Automata-Based Software Model Checking of Hyperproperties

Bernd Finkbeiner[1], Hadar Frenkel[1], Jana Hofmann[2], and Janine Lohse[3(✉)]

[1] CISPA Helmholtz Center for Information Security, Saarbrücken, Germany
[2] Azure Research, Microsoft, Cambridge, UK
[3] Saarland University, Saarbrücken, Germany
s8jalohs@stud.uni-saarland.de

Abstract. We develop model checking algorithms for Temporal Stream Logic (TSL) and Hyper Temporal Stream Logic (HyperTSL) modulo theories. TSL extends Linear Temporal Logic (LTL) with memory cells, functions and predicates, making it a convenient and expressive logic to reason over software and other systems with infinite data domains. HyperTSL further extends TSL to the specification of hyperproperties – properties that relate multiple system executions. As such, HyperTSL can express information flow policies like noninterference in software systems. We augment HyperTSL with theories, resulting in HyperTSL(T), and build on methods from LTL software verification to obtain model checking algorithms for TSL and HyperTSL(T). This results in a sound but necessarily incomplete algorithm for specifications contained in the $\forall^*\exists^*$ fragment of HyperTSL(T). Our approach constitutes the first software model checking algorithm for temporal hyperproperties with quantifier alternations that does not rely on a finite-state abstraction.

1 Introduction

Hyperproperties [20] generalize trace properties [2] to system properties, i.e., properties that reason about a system in its entirety and not just about individual execution traces. Hyperproperties comprise many important properties that are not expressible as trace properties, e.g., information flow policies [20], sensitivity and robustness of cyber-physical systems, and linearizability in distributed computing [11]. For software systems, typical hyperproperties are program refinement or fairness conditions such as symmetry.

For the specification of hyperproperties, Linear Temporal Logic [50] (LTL) has been extended with trace quantification, resulting in Hyper Linear Temporal Logic [19] (HyperLTL). There exist several model checking algorithms for HyperLTL [19,23,37], but they are designed for finite-state systems and are therefore

This work was partially supported by the European Research Council (ERC) Grant HYPER (No. 101055412).

J. Hofmann—Research carried out while at CISPA Helmholtz Center for Information Security.

K. Y. Rozier and S. Chaudhuri (Eds.): NFM 2023, LNCS 13903, pp. 361–379, 2023.
https://doi.org/10.1007/978-3-031-33170-1_22

not directly applicable to software. Existing algorithms for software verification of temporal hyperproperties (e.g., [1,9]) are, with the exception of [10], limited to universal hyperproperties, i.e., properties without quantifier alternation.

In this paper, we develop algorithms for model checking software systems against $\forall^*\exists^*$ hyperproperties. Our approach is complementary to the recently proposed approach of [10]. They require to be given a finite-state abstraction of the system, based on which they can both prove and disprove $\forall^*\exists^*$ hyperproperties. We do not require abstractions and instead provide sound but necessarily incomplete approximations to detect counterexamples of the specification.

The class of $\forall^*\exists^*$ hyperproperties contains many important hyperproperties like program refinement or *generalized noninterference* [47]. Generalized noninterference states that it is impossible to infer the value of a high-security input by observing the low-security outputs. Unlike *noninterference*, it does not require the system to be deterministic. Generalized noninterference can be expressed as $\varphi_{gni} = \forall\pi\exists\pi'.\square(i_{\pi'} = \lambda \wedge c_\pi = c_{\pi'})$. The formula states that replacing the value of the high-security input i with some dummy value λ does not change the observable output c.

The above formula can only be expressed in HyperLTL if i and c range over a finite domain. This is a real limitation in the context of software model checking, where variables usually range over infinite domains like integers or strings. To overcome this limitation, our specifications build on Hyper Temporal Stream Logic (HyperTSL) [22]. HyperTSL replaces HyperLTL's atomic propositions with memory cells together with predicates and update terms over these cells. Update terms use functions to describe how the value of a cell changes from the previous to the current step. This makes the logic especially suited for specifying software properties.

HyperTSL was originally designed for the synthesis of software systems, which is why all predicates and functions are uninterpreted. In the context of model checking, we have a concrete system at hand, so we should interpret functions and predicates according to that system. We therefore introduce HyperTSL(T) – HyperTSL with interpreted theories – as basis for our algorithms.

Overview. Following [41], we represent our system as a symbolic automaton labeled with program statements. Not every trace of such an automaton is also a valid program execution: for example, a trace $assert(n = 0)\,; n{-}{-}; \,(assert(n = 0))^{\omega}$[1] cannot 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 can, in theory, succeed. As a first step, we tackle TSL model checking (Sect. 4) by constructing a program automaton whose feasible accepted traces correspond to program executions that violate the TSL specification. To do so, we adapt the algorithm of [27], which constructs such an automaton for LTL, combining the given program automaton and an automaton for the negated specification.

We then extend this algorithm for HyperTSL(T) formulas without quantifier alternation (Sect. 5.1) by applying *self-composition*, a technique commonly used for the verification of hyperproperties [5,6,30].

