Saarland University
Faculty of Mathematics and Computer Science

Bachelor's Thesis

Model Checking for Temporal Stream Logic and Hyper Temporal Stream Logic

Author
Janine Lohse

Advisors
Dr. Hadar Frenkel
Jana Hofmann

Supervisor
Prof. Dr. Bernd Finkbeiner

Reviewers
Prof. Dr. Bernd Finkbeiner
Dr. Hadar Frenkel

Submitted: 16th August 2022

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement in Lieu of an Oath

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken, 16th August, 2022

Abstract

In this thesis, we develop model checking algorithms for both Temporal Stream Logic (TSL) and Hyper Temporal Stream Logic (HyperTSL). TSL extends Linear Temporal Logic (LTL) with predicates over inputs and memory cells, and with update terms that specify how the value of a memory cell should change. Similar to the extension of LTL to HyperLTL, HyperTSL further extends TSL for the specification of hyperproperties, that is, properties relating multiple system executions. Unlike HyperLTL, HyperTSL can express many important security properties like noninterference even if there is an infinite data domain.

Both TSL and HyperTSL were originally defined for synthesis – to the best of our knowledge, there is no algorithm explicitly designed for model checking yet. We first study the decidable case of (Hyper)TSL model checking of finite-state systems. We show that in this case, (Hyper)TSL is not more expressive than (Hyper)LTL by giving a translation algorithm. Still, many properties can be expressed more compactly using (Hyper)TSL. Thus, we also develop direct model checking algorithms for TSL and HyperTSL with at most one quantifier alternation that are more efficient than model checking an equivalent HyperLTL formula.

Next, we study (Hyper)TSL model checking of infinite-state systems, which is called software model checking. We extend a known LTL software model checking algorithm to TSL and, by applying the technique of self-composition, to alternation-free HyperTSL. We further extend this algorithm to an algorithm for finding counterexamples for $\forall^*\exists^*$ HyperTSL formulas (or, dually, witnesses of $\exists^*\forall^*$ HyperTSL formulas) that is sound but not complete. This also gives the, to the best of our knowledge, first infinite-state software model checking algorithm for finding counterexamples for $\forall^*\exists^*$ and witnesses for $\exists^*\forall^*$ HyperLTL formulas.

Acknowledgements

First of all, thanks a lot to my advisors Hadar and Jana for guiding me while I was writing this thesis. I know you also had a whole range of other obligations, but you always took your time for discussing algorithm ideas, helping me out when I was stuck and giving me valuable and constructive feedback.

Furthermore, I would like to thank Professor Finkbeiner for offering me a thesis at his chair and putting so much effort into finding the optimal thesis topic for me.

I want to thank Niklas for all his support. You motivated me when I was unproductive and gave me so much valuable intermediate advice and feedback. I don't know if you are aware of how much this helped me.

Also thanks to Lobita for her emotional support and to all my friends and family for accompanying me during my studies.

Contents

Abstract **iii**

1 Introduction **1**
1.1 Overview . . . 3
1.1.1 (Hyper) Temporal Stream Logic with Theories . . . 3
1.1.2 Finite-State Model Checking . . . 3
1.1.3 Software Model Checking . . . 5

2 Related Work **9**

3 Preliminaries **13**
3.1 Kripke Structures and Büchi Automata . . . 13
3.2 (Hyper) Linear Temporal Logic . . . 14

4 (Hyper) Temporal Stream Logic with Theories **17**
4.1 Recap: Temporal Stream Logic with Theories . . . 17
4.2 HyperTSL with Theories . . . 19
4.3 Similiarity of LTL and TSL . . . 22

5 Finite-State Model Checking **25**
5.1 TSL Kripke Structures . . . 25
5.2 Translation to (Hyper)LTL . . . 26
5.3 Update Term Elimination . . . 31
5.3.1 TSL Model Checking by Update Term Elimination . . . 34
5.4 HyperTSL Model Checking . . . 35
5.4.1 The Alternation-Free Fragment . . . 37
5.4.2 The $\forall^*\exists^*$ and $\exists^*\forall^*$ Fragments . . . 41

6 Software Model Checking **45**
6.1 TSL Software Model Checking . . . 45
6.1.1 The Algorithm . . . 45

6.1.2 Correctness 50
6.2 Alternation-Free HyperTSL Software Model Checking 54
6.2.1 The Algorithm 54
6.2.2 Correctness 54
6.3 Finding Counterexamples for the $\forall^*\exists^*$ - Fragment 59
6.3.1 The Algorithm 60
6.3.2 Correctness 63
6.3.3 Example Applications 64

7 Discussion **71**
7.1 Conclusion 71
7.2 Future Work 72

Bibliography **75**

Chapter 1

Introduction

With the increasing use of digital technology in all areas of life, the damage that can be caused by undetected bugs is also becoming ever greater. While testing can only reveal bugs but never verify their absence, *model checking*, the process of checking automatically whether a system meets a given specification, can prove a system correct – or provide a counterexample. The object of this thesis is to develop model checking algorithms for both Temporal Stream Logic and Hyper Temporal Stream Logic.

Model checking has already been very successfully applied in practice, for example on the control algorithms of a flood barrier in the Netherlands [36]. In this case and most commonly, *linear temporal logic* [43] (LTL) was used as the language for expressing the specification. However, LTL has its limitations: First of all, it is not possible to relate multiple executions of the system within an LTL formula. Relating multiple executions is necessary to express many important security properties like *noninterference* [40], a property stating that by observing the output, an observer can not gain any information about a specified secret input. More specifically, noninterference states that for every pair of executions, if in both executions all inputs except the secret one are equal, then also the output should be equal. Noninterference is an example for a *hyperproperty*, that is, a property relating multiple executions. Due to the great relevance of hyperproperties, LTL has been extended to HyperLTL [17], making many of them expressible. Several algorithms have been developed for model checking HyperLTL [17, 18, 27]

Nevertheless, there is still another limitation of HyperLTL: like LTL, HyperLTL is based on boolean atomic propositions, each describing a property that can at any time during an execution be true or false. Each atomic proposition is attached to a single execution. Using a finite set of atomic propositions for encoding all possible values of some inputs and outputs is only possible if the inputs and outputs both have a finite value domain. Consequently, properties like noninterference are not expressible

in HyperLTL if the relevant inputs and outputs have infinitely many possible values.

Hyper Temporal Stream Logic (HyperTSL) [19], which itself extends *Temporal Stream Logic (TSL)* [28] for hyperproperties does not have this limitation: In contrast to LTL, TSL is based on arbitrary predicates over memory cells and inputs. This is why HyperTSL can express properties like noninterference even if there is an infinite value domain. Noninterference is formally expressed as a HyperTSL formula as follows:

Example 1.1. (Noninterference as a HyperTSL Formula)

$$\forall \pi.\ \forall \pi'.\ \Box\Big(\bigwedge_{i \in \mathbb{I} \setminus h} i_\pi = i_{\pi'}\Big) \rightarrow \Box(o_\pi = o_{\pi'})$$

The formula states that all pairs of executions π and π' on which all inputs in $\mathbb{I}$ except the secret input h are always equal, must also always have the same output o. Thus, by observing the output o, an observer cannot gain information about h.

TSL and HyperTSL additionally feature so-called *update terms*, terms that state how the value of a memory cell should change. This, for example, makes the statement 'the program variable x should be increased by one' expressible in TSL and HyperTSL, as shown in the following example.

Example 1.2. (Update Term)
This HyperTSL formula states that the cell `counter` should count how often the cell `money` drops below zero.

$$\forall \pi.\ \Box\left((money_\pi > 0) \land \bigcirc(money_\pi \leq 0) \rightarrow \bigcirc\bigcirc[\![counter_\pi \leftarrowtail counter_\pi + 1]\!]\right)$$

Always, when `money` has a positive value, but in the next timestep not anymore, then in the step after, the cell `counter` should be increased by one.

This formula has only one universal quantifier. HyperTSL formulas with one universal quantifier are exactly the set of TSL formulas. When writing a TSL formula, we usually omit the universal quantifier.

Both TSL and HyperTSL were originally not defined for model checking, but for the automatic construction of a system from a given specification (*synthesis*). To the best of our knowledge, there is no algorithm explicitly designed for model checking for TSL or HyperTSL yet. The object of this thesis is to develop model checking algorithms for both TSL and HyperTSL.

Thereby, we first study the easier, decidable case of a finite-state system, that is, a system with only finitely many possible values for each input and memory cell. We show that in this case, as there is a finite value domain, (Hyper)TSL is not more expressive than (Hyper)LTL. Still, many formulas can be expressed more compactly in TSL. Thus, we also provide model checking algorithms for TSL and HyperTSL with at most one quantifier alternation that are more efficient on a large system

than model checking an equivalent (Hyper)LTL formula. For TSL, our algorithm is a second TSL to LTL model checking reduction that is now based on the modification of the system instead of the modification of the formula. For HyperTSL, we adapt the HyperLTL model checking algorithm by Finkbeiner et. al. [27] for HyperTSL with at most one quantifier alternation.

Then, we study the case of an infinite-state system, where the model checking problem is called *software model checking*. As the system, we use an automaton labeled with program statements. The software model checking problem is undecidable already for LTL, so it is also undecidable for TSL and HyperTSL. Thus, we develop partial algorithms that might diverge or return 'unknown'. To obtain an algorithm for TSL software model checking, we adapt the automata-based LTL software model checking algorithm by Dietsch et. al. [25]. This algorithm is extendable to HyperTSL formulas without quantifier alternations by applying the technique of *self-composition*, a technique commonly used for the verification of hyperproperties [3, 4, 26]. Next, we further extend this to an algorithm for finding counterexamples disproving $\forall^*\exists^*$ HyperTSL formulas (and, by negating, witnesses proving $\exists^*\forall^*$ formulas) that is sound but not complete. As such an algorithm does, to the best of our knowledge, also not yet exist for HyperLTL, this also gives the first software model checking algorithm for finding counterexamples for $\forall^*\exists^*$ HyperLTL formulas (and proving $\exists^*\forall^*$ HyperLTL formulas). The idea of this algorithm is to construct an automaton that overapproximates the set of program executions satisfying the existential part of the formula.

1.1 Overview

1.1.1 (Hyper) Temporal Stream Logic with Theories

We present the definitions of TSL and HyperTSL, modified for making them more suitable for model checking. In their original presentations by Finkbeiner et. al. in [19, 28], predicates and functions were left uninterpreted. This for example means that the equals predicate used in the formulation of noninterference would not have any concrete meaning, which is undesirable in our setting. Therefore, we evaluate the predicates and functions based on a given theory. This was already done for TSL in [29], and we also add an interpretation for predicates and functions in HyperTSL. Moreover, as relating multiple executions within one predicate is necessary to express properties like noninterference, we choose the version of HyperTSL that was originally called HyperTSL$_{rel}$ by Finkbeiner et. al. and allows such predicates.

1.1.2 Finite-State Model Checking

Next, we study algorithms for model checking TSL and HyperTSL in a finite-state system. For the system, we use an automaton that specifies the current values of

the memory cells and inputs at each automaton state. Because an automaton can only have finitely many states, such a system only allows to express finitely many values. This is the reason why in this case, (Hyper)TSL model checking is reducible to (Hyper)LTL model checking, which we prove by describing a translation algorithm that given a (Hyper)TSL formula and a finite-state system, constructs a (Hyper)LTL formula that is equivalent for this system. The idea of this translation is to 'hardcode' the predicate and update term evaluations for all possible values of the memory cells and inputs in the (Hyper)LTL formula.

The existence of such a translation means that if there is a finite value domain, one does not gain expressiveness by using (Hyper)TSL instead of (Hyper)LTL. Still, many properties are expressible more compactly in (Hyper)TSL. Therefore, we also design an explicit algorithm for model checking (Hyper)TSL that is more efficient than applying the translation if the system is large.

For TSL, our model checking algorithm is another reduction to LTL model checking, but now a reduction that modifies the system instead of the formula. This reduction is based on the observation that TSL predicates can also be interpreted as atomic propositions in LTL. This does not work for the update terms, since whether an update term is true, depends not only on the current but also on the previous time step. Therefore, we describe an algorithm for eliminating all update terms from the TSL formula: to this end, our algorithm modifies the system such that each state 'remembers' the previous values of the cells and update terms. This allows expressing update terms as predicates. In a TSL formula without update terms, we can interpret the predicates as atomic propositions and then relabel our system, specifying at each state which predicates are currently true. Then, any LTL model checking algorithm can be used.

Update term elimination also works for HyperTSL. However, interpreting predicates as atomic propositions, does not, since atomic propositions can only refer to one trace. We instead adapt the HyperLTL model checking algorithm by Finkbeiner et. al [27]. Translating the formula to HyperLTL and then using the HyperLTL model checking algorithm by Finkbeiner et. al. would roughly mean instantiating all possible values in the formula, translating the quantifier-free part of the formula to an automaton and then combining this automaton with the system. Our idea for improving this algorithm is to first translate the quantifier-free part of the TSL formula into an automaton and then only instantiate the concrete values when combining this automaton with the system. This avoids an unnecessary blow-up of the formula automaton.

However, having predicates instead of concrete values as transition labels makes an important step of the HyperLTL model checking algorithm, the *projection*, impossible in our setting. This is why our HyperTSL model checking algorithm is limited to the fragment of HyperTSL with at most one quantifier alternation. To avoid the

projection, the idea when verifying a $\forall^*\exists^*$ HyperTSL formula is to construct an automaton containing all the execution combinations that satisfy the existential part of the formula. Then, it only remains to test whether this automaton contains all executions of the system.

Using our algorithms for TSL and HyperTSL model checking is indeed more efficient than translating the formula and using the HyperLTL model checking algorithm if the system is large, as we show in Chapter 5. Therefore, if one wants to model check a property that can be more compactly described using a (Hyper)TSL formula, like the property in Example 1.2, using Temporal Stream Logic as the specification language is a good choice, even if (Hyper)TSL is not more expressive for a finite-state system.

1.1.3 Software Model Checking

The more interesting use for (Hyper)TSL model checking is the case of an infinite-state system like a program. Model checking with a program as the system is called *software model checking*. We discuss a software model checking algorithm for (Hyper)TSL in Chapter 6. For TSL, our algorithm is an adaption of the automata-based LTL software model checking algorithm by Dietsch et. al [25]. They already express atomic propositions using predicates over memory cells, but we need to adapt the algorithm for update terms and predicates over inputs. Dietsch et. al. model the program as an automaton, where the states correspond to program locations and the transitions are labeled with program statements that can be assertions or variable assignments. Not every trace of this automaton is then a real program execution: for example, a trace $assert(n = 0)\ ; n--;\ (assert(n = 0))^\omega$ [1] can never be a program execution, as the second assertion will always fail. Such a trace is called *infeasible*. In contrast, in a *feasible* trace, all assertions might succeed.

In our algorithm, as in the finite-state case, the checked formula is translated to an automaton, but this time, the update terms are also treated as atomic propositions. As done by Dietsch et. al., we combine the formula automaton and the program automaton into a new program automaton called the *Büchi program product* that accepts any feasible trace if and only if the program satisfies the formula. Whether the automaton accepts a feasible trace is undecidable in general. Nevertheless, Dietsch et. al. describe an algorithm based on counterexample guided abstraction refinement [16], that can often decide whether there is a feasible accepted trace or not. This algorithm is also applicable in our setting.

The Büchi program product is similar to the automaton product of the formula automaton and the program automaton. Thereby, in our algorithm, two transition labels, a statement s and a set l of predicate and update terms are combined into a program statement in such a way, that the assertions in the resulting statement suc-

[1] The superscript ω denotes an infinite repetition of the program statement

ceed if the predicates and update terms are true after the execution of s. A feasible trace of the Büchi program product then corresponds to a program execution that does not fulfill the TSL formula.

We extend this algorithm for HyperTSL-formulas with only existential or only universal quantifiers by applying the technique of *self-composition*, a technique commonly used for the verification of hyperproperties [3, 4, 26]. If there are n quantifiers, the program automaton is combined $n - 1$ times with itself, such that a trace of the created automaton corresponds to n traces of the original system.

While there are already many important hyperproperties that can be expressed with a single type of quantifiers, some especially important ones require a combination of both. As an example, observe that a nondeterministic system can never satisfy noninterference as stated in Example 1.1 even if the output is not influenced by the secret input, as the output of two executions can differ even if all inputs are equal. It is possible to generalize noninterference to allow nondeterminism. *Generalized noninterference* [40] states that for any execution, there exists another execution with the same output, but with the secret input always being some dummy value. For expressing generalized noninterference, a combination of existential and universal quantifiers is necessary.

To the best of our knowledge, there does not exist a software model checking algorithm for HyperLTL formulas with both universal and existential quantifiers yet. So for this problem, there is no HyperLTL algorithm that we could extend to HyperTSL. In finite-state HyperLTL model checking algorithms, there is usually a complementation of the constructed automaton involved. When complementing a program automaton containing infeasible traces, the infeasible traces are lost. This makes the task of software model checking HyperLTL or HyperTSL formulas with different quantifiers especially difficult.

We develop an algorithm that, in many cases, can find counterexamples disproving $\forall^*\exists^*$-HyperTSL formulas (and therefore also witnesses proving $\exists^*\forall^*$-HyperTSL formulas by finding a counterexample for the negated formula). However, our algorithm is partial in the sense that if our algorithm does not find a counterexample, this does not necessarily mean that the system satisfies the formula.

For this algorithm, we again use an idea from the finite-state algorithm: Consider again a $\forall^*\exists^*$ HyperTSL formula. In the finite-state algorithm, we have constructed an automaton containing all the execution combinations from our system that satisfy the existential part of the formula. As this is not possible for an infinite-state system, we now instead construct an automaton that *overapproximates* the combinations of program executions satisfying the existential part of the formula. If there is a program execution that is not included in the overapproximation, this execution is a counterexample proving that the program does not satisfy the HyperTSL-formula.

To obtain such an overapproximation, we construct the Büchi program product of the automaton for ψ and the n-fold self-composition of the program automaton. Every feasible trace of the Büchi program product then corresponds to a choice of executions for the variables $\pi_1, \dots, \pi_n$ that makes ψ true. Then, we remove two kinds of infeasibility from the Büchi program product: first of all, we choose some k and remove all *k-infeasibility*, that is, a local inconsistency in a trace appearing within k consecutive timesteps. For example, the trace $assert(n = 0);\ n--;\ (assert(n = 0))^\omega$ is 3-infeasible. Whether a sequence of program executions is inconsistent can be checked using an SMT-solver. If k is larger, more counterexamples can be identified, but the algorithm is also slower.

As the second kind of infeasibility, we remove some *infeasible accepting cycles* from the automaton. For example, the trace $(assert(n \geq 0);\ n--)^\omega$ is not k-infeasible for any k, but still infeasible, as n cannot decrease forever without eventually dropping below zero. This trace is accepted by a program automaton if $assert(n \geq 0);\ n--;\ assert(n \geq 0)$ is an infeasible accepting cycle of the automaton. The infeasibility of a cycle can be proven using a ranking function synthesizer [5, 10, 20, 32, 44].

After some infeasible traces were removed using those algorithms, we project the automaton to the universally quantified traces, obtaining an overapproximation of the trace combinations satisfying the existential part of the formula. It remains to check whether the overapproximation includes all the combinations of feasible traces.

We apply the algorithm to two examples to show that there are indeed cases where it suffices to remove those kinds of infeasibility from the automaton to be able to identify a counterexample.

Contributions.

- We extend HyperTSL with theories, creating a version of HyperTSL suitable for model checking.
- For a finite-state system, we describe a reduction of (Hyper)TSL to (Hyper)LTL model checking that is based on 'hardcoding' all possible values for the memory cells and inputs inside the formula.
- We also describe direct finite-state model checking algorithms for TSL and $\forall^*\exists^*$ and $\exists^*\forall^*$ HyperTSL, which are more efficient than applying the reduction based on hardcoding all possible values if the system is large.
- To obtain an algorithm for TSL software model checking, we adapt the automaton-based LTL software model checking algorithm from [25].
- We extend this algorithm to HyperTSL formulas with only universal or existential quantifiers by applying the technique of self-composition.
- We further extend this to an algorithm for finding counterexamples disproving

$\forall^*\exists^*$ HyperTSL formulas (and, conversely, witnesses proving $\exists^*\forall^*$ formulas) that is sound but not complete. This also gives the, to the best of our knowledge, first software model checking algorithm for finding counterexamples for $\forall^*\exists^*$ HyperLTL formulas and proving $\exists^*\forall^*$ HyperLTL formulas

Chapter 2

Related Work

Temporal Stream Logic. Temporal stream logic extends linear temporal logic [43] and was originally designed for synthesis [28]. For synthesis, the logic was already successfully applied, for example for synthesizing the computer game 'Syntroids' [31] or for synthesizing smart contracts [30]. For advancing the smart contract synthesis, TSL was extended to HyperTSL [19], thereby adding the possibility to relate multiple program executions. All those applications however use a different version of TSL than presented in this thesis - there, all functions and predicates were left uninterpreted. While this choice made perfect sense for synthesis, having interpreted predicates is more desirable for other purposes like model checking. TSL was extended with theories by Finkbeiner et al. in [29]. Our definition differs slightly from theirs: Finkbeiner et al. define the satisfaction of an update term by syntactic comparison of the current program statement and the update term. Thus, for example, the program statement $c := c + 1$ would satisfy the update term $[\![c \leftarrowtail c + 1]\!]$ while the statement $c := c + 2 - 1$ would not. In model checking, we usually do not want to reason about the syntax of a program, thus in this thesis, we interpret update terms based on the current and the previous value assignment, as done in [39].

While there are algorithms for synthesizing a system from a TSL specification, to the best of our knowledge, there was no algorithm yet that is explicitly designed for model checking. There is some recent work on TSL satisfiability [29]. As TSL can encode programs, a satisfiability checker can also be used for model checking. However, an algorithm explicitly designed for model checking is likely to be more efficient. The reason for that is that encoding the program in the formula and then translating the formula to an automaton already leads to an automaton with a size exponential in the size of the program.
For LTL and HyperLTL, the model checking problem has been extensively studied both on finite and infinite-state models [7, 9, 14, 18, 23, 25, 27]. In this thesis, we rely on the fact that TSL extends LTL in order to extend LTL model-checking algorithms

to TSL.

Finite-State Model Checking. For a finite-state system, the model checking problem for LTL is well-known to be decidable. The classical approach is based on Büchi-automata: the negated property is translated into an automaton [2, 41, 47], then it is examined whether the intersection of the language of the automaton and the traces of the model is empty. This approach was adapted for hyperproperties, developing model checking algorithms for HyperLTL [17, 18, 27]. We show in Chapter 5 that for a finite-state system, (Hyper)TSL model checking is reducible to (Hyper)LTL model checking. Moreover, the LTL to Büchi automaton translation is used as the basis for all TSL model checking algorithms in this thesis.

Self-Compositon. Self-composition is a technique often used for the verification of hyperproperties, both in finite and infinite-state systems [3, 4, 26], exploiting the fact that the model checking problem for universal or existential hyperproperties can be reduced to a simple model checking problem in the composed system – each trace of the composed system corresponds to an interleaving of multiple traces in the original system. We use self-composition both for finite and infinite-state systems in Chapters 5 and 6.

Software Model Checking. The software model checking problem is already undecidable for LTL [8]. There are many approaches for partially solving the problem, like semi-algorithms that might diverge or return "unknown" if they can not determine the solution [14, 22, 23, 25]. Most of these approaches use counterexample guided abstraction refinement (CEGAR) [16]. The program is first abstracted to a finite-state model in a sound but incomplete manner, for example as a Büchi automaton, having program statements as its alphabet [33] or based on automatically inferred predicates [21]. Then, if in the abstraction an execution is found that does not fulfill the property, this execution is checked for *feasibility*, i.e. it is examined whether it corresponds to an execution of the original system. If not, the abstraction is refined. The problem of determining whether a given execution is feasible also is undecidable. In general, the construction of a ranking function is necessary - a proof that such a sequence must eventually terminate. Many algorithms have been developed for termination-proving [5, 10, 20, 32, 44]. They are often able to prove or disprove termination despite the general undecidability of the problem, but they are still very inefficient. For tackling this issue, the automata-based LTL software model checking algorithm from [25], which is the algorithm we extend for (Hyper)TSL in this thesis, exploits the fact that proving the infeasibility of a finite prefix of the sequence is mostly easier - first, some prefixes of the sequence are checked for feasibility by using SMT-solvers, then, only if they are feasible, a ranking function is synthesized.

However, there exist more recent approaches for LTL software model checking that are even more successful in practice. Many of them are based on the observation that model checking algorithms for safety properties are already well-studied and work

well in practice. The most known algorithm is called `IC3` [9] and has been extended for software model checking [14]. Reducing the model checking problem of a liveness property to that of a safety propery in another system leads to good model checking results [14, 23, 42]. This approach is called *liveness-to-safety*. It would be interesting to examine whether such an approach can also be extended to TSL.

There are also some existing algorithms for verifying hyperproperties on programs, for example, algorithms based on type theory [1, 6]. They are usually limited to universal hyperproperties. For example, [1] axiomatically introduces a new relational logic that can be used to prove relational properties.

Recently, an algorithm has been proposed for verifying the $\forall\exists$-fragment of the logic OHyperLTL that, like TSL, extends LTL with predicates, but is also able to express asynchronous hyperproperties [7]. However, this algorithm relies on a given abstraction and does not specify how to check the result for feasibility or refine the abstraction. Thus, it can only be applied for software model checking if a suitable abstraction is available.

Chapter 3

Preliminaries

3.1 Kripke Structures and Büchi Automata

Kripke structures are commonly used as the system that is model checked. A Kripke structure [38] is a finite transition system where the states are labeled with *atomic propositions*, each describing a property that at any time can be true or false.

Definition 3.1. A **Kripke structure** is a tuple $K = (S, s_0, \delta, AP, L)$ where S is a set of states, s_0 is the initial state, $\delta \subseteq S \times S$ is the transition relation, AP is the set of atomic propositions and $L : S \to 2^{AP}$ is the labeling function.

A **path** of K is an infinite sequence $s_0\ s_1\ s_2\ \cdots \in S^\omega$ of states of K, where $(s_i, s_{i+1}) \in \delta$ for all $i \in \mathbb{N}$. We denote all paths of K as $Paths(K)$. A **trace** of K is an infinite sequence $L(s_0)\ L(s_1)\ L(s_2) \cdots \in (2^{AP})^\omega$ where $s_0\ s_1\ s_2$ is a path of K.

A *Büchi automaton* [12] is also a transition system, but with labeled transitions instead of labeled states. Moreover, there is a distinguished subset of *accepting states*. Intuitively, a word is accepted by the Büchi automaton if the automaton, while reading it, can visit some accepting state infinitely often.

Definition 3.2. A **Büchi automaton** is a tuple $A = (\Sigma, Q, \delta, q_0, Q_{acc})$ where Σ is the finite alphabet, Q is a set of states, $\delta \subseteq Q \times \Sigma \times Q$ is the transition relation, q_0 is the initial state and $Q_{acc} \subseteq Q$ is the set of accepting states.

