25): $\{I\}S'\{\overline{IS_1'}\}$.

(27) Composition (9), (10): $\{I \vee IS_1'\}S'\{I \vee IS_1'\}$.

(28) Conjunction (26), (27): $\{I \vee IS_1'\}S'\{(I \wedge \overline{IS_1'}) \vee (IS_1' \wedge \overline{IS_1'})\}$.

(29) Consequence (28): $\{I\}S'\{I\}$.

(30) Because of (17) and (29) it follows that: I is a **Reduced**-invariant. □

We have proven both $\{I\}R'\{I\}, \{I\}S'\{I\} \Longrightarrow \{I \vee IS'_1\}R'\{I \vee IS'_1\}, \{I \vee IS'_1\}S'_1\{I \vee IS'_1\}, \{I \vee IS'_1\}S'_2\{I \vee IS'_1\}$ and $\{I\}R'\{I\}, \{I\}S'\{I\} \Longleftarrow \{I \vee IS'_1\}R'\{I \vee IS'_1\}, \{I \vee IS'_1\}S'_1\{I \vee IS'_1\}, \{I \vee IS'_1\}S'_2\{I \vee IS'_1\}$. Thus:

$\{I\}R'\{I\}, \{I\}S'\{I\} \Longleftrightarrow \{I \vee IS'_1\}R'\{I \vee IS'_1\}, \{I \vee IS'_1\}S'_1\{I \vee IS'_1\}, \{I \vee IS'_1\}S'_2\{I \vee IS'_1\}$. □

By transforming **Unreduced** into **Reduced**, Theorem 2 merges S_1 and S_2 into S. This merging is transaction reduction and relies on weak conditional independence, as condition (4) indicates; Theorem 2 performs TR using weak conditional independence.

Chapter 4

Related work

We would like to mention a bit of history of partial-order reduction and transaction reduction in relation to global, conditional and strongly conditional independence. See also Figure 4.1.

POR in Katz [5] and Valmari [4] still uses global independence. Godefroid [2] introduced sleep sets which can also be used orthogonally with globally independent POR [4, 5] as shown in Flanagan et al. [11]. Most recently (CAV2018) Albert et al. [6] have used strong conditional independence for POR. Using weak conditional independence for POR had not been attempted yet before this thesis. Peephole POR [14] implements a symbolic version of Godefroid's [2] sleep sets in which only transitions are pruned.

Introduced by Lipton [3] and improved on by Lamport et al. [12], TR has also been used in Gribomont [8], whose framework has been very relevant to Chapters 2 and 3. The citations thus far have used global independence for TR, but Günther et al. [7] have used strong conditional independence for TR. Laarman [1] introduces stubborn sets to this type of TR. Performing transaction reduction using weak conditional independence, had not been attempted yet, and is what we use in this thesis. Also relevant to TR is Gueta et al. [13]'s Cartesian POR, which we felt was hard to categorize in Figure 4.1

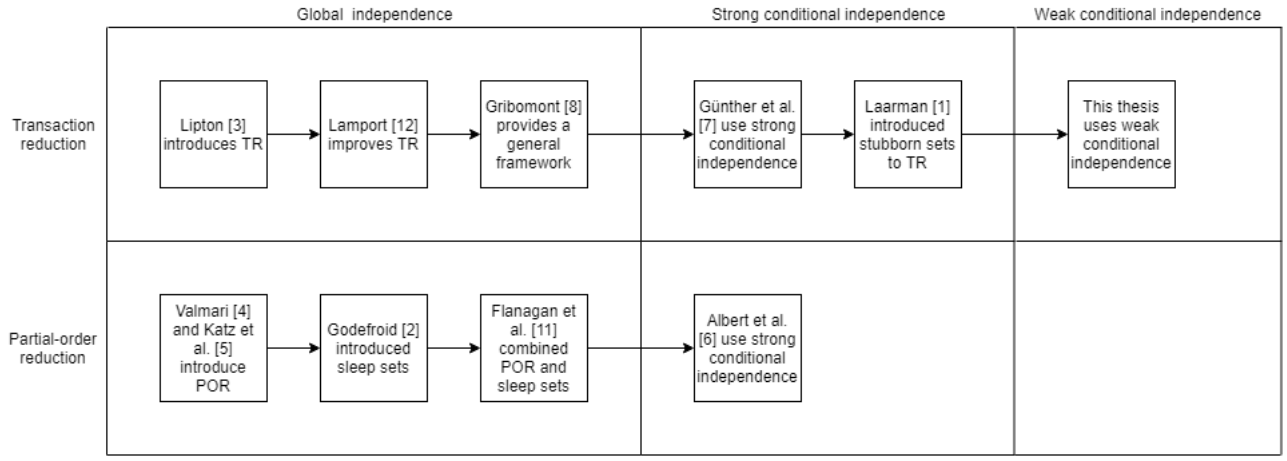

Figure 4.1: [1] Ordering previous work on POR [2, 4–6, 11] and TR [1, 3, 7, 8, 12, 13] according to three categories of independence discussed in Chapters 1 and 2.