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

Next, in Sect. 5.2, we further extend this algorithm to finding counterexamples for $\forall^*\exists^*$-HyperTSL(T) specifications (and, dually, witnesses for $\exists^*\forall^*$ formulas). We construct an automaton that over-approximates the combinations of program executions that satisfy the existential part of the formula. If some program execution is not included in the over-approximation, this execution is a counterexample proving that the program violates the specification.

More concretely, for a HyperTSL(T) formula $\forall^m\exists^n\psi$, we construct the product of the automaton for ψ and the n-fold self-composition of the program automaton. Every feasible trace of this product corresponds to a choice of executions for the variables $\pi_1, \ldots, \pi_n$ such that ψ is satisfied. Next, we remove (some) spurious witnesses by removing infeasible traces. We consider two types of infeasibility: *k-infeasibility*, that is, a local inconsistency in a trace appearing within k consecutive timesteps; and infeasibility that is not local, and is the result of some *infeasible accepting cycles* in the automaton. In the next step, we project the automaton to the universally quantified traces, obtaining an over-approximation of the trace combinations satisfying the existential part of the formula. Finally, all that remains to check is whether the over-approximation includes all combinations of feasible traces.

Lastly, in Sect. 6, we demonstrate our algorithm for two examples, including generalized noninterference.

Contributions. We present an automata-based algorithm for software model checking of $\forall^*\exists^*$-hyperproperties. We summarize our contributions as follows.

- We extend HyperTSL with theories, a version of HyperTSL that is suitable for model checking.
- We adapt the approach of [27] to TSL(T) and alternation-free HyperTSL(T), and thereby suggest the first model checking algorithm for both TSL(T) and HyperTSL(T).
- We further extend the algorithm for disproving $\forall^*\exists^*$ hyperproperties and proving $\exists^*\forall^*$ hyperproperties using a feasibility analysis.

Related Work. Temporal stream logic extends linear temporal logic [50] and was originally designed for synthesis [35]. For synthesis, the logic has been successfully applied to synthesize the FPGA game 'Syntroids' [39], and to synthesize smart contracts [34]. To advance smart contract synthesis, TSL has been extended to HyperTSL in [22]. The above works use a version TSL that leaves functions and predicates uninterpreted. While this choice is very well suited for the purpose of synthesis, for model checking it makes more sense to use the interpretation of the program at hand. TSL was extended with theories in [33], which also analyzed the satisfiability problem of the logic. Neither TSL nor HyperTSL model checking has been studied so far (with or without interpreted theories).

For LTL, the model checking problem for infinite-state models has been extensively studied, examples are [13,16,25,27,38]. Our work builds on the automata-based LTL software model checking algorithm from [27]. There are also various algorithms for verifying universal hyperproperties on programs, for

example, algorithms based on type theory [1,9]. Major related work is [10], which (in contrast to our approach) requires on predicate abstractions to model check software against $\forall^*\exists^*$ HyperLTL specifications. They can also handle asynchronous hyperproperties, which is currently beyond our scope. Another proposal for the verification of $\forall\exists$ hyperproperties on software is [52]. Here, generalized constrained horn clauses are used to verify functional specifications. The approach is not applicable to reactive, non-terminating programs. Recently, it was also proposed to apply model checkers for TLA (a logic capable of expressing software systems as well as their properties) to verify $\forall^*\exists^*$ hyperproperties [45].

Beyond the scope of software model checking, the verification of hyperproperties has been studied for various system models and classes of hyperproperties. Model checking has been studied for ω-regular properties [21,31,37] and asynchronous hyperproperties [7,12] in finite-state Kripke structures, as well as timed systems [43], real-valued [49] and probabilistic hyperproperties [3,28,29] (some of which study combinations of the above).

2 Preliminaries

A Büchi Automaton is a tuple $\mathcal{A} = (\Sigma, Q, \delta, q_0, F)$ where Σ is a finite alphabet; Q is a set of states; $\delta \subseteq Q \times \Sigma \times Q$ is the transition relation; $q_0 \in Q$ is the initial state; and $F \subseteq Q$ is the set of accepting states. A *run* of the Büchi automaton $\mathcal{A}$ 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 $\mathcal{A}$ if there is a run on σ with infinitely many $i \in \mathbb{N}$ such that $q_i \in F$. The language of $\mathcal{A}$, $\mathcal{L}(\mathcal{A})$, is the set of words accepted by $\mathcal{A}$.

2.1 Temporal Stream Logic Modulo Theories TSL(T)

Temporal Stream Logic (TSL) [35] extends Linear Temporal Logic (LTL) [50] by replacing Boolean atomic propositions with predicates over memory cells and inputs, and with *update terms* that specify how the value of a cell should change.

We present the formal definition of TSL modulo theories – TSL(T), based on the definition of [33], which extends the definition [35]. The definition we present is due to [46] and it slightly differs from the definition of [33]; The satisfaction of an update term is not defined by syntactic comparison, but relative to the current and previous values of cells and inputs. This definition suites the setting of model checking, where a concrete model is given.