A **run** of the Büchi automaton on a word $\sigma \in \Sigma^\omega$ is an infinite sequence $q_0\ q_1\ q_2 \cdots \in Q^\omega$ of states such that for all $i \in \mathbb{N}, (q_i, \sigma_i, q_{i+1}) \in \delta$. An infinite word σ is **accepted** by the Büchi automaton if there is a run on σ with infinitely many $i \in \mathbb{N}$ such that $q_i \in Q_{acc}$. We denote with $\mathcal{L}(A)$ the set of words accepted by A.

Büchi automata are closed under intersection, union and complementation [12, 37, 45, 46]. Complementation however leads to an exponential blow-up in the number of states.

3.2 (Hyper) Linear Temporal Logic

Linear Temporal Logic (LTL) is a language for specifying trace properties, that is, properties of sequences of sets of atomic propositions. Besides the standard boolean operators, LTL also includes modal operators, namely

- the next-operator $\bigcirc$. $\bigcirc\varphi$ states that the formula φ should hold in the next timestep.
- the eventually-operator $\Diamond$. $\Diamond\varphi$ states that φ should hold at some timepoint in the future.
- the until-operator $\mathcal{U}$. $\varphi\mathcal{U}\psi$ states that the formula ψ must hold eventually and, until this is the case, φ must hold.
- the globally-operator $\Box$. $\Box\varphi$ states that φ must hold from now on and forever.

We only treat the next-operator and the until-operator as native and derive the other operators.

Definition 3.3. An **LTL Formula** is defined by the grammar

$$\varphi ::= a \mid \neg\varphi \mid \varphi \wedge \varphi \mid \bigcirc\varphi \mid \varphi\mathcal{U}\varphi \qquad \text{where } a \in AP$$

The operators $\vee, \Diamond, \Box$ can be derived using the equations $\varphi \vee \psi = \neg(\neg\varphi \wedge \neg\psi)$, $\Diamond\varphi = true\mathcal{U}\varphi$, $\Box\varphi = \neg\Diamond\neg\varphi$.

Definition 3.4. The **satisfaction of an LTL formula** with respect to a trace $s \in (2^{AP})^\omega$ and a time point t is recursively defined by

$$\begin{aligned}
t, s \vDash_{LTL} a &\iff a \in s_t \\
t, s \vDash_{LTL} \neg\varphi &\iff \neg(t, s \vDash_{LTL} \varphi) \\
t, s \vDash_{LTL} \varphi \wedge \psi &\iff t, s \vDash_{LTL} \varphi \wedge t, s \vDash_{LTL} \psi \\
t, s \vDash_{LTL} \bigcirc\varphi &\iff t+1, s \vDash_{LTL} \varphi \\
t, s \vDash_{LTL} \varphi\mathcal{U}\psi &\iff \exists t' \geq t.\ t', s \vDash_{LTL} \psi \wedge \forall t \leq t'' < t'.\ t'', s \vDash_{LTL} \varphi
\end{aligned}$$

We define $s \vDash_{LTL} \varphi$ as $0, s \vDash_{LTL} \varphi$

It is well known that it is possible to translate an LTL formula into an equivalent Büchi automaton that accepts exactly the traces satisfying the LTL formula [47]. However, if the formula is of size n, the Büchi automaton might have up to $2^{\mathcal{O}(n)}$ states. The classical translation algorithm has been improved, making it faster in practice and reducing the size of the automaton [2, 41].

HyperLTL [17] extends LTL with explicit trace quantification. An LTL formula has to hold for all traces of the system, so this is an implicit universal quantification.

Within an HyperLTL formula, we can use multiple (different) quantifiers and thereby relate multiple traces. Every atomic proposition is now attached to a trace variable. In the following, let Π be a set of trace variables.

Definition 3.5. A **HyperLTL formula** is defined by the grammar

$$\varphi ::= a_\pi \mid \neg\varphi \mid \varphi \wedge \varphi \mid \bigcirc\varphi \mid \varphi\mathcal{U}\varphi \mid \forall\pi.\ \varphi \mid \exists\pi.\ \varphi \quad \text{where } a \in AP \text{ and } \pi \in \Pi$$

The satisfaction of a HyperLTL formula is defined with respect to a mapping $m : \Pi \to (2^{AP})^\omega$ of trace variables to traces. For treating the quantifiers, we need the notion of extending such a mapping for a new trace variable. We define

$$\begin{aligned} m[\pi \to s](\pi) &= s \\ m[\pi \to s](\pi') &= m(\pi') && \text{for } \pi \neq \pi' \end{aligned}$$

Definition 3.6. The **satisfaction of a HyperLTL formula** with respect to a set of traces $Z \subseteq 2^{AP^\omega}$, a mapping of trace variables to traces $m : \Pi \to 2^{AP^\omega}$ and a time point t is recursively defined by

$$\begin{aligned}
Z, t, m &\vDash_{LTL} a_\pi && \Leftrightarrow a \in m(\pi)_t \\
Z, t, m &\vDash_{LTL} \neg\varphi && \Leftrightarrow \neg(Z, t, m \vDash_{LTL} \varphi) \\
Z, t, m &\vDash_{LTL} \varphi \wedge \psi && \Leftrightarrow Z, t, m \vDash_{LTL} \varphi \wedge Z, t, m \vDash \psi \\
Z, t, m &\vDash_{LTL} \bigcirc\varphi && \Leftrightarrow Z, t+1, m \vDash_{LTL} \varphi \\
Z, t, m &\vDash_{LTL} \varphi\mathcal{U}\psi && \Leftrightarrow \exists t' \geq t.\ Z, t', m \vDash_{LTL} \psi \wedge \forall t \leq t'' < t'.\ Z, t'', m \vDash_{LTL} \varphi \\
Z, t, m &\vDash_{LTL} \forall\pi.\ \varphi && \Leftrightarrow \forall s \in Z.\ m[\pi \to s] \vDash_{LTL} \varphi \\
Z, t, m &\vDash_{LTL} \exists\pi.\ \varphi && \Leftrightarrow \exists s \in Z.\ m[\pi \to s] \vDash_{LTL} \varphi
\end{aligned}$$

We define $Z \vDash_{LTL} \varphi$ as $Z, 0, \emptyset \vDash_{LTL} \varphi$.

Chapter 4

(Hyper) Temporal Stream Logic with Theories

4.1 Recap: Temporal Stream Logic with Theories

Temporal Stream Logic [28] extends Linear Temporal Logic [43]. While LTL is based on boolean atomic propositions, TSL is instead based on predicates over memory cells and inputs. Moreover, it supports *update terms* - terms that specify how the value of a cell should change. The following example illustrates the basic usage of TSL.

Example 4.1. This formula states that the cell `counter` should count how often the cell `money` drops below zero.

$$\square((money > 0) \land \bigcirc(money \leq 0) \rightarrow \bigcirc\bigcirc[\![counter \leftarrowtail counter + 1]\!])$$

Whenever `money` has a positive value, but in the next timestep not anymore, then in the step after the cell `counter` should be increased by one.

We present the formal definition of temporal stream logic with theories, based on the definition by Finkbeiner et al. [29], which is an extension of the original TSL definition [28]. The definition we present slightly differs from the definition by Finkbeiner et al.: the satisfaction of an update term is not defined by syntactic comparison, but relative to the current and the previous values of cells and inputs. This adaption was already presented in [39]. We decided on this version because we usually do not want to reason about the syntax used in the description of the system when model checking.

TSL is defined based on a set of *values* $\mathbb{V}$ containing distinguished elements *true* and *false*, a set of inputs $\mathbb{I}$ and a set of memory cells $\mathbb{C}$. Update terms and predicates are interpreted with respect to a given theory.

Definition 4.2. A **theory** is a tuple $(\mathbb{F}, \varepsilon)$

- $\mathbb{F}$ is the set of function symbols, each one with an arity n.
- $\varepsilon : \mathbb{F} \times \mathbb{V}^n \to \mathbb{V}$ is the interpretation function, evaluating a function with arity n given concrete argument values

In this thesis, we now assume a concrete theory $(\mathcal{T}_{\mathcal{F}}, \varepsilon)$ containing at least the equals-predicate, disjunction and negation. Next, we formally present how function terms, predicate terms and TSL formulas are constructed using function symbols, cells and inputs.

Definition 4.3. A **function term** τ_F is defined by the following grammar:

$$\tau_F ::= c \mid i \mid f(\tau_F,\ \tau_F,\ \dots\ \tau_F)$$

where $c \in \mathbb{C}, i \in \mathbb{I}, f \in \mathbb{F}$, and the number of tuple elements matches the function arity.

Definition 4.4. An **assignment** $a : (\mathbb{I} \cup \mathbb{C}) \to \mathbb{V}$ is a function assigning values to inputs and cells. The set of all assignments is $\mathcal{A}$.

Given a concrete assignment, the value of a function term can be computed.

Definition 4.5. The **evaluation function** $\eta : \mathcal{T}_{\mathcal{F}} \times \mathcal{A} \to \mathbb{V}$ is defined by

$$\begin{aligned} \eta(c, a) &= a(c) && \text{for } c \in \mathbb{C} \\ \eta(i, a) &= a(i) && \text{for } i \in \mathbb{I} \\ \eta(f\ (\tau_{F1}, \tau_{F2}, \dots, \tau_{Fn}), a) &= \varepsilon(f, (\eta(\tau_{F1}), \eta(\tau_{F2}), \dots, \eta(\tau_{Fn}))) && \text{for } f \in \mathbb{F} \end{aligned}$$

Definition 4.6. A **predicate term** τ_P is a function term only evaluating to true or false:

$$\forall a \in \mathcal{A}.\ \eta(\tau_P, a) = true \lor \eta(\tau_P, a) = false$$

The set of all predicate terms is $\mathcal{T}_P$.

Definition 4.7. Let $c \in \mathbb{C}$ and $\tau_F \in \mathcal{T}_{\mathcal{F}}$. Then, $[\![c \leftarrowtail \tau_F]\!]$ is called an **update term**.

Intuitively, the update term $[\![c \leftarrowtail \tau_F]\!]$ states that c should be updated to τ_F. If in the previous time step τ_F evaluated to $v \in \mathbb{V}$, then in the current time step c should have value v. The set of all update terms is $\mathcal{T}_U$.

Definition 4.8. A **TSL-Formula** is defined by the grammar:

$$\varphi ::= \tau_P \mid [\![c \leftarrowtail \tau_F]\!] \mid \neg\varphi \mid \varphi \land \varphi \mid \bigcirc\varphi \mid \varphi\,\mathcal{U}\,\varphi$$

where $c \in \mathbb{C}, \tau_P \in \mathcal{T}_P, \tau_F \in \mathcal{T}_{\mathcal{F}}$.

The usual operators $\vee, \Diamond, \Box$ can be derived using the equations
$\varphi \vee \psi = \neg(\neg\varphi \wedge \neg\psi)$, $\Diamond\varphi = true\ \mathcal{U}\ \varphi$, $\Box\varphi = \neg\Diamond\neg\varphi$, where $\varepsilon(true, ()) = true$

Next, we present the definition of the semantics of TSL. For that, assume a fixed initial variable assignment ζ_{-1}. (This could for example map all cells and inputs to zero)

Definition 4.9. The **satisfaction of a TSL-Formula** with respect to a *computation* $\zeta \in \mathcal{A}^{\omega}$ and a time point t is recursively defined by

$$\begin{aligned}
t, \zeta &\vDash \tau_P && \Leftrightarrow \eta(\tau_P, \zeta_t) = true \\
t, \zeta &\vDash [\![c \leftarrowtail \tau_F]\!] && \Leftrightarrow \eta(\tau_F, \zeta_{t-1}) = \zeta_t(c) \\
t, \zeta &\vDash \neg\varphi && \Leftrightarrow \neg(t, \zeta \vDash \varphi) \\
t, \zeta &\vDash \varphi \wedge \psi && \Leftrightarrow t, \zeta \vDash \varphi \wedge t, \zeta \vDash \psi \\
t, \zeta &\vDash \bigcirc\varphi && \Leftrightarrow t+1, \zeta \vDash \varphi \\
t, \zeta &\vDash \varphi\,\mathcal{U}\,\psi && \Leftrightarrow \exists t' \geq t.\ t', \zeta \vDash \psi \wedge \forall t \leq t'' < t'.\ t'', \zeta \vDash \varphi
\end{aligned}$$

We define $\zeta \vDash \varphi$ as $0, \zeta \vDash \varphi$

4.2 HyperTSL with Theories

The definition from the previous section is now extended with path quantifiers, leading to the definition of the logic HyperTSL [19]. Using HyperTSL, many important security properties are expressible by relating multiple program executions.

Example 4.10. (Noninterference)
By observing the output o, an observer cannot gain information about the secret input h

$$\forall\pi.\ \forall\pi'.\ \Box(\bigwedge_{i \in \mathbb{I} \setminus h} i_\pi = i_{\pi'}) \rightarrow \Box(o_\pi = o_{\pi'})$$

The HyperLTL specification of noninterference looks very similar - however, there the inputs and outputs can only be boolean atomic propositions, while in HyperTSL, they can have arbitrary values. This makes HyperTSL more expressive, as there is no need for a finite value domain.

HyperTSL was first presented for the synthesis of smart contracts [19]. There, two different versions of HyperTSL were introduced and discussed: the first version, called HyperTSL, did not allow the relation of multiple traces within one predicate, while the second one, called HyperTSL$_{rel}$, did allow them. Many important security properties like noninterference require HyperTSL$_{rel}$. Nevertheless, the authors of [19] focused on the more restrictive version, as otherwise, the synthesis problem would have been

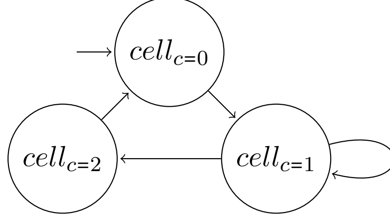

Figure 4.1: An automaton. The annotation $cell_{c=v}$ means that at this automaton state, the cell c has value 0

even harder. In this thesis, we instead use the more expressive version (we even allow update terms that relate multiple traces), as this version seems suitable for model checking. Furthermore, we extend the originally uninterpreted functions and predicates with an interpretation. For simplicity, we still call the logic HyperTSL.

The syntax of HyperTSL is like that of TSL, with the difference that the cells and inputs are now each assigned to a trace variable that represents a computation. For example, c_π now refers to the memory cell c in the computation represented by π. Formally, let Π be a set of trace variables. We define a *hyper-function term* $\hat{\tau_F} \in \hat{\mathcal{T}_F}$ as a function term using $(\mathbb{I} \times \Pi)$ as the set of inputs and $(\mathbb{C} \times \Pi)$ as the set of cells:

Definition 4.11. A **hyper-function term** $\hat{\tau_F}$ is defined by the following grammar:

$$\hat{\tau_F} ::= c_\pi \mid i_\pi \mid f(\hat{\tau_F},\ \hat{\tau_F},\ \dots\ \hat{\tau_F})$$

where $c_\pi \in \mathbb{C} \times \Pi, i_\pi \in \mathbb{I} \times \Pi, f \in \mathbb{F}$, and the number of tuple elements matches the function arity. $\hat{\mathcal{T}_F}$ is the set of all hyper-function terms.

Analogously, we also define *hyper-predicate terms* $\hat{\tau_P} \in \hat{\mathcal{T}_P}$ as hyper-function terms only evaluating to true or false, *hyper-assignments* $\hat{\mathcal{A}} = (\mathbb{I} \cup \mathbb{C}) \times \Pi \to \mathbb{V}$ as functions that map each cell and input of each trace to their current value, and *hyper-computations* $\hat{\zeta} \in \hat{\mathcal{A}}^\omega$ as sequences of hyper-assignments.

Example 4.12. Consider the automaton shown in Figure 4.1 and the two automaton traces

$$\pi = cell_{c=0}\ {cell_{c=1}}^\omega$$
$$\pi' = (cell_{c=0}\ cell_{c=1}\ cell_{c=2})^\omega$$

Those automaton traces can be each interpreted as a computation. When executing both traces simultaneously, at each point in time, there is a corresponding hyper-assignment - an assignment of values to c_π and $c_{\pi'}$. In the first time step, c is 0 at both traces, so the corresponding hyper-assignment is $\hat{a}_1(c_\pi) = \hat{a}_1(c'_\pi) = 0$. Analogously,

for the next timesteps we have

$$\hat{a}_2(c_\pi) = 1, \qquad \hat{a}_2(c'_\pi) = 1 \qquad \text{second time step}$$
$$\hat{a}_3(c_\pi) = 1, \qquad \hat{a}_3(c'_\pi) = 2 \qquad \text{third time step}$$
$$\hat{a}_4(c_\pi) = 1, \qquad \hat{a}_4(c'_\pi) = 0 \qquad \text{fourth time step}$$

Thus, the two traces define a hyper-computation $\hat{a}_1\ (\hat{a}_2\ \hat{a}_3\ \hat{a}_4)^\omega$

We can now define the syntax of HyperTSL. It is similar to the syntax of TSL but allows path quantifiers, hyper-predicate terms instead of predicate terms and hyper-function terms instead of function terms.

Definition 4.13. A **HyperTSL-formula** is defined by the following grammar:

$$\varphi ::= \psi \mid \forall \pi.\ \varphi \mid \exists \pi.\ \varphi$$
$$\psi ::= \hat{\tau_P} \mid [\![c_\pi \leftarrowtail \hat{\tau_F}]\!] \mid \neg\psi \mid \psi \wedge \psi \mid \bigcirc \psi \mid \psi \mathcal{U} \psi$$

where $c_\pi \in \mathbb{C} \times \Pi, \hat{\tau_P} \in \hat{\mathcal{T}_P}, \hat{\tau_F} \in \hat{\mathcal{T}_F}$.

When defining the semantics of HyperTSL, we need to extend a hyper-computation by a computation for a new trace variable. Then, we can extend the current set of traces with a new one for each path quantifier.

Definition 4.14. (**Extension of a Hyper-Computation**)
Let $\hat{\zeta} \in \hat{\mathcal{A}}^\omega, \pi, \pi' \in \Pi, \zeta \in \mathcal{A}^\omega, x \in (\mathbb{I} \cup \mathbb{C})$. We define $\hat{\zeta}[\pi, \zeta]$ as

$$\hat{\zeta}[\pi, \zeta]\ (x_\pi) = \zeta(x_\pi)$$
$$\hat{\zeta}[\pi, \zeta]\ (x_{\pi'}) = \hat{\zeta}(x_{\pi'}) \quad \text{for } \pi' \neq \pi$$

Definition 4.15. The **satisfaction of a HyperTSL-Formula** with respect to a hyper-computation $\hat{\zeta} \in \hat{\mathcal{A}}^\omega$, a set of computations Z and a time point t is recursively defined by

$$\begin{aligned}
t, Z, \hat{\zeta} &\vDash \forall \pi.\ \varphi && \Leftrightarrow \forall \zeta \in Z.\ t,\ Z,\ \hat{\zeta}[\pi, \zeta] \vDash \varphi \\
t, Z, \hat{\zeta} &\vDash \exists \pi.\ \varphi && \Leftrightarrow \exists \zeta \in Z.\ t,\ Z,\ \hat{\zeta}[\pi, \zeta] \vDash \varphi \\
t, Z, \hat{\zeta} &\vDash \hat{\tau_P} && \Leftrightarrow \eta(\hat{\tau_P}, \hat{\zeta}_t) = true \\
t, Z, \hat{\zeta} &\vDash [\![c_\pi \leftarrowtail \hat{\tau_F}]\!] && \Leftrightarrow \eta(\hat{\tau_F}, \hat{\zeta}_{t-1}) = \hat{\zeta}_t(c_\pi) \\
t, Z, \hat{\zeta} &\vDash \neg\varphi && \Leftrightarrow \neg(t, Z, \hat{\zeta} \vDash \varphi) \\
t, Z, \hat{\zeta} &\vDash \varphi \wedge \psi && \Leftrightarrow t, Z, \hat{\zeta} \vDash \varphi \wedge t, Z, \hat{\zeta} \vDash \psi \\
t, Z, \hat{\zeta} &\vDash \bigcirc \varphi && \Leftrightarrow t+1, Z, \hat{\zeta} \cdots \vDash \varphi \\
t, Z, \hat{\zeta} &\vDash \varphi \mathcal{U} \psi && \Leftrightarrow \exists t' \geq t.\ t', Z, \hat{\zeta} \vDash \psi \wedge \forall t \leq t'' < t'.\ t'', Z, \hat{\zeta} \vDash \varphi
\end{aligned}$$

We define $Z \vDash \varphi$ as $0, Z, \varnothing^\omega \vDash \varphi$

4.3 Similiarity of LTL and TSL

In this section, we prove a lemma that states an important relation between (Hyper)TSL and LTL. The LTL semantics is defined with respect to a sequence of subsets of atomic propositions, while the semantics of a TSL-formula or quantifier-free HyperTSL formula is defined with respect to a (hyper-)computation. A crucial observation for this thesis is that we can 'translate' between the two – a (hyper-)computation defines a sequence of predicate and update term subsets. For each time point, the subset contains exactly the predicate and update terms that are true now.

Definition 4.16. Let $\hat{\zeta} \in \hat{\mathcal{A}}^\omega, \rho \subseteq \hat{\mathcal{T}_P}, \upsilon \subseteq \hat{\mathcal{T}_U}$. We define

$$\begin{aligned} Seq(\hat{\zeta}, \rho, \upsilon)_t &= \{\hat{\tau_P} \in \rho \mid t, \varnothing, \hat{\zeta} \vDash \hat{\tau_P}\} \cup \{[\![c \leftarrowtail \hat{\tau_F}]\!] \in \upsilon \mid t, \varnothing, \hat{\zeta} \vDash [\![c \leftarrowtail \tau_F]\!]\} \\ Seq(\hat{\zeta}, \rho, \upsilon) &= Seq(\hat{\zeta}, \rho, \upsilon)_0 \; Seq(\hat{\zeta}, \rho, \upsilon)_1 \; Seq(\hat{\zeta}, \rho, \upsilon)_2 \dots \end{aligned}$$

If ρ and υ are clear from the context, we also omit these arguments.

Lemma 4.17. *Let $t \in \mathbb{N}$. Let φ be a HyperTSL-formula without quantifiers. Let $\rho \subseteq \hat{\mathcal{T}_P}, \upsilon \subseteq \hat{\mathcal{T}_U}$ be the sets of predicate and update terms appearing in φ, respectively. Then*

$$t, Seq(\hat{\zeta}) \vDash_{LTL} \varphi \Leftrightarrow t, \varnothing, \hat{\zeta} \vDash \varphi$$

Proof. Proof by structural induction over φ.

- Case $\varphi = \hat{\tau_P}$

$$t, Seq(\hat{\zeta}) \vDash_{LTL} \hat{\tau_P} \Leftrightarrow \hat{\tau_P} \in Seq(\hat{\zeta})_t \Leftrightarrow t, \varnothing, \hat{\zeta} \vDash \hat{\tau_P}$$

- Case $\varphi = [\![c_\pi \leftarrowtail \hat{\tau_F}]\!]$

$$t, Seq(\hat{\zeta}) \vDash_{LTL} [\![c_\pi \leftarrowtail \hat{\tau_F}]\!] \Leftrightarrow \hat{\tau_P} \in Seq(\hat{\zeta})_t \Leftrightarrow t, \varnothing, \hat{\zeta} \vDash [\![c_\pi \leftarrowtail \hat{\tau_F}]\!]$$

- Case $\varphi = \neg\psi$

$$t, Seq(\hat{\zeta}) \vDash_{LTL} \neg\psi \Leftrightarrow \neg(t, Seq(\hat{\zeta}) \vDash_{LTL} \psi) \Leftrightarrow \neg(t, \varnothing, \hat{\zeta} \vDash \psi) \Leftrightarrow t, \varnothing, \hat{\zeta} \vDash \neg\psi$$

- Case $\varphi = \psi \wedge \psi'$

$$\begin{aligned} & t, Seq(\hat{\zeta}) \vDash_{LTL} \psi \wedge \psi' \\ \Leftrightarrow \quad & t, Seq(\hat{\zeta}) \vDash_{LTL} \psi \wedge t, Seq(\hat{\zeta}) \vDash_{LTL} \psi' \\ \Leftrightarrow \quad & t, \varnothing, \hat{\zeta} \vDash \psi \wedge t, \varnothing, \hat{\zeta} \vDash \psi' \\ \Leftrightarrow \quad & t, \varnothing, \hat{\zeta} \vDash \psi \wedge \psi' \end{aligned}$$

- Case $\varphi = \bigcirc \psi$

$$t, Seq(\hat{\zeta}) \vDash_{LTL} \bigcirc \psi \Leftrightarrow t+1, Seq(\hat{\zeta}) \vDash_{LTL} \psi \Leftrightarrow t+1, \varnothing, \hat{\zeta} \vDash \psi \Leftrightarrow t, \varnothing, \hat{\zeta} \vDash \bigcirc \psi$$

- Case $\varphi = \psi \mathcal{U} \psi'$

$$\begin{aligned}
& t, Seq(\hat{\zeta}) \vDash_{LTL} \psi \mathcal{U} \psi' \\
\Leftrightarrow \quad & \exists t' \geq t.\ t', Seq(\hat{\zeta}) \vDash_{LTL} \psi' \wedge \forall t \leq t'' < t'.\ t'', Seq(\hat{\zeta}) \vDash_{LTL} \psi \\
\Leftrightarrow \quad & \exists t' \geq t.\ t', Z, \hat{\zeta} \vDash \psi' \wedge \forall t \leq t'' < t'.\ t'', Z, \hat{\zeta} \vDash \psi \\
\Leftrightarrow \quad & t, \varnothing, \hat{\zeta} \vDash \psi \mathcal{U} \psi'
\end{aligned}$$

□

Chapter 5

Finite-State Model Checking

In this chapter, we study TSL and HyperTSL model checking algorithms for a finite-state system, that is, a system where each input and cell has only finitely many possible values. As the system, we use a Kripke structure specifying the values of each input and cell at each state. We call such a Kripke structure a *TSL Kripke structure* and define it in section 5.1. In section 5.2 we show that when using a TSL-Kripke structure as the model, (Hyper)TSL model checking is reducible to (Hyper)LTL model checking. For this reduction, we 'hardcode' the predicate and update term evaluations for all possible values that appear in the Kripke structure in the (Hyper)LTL formula. This gives a (Hyper)TSL to (Hyper)LTL translation.

The problem with using this reduction for (Hyper)TSL model checking is that by translating to (Hyper)LTL, the size of the formula is increased a lot, slowing down the model checking algorithm. Therefore, we propose a second reduction from TSL to LTL model checking in section 5.3 that is based on modifying the system instead of the formula. In section 5.4 we directly modify the HyperLTL model checking algorithm by Finkbeiner et. al. [27] for HyperTSL with at most one quantifier alternation. Next, we show that applying our algorithms for TSL and HyperTSL model checking is more efficient than translating the formula and then applying the (Hyper)LTL model checking algorithm.