Chapter 5

Conclusions

We set out to answer two questions with this thesis.

Our main research question is:

Could transaction reduction be implemented using a weak notion of conditional independence?

In Chapter 3 we effectively answer this question by providing a theorem, Theorem 2, which performs transaction reduction using only weak conditional independence. Unlike Günther et al.'s [7] transaction traces, Theorem 2 otherwise limits itself to just two statements, S_1 and S_2. The original invariant, I, is also still contained in $I \vee IS_1$, which is useful for applying the theorem.

Our other research question is:

Could a more dynamic theorem be implemented efficiently?

Unfortunately, we did not implement an algorithm that makes use of Theorem 2. One simple approach we considered is to apply Theorem 2 repeatedly to a program's statements as these are executed. This algorithm would not need to look into future states and can therefore be applied dynamically.

Bibliography

[1] Laarman A.W., *Stubborn Transaction Reduction*, NASA Formal Methods 2018, 10th International Symposium proceedings, pp. 280-298

[2] Godefroid P., *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem* University of Liège, 1993-1994. Web. 19 August. 2018

[3] Lipton R.J., *Reduction: a method of proving properties of parallel programs*, Communications of the ACM, December 1975, vol. 18, no. 12, pp. 717-721

[4] Valmari A., *Eliminating Redundant Interleavings during Concurrent Program Verification*, Parallel Architectures and Languages Europe, 1989 Conference proceedings, vol. 2, pp. 89-103

[5] Katz S., Peled D. A., *Defining conditional independence using collapses*, Theoretical Computer Science, July 1992, vol. 101, issue 2, pp. 337-359

[6] Albert E., Gómez-Zamalloa M., Isabel M., Rubio A., *Constrained Dynamic Partial-Order Reduction*, Computer Aided Verification 2018, 30th International Conference proceedings, pp. 392-410

[7] Günther H., Laarman A., Sokolova A., Weissenbacher G., *Dynamic Reductions for Model Checking Concurrent Software*, Verification, Model Checking, and Abstract Interpretation 2017, 18th International Conference proceedings, pp. 246-265

[8] Gribomont E.P., *Atomicity refinement and trace reduction theorems*, Computer Aided Verification 1996, 8th International Conference proceedings, pp. 311-322

[9] Clarke E.M., Grumberg O., Peled D. A., *Model Checking*, Mit Press Ltd, December 1999

[10] Baier C., Katoen J.P., *Principles of Model Checking*, Mit Press Ltd, April 2008

[11] Flanagan C., Godefroid P., *Dynamic partial-order reduction for model checking software*, Symposium on Principles of Programming Languages 2005, 32nd Annual Symposium proceedings, pp 110-121

[12] Lamport L., Schneider F.B., *Pretending Atomicity*, Technical report, Cornell University, 1989

[13] Gueta G., Flanagan C., Yahav E., Sagiv M., *Cartesian Partial-Order Reduction*, SPIN Workshop on Model Checking of Software 2007, 14th International SPIN Workshop proceedings, pp 95-112

[14] Wang C., Yang Z., Kahlon V., Gupta A., *Peephole Partial Order Reduction*, Tools and Algorithms for the Construction and Analysis of Systems 2008, 14th International Conference proceedings, pp. 382-396

[15] Mazurkiewicz A., *Introduction to Trace Theory*, The Book of Traces Chapter 1, pp. 3-41, World Scientific Publishing Co., March 1995

[16] de Bakker J.W., Meertens L.G.L.T, *On the Completeness of the Inductive Assertion Method*, Journal of Computer and System Sciences 1975, vol. 11, issue 3, pp. 323-357

[17] Floyd R.W., *Assigning Meanings to Programs*, Program Verification Chapter 4, pp. 65-81, Kluwer Academic Publishers, January 1993