5.1 TSL Kripke Structures

When doing model checking for TSL, we first need to define how our model defines the current input and cell values as they are at one state. In this chapter, we use a Kripke structure where for each input $i \in \mathbb{I}$ and possible value $v \in \mathbb{V}$, there is an atomic proposition $in_{i=v}$ that is true when i has value v. Analogously, for each cell $c \in \mathbb{C}$ and possible value $v \in \mathbb{V}$, there is an atomic proposition $cell_{c=v}$ that is true

when cell c is currently v.

Definition 5.1 (TSL-Kripke Structure)**.** A Kripke Structure $K = (S, s_0, \delta, AP, L)$ is called a **TSL-Kripke Structure** if

1. $AP = \bigcup_{i \in \mathbb{I}, v \in \mathbb{V}} in_{i=v} \cup \bigcup_{c \in \mathbb{C}, v \in \mathbb{V}} cell_{c=v}$
2. $\forall s \in S, i \in \mathbb{I}.\ \exists! v \in \mathbb{V}.\ in_{i=v} \in L(s)$
3. $\forall s \in S, c \in \mathbb{C}.\ \exists! v \in \mathbb{V}.\ cell_{c=v} \in L(s)$

The second and third conditions ensure that every cell and input have exactly one current value at any state. A state s of the TSL-Kripke structure thus defines an assignment $a_s \in \mathcal{A}$:

$$\begin{aligned} a_s(i) &:= v && \text{if and only if } in_{i=v} \in L(s) \\ a_s(c) &:= v && \text{if and only if } cell_{c=v} \in L(s) \end{aligned}$$

In the following, we additionally require that $a_{s_0} = \zeta_{-1}$, i.e. the initial state of the TSL-Kripke structure matches the initial assignment.

Then, a path $\sigma = s_0, s_1, \ldots$ of a TSL-Kripke structure defines a computation $\zeta_\sigma = a_{s_1}, a_{s_2}, \ldots$. We say that a TSL-Kripke structure K satisfies TSL-formula φ if for all paths σ of K, $\zeta_\sigma \vDash \varphi$. Analogously, we say that K satisfies a HyperTSL-formula φ if $\{\zeta_\sigma \mid \sigma \in Paths(K)\} \vDash \varphi$. If a TSL-Kripke structure satisfies a TSL or HyperTSL formula φ, we write $K \vDash \varphi$

In the following, we give a translation algorithm that translates a (Hyper)TSL formula to an equivalent (Hyper)LTL formula given a concrete TSL-Kripke structure. This already gives a model checking algorithm, as we can then use (Hyper)LTL model checking to solve the (Hyper)TSL model checking problem.

5.2 Translation to (Hyper)LTL

In this section, we present a translation algorithm that, given a TSL-Formula φ, outputs an LTL-Formula φ', such that K satisfies φ if and only if K satisfies φ'. The idea is that as there are only finitely many value combinations appearing at states of the Kripke structure, we can "hardcode" the function evaluation for these value combinations in the LTL formula.

First, for each state s, we define a formula that is true when we are in state s, i.e. the truth values of the atomic propositions are as in s:

$$cur_s := \bigwedge_{a \in L(s)} a$$

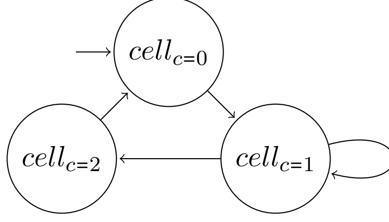

Figure 5.1: A TSL Kripke structure

We now define the translation function t_K that translates TSL to LTL given a concrete Kripke structure K:

$$
\begin{aligned}
t_K(\neg\varphi) &:= \neg t_K(\varphi)\\
t_K(\bigcirc\varphi) &:= \bigcirc\ t_K(\varphi)\\
t_K(\varphi \wedge \psi) &:= t_K(\varphi) \wedge t_K(\psi)\\
t_K(\varphi \mathcal{U} \psi) &:= t_K(\varphi)\ \mathcal{U}\ t_K(\psi)\\
t_K(\tau_P) &:= \bigvee_{s \in S \text{ with } \eta(\tau_P, a_s) = true} \bigcirc cur_s\\
t_K([\![c \leftarrowtail \tau_F]\!]) &:= \bigwedge_{s\in S} (cur_s \Rightarrow \bigcirc\ cell_{c=v_s}) \qquad \text{where } v_s := \eta(\tau_F, a_s)
\end{aligned}
$$

In the case of a predicate term, for the formula to be satisfied, we need to be in a state where the predicate term evaluates to true. The next-operator is necessary because the computation ‘starts one step earlier’ than the path of the Kripke structure, as the initial assignment is also included.

In the case of an update term, depending on what state we are in, the value of the cell has to be changed at the next state. For all combinations of atomic propositions that appear at one of the states, we define how the cell value has to change in that case.

Example 5.2. Let K be the TSL Kripke structure shown in Figure 5.1. Then:

$$
\begin{aligned}
t_K(\ \square(\ [\![c \leftarrowtail (c+1)\ mod\ 3]\!] \vee c = 1)\ ) = \square(((&cell_{c=0} \Rightarrow \bigcirc cell_{c=1}) \wedge\\
(&cell_{c=1} \Rightarrow \bigcirc cell_{c=2}) \wedge\\
(&cell_{c=2} \Rightarrow \bigcirc cell_{c=0})) \vee \bigcirc cell_{c=1})
\end{aligned}
$$

Theorem 5.3. *For a TSL-formula φ and TSL-Kripke structure K, K satisfies φ if and only if K satisfies $t_K(\varphi)$*

Proof. We show that for all $\sigma \in Paths(K), t \in \mathbb{N}$, $t, L(\sigma) \vDash \varphi$ if and only if $t, \zeta_\sigma \vDash \varphi$. Proof by structural induction over φ.

Case $\varphi = \tau_P$

$$
\begin{aligned}
& t, L(\sigma) \vDash_{LTL} t_K(\tau_P) \\
\Leftrightarrow\quad & t, L(\sigma) \vDash_{LTL} \bigvee_{s \in S \text{ with } \eta(\tau_P, a_s) = true} \bigcirc cur_s \\
\Leftrightarrow\quad & t+1, L(\sigma) \vDash_{LTL} \bigvee_{s \in S \text{ with } \eta(\tau_P, a_s) = true} cur_s \\
\Leftrightarrow\quad & \eta(\tau_P, a_{\sigma_{t+1}}) = true && \text{as } cur_s \Leftrightarrow (a_s = a_{\sigma_{t+1}}) \\
\Leftrightarrow\quad & \zeta_\sigma \vDash \tau_P
\end{aligned}
$$

Case $\varphi = [\![c \leftarrowtail \tau_F]\!]$

$$
\begin{aligned}
& t, L(\sigma) \vDash_{LTL} t_K([\![c \leftarrowtail \tau_F]\!]) \\
\Leftrightarrow\quad & t, L(\sigma) \vDash_{LTL} \bigwedge_{s \in S} (cur_s \Rightarrow \bigcirc\ cell_{c=v_s}) && \text{where } v_s := \eta(\tau_F, a_s) \\
\Leftrightarrow\quad & t, L(\sigma) \vDash_{LTL} \bigcirc cell_{c=v_{\sigma t}} && \text{where } v_{\sigma_t} := \eta(\tau_F, a_{\sigma_t}) \\
\Leftrightarrow\quad & t+1, L(\sigma) \vDash_{LTL} cell_{c=v_{\sigma t}} \\
\Leftrightarrow\quad & \eta(\tau_F, a_{\sigma_t}) = a_{\sigma_{t+1}}(c) \\
\Leftrightarrow\quad & t, \zeta_\sigma \vDash [\![c \leftarrowtail \tau_F]\!]
\end{aligned}
$$

Case $\varphi = \neg\psi$

$$
\begin{aligned}
& t, L(\sigma) \vDash_{LTL} t_K(\neg\psi) \\
\Leftrightarrow\quad & t, L(\sigma) \vDash_{LTL} \neg t_K(\psi) \\
\Leftrightarrow\quad & \neg(t, L(\sigma) \vDash_{LTL} t_K(\psi)) \\
\Leftrightarrow\quad & \neg(t, \zeta_\sigma \vDash \psi) \\
\Leftrightarrow\quad & t, \zeta_\sigma \vDash \neg\psi
\end{aligned}
$$

Case $\varphi = \bigcirc\psi$

$$
\begin{aligned}
& t, L(\sigma) \vDash_{LTL} t_K(\bigcirc\psi) \\
\Leftrightarrow\quad & t, L(\sigma) \vDash_{LTL} \bigcirc t_K(\psi) \\
\Leftrightarrow\quad & t+1, L(\sigma) \vDash_{LTL} t_K(\psi) \\
\Leftrightarrow\quad & t+1, \zeta_\sigma \vDash \psi \\
\Leftrightarrow\quad & t, \zeta_\sigma \vDash \bigcirc\psi
\end{aligned}
$$

Case $\varphi = \psi \wedge \psi'$

$$
\begin{aligned}
& t, L(\sigma) \vDash t_K(\psi \wedge \psi') \\
\Leftrightarrow \quad & t, L(\sigma) \vDash t_K(\psi) \wedge t_K(\psi') \\
\Leftrightarrow \quad & t, L(\sigma) \vDash t_K(\psi) \wedge t, L(\sigma) \vDash t_K(\psi') \\
\Leftrightarrow \quad & t, \zeta_\sigma \vDash \psi \wedge t, \zeta_\sigma \vDash \psi' \\
\Leftrightarrow \quad & t, \zeta_\sigma \vDash \psi \wedge \psi'
\end{aligned}
$$

Case $\varphi = \psi \,\mathcal{U}\, \psi'$

$$
\begin{aligned}
& t, L(\sigma) \vDash_{LTL} t_K(\psi \,\mathcal{U}\, \psi') \\
\Leftrightarrow \quad & t, L(\sigma) \vDash_{LTL} t_K(\psi) \,\mathcal{U}\, t_K(\psi') \\
\Leftrightarrow \quad & \exists t' \geq t.\ t', L(\sigma) \vDash_{LTL} t_K(\psi) \wedge \forall t \leq t'' < t'.\ t'', L(\sigma) \vDash_{LTL} t_K(\psi') \\
\Leftrightarrow \quad & \exists t' \geq t.\ t', \zeta_\sigma \vDash \psi \wedge \forall t \leq t'' < t'.\ t'', \zeta_\sigma \vDash \psi' \\
\Leftrightarrow \quad & \zeta_\sigma \vDash \psi \,\mathcal{U}\, \psi'
\end{aligned}
$$

□

We can now check whether K satisfies φ by checking whether K satisfies $t_K(\varphi)$. This can be done by any known model checking algorithm for LTL.

Reducing the size of the formula. To avoid an unnecessary blow-up when translating function terms to LTL, one may only include the propositions in the translated LTL formula that are actually relevant for the result of the computation. This can be done for example by the following function θ that given a function term and a state s constructs an LTL-formula that is true if and only if the values of the relevant cells and inputs are as in s.

$$
\begin{aligned}
\theta(i, s) &:= in_{i=v} && \text{iff } in_{i=v} \in L(s) \\
\theta(c, s) &:= cell_{c=v} && \text{iff } cell_{c=v} \in L(s) \\
\theta(f\ (\tau_0,\ \tau_1,\ ...\tau_{k-1}), s) &:= \theta(\tau_0, s) \wedge \theta(\tau_1, s) \wedge ... \wedge \theta(\tau_{k-1}, s)
\end{aligned}
$$

For the translation of a function term or predicate term τ we can then use $\theta(\tau, s)$ instead of cur_s. Moreover, one may eliminate unnecessary duplicate occurrences of propositions and subformulas that will be created for example when for a function term τ there are multiple states where $\theta(\tau, s)$ is equal. This can dramatically reduce the size of the formula when there are functions whose arguments only have a few possible values.

For every update and predicate term in the TSL-formula φ, at most $|S|$ new subformulas are created. As the size of $\theta(\psi, s)$ is bound by the number of input and cell

terms appearing in the update/predicate term, the sizes of those subformulas are not greater than the size of the original update/predicate term, so the total size of the translated formula is at most $|\varphi| \cdot |S|$ (the exact number depends of course on the exact definition of "size")

HyperTSL. The previous construction can be easily extended to obtain a translation from HyperTSL into HyperLTL, relative to a concrete TSL-Kripke structure K.

Now, the result of a function or predicate term might depend on multiple states instead of only one. Every mapping of the trace variables to states $m : \Pi \to S$ defines a hyper-assignment $\hat{a}_m$:

$$\hat{a}_m(i_\pi) := v \qquad \text{if and only if } in_{i=v} \in L(m(\pi))$$
$$\hat{a}_m(c_\pi) := v \qquad \text{if and only if } cell_{c=v} \in L(m(\pi))$$

For every such mapping m, we can construct a formula that is true if it matches the current hyper-assignment:

$$cur_m = \bigwedge_{\pi \in \Pi} \bigwedge_{a \in L(m(\pi))} a_\pi$$

Using this, we can redefine the translation function for HyperTSL. Let Π be the set of trace variables used in φ. Then

$$\begin{aligned}
t_K(\exists \pi.\ \varphi) &:= \exists \pi.\ t_K(\varphi) \\
t_K(\forall \pi.\ \varphi) &:= \forall \pi.\ t_K(\varphi) \\
t_K(\neg \varphi) &:= \neg t_K(\varphi) \\
t_K(\bigcirc \varphi) &:= \bigcirc\ t_K(\varphi) \\
t_K(\varphi \wedge \psi) &:= t_K(\varphi) \wedge t_K(\psi) \\
t_K(\varphi \mathcal{U} \psi) &:= t_K(\varphi)\ \mathcal{U}\ t_K(\psi) \\
t_K(\hat{\tau_P}) &:= \bigvee_{m \in (\Pi \to S) \text{ with } \eta(\hat{\tau_P}, \hat{a}_m) = true} \bigcirc cur_m \\
t_K([\![c_\pi \leftarrowtail \hat{\tau_F}]\!]) &:= \bigwedge_{m \in (\Pi \to S)} (cur_m \Rightarrow \bigcirc\ (cell_{c=v_s})_\pi) \qquad \text{where } v_s := \eta(\hat{\tau_F}, \hat{a}_m)
\end{aligned}$$

Theorem 5.4. *For a HyperTSL formula φ and a TSL Kripke structure K, K satisfies φ if and only if K satisfies $t_K(\varphi)$*

Proof. Let $Z = \mathcal{L}(K)$. Let $Z' = \{\zeta_\sigma \mid \sigma \in Paths(K)\}$. We show that for all

$t \in \mathbb{N}, \sigma_1 \dots \sigma_n \in Paths(K)$

$$\begin{aligned} & Z, t, \varnothing[\pi_1 \to L(\sigma_1)] \dots [\pi_n \to L(\sigma_n)] \vDash_{LTL} t_K(\varphi) \\ \Leftrightarrow\ & Z', t, \varnothing[\pi_1, a_{L(\sigma_1)}] \dots [\pi_n, a_{L(\sigma_n)}] \vDash \varphi \end{aligned}$$

Proof by structural induction.

- Case $\varphi = \forall \pi.\ \psi$

$$\begin{aligned} & Z, t, \varnothing[\pi_1 \to L(\sigma_1)] \dots \varnothing[\pi_n \to L(\sigma_n)] \vDash_{LTL} t_K(\forall \pi.\ \psi) \\ \Leftrightarrow\quad & Z, t, \varnothing[\pi_1 \to L(\sigma_1)] \dots \varnothing[\pi_n \to L(\sigma_n)] \vDash_{LTL} \forall \pi.\ t_K(\psi) \\ \Leftrightarrow\quad & \forall \sigma \in Paths(K). \\ & Z, t, \varnothing[\pi_1 \to L(\sigma_1)] \dots [\pi_n \to L(\sigma_n)][\pi \to L(\sigma)] \vDash_{LTL} t_K(\psi) \\ \Leftrightarrow\quad & \forall \sigma \in Paths(K).\ Z', t, \varnothing[\pi_1, a_{\sigma_1}] \dots [\pi_n, a_{\sigma_n}][\pi, a_\sigma] \vDash \psi \\ \Leftrightarrow\quad & Z', t, \varnothing[\pi_1, a_{\sigma_1}] \dots [\pi_n, a_{\sigma_n}] \vDash \forall \pi. \psi \end{aligned}$$

- The existential case is analogous.
- The remaining cases are analogous to those in the proof of Theorem 5.3

□

As before, the size of the formula can be reduced by using θ instead of cur_m. The size of the translated formula is then at most $|S|^n \cdot |\varphi|$, where n is the number of quantifiers in φ

5.3 Update Term Elimination

The direct translation of TSL to LTL increases the size of the formula a lot. We can improve the performance of the model checking by observing that the TSL predicates can be treated as atomic propositions: for every predicate appearing in the TSL formula, we can create an atomic proposition that is true at the states where the predicate evaluates to true. For the update terms, however, this is not directly possible because whether an update term is satisfied not only depends on the current but also the previous state. Therefore, in this section, we present the function *elimUpd* that modifies the TSL-Kripke structure in such a way that all update terms can then be translated to predicate terms.

The idea is to construct the new TSL Kripke structure $elimUpd(K)$ in such a way that each state of $elimUpd(K)$ corresponds to a pair of consecutive states of K – a current and a previous state. Thereby, when going to a new state, we remember the state we came from. Whether an update term is true in K depends on two states of K, but only on a single state of $elimUpd(K)$. An example is shown in Figure 5.2.

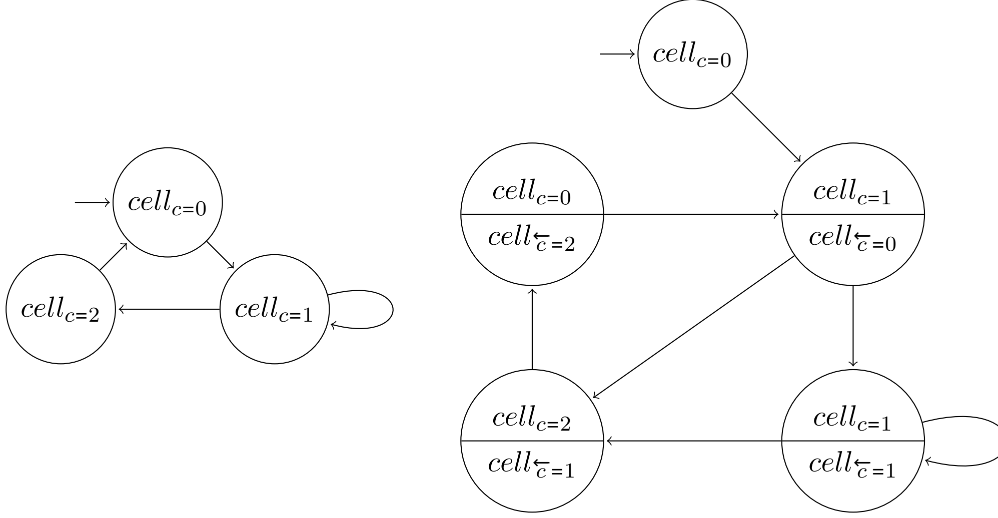

Figure 5.2: Let K be the TSL Kripke structure on the left. Then the TSL-Kripke structure on the right is $elimUpd(K)$: Hereby denotes $cell_{\overleftarrow{c}=v}$ that the previous value of the cell c was v.

Definition 5.5. Let $K = (S, s_0, \delta, AP, L)$ be a TSL-Kripke structure. We define $elimUpd(K) = (S', s_0, \delta', AP', L')$ where

$$\begin{aligned} S' &:= \{s_0\} \cup \{(s_1, s_2) \mid s_1, s_2 \in S \land (s_1, s_2) \in \delta\} \\ \delta' &:= \{((s_1, s_2), (s_2, s_3)) \mid (s_2, s_3) \in \delta\} \cup \{(s_0, (s_0, s_1)) \mid (s_0, s_1) \in \delta\} \\ AP' &:= AP \,\dot\cup\, \{cell_{\overleftarrow{c}=v} \mid c \in \mathbb{C}, v \in \mathbb{V}\} \,\dot\cup\, \{in_{\overleftarrow{i}=v} \mid i \in \mathbb{I}, v \in \mathbb{V}\} \\ L'(s_0) &:= L(s_0) \\ L((s_1, s_2)) &:= L(s_2) \cup \{cell_{\overleftarrow{c}=v} \mid cell_{c=v} \in L(s_1)\} \cup \{in_{\overleftarrow{i}=v} \mid in_{i=v} \in L(s_1)\} \end{aligned}$$

Note that $elimUpd(K)$ is again a TSL-Kripke structure, but over the new set of cells $\mathbb{C} \,\dot\cup\, \{\overleftarrow{c} \mid c \in \mathbb{C}\}$ and the new set of inputs $\mathbb{I} \,\dot\cup\, \{\overleftarrow{i} \mid i \in \mathbb{I}\}$. K and $elimUpd(K)$ are 'equivalent' as the following theorem says.

Theorem 5.6. *Let K be a TSL-Kripke structure over the cells $\mathbb{C}$ and inputs $\mathbb{I}$. A TSL-formula over $\mathbb{C}$ and $\mathbb{I}$ is satisfied by $elimUpd(K)$ if and only if it is satisfied by K.*

Proof. Let AP be the set of atomic propositions of K. We show that for all $\sigma \in AP^\omega, \sigma \in \mathcal{L}(K)$ if and only if $\sigma_{|AP} \in \mathcal{L}(elimUpd(K))$, where $\sigma_{|AP}$ means restricting σ to AP. Note that

$$\sigma_0, \sigma_1, \sigma_2 \cdots \in Paths(K) \Leftrightarrow \sigma_0(\sigma_0, \sigma_1)(\sigma_1, \sigma_2) \cdots \in Paths(elimUpd(K))$$

moreover, $L(\sigma_i, \sigma_{i+1})_{|AP} = L(\sigma_{i+1})$ □

The states of $elimUpd(K)$ however additionally contain information about the previous state - this allows us to eliminate the update terms. To do that, we also define *elimUpd* on TSL formulas.

Definition 5.7.

$$
\begin{aligned}
&elimUpd(\neg\varphi) && := \neg elimUpd(\varphi)\\
&elimUpd(\bigcirc\varphi) && := \bigcirc\, elimUpd(\varphi)\\
&elimUpd(\varphi \wedge \psi) && := elimUpd(\varphi) \wedge elimUpd(\psi)\\
&elimUpd(\varphi\,\mathcal{U}\,\psi) && := elimUpd(\varphi)\,\mathcal{U}\, elimUpd(\psi)\\
&elimUpd(\tau_P) && := \tau_P\\
&elimUpd([\![c \leftarrowtail \tau_F]\!]) && := c = \overleftarrow{\tau_F}
\end{aligned}
$$

where $\overleftarrow{\tau_F}$ is τ_F with each cell $c \in \mathbb{C}$ renamed to $\overleftarrow{c}$ and each input $i \in \mathbb{I}$ renamed to $\overleftarrow{i}$. $\overleftarrow{\tau_F}$ thus evaluates to the value of τ_F in the previous step.

Theorem 5.8. *Let K be a TSL Kripke structure and φ a TSL-formula. Then K satisfies φ if and only if $elimUpd(K)$ satisfies $elimUpd(\varphi)$.*

Proof. For a path $\sigma \in S^\omega$, define $elimUpd(\sigma)$ as $\sigma_0(\sigma_0, \sigma_1)(\sigma_1, \sigma_2)\dots$. Every path σ of the Kripke structure K has a corresponding path $elimUpd(\sigma)$ in $elimUpd(K)$ and vice versa. We show by structural induction over φ that for every path σ and time point t

$$t, \zeta_\sigma \vDash \varphi \Leftrightarrow t, \zeta_{elimUpd(\sigma)} \vDash elimUpd(\varphi)$$

- Case $\varphi = [\![c \leftarrowtail \tau_F]\!]$

$$
\begin{aligned}
& && t, \zeta_{elimUpd(\sigma)} \vDash elimUpd([\![c \leftarrowtail \tau_F]\!])\\
&\Leftrightarrow && t, \zeta_{elimUpd(\sigma)} \vDash c = \overleftarrow{\tau_F}\\
&\Leftrightarrow && \eta(c = \overleftarrow{\tau_F}, (\zeta_{elimUpd(\sigma)})_t) = true\\
&\Leftrightarrow && \eta(c = \overleftarrow{\tau_F}, a_{(\sigma_{t-1}, \sigma_t)}) = true\\
&\Leftrightarrow && \eta(\tau_{F,} a_{\sigma_{t-1}}) = a_{\sigma_t}(c)\\
&\Leftrightarrow && t, \zeta_\sigma \vDash [\![c \leftarrowtail \tau_F]\!]
\end{aligned}
$$

- All other cases are straightforward.

□

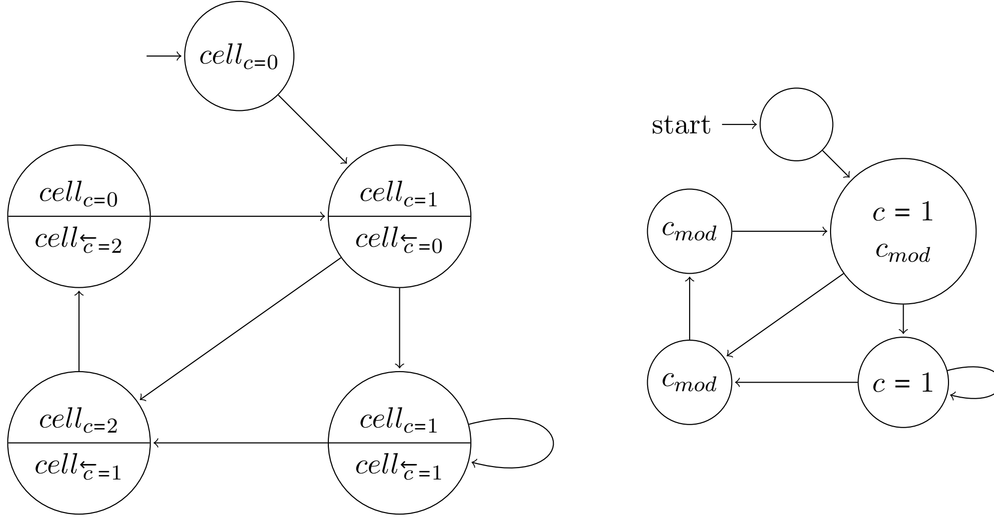

Figure 5.3: Let $elimUpd(K)$ be the TSL-Kripke structure on the left and consider the TSL-Formula $\varphi = \square(\ [\![c \leftarrowtail (c+1)\ mod\ 3]\!] \lor c = 1)$. $elimUpd(\varphi) = \square(\ (c = (\overleftarrow{c}+1)\ mod\ 3) \lor c = 1)$. Relabeling K according to definition 5.9 leads to the Kripke structure on the right. In the drawing, we abbreviate $c_{mod} = (c = (\overleftarrow{c}+1)\ mod\ 3)$.

5.3.1 TSL Model Checking by Update Term Elimination

Theorem 5.8 allows us to simplify the model checking problem for TSL. Now it suffices to give a model checking algorithm for TSL formulas without update terms to get an algorithm for the full logic. This is what we will do in this section.

The idea is to treat all predicates as atomic propositions. We evaluate each predicate at each state of the TSL-Kripke structure and then create a new Kripke structure that is labeled with a predicate if this predicate is true at the corresponding state of the TSL-Kripke structure. This process is formalized using the function *relabel*. An example is shown in Figure 5.3

Definition 5.9. Let $K = (S, s_0, \delta, AP, L)$ be a TSL-Kripke structure. Let $\rho \subseteq \mathcal{T}_P$. We define $relabel(K) = (S, s_0, \delta, \mathcal{T}_P, L')$ where

$$L'(s_0) = \emptyset$$
$$L'(s) = \{\tau_P \in \rho \mid \eta(\tau_P, a_s) = true\}$$

As the example in Figure 5.3 shows, directly using the LTL model checking algorithm after eliminating the update terms and relabeling the TSL-Kripke structure would not lead to correct results. The reason for that is an off-by-one problem: the path of the TSL-Kripke structure starts one step earlier than the corresponding computation. This issue can be solved for example by adding a next-operator in front of the formula.

Theorem 5.10. *Let K be a TSL-Kripke structure over the cells $\mathbb{C}$ and the inputs $\mathbb{I}$. Let φ be a TSL formula over $\mathbb{C}$ and $\mathbb{I}$. Let $\rho \subseteq \mathcal{T}_P$ be the set of predicate terms appearing in $elimUpd(\varphi)$. Then*

$$K \vDash \varphi \Leftrightarrow relabel(elimUpd(K)) \vDash_{LTL} \bigcirc(elimUpd(\varphi))$$

Proof. Let $\sigma' \in Paths(elimUpd(K))$. We define $P(\sigma')_t = \{\tau_P \in \rho \mid \eta(\tau_P, (\zeta_{\sigma'})_{t-1}) = true\}$ and $P(\sigma') = P(\sigma')_0\ P(\sigma')_1 \dots$.

Relabeling $elimUpd(K)$ changes a path σ' to $P(\sigma')$. This means that every path of $relabel(elimUpd(K))$ is of the form $P(\sigma')$ for some $\sigma' \in elimUpd(K)$. Also note that for all t, $P(\sigma')_{t+1} = Seq(\zeta_{\sigma'}, \rho, \varnothing)_t$ (Seq was defined in Definition 4.16). Now, let $\sigma \in Paths(K)$.

$$\begin{aligned} t, \zeta_\sigma \vDash \varphi &\Leftrightarrow t, \zeta_{elimUpd(\sigma)} \vDash elimUpd(\varphi) && \text{(Theorem 5.8)} \\ &\Leftrightarrow t, Seq(\zeta_{elimUpd(\sigma)}, \rho, \varnothing) \vDash_{LTL} elimUpd(\varphi) && \text{(Lemma 4.17)} \\ &\Leftrightarrow t, P(elimUpd(\sigma)) \vDash_{LTL} \bigcirc elimUpd(\varphi) \end{aligned}$$

□

Theorem 5.10 gives a much more efficient reduction to LTL model checking than the direct translation. The size of the formula is not increased, the size of the Kripke structure is increased quadratically. This leads to much better performance, as the complexity of the common LTL model checking algorithms is exponential in the size of the formula, but only linear in the size of the system - therefore, a huge blow-up of the size of the formula is more expensive than a quadratic blow-up of the size of the system. The complexity of both approaches is analyzed formally at the end of section 5.4.1.

5.4 HyperTSL Model Checking

Update term elimination works also for HyperTSL. However, the reduction from TSL to LTL model checking from the previous section can still not be extended to HyperTSL. The reason for that is that in the TSL Kripke structure, predicates relating multiple traces, like $c_\pi = c'_\pi$, can not be treated as atomic propositions anymore, as atomic propositions always belong to one trace only. In this section, we instead modify the HyperLTL model checking algorithm by Finkbeiner et. al. [27] for HyperTSL, supporting at most one quantifier alternation.

Translating the checked formula to HyperLTL and then using the HyperLTL model checking algorithm by Finkbeiner et. al. would roughly mean instantiating all possible values in the formula, translating the quantifier-free part of the formula to an

automaton and then combining this automaton with the system. Our idea for improving this algorithm is to first translate the quantifier-free part of the TSL formula into an automaton and then only instantiate the concrete values when combining this automaton with the system. This avoids an unnecessary blow-up of the formula automaton. However, having predicates instead of concrete values as transition labels makes an important step of the HyperLTL model checking algorithm, the projection, impossible in our setting. This is why our HyperTSL model checking algorithm is limited to the fragment of HyperTSL with at most one quantifier alternation.

As before, we first eliminate update terms using the function *elimUpd* as defined in Definitions 5.5, 5.7. Moreover, when given a HyperTSL formula $\varphi = Q_1\pi_1. \dots Q_n\pi_n. \psi$, $Q_i \in \{\forall, \exists\}$, we translate the underlying TSL formula ψ to an automaton A_ψ, treating all predicates as atomic propositions. The transitions of this automaton are first labeled with subsets of predicates, but we can reduce them to one predicate by constructing the conjunction.

Definition 5.11. Let ψ be a HyperTSL formula without quantifiers or update terms. Let $A = (2^{\hat{\mathcal{T}_P}}, Q, \delta, q_0, Q_{acc})$ be the automaton obtained by treating all predicates as atomic propositions and then using the LTL to Büchi automaton translation algorithm on ψ. We define **the automaton for** ψ as $A_\psi = (\hat{\mathcal{T}_P}, Q, \delta', q_0, Q_{acc})$ where

$$\delta' = \{(q, \hat{\tau_P}, q') \mid (q, S, q') \in \delta \land \hat{\tau_P} = \bigwedge_{\hat{\tau_P}' \in S} \hat{\tau_P}' \land \bigwedge_{\hat{\tau_P}' \notin S} \neg\hat{\tau_P}'\}$$

Observe that when choosing concrete paths $\sigma_1, \dots \sigma_n$ for the quantified trace variables, we can determine using A_ψ whether ψ is true for them. That is because $\zeta_{\sigma_1} \dots \zeta_{\sigma_n}$ define a sequence of predicate sets, containing all predicates that are true per time point (see Definition 4.16). If this sequence is accepted by A, the formula is true for this concrete choice of paths. Equivalently, we can ask whether there is a predicate sequence accepted by A_ψ, such that when plugging in the values according to $\zeta_{\sigma_1} \dots \zeta_{\sigma_n}$ into each predicate at each time point, all predicates are true. We name a predicate sequence where all predicates evaluate to true a *true predicate sequence*. The idea is now to gradually plug in the cell and input values for every quantified trace variable into the labels of the formula automaton.

The operation of plugging in cell and input values for one trace variable is formalized in the following: roughly, $\hat{\tau_F}[a/\pi]$ means replacing the cells labeled with π in τ_F with their values according to a. We can lift this to traces by applying the definition pointwise.

Definition 5.12. Let $\hat{\tau_F} \in \hat{\mathcal{T}_F}$, $a \in \mathcal{A}$, $\pi \in \Pi$. We define 'a **plugged in for** π **into**

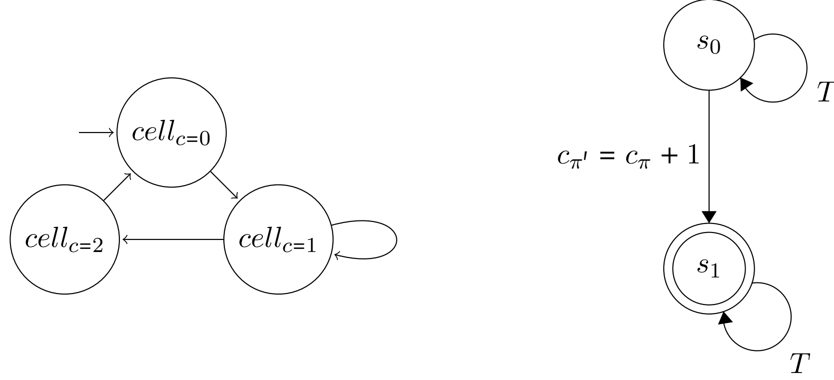

Figure 5.4: Left: a TSL Kripke structure, right: the automaton for the TSL formula $\diamondsuit(c_{\pi'} = c_\pi + 1)$

$\hat{\tau_F}$', written $\hat{\tau_F}[a/\pi]$ as

$$
\begin{aligned}
c_\pi[a/\pi] &:= a(c) && \text{if } \hat{\tau_F} = c_\pi \\
c_{\pi'}[a/\pi] &:= c_{\pi'} && \text{if } \hat{\tau_F} = c_{\pi'} \wedge \pi \neq \pi' \\
i_\pi[a/\pi] &:= a(i) && \text{if } \hat{\tau_F} = i_\pi \\
i_{\pi'}[a/\pi] &:= i_{\pi'} && \text{if } \hat{\tau_F} = i_{\pi'} \wedge \pi \neq \pi' \\
f(\hat{\tau_{F1}}, \dots, \hat{\tau_{Fk}})[a/\pi] &:= f(\hat{\tau_{F1}}[a/\pi], \dots, \hat{\tau_{Fk}}[a/\pi]) && \text{if } \hat{\tau_F} = f(\hat{\tau_{F1}}, \dots, \hat{\tau_{Fk}})
\end{aligned}
$$

Let $\zeta \in \mathcal{A}^\omega, P \in \hat{\mathcal{T}_F}^\omega$. We define '$\zeta$ plugged in for π into P' as

$$P[\zeta/\pi] := P_1[\zeta_1/\pi] \;\; P_2[\zeta_2/\pi] \dots$$

5.4.1 The Alternation-Free Fragment

Consider first the case of an existential formula $\exists\pi.\ \exists\pi'.\ \psi$. For HyperLTL, ψ is first translated to an automaton and then intersected with the Kripke structure in such a way that the resulting automaton now accepts all traces over the atomic propositions corresponding to π, such that $\exists\pi'.\ \psi$ is true for this choice of π. We do something similar for HyperTSL: we first translate ψ to an automaton and then construct an automaton that when *plugging in* a concrete trace for π into its traces, the automaton accepts a true predicate sequence if and only if $\exists\pi'.\ \psi$ is true for this choice of π. For achieving this, the idea is to combine the Kripke structure and the formula automaton in such a way that an accepted predicate sequence of the resulting automaton corresponds to a concrete choice of a trace for π', and still captures the conditions for the choice of π that are necessary to make ψ true.

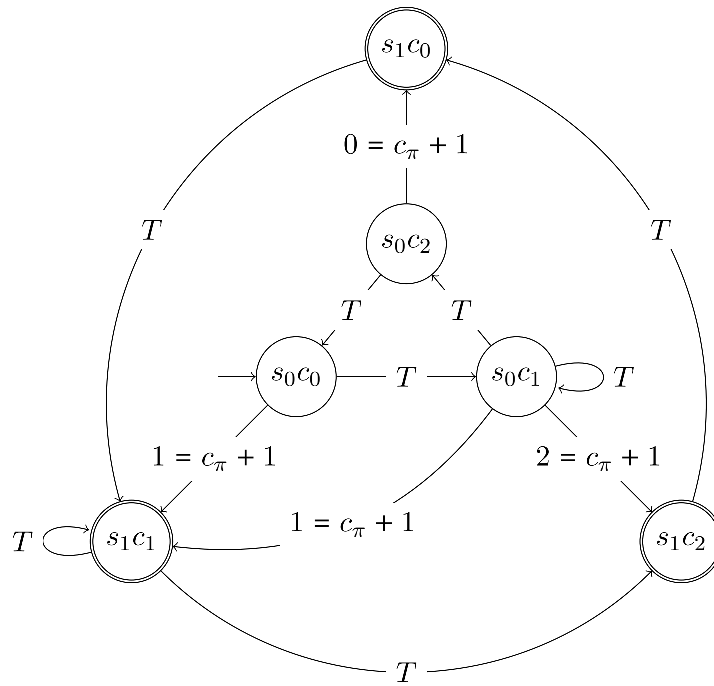

Figure 5.5: The automaton for the TSL formula $\exists\pi'.\ \Diamond(c_{\pi'} = c_\pi + 1)$

Example 5.13. Consider the Kripke structure shown in Figure 5.4 on the left and the HyperTSL-formula $\exists\pi.\ \exists\pi'.\ \diamondsuit(c_{\pi'} = c_\pi + 1)$. The automaton for $\diamondsuit(c_{\pi'} = c_\pi + 1)$ is shown in Figure 5.4 on the right.

The automaton for $\exists\pi'.\ \diamondsuit(c_{\pi'} = c_\pi + 1)$ is shown in Figure 5.5. This automaton accepts for example the trace $P = (1 = c_\pi + 1)\ T^\omega$. This corresponds to choosing for π' the computation $(a(c) = 1)^\omega$ which itself corresponds to the trace $cell_{c=0}\ {cell_{c=1}}^\omega$ of the Kripke structure. One choice for π that makes $P[\zeta/\pi]$ a true predicate sequence would be $\zeta = (a(c) = 0)^\omega$

Definition 5.14. Let $(\hat{\mathcal{T}_P}, Q, \delta_\psi, q_0, Q_{acc})$ be a Büchi automaton (for a formula ψ). Let $(S, s_0, \delta_K, AP, L)$ be a TSL-Kripke structure. We define **the automaton for** $\exists\pi.\ \psi$ as the Büchi automaton $(\hat{\mathcal{T}_P}, S \times Q, \delta, (s_0, q_0), Q_{acc} \times S)$ where

$$\delta = \{((s,q), \hat{\tau_P}[a_{s'}/\pi], (s', q')) \mid (q, \hat{\tau_P}, q') \in \delta_\psi \land (s, s') \in \delta_K\}$$

We get the automaton for a formula with multiple existential quantifiers by repeated application of Definition 5.14. In the end, the automaton for the complete HyperTSL-formula accepts any true predicate sequence if and only if the formula is satisfied.

Correctness Proof

To be able to prove the correctness of our algorithm inductively, we need the following definition that is fulfilled by the constructed automaton A_φ for any existential formula φ. Intuitively, the definition states that when choosing and plugging in computations for the remaining free variables, there should be an accepted true predicate sequence if and only if the formula is true for this choice of the remaining computations.

Definition 5.15. Let K be a TSL-Kripke structure. Let $Z = \{\zeta_\sigma \mid \sigma \in Paths(K)\}$. We say that an automaton **correctly models** a HyperTSL formula φ with free trace variables $\pi_1 \dots \pi_n$ if for all $\zeta_1, \dots \zeta_n \in \mathcal{A}^\omega$ there is an accepted predicate sequence $P \in \hat{\mathcal{T}_P}^\omega$ with $P[\zeta_1/\pi_1]\dots[\zeta_n/\pi_n]$ a true predicate sequence if and only if

$$Z, \varnothing[\pi_1, \zeta_1]\dots[\pi_n, \zeta_n], 0 \vDash \varphi$$

Lemma 5.16. *Let ψ be a HyperTSL-formula without quantifiers or update terms. Let A_ψ be the automaton for ψ. A_ψ correctly models ψ.*

Proof. Let $\zeta_1 \dots \zeta_n \in \mathcal{A}^\omega$. We define $\hat{\zeta} = \varnothing[\pi_1, \zeta_1]\dots[\pi_n, \zeta_n]$ and

$$P_t = \bigwedge_{\hat{\tau_P} \in Seq(\hat{\zeta})_t} \hat{\tau_P} \land \bigwedge_{\hat{\tau_P} \notin Seq(\hat{\zeta})_t} \neg\hat{\tau_P}$$

$$P = P_0\ P_1 \dots$$

We require that this induces for the same predicate sets the same ordering the as Definition 5.11. It is clear that $P[\zeta_1/\pi_1]\dots[\zeta_n/\pi_n]$ is always a true predicate sequence as $\hat{\tau_P}[\zeta_1/\pi_1]\dots[\zeta_n/\pi_n]$ true is equivalent to $\hat{\zeta} \vDash \hat{\tau_P}$. Moreover, this holds for no other predicate sequence accepted by A_ψ. By the correctness of the LTL to Büchi automaton algorithm, we also know that P is accepted by A_ψ if and only if $Seq(\hat{\zeta}) \vDash_{LTL} \psi$ Thus it suffices to show that,

$$Seq(\hat{\zeta}) \vDash_{LTL} \psi \Leftrightarrow 0, Z, \hat{\zeta} \vDash \psi$$

This is true by Lemma 4.17. □

Lemma 5.17. *Let K be a TSL-Kripke structure. Let ψ be a HyperTSL-formula and $\pi_1, \dots \pi_n$ be the free trace variables appearing in ψ. Let A_ψ be an automaton that correctly models ψ. Then, the automaton A_φ for $\varphi = \exists \pi_1.\ \psi$ correctly models $\exists \pi_1.\ \psi$*

Proof. Let $\zeta_2, \dots \zeta_n \in \mathcal{A}^\omega$.
$\Rightarrow$ Assume that A_φ accepts the predicate sequence $P_\varphi \in \hat{\mathcal{T}_P}^\omega$, and assume that $P_\varphi[\zeta_2/\pi_2]\dots[\zeta_n/\pi_n]$ is a true predicate sequence. Then there is an accepting run $(s_0, q_0)(s_1, q_1)(s_2, q_2)\dots$. By the construction of A_φ, $p = s_0 s_1 \dots$ is a path of the Kripke structure and $q_0 q_1 \dots$ is an accepting run of the automaton for ψ. Let P_ψ be the predicate sequence induced by the accepting run of the automaton for ψ. We show that $P_\psi[\zeta_p/\pi_1]\dots[\zeta_n/\pi_n]$ is a true predicate sequence: for all i, $P_{\varphi,i} = P_{\psi,i}[\zeta_i/\pi_1]$ by Definition 5.14. Thus, $P_\psi[\zeta_p/\pi_1]\dots[\zeta_n/\pi_n] = P_\varphi[\zeta_2/\pi_2]\dots[\zeta_n/\pi_n]$ which is a true predicate sequence by assumption. By the correctness of A_ψ, this means that $Z, \varnothing[\pi_1, \zeta_p]\dots[\pi_n, \zeta_n], 0 \vDash \varphi$. Moreover, $\zeta_p \in Z$, so $Z, \varnothing[\pi_2, \zeta_2]\dots[\pi_n, \zeta_n], 0 \vDash \exists \pi_1.\ \psi$

$\Leftarrow$ Let $Z, \varnothing[\pi_2, \zeta_2]\dots[\pi_n, \zeta_n], 0 \vDash \exists \pi_1.\ \psi$. Then, there exists a ζ_p for a path $p = p_0 p_1 \dots$ of the Kripke structure such that $Z, \varnothing[\pi_1, \zeta_p]\dots[\pi_n, \zeta_n], 0 \vDash \psi$. By the correctness of A_ψ, this means that there is an accepted predicate sequence $P_\psi \in \hat{\mathcal{T}_P}^\omega$ with $P[\pi_1/\zeta_p]\dots[\pi_n, \zeta_n]$ true. Let $s_0 s_1 \dots$ be the accepting run. We show that $(s_0, p_0)(s_1, p_1)\dots$ is an accepting run of A_φ. For all i, note that

$$((s_i, p_i), (P_\psi)_i[(\zeta_p)_i/\pi_1], (s_{i+1}, p_{i+1})) \in \delta$$

because $(p_i, (P_\psi)_i, p_{i+1}) \in \delta_\psi$ and $(s_i, s_{i+1}) \in \delta_K$. Let P_φ be the induced predicate sequence. As before, $P_\varphi[\zeta_2/\pi_2]\dots[\zeta_n/\pi_n] = P_\psi[\zeta_p/\pi_1]\dots[\zeta_n/\pi_n]$ which is true by assumption. □

Theorem 5.18. *Let φ be a HyperTSL formula with only existential quantifiers and without update terms. Let K be a TSL-Kripke structure. The automaton for φ accepts any true predicate sequence if and only if $K \vDash \varphi$*

Proof. By induction, Lemma 5.16 and Lemma 5.17 we know that the automaton for φ correctly models φ. As there are no free trace variables, this means that there is an accepted true predicate sequence if and only if $Z, \emptyset, 0 \models \varphi$ which is the definition of $K \models \varphi$. □

Theorem 5.18 gives an algorithm for model checking the existential fragment of HyperTSL: we repeatedly apply Definition 5.14 to construct an automaton for the formula, then delete all transitions where the predicate label evaluates to false, and then test whether the automaton is nonempty. For the universal fragment, test the negated formula instead, which is existential.

Complexity Comparision

Let $\varphi = \exists\pi_1. \dots \exists\pi_n. \psi$ be a HyperTSL formula and $K = (S, s_0, \delta, AP, L)$ be a TSL Kripke structure.

Translation. As analyzed before, the size of the HyperLTL formula for a HyperTSL formula is at most $|S|^n \cdot |\psi|$. The HyperLTL model checking algorithm presented in [27] first translates the quantifier-free part of the formula to an alternating automaton. In this process, a new state is added for every operator and atomic proposition, so the size of the alternating automaton is then in $\mathcal{O}(|S|^n \cdot |\psi|)$. Next, the alternating automaton is translated into an equivalent Büchi automaton, raising the automaton size to $2^{\mathcal{O}(|S|^n \cdot |\psi|)}$. Then, this automaton is intersected with the system n times and checked for emptiness, leading to a total complexity of $\mathcal{O}(|S|^n \cdot 2^{\mathcal{O}(|S|^n \cdot |\psi|)})$

Direct TSL Model Checking. Here, first, the update terms are eliminated. This does not increase the asymptotic size of φ, but introduces a new system state for every transition of K, thus there $|\delta|$ states in *elimUpd*(K). The number of transitions is at most $|\delta| \cdot |S|$. Then, *elimUpd*(ψ) is also translated into an automaton of size $\mathcal{O}(2^{\mathcal{O}(|\psi|)})$. Next, Definition 5.14 is applied. This leads to an automaton with $|\delta|^n \cdot \mathcal{O}(2^{\mathcal{O}(|\psi|)})$ states and $|\delta|^n \cdot |S|^n \cdot \mathcal{O}(2^{\mathcal{O}(|\psi|)})$ transitions as the number of states is for every quantifier multiplied by the number of states of *elimUpd*(K), and the number of transitions is for every quantifier multiplied by the number of transitions of *elimUpd*(K). This leads to a total running time of

$$\mathcal{O}(|\delta|^n \cdot |S|^n \cdot 2^{\mathcal{O}(|\psi|)})$$

this is better than the translation if the system is significantly larger than the formula (as $|\delta| \in \mathcal{O}(|S|^2)$, which is usually the case in model checking.

5.4.2 The $\forall^*\exists^*$ and $\exists^*\forall^*$ Fragments

Let us now consider the case of a HyperTSL-formula $\forall\pi. \exists\pi_1. \dots \exists\pi_n.\psi$. We extend the previously presented algorithm to this fragment. Recall that when plugging in

a concrete trace for π into the traces of the automaton for the existential part, it accepts a true predicate sequence if and only if the existential part of the formula is fulfilled for this choice of π. Thus, we need to check whether this is the case for all the traces of the Kripke structure. We first intuitively describe the algorithm for checking this: First, we again construct the product automaton similar to Definition 5.14. As there are now no free trace variables anymore, we know whether a transition label evaluates to true. If this is the case, we relabel this transition with the labels of the trace that was plugged in for π. If this is not the case, the transition is deleted. The resulting automaton now accepts all traces of the Kripke structure (up to an off-by-one problem) that fulfill the existential part of the formula - it remains to test whether these are all the traces of the Kripke structure.

The algorithm can be extended to multiple universal quantifiers by applying the technique of *self-composition*. Self-Composition is a technique commonly used to reduce the model checking problem for alternation-free hyperproperties to a simpler model checking problem for another system [3, 4, 26]. The idea is to compose multiple instances of the system such that each trace of the composed system corresponds to an interleaving of multiple traces of the original system. Self-composition for TSL-Kripke structures is formally defined as follows:

Definition 5.19. Let $\Pi = \{\pi_1, \dots \pi_n\}$ be a set of trace variables and let $|\Pi| = n$ Let $K = (S, s_0, \delta, AP, L)$ be a TSL Kripke structure. We define the $n-$fold **self-composition** of K as $K^n = (S^n, {s_0}^n, \delta^n, AP \times \Pi, L^n)$ where

$$\begin{aligned}
S^n =& S \times \dots \times S \\
{s_0}^n =& (s_0, \dots, s_0) \\
\delta^n =& \{((s_1 \dots s_n), ({s_1}' \dots {s_n}')) \mid (s_1, {s_1}') \in \delta \wedge \dots \wedge (s_n, {s_n}') \in \delta\} \\
L^n((s_1, \dots, s_n)) =& \{cell_{c_{\pi_j}=v} \mid 1 \leq j \leq n \wedge cell_{c=v} \in L(s_j)\} \cup \\
& \{in_{i_{\pi_j}=v} \mid 1 \leq j \leq n \wedge in_{i=v} \in L(s_j)\}
\end{aligned}$$

Note that the self-composition is again a TSL-Kripke structure over the new set of inputs $\mathbb{I} \times \Pi$ and the new set of cells $\mathbb{C} \times \Pi$.

The next definition formalizes the construction of the automaton containing all the traces satisfying the existential part of the formula (up to an off-by-one problem)

Definition 5.20. Let $n \in \mathbb{N}$. Let $(\hat{\mathcal{T}_P}, Q, \delta_\psi, q_0, Q_{acc})$ be a Büchi automaton (for a formula ψ). Let K be a TSL-Kripke structure and let $K^n = (S, s_0, \delta_K, AP, L)$ be its n-fold self-composition. We define the automaton for $\forall \pi_1 \dots \forall \pi_n.\ \psi$ as the Büchi automaton $(AP, S \times Q, (s_0, q_0), \delta, Q_{acc} \times S)$ where

$$\delta = \{((s, q), L(s'), (s', q')) \mid \exists \hat{\tau_P}.\ (q, \hat{\tau_P}, q') \in \delta_\psi \wedge (s, s') \in \delta_K \wedge \eta(\hat{\tau_P}, a_{s'}) = true\}$$

Theorem 5.21. *Let K be a TSL-Kripke structure. Let A_ψ be a Büchi automaton that correctly models ψ. Let A_φ be the Büchi automaton for $\varphi = \forall\pi_1 \dots \forall\pi_n.\ \psi$. Then $K \vDash \varphi$ if and only if*

$$\{\sigma_1\sigma_2\cdots \mid \sigma_0\sigma_1\sigma_2\cdots \in Paths(K^n)\} \subseteq Paths(A_\varphi)$$

Proof. $\Rightarrow$ Let $K \vDash \varphi$. Then $\{\zeta_\sigma \mid \sigma \in Paths(K)\} \vDash \forall\pi_1 \dots \forall\pi_n.\ \psi$. Let $\sigma = \sigma_0\sigma_1\cdots \in Paths(K^n)$. σ corresponds to n paths of K: for all i, $\sigma_i = (\sigma_i^1, \dots, \sigma_i^n)$ and for $1 \le j \le n, \sigma^j = \sigma_0^j, \sigma_1^j \cdots \in Paths(K)$. By assumption, $\varnothing[\pi_1, \zeta_{\sigma^1}] \dots [\pi_n, \zeta_{\sigma^n}] \vDash \psi$. By the correctness of A_ψ, this means that there is an accepted predicate sequence P with $P[\zeta_{\sigma^1}/\pi_1] \dots [\zeta_{\sigma^n}/\pi_n]$ true in A_ψ. Let p be the corresponding accepting run. We show that $(\sigma_0, p_0)\ (\sigma_1, p_1) \dots$ is an accepting run of A_φ. Indeed, for all i, $(p_i, P_i, p_{i+1}) \in \delta_\psi \wedge (\sigma_i, \sigma_{i+1}) \in \delta_{K^n} \ \wedge\ \eta(P_i, a_{\sigma_{i+1}}) = true$ - the latter one is true because $P[\zeta_{\sigma^1}/\pi_1] \dots [\zeta_{\sigma^n}/\pi_n]$ is a true predicate sequence. There are infinitely many accepting states visited because p is an accepting run, thus $(\sigma_0, p_0)(\sigma_1, p_1)$ is an accepting run - the corresponding accepted word is $\sigma_1, \sigma_2 \dots$, so $\sigma_1\sigma_2\cdots \in Paths(A_\varphi)$.

$\Leftarrow$ Let $Paths(A_\varphi) \supseteq \{\sigma_1\sigma_2\cdots \mid \sigma_0\sigma_1\sigma_2\cdots \in Paths(K^n))\}$. We have to show that $\{\zeta_\sigma \mid \sigma \in Paths(K)\} \vDash \forall\pi_1 \dots \forall\pi_n.\ \psi$. Let $\sigma^1 \dots \sigma^n$ be paths of K. Then, $\sigma = (\sigma_0^1, \dots, \sigma_0^n)\ (\sigma_1^1, \dots \sigma_1^n) \dots$ is a path of K^n. By assumption, $\sigma_1\sigma_2\cdots \in Paths(A_\varphi)$. Let $(\sigma_0, p_0)\ (\sigma_1, p_1)$ be the accepting run. We know that for all i, there exists a P_i such that $(p_i, P_i, p_{i+1}) \in \delta_\psi$ and $\eta(P_i, a_{\sigma_{i+1}}) = true$. Then, $P[\zeta_{\sigma^1}/\pi_1] \dots [\zeta_{\sigma^n}/\pi_n]$ is a true predicate sequence. By the correctness of A_ψ, this means that $\varnothing[\pi_1, \zeta_{\sigma^1}] \dots [\pi_n, \zeta_{\sigma^n}] \vDash \psi$. As $\sigma^1, \dots, \sigma^n$ were chosen arbitrarily, we are have shown that $K \vDash \forall\pi_1 \dots \forall\pi_n.\ \psi$. □

Theorem 5.21 gives an algorithm for model checking the $\forall\exists$ fragment. Let φ be a $\forall\exists$-HyperTSL formula with n universal quantifiers. As before, we first translate the existential part ψ of the formula into an automaton A_ψ by applying Definition 5.14. Next, we compute the n-fold self-composition of the Kripke structure and apply Definition 5.20 on the self-composition and A_ψ to get an automaton A_φ. Then, to fix the off-by-one-problem, we add an initial state to A_φ, and a transition labeled with

$$\{cell_{c_{\pi_j}=v} \mid 1 \le j \le n \wedge \zeta_{-1}(c) = v\} \cup \{in_{i_{\pi_j}=v} \mid 1 \le j \le n \wedge \zeta_{-1}(i) = v\}$$

from the new initial to every other state. Name this automaton $A_\varphi{}'$. Next, we check whether $A_\varphi{}' \setminus K^n$ is empty. This involves one complementation (just as the model checking algorithm for $\forall\exists$-HyperLTL [27]) and is thus computationally expensive.

This also gives an algorithm for the $\exists\forall$ fragment - we can test the negated formula and apply the algorithm for the $\forall\exists$ fragment.

Complexity. Following the same reasoning as in Section 5.4.1 leads to a complexity of $\mathcal{O}(|\delta|^n \cdot |S|^n \cdot 2^{\mathcal{O}(|\psi|)})$ for constructing the automaton for a HyperTSL formula φ with n quantifiers and one quantifier alternation. Testing whether $\{\sigma_1\sigma_2\cdots \mid \sigma_0\sigma_1\sigma_2\cdots \in Paths(K^n)\} \subseteq Paths(A_\varphi)$ involves one complementation, thus leading to a total running time of

$$2^{\mathcal{O}(|\delta|^n \cdot |S|^n \cdot 2^{\mathcal{O}(|\psi|)})}$$

model checking by translation instead leads to a running time of

$$2^{\mathcal{O}(|S|^n \cdot 2^{\mathcal{O}(|S|^n \cdot |\psi|)})}$$

Thus, the direct model checking algorithm is again better than the translation if the system is much larger than the formula.

Chapter 6

Software Model Checking

In this chapter, we develop software model checking algorithms for TSL and HyperTSL. As the system, we use a Büchi automaton labeled with program statements [33]. Our TSL software model checking algorithm is an adaption of the automata-based LTL software model checking algorithm by Dietsch et al. [25]. They already express atomic propositions as predicates over memory cells, but we need to extend the algorithm for update terms and predicates over inputs. Then, We further extend this algorithm to alternation-free HyperTSL by applying the technique of *self-composition*. The n-fold self-composition of a program automaton is again a program automaton where each execution is an interleaving of n executions of the original program.

Next, we propose an algorithm for finding counterexamples for $\forall^*\exists^*$ HyperTSL formulas or, dually, witnesses for $\exists^*\forall^*$ HyperTSL formulas.

6.1 TSL Software Model Checking

6.1.1 The Algorithm

We present the LTL software model checking algorithm by Dietsch et al. [25], adapted for Temporal Stream Logic.

We model the program as a Büchi automaton labeled with program statements [33] with all states accepting. As basic program statements, we allow assertions and memory cell assignments. Later, we will also label transitions with multiple statements:

Definition 6.1. We define the set of **basic program statements** $Stmt_0$ by

$$s_0 ::= assert(\tau_P) \mid c := \tau_F \mid c := * \qquad \text{where } c \in \mathbb{C}, \tau_P \in \mathcal{T}_P, \tau_F \in \mathcal{T}_{\mathcal{F}}$$

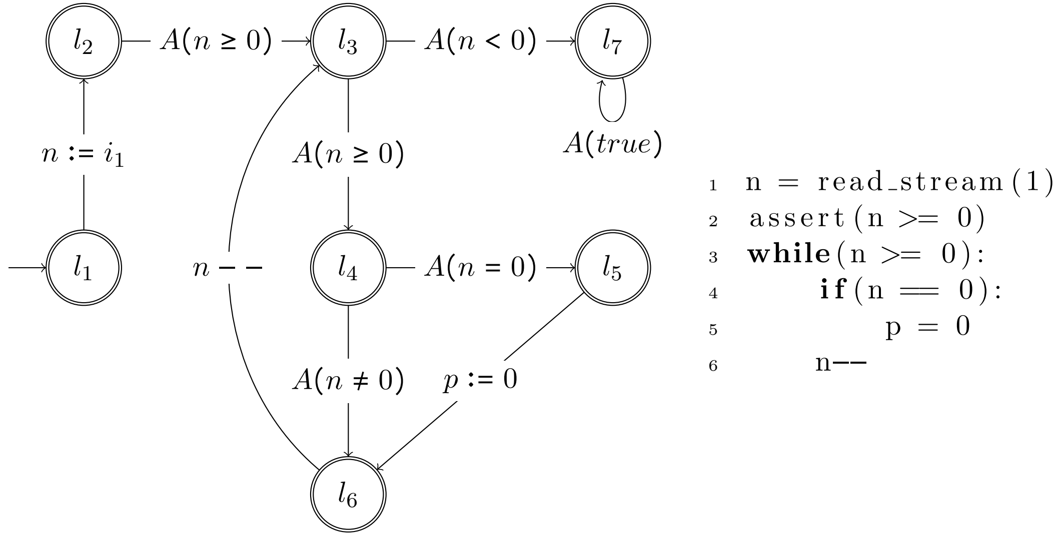

Figure 6.1: The program shown on the right can be modeled as the automaton on the left. Thereby, we abbreviate *assert* as A. For example, the condition of the while-loop is modeled using the two outgoing transitions from state l_3 – one with the assertion that the condition is true, the other with the assertion that the condition is false. A self-loop labeled with $A(true)$ can be added to the state l_7 to extend each trace to infinity.

the set of **program statements** *Stmt* is defined by

$$s ::= s_0 \mid s; s \qquad\qquad \text{where } s_0 \in Stmt_0$$

The assignment $c := *$ means that any value could be assigned to c - for example, if c is chosen as a random number.

A *program automaton* is an automaton $P = (Stmt, Q, q_0, \delta, Q_{acc})$ for some set of program locations Q, an initial location $q_0 \in Q$, a transition relation $\delta \subseteq Q \times Stmt \times Q$ and a set of accepting states $Q_{acc} \subseteq Q$. For now, $Stmt = Stmt_0$ and $Q_{acc} = Q$, but we will later also construct other program automata. An example of a program automaton is shown in Figure 6.1

Note that using such a model with the basic program statements only, we can model `if` statements, `while` loops, and also non-deterministic choices. However, not every trace of the program automaton corresponds to a program execution. For example, the trace $n := input_1$; $assert(n > 0)$; $assert(n < 0)$; $assert(true)^\omega$ does not – the second assertion will always fail. We call such a trace *infeasible*. In contrast, a *feasible* trace corresponds to a program execution where all the assertions may succeed. We now define the feasible traces over the basic statements formally:

Definition 6.2. A computation ζ **matches** a trace $\sigma \in Stmt_0^\omega$ at time point t, written $\zeta \triangleleft_t \sigma$ if

$$\begin{cases} \text{if } \sigma_t = assert(\tau_P): & \eta(\tau_P, \zeta_{t-1}) = true \quad \wedge \forall c \in \mathbb{C}.\ \zeta_t(c) = \zeta_{t-1}(c) \\ \text{if } \sigma_t = c := \tau_F: & \eta(\tau_F, \zeta_{t-1}) = \zeta_t(c) \wedge \forall c' \in \mathbb{C}\backslash\{c\}.\ \zeta_t(c') = \zeta_{t-1}(c') \\ \text{if } \sigma_t = c := *: & \forall c \in \mathbb{C}\backslash\{c\}.\ \zeta_t(c) = \zeta_{t-1}(c) \end{cases}$$

where ζ_{-1} is the initial assignment.

A computation ζ matches a trace $\sigma \in Stmt_0^\omega$, written $\zeta \triangleleft \sigma$ if

$$\forall t \in \mathbb{N}.\ \zeta \triangleleft_t \sigma$$

We can extend the definition to traces over all program statements by 'flattening' the trace, eliminating sequential composition. The following function $flatten : Stmt^\omega \to {Stmt_0}^\omega$ takes a sequence of program statements and transforms it into a sequence of basic program statements by converting a composed program statement into multiple basic program statements.

Example 6.3.

$$\begin{aligned} flatten(\ (i := *;\ n := 8)\ (i := *;\ n := 17;\ assert(true))\ ) = \\ (i := *)\ (n := 8)\ (i := *)\ (n = 17)\ (assert(true)) \end{aligned}$$

Definition 6.4.

$$flatten(s) = \begin{cases} \tau_P & \text{if } s = \tau_P \\ c := \tau_F & \text{if } s = c := \tau_F \\ c := * & \text{if } s = c := * \\ flatten(s)\ flatten(s') & \text{if } s = s; s' \end{cases}$$

$$flatten(s_1\ s_2\ s_3 \ldots) = flatten(s_1)\ flatten(s_2)\ flatten(s_3)\ \ldots$$

Note that flattening changes the notion of time steps: a single time step in the original trace can correspond to multiple time steps in the flattened trace.

Definition 6.5. A trace $\sigma \in Stmt^\omega$ **matches** a computation ζ, written $\zeta \triangleleft \sigma$ if $\zeta \triangleleft flatten(\sigma)$.

A trace $\sigma \in Stmt^\omega$ is **feasible** if there is a computation ζ such that $\zeta \triangleleft \sigma$.

Definition 6.6. A program automaton P over the basic statements $Stmt_0$ satisfies a TSL-formula φ, if for all traces σ of P

$$\forall \zeta \in \mathcal{A}^\omega.\ \zeta \triangleleft \sigma \Rightarrow \zeta \vDash \varphi$$

We now present an algorithm to check whether a program automaton satisfies a TSL formula. It is an adaption of the automaton-based LTL software model checking approach by Podelski et. al [25], where the basic idea is to first translate the negated formula into an automaton $A_{\neg\varphi}$, then combine the formula automaton and the program automaton to a new one that is called the *Büchi program product*. The program satisfies the formula if and only if the Büchi program product accepts no feasible trace.

In [25], the Büchi program product is constructed similarly to the standard product automaton construction. However, to make the result again a program automaton, the transitions are not labeled with pairs $(s,l) \in Stmt_0 \times 2^{AP}$, but instead with the program statement $(s;\ assert(l))$. A feasible accepted trace of the Büchi program product then corresponds to a counterexample proving that the program does not satisfy the formula. In the following, we discuss how we can adapt the Büchi program product construction for TSL such that this property - a feasible trace corresponds to a counterexample - is also true for TSL. Moreover, we want to construct the TSL Büchi program product in such a way that we can use the same algorithm as in [25] for testing if there is a feasible accepted trace.

For the construction of $A_{\neg\varphi}$, we treat all update- and predicate terms as atomic propositions and use any algorithm for the translation of an LTL formula to an automaton. For the construction of the Büchi program product, we have to define how to combine a transition label s of the original program automaton with a transition label l of the formula automaton into a single program statement. We want the combined statement to succeed if and only if l holds for the statement s. l is a set of update and predicate terms and for the update terms $[\![c \leftarrowtail \tau_F]\!]$ we can not just use an assertion to check if they are true. We instead need to 'save' the value of τ_F before the statement s is executed.

There is still a second adaption needed: the feasibility definition from [25] differs slightly from ours, as their program statements do not explicitly involve input streams. To be able to use their algorithm for testing if there is a feasible accepted trace, we model the behavior of the input streams by using fresh memory cells that receive a new value at every time step.

In the following, we formally define the function $combine : Stmt \times \mathcal{P}(\mathcal{T}_P \cup \mathcal{T}_U)$ that combines a program statement s and a transition label l to a new program statement that succeeds if and only if l is true for s.

Definition 6.7. Let $\upsilon = \{[\![c_1 \leftarrowtail \tau_{F1}]\!], \dots, [\![c_n \leftarrowtail \tau_{Fn}]\!]\}$ be the set of update terms appearing in φ, let ρ be the set of predicate terms appearing in φ. Let $l \subseteq (\upsilon \cup \rho)$ be a transition label of $A_{\neg\varphi}$. Let $(tmp_j)_{j\in\mathbb{N}}$ be a family of fresh cells. Let $\mathbb{I} = \{i_1, \dots i_m\}$. In the following, we define the function $combine : Stmt \times \mathcal{P}(\mathcal{T}_P \cup \mathcal{T}_U) \to Stmt$. The result of $combine(s,l)$ is composed of the program statements

in $save_values_l, s, new_inputs, check_preds_l$ and $check_updates_l$

$$
\begin{aligned}
save_values &:= tmp_1 := \tau_{F1};\ \ldots; tmp_n := \tau_{Fn} \\
new_inputs &:= i_1 := *;\ \ldots\ ; i_m := * \\
check_preds_l &:= assert\left(\bigwedge_{\tau_P \in l} \tau_P \wedge \bigwedge_{\tau_P \in \rho \setminus l} \neg\tau_P\right) \\
check_updates_l &:= assert\left(\bigwedge_{[\![c_j \leftarrowtail \tau_{Fj}]\!] \in v} \begin{cases} c_j = tmp_j & \text{if } [\![c_j \leftarrowtail \tau_{Fj}]\!] \in l \\ c_j \neq tmp_j & \text{else} \end{cases}\right) \\
combine(s, l) &:= save_values;\ s;\ new_inputs;\ check_preds_l;\ check_updates_l
\end{aligned}
$$

We can extend this definition for combining a program trace and a predicate trace by applying it per timepoint. This leads to a function $combine : Stmt^{\omega} \times \mathcal{P}(\mathcal{T}_P \cup \mathcal{T}_U)^{\omega} \rightarrow Stmt^{\omega}$.

Note that the result of *combine* is again a program statement in *Stmt* (or a trace $Stmt^{\omega}$) over the new set of cells $\mathbb{C} \cup \mathbb{I} \cup (tmp_j)_{j \in \mathbb{N}}$, which we call $\mathbb{C}^*$.

Example 6.8. Let $\mathbb{I} = \{i\}$

$$
\begin{aligned}
&combine(n := 42, \{[\![n \leftarrowtail n + 7]\!], n > 0\}) = \\
&tmp_0 := n + 7;\ n := 42;\ i := *;\ assert(n > 0);\ assert(n = tmp_0)
\end{aligned}
$$

Definition 6.9. (Büchi Program Product) Let $P = (Stmt, Q, q_0, \delta, Q)$ be a program automaton and $A = (\mathcal{P}(\mathcal{T}_P \cup \mathcal{T}_U), Q', q_0', \delta', Q'_{acc})$ be a Büchi automaton (for example, the automaton $A_{\neg\varphi}$). The Büchi program product $P \otimes A$ is a program automaton $B = (Stmt, Q \times Q', (q_0, q_0'), \delta_B, Q_{accB})$ where

$$\delta_B = \{((p, q), combine(s, l), (p', q')) \mid (p, s, p') \in \delta \wedge (q, l, q') \in \delta'\}$$

and

$$Q_{accB} = \{(q, q') \mid q \in Q \wedge q' \in Q'_{acc}\}$$

Theorem 6.10. *Let P be a program automaton over the basic statements $Stmt_0$ with all states accepting. Let φ be a TSL formula. P satisfies φ if and only if $P \otimes A_{\neg\varphi}$ has no feasible trace.*

For the proof, see subsection 6.1.2.

We can now apply theorem 6.10 to solve the model checking problem: we have to test whether $P \otimes A_{\neg\varphi}$ accepts no feasible trace. This can be done exactly as described

in [25], Section 5. The algorithm is based on counterexample-guided abstraction refinement (CEGAR [16]) - when a trace that is accepted by the automaton is found, the trace is checked for feasibility. First, finite prefixes of the trace are checked for feasibility using an SMT-solver. If they are feasible, a ranking function synthesizer is then used to check whether the whole trace eventually terminates. If the trace is feasible, a counterexample is found. If not, the automaton is refined such that it now does no more include the spurious counterexample trace, and the process is repeated. For more details, we refer to [25].

The limitations of SMT-solvers and ranking function synthesizers also limit the functions and predicates that can be used in both the program and the TSL formula.

6.1.2 Correctness

The main idea of the correctness proof is a construction that, given a computation ζ that matches a program trace σ, constructs a computation matching the combined trace $combine(\sigma, Seq(\zeta))$ and vice versa (Seq was defined in Definition 4.16). This gives us the necessary feasibility proofs. To do so, we define two operations, $\widetilde{(-)}$ and $(-)_{|\sigma}$ that 'nearly' invert each other: we have that $(\tilde{\zeta})_{|\sigma} = \zeta$ and if $\zeta \lhd combine(\sigma, X)$ for some X, we also have that $\widetilde{\zeta_{|\sigma}} = \zeta$. In Lemma 6.13 we show that if $\zeta \lhd combine(\sigma, X)$ for some X, then $\zeta_{|\sigma} \lhd \sigma$. In Lemma 6.14 we show that then, we also have that $X = Seq(\zeta_{|\sigma})$. Lemma 6.15 states the other direction: if $\zeta \lhd \sigma$, then also $\tilde{\zeta} \lhd combine(\sigma, Seq(\zeta))$. Those three lemmata give us the feasibility proofs needed for the algorithm's correctness. Lemma 4.17 then gives the equivalence between the violation of the TSL-formula by ζ and the sequence $Seq(\zeta)$ being accepted by $A_{\neg\varphi}$, needed for reasoning about the existence of a trace $combine(\sigma, Seq(\zeta))$ in the Büchi program product.

We start with definining the operation $\widetilde{(-)}$. Let $\sigma \in Stmt^{\omega}$ and $\zeta \lhd \sigma$. We need to extend this computation to one that matches $combine(\sigma, Seq(\zeta))$. For every time point t, we need to introduce computation steps that match $combine(\sigma_t, Seq(\zeta)_t) = save_values_{Seq(\zeta)_t}; \sigma_t; new_inputs; check_preds_{Seq(\zeta)_t}; check_updates_{Seq(\zeta)_t}$. While executing $save_values_{Seq(\zeta)_t}$, the values of the temporary variables are changed as required by the statements $tmp_j := \tau_{Fj}$. When the actual statement σ_t is executed, the computation changes to ζ_t, but still with the 'old' input values and extended with values for the temporal variables. Next, when executing new_inputs, we stepwise change the input values to those in ζ_t. Then, the assertions are executed and the computation cannot change anymore.

In the following, we also need the notion of extending an assignment: we define $a[c \mapsto v](c) = v$ and $a[c \mapsto v](c') = a(c')$ for $c \neq c'$.

Let $\upsilon \subseteq \mathcal{T}_U$ be in the following the set of update terms, and $\rho \subseteq \mathcal{T}_P$ the set of predicate terms appearing in the formula φ.

Definition 6.11. Let $\mathbb{I} = \{i_1, \dots i_n\}$ be the set of inputs and $\upsilon = \{[\![c_1 \leftarrowtail \tau_{F1}]\!], \dots, [\![c_m \leftarrowtail \tau_{Fm}]\!]\}$. Given a computation ζ, we define the **adapted computation** $\tilde{\zeta}$ as follows.

$$\begin{aligned}
a_t^{tmp_1} &:= \zeta_{t-1}[tmp_1 \mapsto \eta(\tau_{F1}, \zeta_{t-1})] \\
a_t^{tmp_j} &:= a^{tmp_{j-1}}[tmp_j \mapsto \eta(\tau_{Fj}, \zeta_{t-1})] && \text{for } 1 < j \le m \\
a_t &:= \zeta_t[tmp_1 \mapsto \eta(\tau_{F1}, \zeta_{t-1}), \dots, tmp_m \mapsto \eta(\tau_{Fm}, \zeta_{t-1}), \\
&\qquad i_1 \mapsto \zeta_{t-1}(i_1),\ \dots,\ i_n \mapsto \zeta_{t-1}(i_n)] \\
a_t^{i_1} &:= a_t[i_1 \mapsto \zeta_t(i_1)] \\
a_t^{i_j} &:= a^{i_{j-1}}[i_j \mapsto \zeta_t(i_j)] && \text{for } 1 < j \le n \\
\widetilde{\zeta_t} &:= a_t^{tmp_1} \dots a_t^{tmp_m}\ a_t\ a_t^{i_1}\ \dots\ a_t^{i_n}\ a_t^{i_n}\ a_t^{i_n} \\
\tilde{\zeta} &:= \widetilde{\zeta_0}\ \widetilde{\zeta_1} \dots
\end{aligned}$$

Note that this is the only possibility to adapt a computation $\zeta \triangleleft \sigma$ such that the result *could* match $combine(\sigma, X)$ for any X.

Also note that $a_t^{i_n} = \zeta_t[tmp_1 \mapsto \eta(\tau_{F1}, \zeta_{t-1}), \dots, tmp_m \mapsto \eta(\tau_{Fm}, \zeta_{t-1})]$.

We can also define the left inverse of this operation: reducing a computation that matches $combine(\sigma, X)$ to a computation that matches σ.

Definition 6.12. Let $\sigma \in Stmt^\omega, X \in \mathcal{P}(\mathcal{T}_P \cup \mathcal{T}_U)^\omega$ and $\zeta \triangleleft combine(\sigma, X)$. We define the **reduced computation** $\zeta_{|\sigma}$ as follows.

$$\begin{aligned}
\iota(j) &:= (|\mathbb{I}| + |\upsilon| + 3) \cdot (j + 1) - 3 \\
\zeta_{|\sigma}(U) &:= (\zeta_{\iota(0)})_{|(\mathbb{I} \cup \mathbb{C})}\ (\zeta_{\iota(1)})_{|(\mathbb{I} \cup \mathbb{C})} \dots
\end{aligned}$$

where $a_{|(\mathbb{I} \cup \mathbb{C})}$ means restricting the domain of the assignment to the original inputs and cells, thus excluding the temporal variables $tmp_1, tmp_2 \dots$

Note that if $\zeta \triangleleft combine(\sigma, X)$, we also have that $\zeta = \widetilde{\zeta_{|\sigma}}$, as this is the *only* computation that could potentially match $combine(\sigma, X)$ and equals $\zeta_{|\sigma}$ when restricted to σ.

Lemma 6.13. *If $\zeta \triangleleft combine(\sigma, X)$, then $\zeta_{|\sigma} \triangleleft \sigma$.*

Proof. We have to show that $\forall t \in \mathbb{N}.\ \zeta_{|\sigma} \triangleleft_t \sigma$

Recall that $\zeta = \widetilde{\zeta_{|\sigma}}$ and

$$\widetilde{(\zeta_{|\sigma})_t} = a_t^{tmp_1} \dots a_t^{tmp_m}\ a_t\ a_t^{i_1}\ \dots\ a_t^{i_n}\ a_t^{i_n}\ a_t^{i_n}$$

- Case $\sigma_t = assert(\tau_P)$
  We know that $\zeta \triangleleft_{\iota(t)-|\mathbb{I}|} combine(\sigma, X)$. The corresponding statement is σ_j, thus

  $$\eta(\tau_P, \zeta_{\iota(t)-|\mathbb{I}|-1}) = true \quad \wedge \forall c \in \mathbb{C}^*.\ \zeta_{\iota(t)-|\mathbb{I}|}(c) = \zeta_{\iota(t)-|\mathbb{I}|-1}(c)$$

  Moreover, $(\zeta_{\iota(t)-|\mathbb{I}|-1}) = a_t^{tmp_m}$. This equals $(\zeta_{|\sigma})_{t-1}$ extended with values for the temporary variables. As τ_P does not contain the temporal variables, this means that $\eta(\tau_P, ((\zeta_{|\sigma})_{t-1})_{|(\mathbb{I}\cup\mathbb{C})})$ is also true. It remains to show that

  $$\forall c \in \mathbb{C}.\ (\zeta_{\iota(t-1)})_{|(\mathbb{I}\cup\mathbb{C})}(c) = (\zeta_{\iota(t)})_{|(\mathbb{I}\cup\mathbb{C})}(c)$$

  This is true as the only cells changed in $\zeta_{\iota(t-1)} \dots \zeta_{\iota(t)-|\mathbb{I}|-1}$ and in $\zeta_{\iota(t)-|\mathbb{I}|}, \dots \zeta_{\iota(t)}$ are cells from $\mathbb{C}^* \backslash \mathbb{C}$.

- The two remaining cases are analogous. □

Lemma 6.14. *If $\zeta \triangleleft combine(\sigma, X)$, then $X = Seq(\zeta_{|\sigma})$.*

Proof. We prove $\forall t.\ X_t = Seq(\zeta_{|\sigma})_t$. We know that $\zeta \triangleleft_{\iota(t)+1} combine(\sigma, X)$. The corresponding statement is $check_preds_{X_t}$. Set $h = \left(\bigwedge_{\tau_P \in X_t} \tau_P \wedge \bigwedge_{\tau_P \in \rho \backslash X_t} \neg\tau_P\right)$. This means that

$$\eta(h, \zeta_{\iota(t)+1}) = true \quad \wedge \forall c \in \mathbb{C}^*.\ \zeta_{\iota(t)+1}(c) = \zeta_{\iota(t)}(c)$$

This implies that $true = \eta((h, \zeta_{\iota(t)})_{|(\mathbb{I}\cup\mathbb{C})}) = \eta(h, (\zeta_{|\sigma})_t)$. Therefore, for all $\tau_P \in \rho$

$$\tau_P \in Seq(\zeta_{|\sigma})_t \Leftrightarrow t, \zeta_{|\sigma} \models \tau_P \Leftrightarrow \eta(\tau_P, \zeta_{\iota(t)}) = true \Leftrightarrow \tau_P \in X_t$$

For the update terms, we know that $\zeta \triangleleft_{\iota(t)+2} combine(\sigma, X)$. The corresponding statement is $check_updates_{X_t}$. Set $h = \left(\bigwedge_{[\![c_j \leftarrowtail \tau_{Fj}]\!] \in \upsilon} \begin{cases} c_j = tmp_j & \text{if } [\![c_j \leftarrowtail \tau_{Fj}]\!] \in X_t \\ c_j \neq tmp_j & \text{else} \end{cases}\right)$
As before, we know that $\eta(h, (\zeta_{|\sigma})_t) = true$. Moreover, we know that for each j, $(\zeta_{|\sigma})_t(tmp_j) = \eta(\tau_{Fj}, (\zeta_{|\sigma})_{t-1})$ by definition 6.11 Therefore, for every $[\![c_j \leftarrowtail \tau_{Fj}]\!] \in \upsilon$,

$$\begin{aligned} [\![c_j \leftarrowtail \tau_{Fj}]\!] \in Seq(\zeta_{|\sigma})_t &\Leftrightarrow t, \zeta_{|\sigma} \models [\![c_j \leftarrowtail \tau_{Fj}]\!] \\ &\Leftrightarrow \eta(\tau_{Fj}, \zeta_{\iota(t-1)}) = \eta(c_j, \zeta_{\iota(t)}) \\ &\Leftrightarrow \eta(c_j = tmp_j, \zeta_{\iota(t)}) = true \\ &\Leftrightarrow [\![c_j \leftarrowtail \tau_{Fj}]\!] \in X_t \end{aligned}$$

□

Lemma 6.15. *If $\zeta \triangleleft \sigma$, then $\tilde{\zeta} \triangleleft combine(\sigma, Seq(\zeta))$*

Proof. We have to show that for all t, $\tilde{\zeta} \lhd_t \mathit{combine}(\sigma, \mathit{Seq}(\zeta))$. This is clear for all time steps except for those of kind *check_preds* or *check_updates* by the definition of $\tilde{\zeta}$.

First consider *check_preds*. We need to show that $\forall t,\ \tilde{\zeta} \lhd_{\iota(t)+1} \mathit{combine}(\sigma, \mathit{Seq}(\zeta))$. This boils down to

$$\eta\left(\left(\bigwedge_{\tau_P \in \mathit{Seq}(\zeta)_t} \tau_P \wedge \bigwedge_{\tau_P \in \rho \setminus \mathit{Seq}(\zeta)_t} \neg \tau_P\right), \tilde{\zeta}_{\iota(t)}\right) = \mathit{true}$$

As the temporary variables $\mathit{tmp}_1, \mathit{tmp}_2 \dots$ are not used in any $\tau_P \in \rho$, this is by definition 6.11 equivalent to

$$\forall \tau_P \in \rho.\ \tau_P \in \mathit{Seq}(\zeta)_t \Leftrightarrow \eta(\tau_P, \zeta_t) = \mathit{true}$$

This is true by the definition of $\mathit{Seq}(\zeta)_t$.

Now consider *check_updates*. We need to show that $\forall t,\ \tilde{\zeta} \lhd_{\iota(t)+2} \mathit{combine}(\sigma, \mathit{Seq}(\zeta))$. This boils down to

$$\eta\left(\left(\bigwedge_{[\![c_j \leftarrowtail \tau_{Fj}]\!] \in \upsilon} \begin{cases} c_j = \mathit{tmp}_j & \text{if } [\![c_j \leftarrowtail \tau_{Fj}]\!] \in \mathit{Seq}(\zeta)_t \\ c_j \neq \mathit{tmp}_j & \text{else} \end{cases}\right), \tilde{\zeta}_{\iota(t)}\right) = \mathit{true}$$

Which is equivalent to

$$\forall [\![c_j \leftarrowtail \tau_{Fj}]\!] \in \upsilon.\ \eta(c_j = \mathit{tmp}_j, \tilde{\zeta}_{\iota(t)}) = \mathit{true} \Leftrightarrow [\![c_j \leftarrowtail \tau_{Fj}]\!] \in \mathit{Seq}(\zeta)_t$$

We know that $\tilde{\zeta}_{\iota(t)}(\mathit{tmp}_j) = \eta(\tau_{Fj}, \zeta_{t-1})$. Thus this is equivalent to

$$\forall [\![c_j \leftarrowtail \tau_{Fj}]\!] \in \upsilon.\ \eta(\tau_{Fj}, \zeta_{t-1}) = \zeta_t(c) \Leftrightarrow [\![c_j \leftarrowtail \tau_{Fj}]\!] \in \mathit{Seq}(\zeta)_t$$

which is again true by the definition of $\mathit{Seq}(\zeta)_t$. □

Now, we have all the lemmas needed to prove Theorem 6.10

Proof. (Theorem 6.10)
$\Rightarrow$ Assume that $P \otimes A_{\neg\varphi}$ has a feasible trace. Then, this is a trace $\mathit{combine}(\sigma, X)$ for some $\sigma \in \mathcal{L}(P)$ and $X \in \mathcal{L}(A_{\neg\varphi})$. Moreover, $\zeta \lhd \mathit{combine}(\sigma, X)$ for some $\zeta \in \mathcal{A}^\omega$. By Lemma 6.14, we know that $X = \mathit{Seq}(\zeta)$ and by Lemma 6.13 we know that $\zeta \lhd \sigma$. By the correctness of A_φ, we know that $\mathit{Seq}(\zeta) \vDash_{LTL} \neg\varphi$, which by Lemma 4.17 means that $\zeta \vDash \neg\varphi$. Thus ζ is a counterexample that proves that P does not satisfy φ.

$\Leftarrow$ Assume that P does not satisfy φ. Then, there is a trace $\sigma \in \mathcal{L}(P)$ and a computation ζ such that $\zeta \lhd \sigma$ and $\zeta \vDash \neg\varphi$. This means by Lemma 4.17 that $\mathit{Seq}(\zeta) \vDash_{LTL} \neg\varphi$, so $\mathit{Seq}(\zeta)$ is accepted by $A_{\neg\varphi}$. Then, $\mathit{combine}(\sigma, \mathit{Seq}(\zeta))$ is a trace of $P \otimes A_{\neg\varphi}$. By Lemma 6.15, $\tilde{\zeta} \lhd \mathit{combine}(\sigma, \mathit{Seq}(\zeta))$, so this is also a feasible trace.

□

6.2 Alternation-Free HyperTSL Software Model Checking

In this section, we apply the technique of self-composition to extend the algorithm from the previous section for alternation-free HyperTSL, similarily to Section 5.4.2, but now for a program automaton. Self-composition is a technique commonly used for the verification of hyperproperties [3, 4, 26]

6.2.1 The Algorithm

First, we need to define what it means for a program automaton to satisfy a HyperTSL formula.

Definition 6.16. Let P be a program automaton over the basic statements $Stmt_0$, let φ be a HyperTSL formula and let

$$Z = \{\zeta \in \mathcal{A}^\omega \mid \exists \sigma.\ \zeta \triangleleft \sigma \text{ and } \sigma \text{ is a trace of } P\}$$

P satisfies φ if $Z \vDash \varphi$.

Definition 6.17. Let $P = (Stmt, Q, q_0, \delta, Q)$ be a program automaton. We define the n-fold self-composition of P as $P^n = (Stmt', Q^n, q_0^n, \delta^n, Q^n)$ where $Stmt'$ are program statements over the set of inputs $\mathbb{I} \times \Pi$ and the set of cells $\mathbb{C} \times \Pi$ and

$$\begin{aligned} Q^n &= Q \times \cdots \times Q \\ q_0^n &= (q_0, \dots, q_0) \\ \delta^n &= \{((q_1, \dots, q_n), ((s_1)_{\pi_1}; \dots; (s_n)_{\pi_n}), (q_1', \dots, q_n')) \mid \forall 1 \leq i \leq n.\ (q_i, s_i, q_i') \in \delta\} \end{aligned}$$

where $(s)_\pi$ means renaming the cells in s from c to c_π and the inputs from i to i_π.

Theorem 6.18. *A program automaton P over the basic statements $Stmt_0$ satisfies a universal HyperTSL formula $\varphi = \forall \pi_1. \dots \forall \pi_n.\ \psi$ if and only if $P^n \otimes A_{\neg\psi}$ has no feasible trace.*

Theorem 6.19. *A program automaton P over the basic statements $Stmt_0$ satisfies an existential HyperTSL formula $\varphi = \exists \pi_1. \dots \exists \pi_n.\ \psi$ if and only if $P^n \otimes A_\psi$ has any feasible trace.*

The proof is given in the following in Section 6.2.2. As the two theorems are dual, it suffices to give the proof for Theorem 6.18.

6.2.2 Correctness

The proof is analogous to the proof of Theorem 6.10, but we have to deal with multiple traces and thus even more indices now. We give it here for completeness.

Given n program traces $\sigma_{\pi_1}, \dots \sigma_{\pi_n}$, we define $\sigma_j = ((\sigma_{\pi_1})_{\pi_1 j}; (\sigma_{\pi_2})_{\pi_2 j}; \dots (\sigma_{\pi_n})_{\pi_n j})$ and $\sigma = \sigma_1 \sigma_2 \dots$. Let $\zeta_{\pi_1} \triangleleft \sigma_{\pi_1} \wedge \dots \wedge \zeta_{\pi_n} \triangleleft \sigma_{\pi_n}$. Let $\hat{\zeta} = \emptyset[\pi_1, \zeta_{\pi_1}] \dots [\pi_n, \zeta_{\pi_n}]$. Those computations are extendable to a computation that matches $combine(\sigma, Seq(\hat{\zeta}))$ For every time point t, we need to introduce the computation steps that match $combine(\sigma_t, X_t) = save_values;\ \sigma_t;\ new_inputs;\ check_preds_{X_t};\ check_updates_{X_t}$. While executing *save_values*, the values of the relevant temporary variables are changed as required by the statements $tmp_j := \hat{\tau_{Fj}}$. After the actual statements σ_t are executed, the computation changes to $\hat{\zeta}_t$, but still with the 'old' inputs and extended with values for the temporal variables. Next, when executing *new_inputs*, we stepwise change the input values to those in $\hat{\zeta}_t$. Then, the assertions are executed and the computation cannot change anymore.

In the following, we also need the notion of extending a hyper-assignment: we define $\hat{a}[c \mapsto v](c) = v$ and $\hat{a}[c \mapsto v](c') = \hat{a}(c')$ for $c \neq c'$.

Let $\upsilon \subseteq \hat{\mathcal{T}_U}$ be in the following the set of update terms and $\rho \subseteq \hat{\mathcal{T}_P}$ the predicate terms appearing in the formula φ.

Definition 6.20. Let $\mathbb{I} \times \Pi = \{i_1, \dots i_k\}$ be the set of inputs and $\upsilon = \{[\![c_1 \leftarrowtail \hat{\tau_{F1}}]\!], \dots, [\![c_m \leftarrowtail \hat{\tau_{Fm}}]\!]\}$. Given computations $\widetilde{\zeta_{\pi_1} \dots \zeta_{\pi_n}}$, let $\hat{\zeta} = \emptyset[\pi_1, \sigma_{\pi_1}] \dots [\pi_n, \sigma_{\pi_n}]$. We define the **adapted computation** $(\widetilde{\zeta_{\pi_1}, \dots, \zeta_{\pi_n}})$.

$$
\begin{aligned}
\hat{a}_t^{tmp_1} &:= \hat{\zeta}_{t-1}[tmp_1 \mapsto \eta(\tau_{F1}, \hat{\zeta}_{t-1})] \\
\hat{a}_t^{tmp_j} &:= \hat{a}^{tmp_{j-1}}[tmp_j \mapsto \eta(\tau_{Fj}, \hat{\zeta}_{t-1})] && \text{for } 1 < j \leq m \\
\hat{a}_t^{\pi_j} &= (\hat{\zeta}_{t-1}[\pi_1, \zeta_{\pi_1}] \dots [\pi_j, \zeta_{\pi_j}])_t[i_1 \mapsto \zeta_{t-1}(i_1),\ \dots\ , i_k \mapsto \zeta_{t-1}(i_k), \\
&\quad tmp_1 \mapsto \eta(\tau_{F1}, \hat{\zeta}_{t-1}), \dots, tmp_m \mapsto \eta(\tau_{Fm}, \hat{\zeta}_{t-1})] \\
\hat{a}_t^{i_1} &:= \hat{a}_t^{\pi_n}[i_1 \mapsto \hat{\zeta}_t(i_1)] \\
\hat{a}_t^{i_j} &:= \hat{a}^{i_{j-1}}[i_j \mapsto \hat{\zeta}_t(i_j)] && \text{for } 1 < j \leq k \\
(\widetilde{\zeta_{\pi_1}, \dots, \zeta_{\pi_n}})^t &:= \hat{a}_t^{tmp_1} \dots \hat{a}_t^{tmp_n}\ \hat{a}_t^{\pi_1} \dots \hat{a}_t^{\pi_n}\ \hat{a}_t^{i_1}\ \dots\ \hat{a}_t^{i_k}\ \hat{a}_t^{i_k}\ \hat{a}_t^{i_k} \\
(\widetilde{\zeta_{\pi_1}, \dots, \zeta_{\pi_n}}) &:= (\widetilde{\zeta_{\pi_1}, \dots, \zeta_{\pi_n}})^0\ (\widetilde{\zeta_{\pi_1}, \dots, \zeta_{\pi_n}})^1 \dots
\end{aligned}
$$

Note that this is the only possibility to extend the computations $\zeta_{\pi_1} \triangleleft \sigma_{\pi_1}, \dots, \zeta_{\pi_n} \triangleleft \sigma_{\pi_n}$ to a computation that potentially matches $combine(\sigma, X)$ for any X.

We can also define the left inverse of this operation: reducing a computation that matches $combine(\sigma, X)$ to computations that match $\sigma_{\pi_1}, \dots, \sigma_{\pi_n}$ as follows.

Definition 6.21. Let $1 \leq j \leq n, \sigma \in Stmt^\omega, X \in \mathcal{P}(\hat{\mathcal{T}_P} \cup \hat{\mathcal{T}_U})^\omega$ and $\hat{\zeta} \triangleleft combine(\sigma, X)$. We define the index of the computation step of $(\sigma_{\pi_j})_t$ in $combine(\sigma, X)$

$$\iota(t) := (|\mathbb{I} \times \Pi| + |\upsilon| + n + 2) \cdot (t + 1) - 3$$

We define the **reduced computation** $\zeta_{|\pi_j}$.

$$\zeta_{|\pi_j} := (\zeta_{\iota(0)})_{|(\mathbb{I}\cup\mathbb{C})_{\pi_j}} \; (\zeta_{\iota(1)})_{|(\mathbb{I}\cup\mathbb{C})_{\pi_j}} \dots$$

where $\hat{a}_{|(\mathbb{I}\cup\mathbb{C})_{\pi_j}}$ means restricting the domain of the assignment to the cells and inputs labeled with π_j, thus excluding the temporal variables $tmp_1, tmp_2 \dots$ and the variables from other traces. Moreover, the cells and inputs are again renamed from c_{π_j} to c or i_{π_j} to i.

Note that if $\hat{\zeta} \triangleleft combine(\sigma, X)$, we also have that $\hat{\zeta}$ is the adapted computation of $(\hat{\zeta}_{|\pi_1}, \dots, \hat{\zeta}_{|\pi_n})$ as this is the *only* computation that could match $combine(\sigma, X)$ and equals $\zeta_{|\pi_j}$ when restricted to π_j.

Lemma 6.22. *If* $\hat{\zeta} \triangleleft combine(\sigma, X)$ *and* $\sigma = ((\sigma_{\pi_1})_{\pi_1}, \dots, (\sigma_{\pi_n})_{\pi_n})$*, then* $\hat{\zeta}_{|\pi_j} \triangleleft \sigma_{\pi_j}$ *for every* $1 \leq j \leq n$*.*

Proof. We show that $\forall t \in \mathbb{N}.\ \hat{\zeta}_{|\pi_j} \triangleleft_t \sigma_{\pi_j}$.

Recall that $\hat{\zeta}$ is the adapted computation of $(\hat{\zeta}_{|\pi_1}, \dots, \hat{\zeta}_{|\pi_n})$.

- Case $\sigma_t = assert(\tau_P)$
  We know that $\hat{\zeta} \triangleleft_{\iota(t)-|\mathbb{I}\times\Pi|-(n-j)} combine(\sigma, X)$, and thus

  $$\eta(\hat{\tau_P}, \hat{\zeta}_{\iota(t)-|\mathbb{I}\times\Pi|-(n-j)-1}) = true \;\; \wedge$$
  $$\forall c \in \mathbb{C}^*.\ \hat{\zeta}_{\iota(t)-|\mathbb{I}\times\Pi|-(n-j)}(c) = \hat{\zeta}_{\iota(t)-|\mathbb{I}\times\Pi|(n-j)-1}(c)$$

  Moreover, $(\hat{\zeta}_{\iota(t)-|\mathbb{I}\times\Pi|-(n-j)-1})$ equals $\hat{a}_t^{tmp_m}$ if $j = 0$ and else $\hat{a}_t^{\pi_{j-1}}$, which both equals $(\hat{\zeta}_{|\pi_j})_{t-1}$ when restricted to the inputs and variables from π_j. τ_P does not contain variables from other traces or temporary variables, thus $\eta(\tau_P, (\hat{\zeta}_{|\pi_j})_{t-1})$ is also true. It remains to show that

  $$\forall c \in \mathbb{C}.\ ((\hat{\zeta})_{\iota(t-1)})_{|(\mathbb{I}\cup\mathbb{C})_{\pi_j}}(c) = ((\hat{\zeta})_{\iota(t)})_{|(\mathbb{I}\cup\mathbb{C})_{\pi_j}}(c)$$

  This is also true as the only cells changed in $\hat{\zeta}_{\iota(t-1)} \dots \hat{\zeta}_{\iota(t)-|\mathbb{I}\times\Pi|-(n-j)-1}$ and in $\hat{\zeta}_{\iota(t)-(n-j)-|\mathbb{I}\times\Pi|}, \dots \hat{\zeta}_{\iota(t)}$ are cells from $\mathbb{C}^* \backslash \mathbb{C}$ or cells from other traces.
- The two remaining cases are analogous.

□

Lemma 6.23. *If $\hat{\zeta} \triangleleft combine(\sigma, X)$, then $X = Seq(\varnothing[\pi_1, \hat{\zeta}_{|\pi_1}] \ldots [\pi_n, \hat{\zeta}_{|\pi_n}])$*

Proof. Set $\hat{\zeta}' = \varnothing[\pi_1, \hat{\zeta}_{|\pi_1}] \ldots [\pi_n, \hat{\zeta}_{|\pi_n}]$. We prove $\forall t.\ X_t = Seq(\hat{\zeta}')_t$. We know that $\hat{\zeta} \triangleleft_{\iota(t)+1} combine(\sigma, X)$. The corresponding statement is $check_preds_{X_t}$. Set $h = \left(\bigwedge_{\tau_P \in X_t} \hat{\tau_P} \wedge \bigwedge_{\hat{\tau_P} \in \rho \setminus X_t} \neg \tau_P\right)$. This means that

$$\eta(h, \hat{\zeta}_{\iota(t)+1}) = true \quad \wedge\ \forall c \in \mathbb{C}^*.\ \zeta_{\iota(t)+1}(c) = \hat{\zeta}_{\iota(t)}(c)$$

Recall that $\hat{\zeta}_{\iota(t)+1}$ is by Definition 6.20 equal to

$$\begin{aligned}&(\hat{\zeta}'_{t-1}[\pi_1, \hat{\zeta}'_{\pi_1}] \ldots [\pi_j, \hat{\zeta}'_{\pi_n}])_t\ [tmp_1 \mapsto \eta(\tau_{\hat{F}1}, \hat{\zeta}'_{t-1}), \ldots, tmp_m \mapsto \eta(\tau_{\hat{F}m}, \hat{\zeta}'_{t-1})]\\ &= \hat{\zeta}'_t\ [tmp_1 \mapsto \eta(\tau_{\hat{F}1}, \hat{\zeta}'_{t-1}), \ldots, tmp_m \mapsto \eta(\tau_{\hat{F}m}, \hat{\zeta}'_{t-1})]\end{aligned}$$

h does not contain the temporal variables, so this implies that $\eta(h, \hat{\zeta}'_t) = true$. Therefore, for all $\tau_P \in P$

$$\tau_P \in Seq(\hat{\zeta}')_t \Leftrightarrow t, \hat{\zeta}' \vDash \tau_P \Leftrightarrow \eta(\tau_P, \zeta'_t) = true \Leftrightarrow \tau_P \in X_t$$

The last equivalence holds by the definition of h.

For the update terms, we know that $\zeta \triangleleft_{\iota(t)+2} combine(\sigma, X)$. The corresponding statement is $check_updates_{X_t}$. Set $h = \left(\bigwedge_{[\![c_j \leftarrowtail \tau_{\hat{F}j}]\!] \in \upsilon} \begin{cases} c_j = tmp_j & \text{if } [\![c_j \leftarrowtail \tau_{\hat{F}j}]\!] \in X_t \\ c_j \neq tmp_j & \text{else} \end{cases}\right)$
As before, we know that $\eta(h, \hat{\zeta}'_t) = true$. Moreover, we know that for each j, $\hat{\zeta}_{\iota(t)+1}(tmp_j) = \eta(\tau_{\hat{F}j}, \hat{\zeta}'_{t-1})$ again by Definition 6.20 Therefore, for every $[\![c_j \leftarrowtail \tau_{\hat{F}j}]\!] \in \upsilon$,

$$\begin{aligned}[\![c_j \leftarrowtail \tau_{\hat{F}j}]\!] \in Seq(\hat{\zeta}')_t &\Leftrightarrow t, \hat{\zeta}' \vDash [\![c_j \leftarrowtail \tau_{\hat{F}j}]\!]\\ &\Leftrightarrow \eta(\tau_{\hat{F}j}, \hat{\zeta}'_{t-1}) = \eta(c_j, \hat{\zeta}'_t)\\ &\Leftrightarrow \eta(c_j = tmp_j, \hat{\zeta}'_t) = true\\ &\Leftrightarrow [\![c_j \leftarrowtail \tau_{\hat{F}j}]\!] \in X_t\end{aligned}$$

The last equivalence is again true by the definition of h. □

Lemma 6.24. *If $\zeta_{\pi_1} \triangleleft \sigma_{\pi_1} \wedge \cdots \wedge \zeta_{\pi_n} \triangleleft \sigma_{\pi_n}$, then $(\widetilde{\zeta_{\pi_1}, \ldots, \zeta_{\pi_n}}) \triangleleft combine(\sigma, Seq(\hat{\zeta}'))$, where $\hat{\zeta}' = \varnothing[\pi_1, \zeta_{\pi_1}] \ldots [\pi_n, \zeta_{\pi_n}]$*

Proof. Set $\hat{\zeta} = (\widetilde{\zeta_{\pi_1}, \ldots, \zeta_{\pi_n}})$. We have to show that for all t, $\hat{\zeta} \triangleleft_t combine(\sigma, Seq(\hat{\zeta}'))$. This is clear for all time steps except for those of kind *check_preds* or *check_updates* by the definition of $\hat{\zeta}$.

First consider *check_preds*. We need to show that $\forall t,\ \hat{\zeta} \lhd_{\iota(t)+1} combine(\sigma, Seq(\hat{\zeta}'))$. This boils down to

$$\eta\left(\left(\bigwedge_{\hat{\tau_P} \in Seq(\hat{\zeta}')_t} \hat{\tau_P} \wedge \bigwedge_{\hat{\tau_P} \in \rho \setminus Seq(\hat{\zeta}')_t} \neg\hat{\tau_P}\right), \hat{\zeta}_{\iota(t)+1}\right) = true$$

Recall that $\hat{\zeta}_{\iota(t)+1}$ is by Definition 6.20 equal to

$$\begin{aligned}&(\hat{\zeta}'_{t-1}[\pi_1, \hat{\zeta}'_{\pi_1}]\dots[\pi_j, \hat{\zeta}'_{\pi_n}])_t\ [tmp_1 \mapsto \eta(\hat{\tau_{F1}}, \hat{\zeta}'_{t-1}), \dots, tmp_m \mapsto \eta(\hat{\tau_{Fm}}, \hat{\zeta}_{t-1})]\\ &= \hat{\zeta}'_t\ [tmp_1 \mapsto \eta(\hat{\tau_{F1}}, \hat{\zeta}'_{t-1}), \dots, tmp_m \mapsto \eta(\hat{\tau_{Fm}}, \hat{\zeta}_{t-1})]\end{aligned}$$

Thus, as the temporary variables are not used in $\hat{\tau_P}$, this is equivalent to

$$\forall \hat{\tau_P} \in \rho.\ \hat{\tau_P} \in Seq(\zeta')_t \Leftrightarrow \eta((\hat{\tau_P}), \zeta'_t) = true$$

This is true by the definition of $Seq(\zeta)_t$.

Now consider *check_updates*. We need to show that $\forall t,\ \hat{\zeta} \lhd_{\iota(t)+2} combine(\sigma, Seq(\hat{\zeta}'))$. This boils down to

$$\eta\left(\left(\bigwedge_{[\![c_j \leftarrowtail \hat{\tau_{Fj}}]\!] \in \upsilon} \begin{cases} c_j = tmp_j & \text{if } [\![c_j \leftarrowtail \hat{\tau_{Fj}}]\!] \in Seq(\hat{\zeta}')_t \\ c_j \neq tmp_j & \text{else}\end{cases}\right), \hat{\zeta}_{\iota(t)+2}\right) = true$$

Which is again equivalent to

$$\forall [\![c_j \leftarrowtail \tau_{Fj}]\!] \in \upsilon.\ \eta(c_j = tmp_j, \hat{\zeta}_{\iota(t)+2}) = true \Leftrightarrow [\![c_j \leftarrowtail \tau_{Fj}]\!] \in Seq(\hat{\zeta}')_t$$

We know that $\hat{\zeta}_{\iota(t)+2}(tmp_j) = \hat{\zeta}'_{t-1}(\tau_{Fj})$. Thus this is equivalent to

$$\forall [\![c_j \leftarrowtail \hat{\tau_{Fj}}]\!] \in \upsilon.\ \eta(\hat{\tau_{Fj}}, \hat{\zeta}'_{t-1}) = \hat{\zeta}'_t(c) \Leftrightarrow [\![c_j \leftarrowtail \hat{\tau_{Fj}}]\!] \in Seq(\hat{\zeta}')_t$$

Which is again true by the definition of $Seq(\hat{\zeta}')_t$. □

Now, we have all the lemmas needed to prove Theorem 6.18

Proof. (Theorem 6.18)
$\Rightarrow$ Assume that $P^n \otimes A_{\neg\varphi}$ has a feasible trace. Then, this is a trace $combine(\sigma, X)$ for some $\sigma \in \mathcal{L}(P^n)$ and $X \in \mathcal{L}(A_{\neg\varphi})$. We know that $\sigma_t = ((\sigma_{\pi_1})_{\pi_1 t}; \dots; (\sigma_{\pi_n})_{\pi_n t})$. Moreover, $\hat{\zeta} \lhd combine(\sigma, X)$ for some $\hat{\zeta} \in \hat{\mathcal{A}}^\omega$. By Lemma 6.22, we know that $\hat{\zeta}_{|\pi_1} \lhd \sigma_{\pi_1} \wedge \dots \wedge \hat{\zeta}_{|\pi_n} \lhd \sigma_{\pi_n}$. Set $\hat{\zeta}' = \varnothing[\pi_1, \hat{\zeta}_{|\pi_1}]\dots[\pi_n, \hat{\zeta}_{|\pi_n}]$. By Lemma 6.23, we know that $X = Seq(\hat{\zeta}')$. By the correctness of A_ψ this means that $Seq(\hat{\zeta}') \vDash_{LTL} \neg\varphi$ which

by Lemma 4.17 means that $\hat{\zeta}' \vDash \neg\varphi$. Thus, $\hat{\zeta}_{|\pi_1} \dots \hat{\zeta}_{|\pi_n}$ are feasible counterexample traces proving that $\forall \pi_1. \dots \forall \pi_n. \psi$ does not hold.

$\Leftarrow$ Assume that P does not satisfy φ. Then, there are trace $\sigma_{\pi_1}, \dots \sigma_{\pi_n} \in \mathcal{L}(P)$ and computations $\zeta_{\pi_1}, \dots \zeta_{\pi_n}$ such that $\zeta_{\pi_1} \lhd \sigma_{\pi_1} \wedge \dots \wedge \zeta_{\pi_n} \lhd \sigma_{\pi_n}$ and $\hat{\zeta}' = \varnothing[\pi_1, \zeta_{\pi_1}] \dots [\pi_n, \zeta_{\pi_n}] \vDash \neg\varphi$. This means by Lemma 4.17 that $Seq(\hat{\zeta}') \vDash_{LTL} \neg\varphi$, so $Seq(\hat{\zeta}')$ is accepted by $A_{\neg\varphi}$. Set $\sigma_t = ((\sigma_{\pi_1})_{\pi_1 t}; \dots; (\sigma_{\pi_n})_{\pi_n t})$ and $\sigma = \sigma_0 \sigma_1 \dots$. Then, $combine(\sigma, Seq(\zeta))$ is a trace of $P \otimes A_{\neg\varphi}$. By Lemma 6.24, $(\zeta_{\pi_1}, \dots, \zeta_{\pi_n}) \lhd combine(\sigma, Seq(\zeta'))$, so this is also a feasible trace.

□

6.3 Finding Counterexamples for the $\forall^*\exists^*$ - Fragment

In this section, we give a sound but incomplete algorithm for finding counterexamples for $\forall^*\exists^*$ HyperTSL formulas in software model checking. To the best of our knowledge, there is also for HyperLTL no such algorithm yet, thus this also gives the first software model checking algorithm for finding counterexamples for $\forall^*\exists^*$ HyperLTL formulas. We also obtain an algorithm for finding witnesses proving $\exists^*\forall^*$ HyperTSL or HyperLTL formulas as such witnesses are counterexamples for the negated formula.

It is not directly possible to combine the algorithm from the previous sections with the finite-state HyperLTL model checking algorithm [27] to obtain an algorithm for infinite-state $\forall^*\exists^*$ HyperTSL model checking. The reason for that is that the finite-state HyperLTL model checking algorithm involves complementation. The automaton for $\exists^*\psi$ is complemented, constructing an automaton containing all the traces that are either not from the system or do not satisfy ψ. However, in our infinite-state algorithm, the automaton for $\exists^*\psi$ still contains all the system traces that do not satisfy ψ (they correspond to infeasible traces in the automaton for $\exists^*\psi$). This means that when complementing the automaton in our setting, the traces that do not satisfy ψ would not be accepted by the complemented automaton. To summarize, when complementing an automaton containing infeasible traces, the infeasible traces are lost.

Removing all infeasible traces from the automaton is impossible in general as the resulting automaton would be a finite-state system describing an infinite-state system. Therefore, the main idea of this section is to remove only parts of the infeasible traces from the automaton before complementing it. This can already be sufficient for identifying a counterexample.

6.3.1 The Algorithm

For finding counterexamples for $\forall^*\exists^*$ HyperTSL formulas, we again construct an automaton from the program and the formula with the property that if it has a feasible trace, then the property is violated. However, the converse is not true anymore - if it has no feasible trace, then the formula will not necessarily hold. Consequently, the algorithm can only find counterexamples, but not verify correctness.

Consider in the following a HyperTSL formula $\varphi = \forall \pi_1. \dots \forall \pi_m. \exists \pi_{m+1}. \dots \exists \pi_n. \psi$ and a program automaton P. First, we construct the Büchi program product $P^n \otimes A_\varphi$, whose feasible traces correspond to tuples of n traces that satisfy φ. Next, we remove some infeasibility from the automaton.

More precisely, we propose two techniques for removing infeasibility. The first technique removes *k-infeasibility* from the automaton, that is, a local inconsistency in a trace, occurring within k consecutive time steps. When choosing k, there is a tradeoff: if k is larger, more counterexamples can be identified, but the algorithm is also slowed down.

The second technique removes *infeasible accepting cycles* from the automaton. It might not be possible to remove all of them, thus the number of iterations has to be bounded.

Example 6.25. The trace

$$(n--;\ assert(n>=0))\ (n:=1;\ assert(n>=0))\ (n--;\ assert(n>=0))^\omega$$

is 3-infeasible because no matter what the value of n is before the second time step, the assertion in the fourth time step will fail. In contrast, the trace

$$(n:=*)\ (n--;\ assert(n>=0))^\omega$$

is not k-infeasible for any k, because the value of n can always be large enough to make the first k assertions succeed. Still, the trace is infeasible because n cannot decrease forever without dropping below zero. If such a trace is accepted by an automaton, $n--;\ assert(n>=0)$ corresponds to an infeasible accepting cycle of this automaton.

Definition 6.26. Let $k \in \mathbb{N}$, $\sigma \in Stmt^\omega$. We call σ k-infeasible if there exists a $j \in \mathbb{N}$ such that $\sigma_j\ \sigma_{j+1} \dots \sigma_{j+k-1}\ assert(true)^\omega$ is infeasible for all possible initial assignments ζ_{-1}. We then also call the subsequence $\sigma_j\ \sigma_{j+1}\ \dots \sigma_{j+k-1}$ infeasible. If a trace is not k-infeasible, we call it k-feasible.

Whether a subsequence $\sigma_j\ \sigma_{j+1} \dots \sigma_{j+k-1}$ is a witness of k-infeasibility can be checked using an SMT-solver e.g [11, 13, 15, 24]. For removing k-infeasibility from an automaton, we reuse the idea from the update term elimination in Section 5.3 to construct an automaton that 'remembers' the $k-1$ previous statements. The nodes of the new automaton correspond to paths of k nodes in the original automaton. We only add a

transition labeled with l between two nodes p and q if we can extend the trace of the path of p with l such that the resulting trace subsequence is k-feasible.

Definition 6.27. Let $P = (Stmt, Q, q_0, \delta, Q_{acc})$ be a program automaton. Let $k \in \mathbb{N}$. We define P without k-infeasibility, as $P_k = (Stmt, Q', q_0, \delta', Q'_{acc})$ where

$$\begin{aligned}
Q' :=& \{(q_1, s_1, q_2 \dots, s_{k-1}, q_k) \mid (q_1, s_1, q_2) \in \delta \wedge \dots \wedge (q_{k-1}, s_{k-1}, q_k) \in \delta\} \cup \\
& \{(q_0, s_0, q_1 \dots, s_{k'-1}, q_{k'}) \mid k' < k-1 \wedge (q_0, s_0, q_1) \in \delta \wedge \dots \wedge (q_{k'-1}, s_{k'-1}, q_{k'}) \in \delta\} \\
\delta' :=& \{((q_1, s_1, q_2 \dots, s_{k-1}, q_k), s_k, (q_2, s_2, \dots, q_k, s_k, q_{k+1})) \in Q' \times Stmt \times Q' \\
& \mid s_1 \dots s_k \text{ feasible}\} \cup \\
& \{((q_0, s_0, q_1 \dots, s_{k'-1}, q_{k'}), s_{k'}, (q_0, s_0, \dots, q_{k'}, s_{k'}, q_{k'+1})) \in Q' \times Stmt \times Q' \\
& \mid k' < k-1 \wedge s_0 \dots s_{k'} \text{ feasible}\} \\
Q'_{acc} :=& \{(q_1, s_1, q_2 \dots, s_{k-1}, q_k) \in Q' \mid q_k \in Q_{acc}\} \cup \\
& \{(q_0, s_0, q_1 \dots, s_{k'-1}, q_{k'}) \in Q' \mid k' < k-1 \wedge q_{k'} \in Q_{acc}\}
\end{aligned}$$

Theorem 6.28. *P_k accepts exactly the k-feasible traces of the program automaton P.*

Proof. $\Rightarrow$ Let $q_0, q_1, q_2 \dots \in Q^\omega$ be a run of P on the k-feasible trace σ. Then, for every $j \in \mathbb{N}$,

$$e_j = ((q_j, \sigma_j, q_{j+1} \dots, \sigma_{j+k-2}, q_{j+k-1}), \sigma_{j+k-1}, (q_{j+1}, \sigma_{j+1}, \dots, q_{j+k-1}, \sigma_{j+k-1}, q_{j+k}))$$

is a transition of P_k. Moreover, for every $k' < k$,

$$e_{k'} = ((q_0, s_0, q_1 \dots, \sigma_{k'-1}, q_{k'}), \sigma_{k'}, (q_0, \sigma_0, \dots, q_{k'}, \sigma_{k'}, q_{k'+1}))$$

is also a transition of P_k. Thus, $q_0, q_1, \dots$ is accepted by P_k.

$\Leftarrow$ Let σ be a trace of P accepted by P_k. Then, there exist states of P $q_0, q_1 \dots$ such that for every $j \in \mathbb{N}$, e_j from above is a transition of P_k. Thus, by the definition of P_k for every j, $\sigma_j \dots \sigma_{j+k-1}$ is feasible. Thus, σ is k-feasible. □

Definition 6.29. Let $P = (Stmt, Q, q_0, \delta, Q_{acc})$ be a program automaton. Let $\varrho = (q_1, s_1, q_2)\ (q_2, s_2, q_3)\ \dots\ (q_n, s_n, q_1)$ be a sequence of transitions of P. We call ϱ an **infeasible accepting cycle** if there is a $1 \leq j \leq n$ with $q_j \in Q_{acc}$ and $(s_1\ s_2 \dots s_{n-1})^\omega$ is infeasible for all possible initial assignments ζ_{-1}.

For removing infeasible accepting cycles, we first enumerate all simple cycles of the automaton. To do so, we use the algorithm from [35]. This algorithm does not include the cycles induced by self-loops, so we have to add them to the set of cycles afterward. For each cycle ϱ that contains at least one accepting state, we test its feasibility using first an SMT-solver to test if ϱ is locally infeasible and then using a ranking function

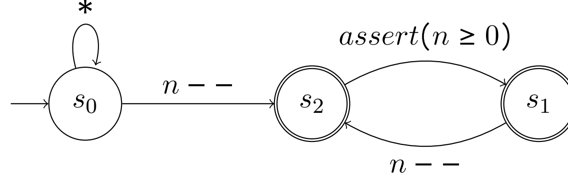

Figure 6.2: The automaton A_ϱ for the infeasible cycle $\varrho = (s_1,\ n--,\ s_2)$ $(s_2,\ assert(n > 0),\ s_1)$. The edge label $*$ denotes an edge for every (relevant) statement

synthesizer e.g [5, 10, 20, 32, 44] to test if ϱ^ω is infeasible. If we successfully prove infeasibility, we refine the model, using the methods from [33, 34]. This refinement is formalized in the following.

Definition 6.30. Let P be a program automaton and $C \subseteq (Q \times Stmt \times Q)^\omega$ be a set of infeasible accepting cycles of P. Let $\varrho = (q_1, s_1, q_2)\ (q_2, s_2, q_3)\ \dots\ (q_{n-1}, s_{n-1}, q_n) \in C$. We define the automaton for ϱ, written A_ϱ as

$$A_\varrho = (Stmt, Q = \{q_0, q_1, \dots q_n\}, q_0, \delta, Q \setminus \{q_0\}) \text{ where}$$
$$\delta = \{(q_0, s, q_0) \mid s \in Stmt\} \cup \{(q_j, s_j, q_{j+1}) \mid 1 \leq j < n\} \cup \{(q_0, s_1, q_2), (q_n, s_n, q_1)\}$$

A_ϱ accepts exactly the traces that end with ϱ^ω, without any restriction on the prefix. See Figure 6.2 for an example. We want to exclude the traces in A_ϱ from P. Thus we define

$$P_C := P \setminus \left(\bigcup_{\varrho \in C} A_\varrho \right)$$

This construction can be repeated to exclude the infeasible accepted cycles that are newly created in P_C. If we iterate this process k' times, we name the result $P_{C(k')}$.

Consider now a HyperTSL formula $\varphi = \forall \pi_1.\ \dots \forall \pi_m.\ \exists \pi_{m+1} \dots \exists \pi_n.\ \psi$ and a program automaton P. For finding a counterexample, we first construct the Büchi program product $P^n \otimes A_\psi$. Each feasible accepted trace of $P^n \otimes A_\psi$ corresponds to a combination of n feasible program traces that make ψ true. Next, we eliminate k-infeasibility and remove k'-times infeasible accepting cycles from the Büchi program product, constructing an automaton $(P^n \otimes A_\psi)_{k,C(k')}$. Using this modified Büchi program product, we can obtain an overapproximation of the program execution combinations satisfying the existential part of the formula. Each trace of the Büchi program product is a combination of n program executions and a predicate/update term sequence. We can 'extract' the m universally quantified program executions from a feasible trace, obtaining a tuple of m program executions that fulfill the existential part of the formula. Applying this 'extraction' (formally called *projection*) to

all traces of $(P^n \otimes A_\psi)_{k,C(k')}$ leads to an overapproximation of the program executions satisfying the existential part of the formula. In the following, we formally define the projection.

Definition 6.31. Let P be a program automaton, $n, m \in \mathbb{N}, m \leq n$ and A_ψ be the automaton for the formula ψ. Let $(P^n \otimes A)_{k,C(k')} = (Stmt, Q, q_0, \delta, Q_{acc})$. We define the projected automaton $(P^m \otimes A)^\forall_{k,C(k')} = (Stmt, Q, q_0, \delta^\forall, Q_{acc})$ where

$$\delta^\forall = \{(q, (s_1; \ldots ;\ s_m), q') \mid \exists s_{m+1}, \ldots s_n, l.\ (q, combine(s_1; \ldots ;\ s_n, l), q') \in \delta\}$$

Now, it only remains to check whether the overapproximation contains all tuples of m feasible program executions. If not, a counterexample is found. This boils down to testing if $P^m \backslash (P^n \otimes A_\psi)^\forall_{k,C(k')}$ has any feasible trace, which we can check as described in the previous sections.

The following theorem states the soundness of our algorithm:

Theorem 6.32. *Let $\varphi = \forall \pi_1. \ \ldots \forall \pi_m. \ \exists \pi_{m+1}. \ \ldots \exists \pi_n$. If $P^m \backslash (P^n \otimes A_\psi)^\forall_{k,C(k')}$ has a feasible trace, then P does not satisfy φ.*

For the proof, see section 6.3.2.

6.3.2 Correctness

We now prove Theorem 6.32. To do that, we need the following lemma: recall that for a program execution σ_π, $(\sigma_\pi)_\pi$ means renaming every cell c in σ_π to c_π and every input i to i_π.

Lemma 6.33. *Let $\sigma \in Stmt^\omega$ be feasible and $\sigma_t = (((\sigma_{\pi_1})_{\pi_1})_t, \ldots, ((\sigma_{\pi_n})_{\pi_n})_t)$ for some $\sigma_{\pi_1}, \ldots \sigma_{\pi_n}$. Then $\sigma_{\pi_1}, \ldots, \sigma_{\pi_n}$ are also all feasible.*

Proof. As σ is feasible, we know that $\hat{\zeta} \lhd \sigma$ for some $\hat{\zeta}$. For all $1 \leq j \leq n$, we define ζ_{π_j} by

$$(\zeta_{\pi_j})_t = (\hat{\zeta}_{t \cdot n + j - 1})_{|(\mathbb{I} \cup \mathbb{C})_{\pi_j}}$$
$$\zeta_{\pi_j} = (\zeta_{\pi_j})_0 \ (\zeta_{\pi_j})_1 \ldots$$

where $\hat{a}_{|(\mathbb{I} \cup \mathbb{C})_{\pi_j}}$ as before means restricting the domain of the assignment to the cells and inputs labeled with π_j, thus excluding the variables from other traces. Moreover, the cells and inputs are again renamed from c_{π_j} to c or i_{π_j} to i. $t \cdot m + j - 1$ is the index of $(\sigma_{\pi_j})_t$ in σ.

We show that for all time points t, $\zeta_{\pi_j} \lhd_t \sigma_{\pi_j}$

- Case $(\zeta_{\pi_j})_t = assert(\tau_P)$
  We know that $\hat{\zeta} \lhd_{t \cdot m+j-1} \sigma$ and thus

  $$\eta(\hat{\tau_P}, \hat{\zeta}_{t \cdot m+j-1}) = true \land \forall c \in \mathbb{C} \times \Pi.\ \hat{\zeta}_{t \cdot m+j-1}(c) = \hat{\zeta}_{t \cdot m+j-2}.$$

  Moreover $\eta(\tau_P, (\zeta_{\pi_j})_t)$ is also true as τ_P does not contain variables from other traces. It remains to show that

  $$\forall c \in \mathbb{C}.\ ((\zeta_{\pi_j})_t)(c) = (\zeta_{\pi_j})_{t-1}(c)$$

  This is also true as the only cells changed in $\hat{\zeta}_{(t-1) \cdot m+j-1}, \ldots \hat{\zeta}_{t \cdot m+j-2}$ are cells from other traces.

- The remaining two cases are analogous.

□

We now proof Theorem 6.32

Proof. Assume that $P^m \backslash (P^n \otimes A_\psi)^{\forall}_{k,C(k')}$ has a feasible trace σ with $\hat{\zeta} \lhd \sigma$. By Lemma 6.33, this means that $\zeta_{\pi_1} \lhd \sigma_{\pi_1} \land \cdots \land \zeta_{\pi_m} \lhd \sigma_{\pi_m}$. It suffices to show that $\varnothing[\pi_1, \zeta_1] \ldots [\pi_m, \zeta_m] \nvDash \exists \pi_{m+1}.\ \ldots \exists \pi_n.\ \psi$ as this implies that $\zeta_1, \ldots \zeta_m$ are a counterexample proving that P does not satifsfy φ.

Proof by contradiction. Assume that $\varnothing[\pi_1, \zeta_1] \ldots [\pi_m, \zeta_m] \vDash \exists \pi_{m+1}.\ \ldots \exists \pi_n.\ \psi$. Then, there are traces $\sigma_{\pi_{m+1}}, \ldots \sigma_{\pi_n}$ and computations $\zeta_{\pi_{m+1}} \ldots \zeta_{\pi_n}$ such that $\zeta_{\pi_{m+1}} \lhd \sigma_{\pi_{m+1}} \land \cdots \land \zeta_{\pi_n} \lhd \sigma_{\pi_n}$ and $\hat{\zeta}' = \varnothing[\pi_1, \zeta_{\pi_1}] \ldots [\pi_n, \zeta_{\pi_n}] \vDash \psi$.
Set $\sigma'_t = ((\sigma_{\pi_1})_{\pi_1 t}; \ldots; (\sigma_{\pi_n})_{\pi_n t})$ and $\sigma' = \sigma'_0\ \sigma'_1 \ldots$. Now, by Lemma 4.17 and the correctness of A_ψ, we know that $Seq(\hat{\zeta}')$ is accepted by A_ψ, thus $combine(\sigma', Seq(\hat{\zeta}'))$ is accepted by $P^n \otimes A_\psi$. Moreover, by Lemma 6.23, we know that $combine(\sigma', Seq(\hat{\zeta}'))$ is also feasible, so it is also k-feasible and thus accepted by $(P^n \otimes A_\psi)_k$. Moreover, does not end with an infeasible cycle and is thus also accepted by $(P^n \otimes A_\psi)_{k,C(k')}$ But this means that σ is accepted by $(P^n \otimes A_\psi)^{\forall}_{k,C(k')}$. and thus not by $P^m \backslash (P^n \otimes A_\psi)^{\forall}_{k,C(k')}$. Contradiction. □

6.3.3 Example Applications

In this section, we apply the algorithm to two simple examples to show that removing some infeasibility can already be sufficient for identifying a counterexample.

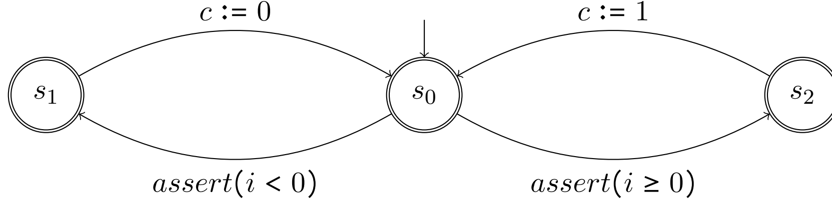

Figure 6.3: The program automaton P used in the first example

Example 1

In the first example, removing local inconsistency is already sufficient. The property we check here is *generalized noninterference* [40], a formula stating that when observing an output cell c, an observer can not gain any information about the secret input i. We've already seen noninterference as an example in Section 4.2. Compared to noninterference, generalized noninterference is a weaker property as it does not require determinism, but is already sufficient for the system to be secure. Intuitively, generalized noninterference states that when replacing the values of the secret input i with a dummy value λ, the output c should not be changed.

$$\forall \pi.\ \exists \pi'.\ \Box(i_{\pi'} = \lambda \wedge c_\pi = c_{\pi'})$$

We model-check this property on the simple program automaton P shown in Figure 6.3 that violates it since for the trace $(assert(i < 0)\ c := 0)^\omega$ there exists no other trace where c is always equal, but i is always zero.

P^2 is shown in Figure 6.4. For brevity, we write A for *assert* and write consecutive assertions as one.

We choose as a dummy value $\lambda = 0$. The automaton for $\psi = \Box(i_{\pi'} = 0 \wedge c_\pi = c_{\pi'})$ consists of a single accepting state with the self-loop labeled with $\tau_P = (i_{\pi'} = 0 \wedge c_\pi = c_{\pi'})$. The Büchi program product is shown in Figure 6.5

For this simple example, it suffices to choose $k = 1$. Then $(P^2 \otimes A_\psi)_k$ is the Büchi program product with all locally inconsistent transitions removed. The automaton $(P^2 \otimes A_\psi)_k^\forall$ is shown in Figure 6.6

This automaton for example does not contain the trace $assert(i < 0)\ (c := 0)^\omega$ which is a feasible trace of P. This trace is therefore a feasible trace accepted by $P \backslash (P^2 \otimes A_\psi)_k^\forall$. Thus, we know that this trace is a counterexample proving that P does not satisfy generalized noninterference – there is no feasible trace that agrees on the value of the output cell c but has always $i = 0$. For this example, it is not necessary to remove infeasible cycles.

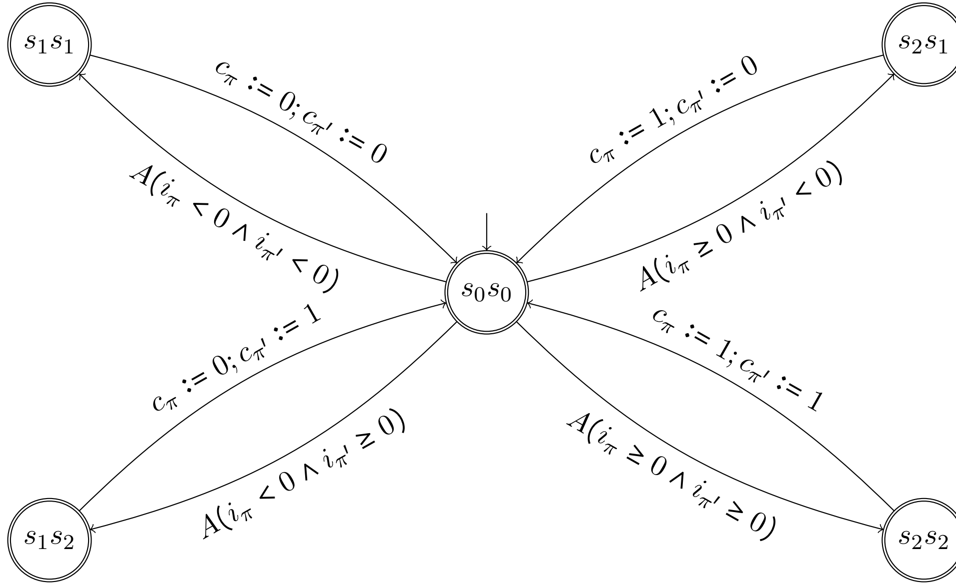

Figure 6.4: The program automaton P^2

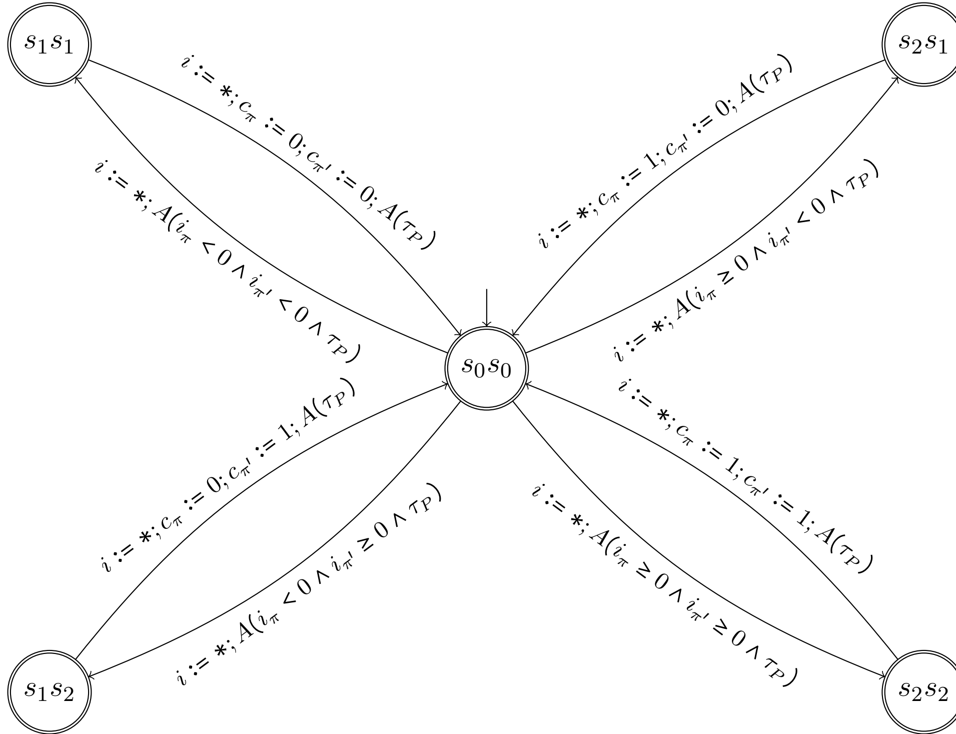

Figure 6.5: The program automaton $P^2 \otimes A_\varphi$

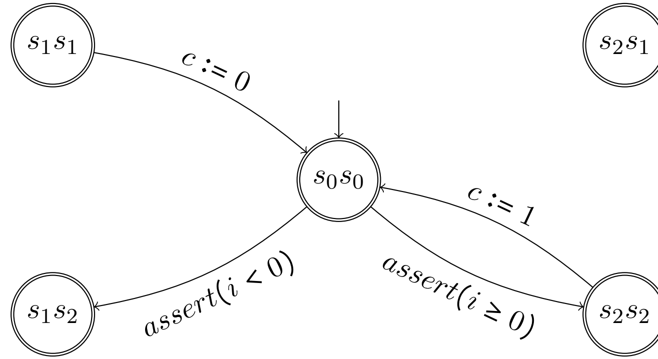

Figure 6.6: The program automaton $(P^2 \otimes A_\psi)_k^\forall$

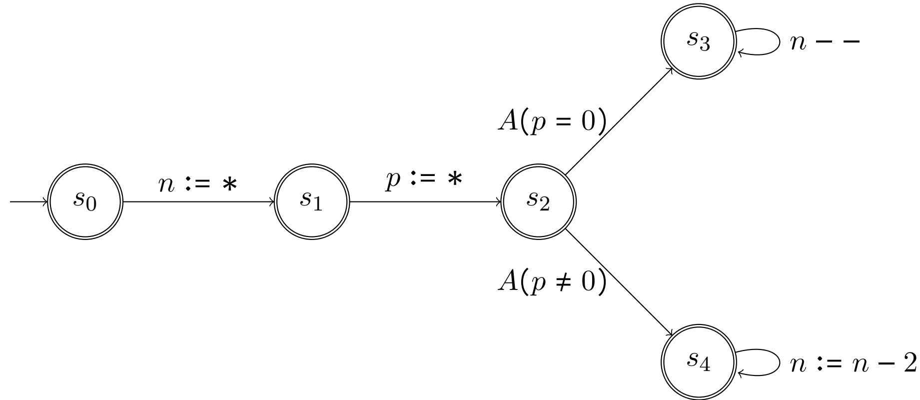

Figure 6.7: The program automaton P used in the second example

Example 2

Next, we show an example where removing k-infeasibility is not sufficient, but removing infeasible accepting cycles leads to a counterexample. We check the property

$$\varphi = \forall\pi.\ \exists\pi'.\ \Box(p_\pi \neq p_{\pi'} \land n_\pi < n'_\pi)$$

on the program automaton shown in Figure 6.7.

φ states that for every trace π, there is another trace π' where p differs, but n is always greater. This property is not fulfilled, the trace $n := *;\ p := *;\ assert(p = 0);\ (n--)^\omega$ is a counterexample, as any trace π' where p is different will decrease its n by two in every time step, and thus $n_{\pi'}$ will eventually drop below n_π. P^2 is shown in Figure 6.8.

In the Büchi program product, the structure of the automaton stays the same, but the assertion $assert(p_\pi \neq p_{\pi'} \land n_\pi < n'_\pi)$ is added to every state. Then, we remove

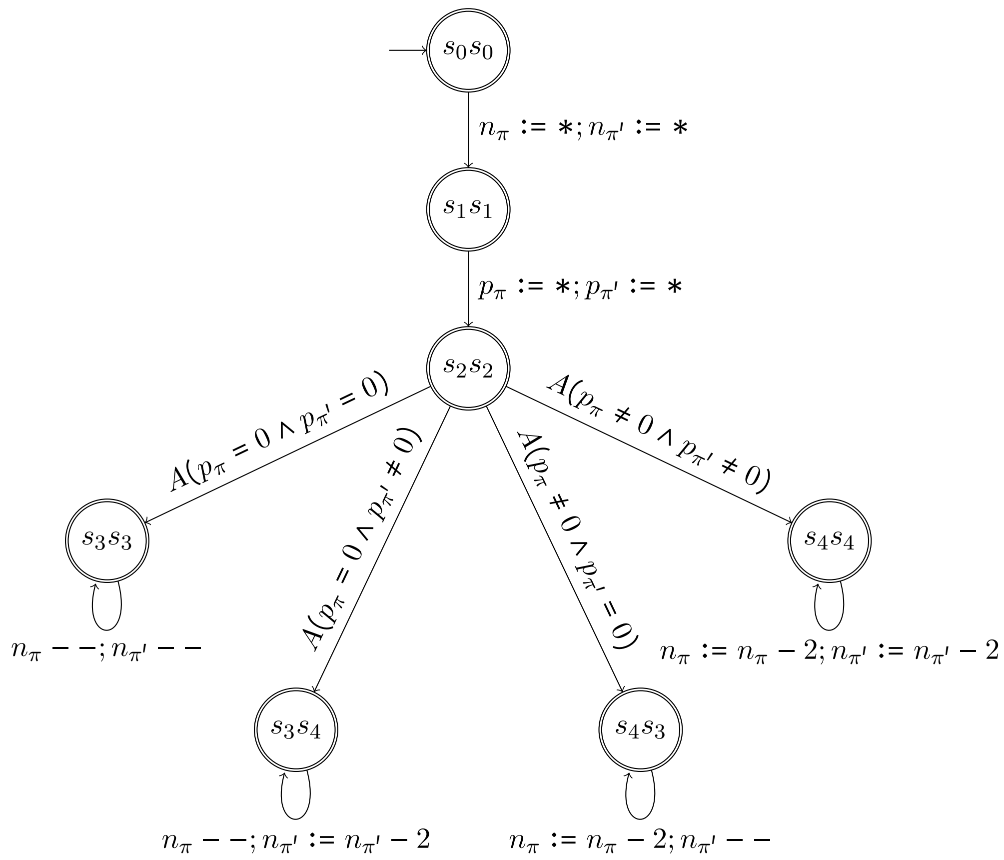

Figure 6.8: The program automaton P^2

local infeasibility. We again choose $k = 1$, but the only 1-infeasible transition is the transition from s_2s_2 to s_3s_3 - our counterexample trace is still in the automaton. Choosing a greater k is also not helpful here, as the remaining traces of the Büchi program product are not k-infeasible for any k.

However, the self-loop at s_3s_4 is an infeasible accepting cycle – the sequence

$$(n_\pi --;\ n_{\pi'} := n_{\pi'} - 2;\ assert(n_\pi < n_{\pi'}))^\omega$$

must eventually terminate. We choose $k' = 1$ and thus remove all traces ending with this cycle. Next, we project the automaton to the universal part. Then, the trace $n := *;\ p := *;\ assert(p = 0); (n--)^\omega$ which we identified as a counterexample at the very beginning of this section, is not accepted by the automaton $(P^2 \otimes A_\psi)^\forall_{1,C(1)}$. But it is in P and feasible, thus it is identified as a counterexample when computing $P \setminus (P^2 \otimes A_\psi)^\forall_{1,C(1)}$.

Chapter 7

Discussion

7.1 Conclusion

In this thesis, we have developed TSL and HyperTSL model checking algorithms both for finite-state (Chapter 5) and infinite-state systems (Chapter 6). We have shown that for a concrete finite data domain, (Hyper)TSL is not more expressive than (Hyper)LTL by giving a translation algorithm from the former to the latter. Still, many properties can be expressed more compactly in (Hyper)TSL. Therefore, we have also given direct model checking algorithms for TSL and HyperTSL with at most one quantifier alternation which are more efficient than model checking an equivalent (Hyper)LTL formula. We leave the implementation of these algorithms to future work. The implementation will allow evaluating whether the (Hyper)TSL model checking algorithms are indeed also faster in practice on suitable systems and properties.

In the infinite-state case, (Hyper)TSL is more expressive than (Hyper)LTL. We have extended the automata-based LTL software model checking algorithm by Dietsch et al. [25] for TSL. Remarkably, this extension comes at no additional cost in the sense that the size of the automaton that is checked for a feasible trace is not increased. By applying the technique of self-composition, we have further generalized the algorithm for model checking alternation-free HyperTSL.

We have extended this further to a sound but incomplete algorithm for finding counterexamples disproving $\forall^*\exists^*$ HyperTSL formulas and witnesses proving $\exists^*\forall^*$ HyperTSL formulas. Our algorithm makes it possible to find program executions violating important security properties like generalized noninterference, that are only expressible by using a combination of universal and existential quantifiers. For finding a counterexample, the key idea was to remove some infeasibility from the Büchi program product for overapproximating the set of program execution combinations

satisfying the existential part of the formula. We remove the local k-infeasibility and traces ending with infeasible accepting cycles.

For software model checking hyperproperties with quantifier alternations, there was, to the best of our knowledge, also no algorithm yet for HyperLTL. It remains open whether there is also an algorithm for proving $\forall^*\exists^*$ hyperproperties.

7.2 Future Work

Evaluation of the Model Checking Algorithms. The contributions of this thesis are purely theoretical. It would be interesting to implement our model checking algorithms to evaluate how efficient and scalable they are in practice. For the finite-state case, this would allow comparing the practical performance of (Hyper)TSL model checking with the performance of model checking equivalent (Hyper)LTL formulas to examine whether (Hyper)TSL model checking could be a promising, more efficient alternative for suitable systems and properties.
For the infinite-state case, the implementation would in particular make it possible to test whether the partial algorithm for finding counterexamples disproving $\forall^*\exists^*$-HyperTSL formulas like generalized noninterference succeeds also on larger program snippets than the examples examined in Chapter 6.

Proving $\forall^*\exists^*$ hyperproperties on programs. Our HyperTSL software model checking algorithm can only find counterexamples disproving $\forall^*\exists^*$ hyperproperties. If the algorithm does not find a counterexample, this does not necessarily mean that the formula holds. Proving $\forall^*\exists^*$ hyperproperties is more difficult - while a counterexample is a single trace, a proof is a function mapping each trace to a witness for the existential part of the formula. As our algorithm *overapproximates* the set of executions satisfying the existential part of the formula, the naive approach for adapting this algorithm for proving $\forall^*\exists^*$ hyperproperties would be to *underapproximate* the set of traces satisfying the existential part. However, an underapproximation that contains all the system executions is not an approximation, but an automaton that exactly contains all program executions. The set of all program executions is usually not an ω-regular language and can thus not be the language of a Büchi automaton. Therefore this approach could only succeed on very special programs. This means that proving $\forall^*\exists^*$ hyperproperties for infinite-state systems requires a different approach than ours.

Extensions for more quantifier alternations. Both our finite-state and infinite-state HyperTSL model checking algorithms are limited to the fragment with one quantifier alternation. For the finite-state algorithm, translating the formula to HyperLTL and applying HyperLTL model checking also gives a full HyperTSL model checking algorithm, but it would be interesting if there is also a more efficient one. For the infinite-state case, there is no algorithm yet for model checking hyperproperties

with more than one quantifier alternation, neither for HyperTSL nor for HyperLTL. It would be interesting whether partial algorithms for more quantifier alternations could be found.

Bibliography

[1] Alejandro Aguirre, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Pierre-Yves Strub. A relational logic for higher-order programs. *Proc. ACM Program. Lang.*, 1(ICFP):21:1–21:29, 2017. doi:10.1145/3110265.

[2] Tomás Babiak, Mojmír Kretínský, Vojtech Rehák, and Jan Strejcek. LTL to büchi automata translation: Fast and more deterministic. In Cormac Flanagan and Barbara König, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 18th International Conference, TACAS 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*, volume 7214 of *Lecture Notes in Computer Science*, pages 95–109. Springer, 2012. doi:10.1007/978-3-642-28756-5_8.

[3] Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. Secure information flow by self-composition. *Math. Struct. Comput. Sci.*, 21(6):1207–1252, 2011. doi:10.1017/S0960129511000193.

[4] Gilles Barthe, Juan Manuel Crespo, and César Kunz. Beyond 2-safety: Asymmetric product programs for relational program verification. In Sergei N. Artëmov and Anil Nerode, editors, *Logical Foundations of Computer Science, International Symposium, LFCS 2013, San Diego, CA, USA, January 6-8, 2013. Proceedings*, volume 7734 of *Lecture Notes in Computer Science*, pages 29–43. Springer, 2013. doi:10.1007/978-3-642-35722-0_3.

[5] Amir M. Ben-Amram and Samir Genaim. On the linear ranking problem for integer linear-constraint loops. In Roberto Giacobazzi and Radhia Cousot, editors, *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, pages 51–62. ACM, 2013. doi:10.1145/2429069.2429078.

[6] Nick Benton. Simple relational correctness proofs for static analyses and program transformations. In Neil D. Jones and Xavier Leroy, editors, *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming*

Languages, POPL 2004, Venice, Italy, January 14-16, 2004, pages 14–25. ACM, 2004. doi:10.1145/964001.964003.

[7] Raven Beutner and Bernd Finkbeiner. Software verification of hyperproperties beyond k-safety. In Sharon Shoham and Yakir Vizel, editors, *Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part I*, volume 13371 of *Lecture Notes in Computer Science*, pages 341–362. Springer, 2022. doi:10.1007/978-3-031-13185-1_17.

[8] Laura Bozzelli, Mojmír Kretínský, Vojtech Rehák, and Jan Strejcek. On decidability of LTL model checking for process rewrite systems. In S. Arun-Kumar and Naveen Garg, editors, *FSTTCS 2006: Foundations of Software Technology and Theoretical Computer Science, 26th International Conference, Kolkata, India, December 13-15, 2006, Proceedings*, volume 4337 of *Lecture Notes in Computer Science*, pages 248–259. Springer, 2006. doi:10.1007/11944836_24.

[9] Aaron R. Bradley. Sat-based model checking without unrolling. In Ranjit Jhala and David A. Schmidt, editors, *Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings*, volume 6538 of *Lecture Notes in Computer Science*, pages 70–87. Springer, 2011. doi:10.1007/978-3-642-18275-4_7.

[10] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. Linear ranking with reachability. In Kousha Etessami and Sriram K. Rajamani, editors, *Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005, Proceedings*, volume 3576 of *Lecture Notes in Computer Science*, pages 491–504. Springer, 2005. doi:10.1007/11513988_48.

[11] Roberto Bruttomesso, Edgar Pek, Natasha Sharygina, and Aliaksei Tsitovich. The opensmt solver. In Javier Esparza and Rupak Majumdar, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 16th International Conference, TACAS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, volume 6015 of *Lecture Notes in Computer Science*, pages 150–153. Springer, 2010. doi:10.1007/978-3-642-12002-2_12.

[12] J Richard Büchi. On a decision method in restricted second order arithmetic. In *The collected works of J. Richard Büchi*, pages 425–435. Springer, 1990.

[13] Jürgen Christ, Jochen Hoenicke, and Alexander Nutz. Smtinterpol: An interpolating SMT solver. In Alastair F. Donaldson and David Parker, editors, *Model Checking Software - 19th International Workshop, SPIN 2012, Oxford, UK, July 23-24, 2012. Proceedings*, volume 7385 of *Lecture Notes in Computer Science*, pages 248–254. Springer, 2012. doi:10.1007/978-3-642-31759-0_19.

[14] Alessandro Cimatti and Alberto Griggio. Software model checking via IC3. In

P. Madhusudan and Sanjit A. Seshia, editors, *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, volume 7358 of *Lecture Notes in Computer Science*, pages 277–293. Springer, 2012. doi:10.1007/978-3-642-31424-7_23.

[15] Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. The mathsat5 SMT solver. In Nir Piterman and Scott A. Smolka, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, volume 7795 of *Lecture Notes in Computer Science*, pages 93–107. Springer, 2013. doi:10.1007/978-3-642-36742-7_7.

[16] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In E. Allen Emerson and A. Prasad Sistla, editors, *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer, 2000. doi:10.1007/10722167_15.

[17] Michael R. Clarkson, Bernd Finkbeiner, Masoud Koleini, Kristopher K. Micinski, Markus N. Rabe, and César Sánchez. Temporal logics for hyperproperties. In Martín Abadi and Steve Kremer, editors, *Principles of Security and Trust - Third International Conference, POST 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*, volume 8414 of *Lecture Notes in Computer Science*, pages 265–284. Springer, 2014. doi:10.1007/978-3-642-54792-8_15.

[18] Norine Coenen, Bernd Finkbeiner, César Sánchez, and Leander Tentrup. Verifying hyperliveness. In Isil Dillig and Serdar Tasiran, editors, *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I*, volume 11561 of *Lecture Notes in Computer Science*, pages 121–139. Springer, 2019. doi:10.1007/978-3-030-25540-4_7.

[19] Norine Coenen, Bernd Finkbeiner, Jana Hofmann, and Julia Tillman. Smart contract synthesis modulo hyperproperties. *To appear at the 36th IEEE Computer Security Foundations Symposium (CSF 2023)*, 2023.

[20] Michael Colón and Henny Sipma. Practical methods for proving program termination. In Ed Brinksma and Kim Guldstrand Larsen, editors, *Computer Aided Verification, 14th International Conference, CAV 2002,Copenhagen, Denmark, July 27-31, 2002, Proceedings*, volume 2404 of *Lecture Notes in Computer Science*, pages 442–454. Springer, 2002. doi:10.1007/3-540-45657-0_36.

[21] Michael Colón and Tomás E. Uribe. Generating finite-state abstractions of reactive systems using decision procedures. In Alan J. Hu and Moshe Y. Vardi, editors, *Computer Aided Verification, 10th International Conference, CAV '98, Vancouver, BC, Canada, June 28 - July 2, 1998, Proceedings*, volume 1427 of *Lecture Notes in Computer Science*, pages 293–304. Springer, 1998. doi:10.1007/BFb0028753.

[22] Byron Cook, Eric Koskinen, and Moshe Y. Vardi. Temporal property verification as a program analysis task. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 333–348. Springer, 2011. doi:10.1007/978-3-642-22110-1_26.

[23] Jakub Daniel, Alessandro Cimatti, Alberto Griggio, Stefano Tonetta, and Sergio Mover. Infinite-state liveness-to-safety via implicit abstraction and well-founded relations. In Swarat Chaudhuri and Azadeh Farzan, editors, *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I*, volume 9779 of *Lecture Notes in Computer Science*, pages 271–291. Springer, 2016. doi:10.1007/978-3-319-41528-4_15.

[24] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008. doi:10.1007/978-3-540-78800-3_24.

[25] Daniel Dietsch, Matthias Heizmann, Vincent Langenfeld, and Andreas Podelski. Fairness modulo theory: A new approach to LTL software model checking. In Daniel Kroening and Corina S. Pasareanu, editors, *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, volume 9206 of *Lecture Notes in Computer Science*, pages 49–66. Springer, 2015. doi:10.1007/978-3-319-21690-4_4.

[26] Marco Eilers, Peter Müller, and Samuel Hitz. Modular product programs. *ACM Trans. Program. Lang. Syst.*, 42(1):3:1–3:37, 2020. doi:10.1145/3324783.

[27] Bernd Finkbeiner, Markus N. Rabe, and César Sánchez. Algorithms for model checking hyperltl and hyperctl ˆ*. In Daniel Kroening and Corina S. Pasareanu, editors, *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, volume 9206 of *Lecture Notes in Computer Science*, pages 30–48. Springer, 2015. doi:10.1007/978-3-319-21690-4_3.

[28] Bernd Finkbeiner, Felix Klein, Ruzica Piskac, and Mark Santolucito. Temporal stream logic: Synthesis beyond the bools. In Isil Dillig and Serdar Tasiran, editors, *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I*, volume 11561 of *Lecture Notes in Computer Science*, pages 609–629. Springer, 2019. doi:10.1007/978-3-030-25540-4_35.

[29] Bernd Finkbeiner, Philippe Heim, and Noemi Passing. Temporal stream logic modulo theories. In Patricia Bouyer and Lutz Schröder, editors, *Foundations of Software Science and Computation Structures - 25th International Conference, FOSSACS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings*, volume 13242 of *Lecture Notes in Computer Science*, pages 325–346. Springer, 2022. doi:10.1007/978-3-030-99253-8_17.

[30] Bernd Finkbeiner, Jana Hofmann, Florian Kohn, and Noemi Passing. Reactive synthesis of smart contract control flows. *CoRR*, abs/2205.06039, 2022. doi:10.48550/arXiv.2205.06039.

[31] Gideon Geier, Philippe Heim, Felix Klein, and Bernd Finkbeiner. Syntroids: Synthesizing a game for fpgas using temporal logic specifications. In Clark W. Barrett and Jin Yang, editors, *2019 Formal Methods in Computer Aided Design, FMCAD 2019, San Jose, CA, USA, October 22-25, 2019*, pages 138–146. IEEE, 2019. doi:10.23919/FMCAD.2019.8894261.

[32] Matthias Heizmann, Jochen Hoenicke, Jan Leike, and Andreas Podelski. Linear ranking for linear lasso programs. In Dang Van Hung and Mizuhito Ogawa, editors, *Automated Technology for Verification and Analysis - 11th International Symposium, ATVA 2013, Hanoi, Vietnam, October 15-18, 2013. Proceedings*, volume 8172 of *Lecture Notes in Computer Science*, pages 365–380. Springer, 2013. doi:10.1007/978-3-319-02444-8_26.

[33] Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. Software model checking for people who love automata. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, volume 8044 of *Lecture Notes in Computer Science*, pages 36–52. Springer, 2013. doi:10.1007/978-3-642-39799-8_2.

[34] Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. Termination analysis by learning terminating programs. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, volume 8559 of *Lecture Notes in Computer Science*, pages 797–813. Springer, 2014. doi:10.1007/978-3-319-08867-9_53.

[35] Donald B. Johnson. Finding all the elementary circuits of a directed graph. *SIAM J. Comput.*, 4(1):77–84, 1975. doi:10.1137/0204007.

[36] Pim Kars. The application of promela and spin in the BOS project. In Jean-Charles Grégoire, Gerard J. Holzmann, and Doron A. Peled, editors, *The Spin Verification System, Proceedings of a DIMACS Workshop, New Brunswick, New Jersey, USA, August, 1996*, volume 32 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 51–63. DIMACS/AMS, 1996. doi:10.1090/dimacs/032/05.

[37] Nils Klarlund. Progress measures for complementation of omega-automata with applications to temporal logic. In *32nd Annual Symposium on Foundations of Computer Science, San Juan, Puerto Rico, 1-4 October 1991*, pages 358–367. IEEE Computer Society, 1991. doi:10.1109/SFCS.1991.185391.

[38] Saul Kripke. Semantical considerations of the modal logic. *Studia Philosophica*, 1, 2007.

[39] Benedikt Maderbacher and Roderick Bloem. Reactive synthesis modulo theories using abstraction refinement. *CoRR*, abs/2108.00090, 2021.

[40] Daryl McCullough. Noninterference and the composability of security properties. In *Proceedings of the 1988 IEEE Symposium on Security and Privacy, Oakland, California, USA, April 18-21, 1988*, pages 177–186. IEEE Computer Society, 1988. doi:10.1109/SECPRI.1988.8110.

[41] Shohei Mochizuki, Masaya Shimakawa, Shigeki Hagihara, and Naoki Yonezaki. Fast translation from LTL to büchi automata via non-transition-based automata. In Stephan Merz and Jun Pang, editors, *Formal Methods and Software Engineering - 16th International Conference on Formal Engineering Methods, ICFEM 2014, Luxembourg, Luxembourg, November 3-5, 2014. Proceedings*, volume 8829 of *Lecture Notes in Computer Science*, pages 364–379. Springer, 2014. doi:10.1007/978-3-319-11737-9_24.

[42] Oded Padon, Jochen Hoenicke, Giuliano Losa, Andreas Podelski, Mooly Sagiv, and Sharon Shoham. Reducing liveness to safety in first-order logic. *Proc. ACM Program. Lang.*, 2(POPL):26:1–26:33, 2018. doi:10.1145/3158114.

[43] Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*, pages 46–57. IEEE Computer Society, 1977. doi:10.1109/SFCS.1977.32.

[44] Andreas Podelski and Andrey Rybalchenko. A complete method for the synthesis of linear ranking functions. In Bernhard Steffen and Giorgio Levi, editors, *Verification, Model Checking, and Abstract Interpretation, 5th International*

Conference, VMCAI 2004, Venice, Italy, January 11-13, 2004, Proceedings, volume 2937 of *Lecture Notes in Computer Science*, pages 239–251. Springer, 2004. doi:10.1007/978-3-540-24622-0_20.

[45] Shmuel Safra. On the complexity of omega-automata. In *29th Annual Symposium on Foundations of Computer Science, White Plains, New York, USA, 24-26 October 1988*, pages 319–327. IEEE Computer Society, 1988. doi:10.1109/SFCS.1988.21948.

[46] A. Prasad Sistla, Moshe Y. Vardi, and Pierre Wolper. The complementation problem for büchi automata with applications to temporal logic (extended abstract). In Wilfried Brauer, editor, *Automata, Languages and Programming, 12th Colloquium, Nafplion, Greece, July 15-19, 1985, Proceedings*, volume 194 of *Lecture Notes in Computer Science*, pages 465–474. Springer, 1985. doi:10.1007/BFb0015772.

[47] Yih-Kuen Tsay and Moshe Y. Vardi. From linear temporal logics to büchi automata: The early and simple principle. In Ernst-Rüdiger Olderog, Bernhard Steffen, and Wang Yi, editors, *Model Checking, Synthesis, and Learning - Essays Dedicated to Bengt Jonsson on The Occasion of His 60th Birthday*, volume 13030 of *Lecture Notes in Computer Science*, pages 8–40. Springer, 2021. doi:10.1007/978-3-030-91384-7_2.