TSL(T) is defined based on a set of *values* $\mathbb{V}$ with $\mathit{true}, \mathit{false} \in \mathbb{V}$, 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. A *theory* is a tuple $(\mathbb{F}, \varepsilon)$, where $\mathbb{F}$ is a set of function symbols; $\mathbb{F}_n$ is the set of functions of arity n; and $\varepsilon : \left(\bigcup_{n\in\mathbb{N}} \mathbb{F}_n \times \mathbb{V}^n\right) \to \mathbb{V}$ is the interpretation function, evaluating a function with arity n. For our purposes, we assume that every theory $(\mathcal{T}_\mathcal{F}, \varepsilon)$ contains at least $\{=, \vee, \neg\}$ with their usual interpretations.

A *function term* τ_F is defined by the grammar

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

where $c \in \mathbb{C}, i \in \mathbb{I}, f \in \mathbb{F}$, and the number of elements in f matches its arity. An *assignment* $a : (\mathbb{I} \cup \mathbb{C}) \to \mathbb{V}$ is a function assigning values to inputs and cells. We denote the set of all assignments by A. Given a concrete assignment, we can compute the value of a function term.

The *evaluation function* $\eta : \mathcal{T}_{\mathcal{F}} \times \mathsf{A} \to \mathbb{V}$ is defined as

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

A *predicate term* τ_P is a function term only evaluating to *true* or *false*. We denote the set of all predicate terms by $\mathcal{T}_P$.

For $c \in \mathbb{C}$ and $\tau_F \in \mathcal{T}_{\mathcal{F}}$, $[\![c \leftarrowtail \tau_F]\!]$ is called an *update term*. Intuitively, the update term $[\![c \leftarrowtail \tau_F]\!]$ states that c should be updated to the value of 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 $\mathcal{T}_U$. TSL formulas are constructed as follows, for $c \in \mathbb{C}, \tau_P \in \mathcal{T}_P, \tau_F \in \mathcal{T}_{\mathcal{F}}$.

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

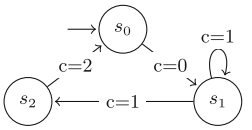
The usual operators $\vee, \Diamond$ ("eventually"), and $\Box$ ("globally") can be derived using the equations $\varphi \vee \psi = \neg(\neg\varphi \wedge \neg\psi)$, $\Diamond\varphi = \mathit{true}\ \mathcal{U}\varphi$ and $\Box\varphi = \neg\Diamond\neg\varphi$.

Assume a fixed initial variable assignment ζ_{-1} (e.g., setting all values to zero). The satisfaction of a TSL(T) formula with respect to a *computation* $\zeta \in \mathsf{A}^\omega$ and a time point t is defined as follows, where we define $\zeta \vDash \varphi$ as $0, \zeta \vDash \varphi$.

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

3 HyperTSL Modulo Theories

In this section, we introduce HyperTSL(T), HyperTSL with theories, which enables us to interpret predicates and functions depending on the program at hand. In [22], two versions of HyperTSL are introduced: HyperTSL and HyperTSL$_{rel}$. The former is a conservative extension of TSL to hyperproperties, meaning that predicates only reason about a single trace. In HyperTSL$_{rel}$, predicates may relate multiple traces, which opens the door to expressing properties like noninterference in infinite domains. Here, we build on HyperTSL$_{rel}$, allowing, in addition, update terms ranging over multiple traces. Furthermore, we extend the originally uninterpreted functions and predicates with an interpretation over theories. We denote this logic by HyperTSL(T).



$\pi := (c = 0)\ (c = 1)^\omega,\ \pi' := ((c = 0)\ (c = 1)\ (c = 2))^\omega$
$\hat{a}_1 : \{c_\pi \mapsto 0, c_{\pi'} \mapsto 0\}, \hat{a}_2 : \{c_\pi \mapsto 1, c_{\pi'} \mapsto 1\}$
$\hat{a}_3 : \{c_\pi \mapsto 1, c_{\pi'} \mapsto 2\}, \hat{a}_4 : \{c_\pi \mapsto 1, c_{\pi'} \mapsto 0\}$

Fig. 1. Left: A program automaton. Right: two traces π and π' of the program automaton. We interpret each trace as a computation. When executing both traces simultaneously, every time point has a corresponding hyper-assignment that assigns values to c_π and $c_{\pi'}$. Those for the first four time steps are shown on the right. Together, they define the hyper-computation $\hat{\zeta} := \hat{a}_1(\hat{a}_2\ \hat{a}_3\ \hat{a}_4)^\omega$, matching π and π'.

The syntax of HyperTSL(T) is that of TSL(T), with the addition that 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 the trace 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 1. *A* hyper-function term $\hat{\tau_F}$ *is defined by the grammar*

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

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

Analogously, we define *hyper-predicate terms* $\hat{\tau_P} \in \hat{\mathcal{T}_P}$ as hyper-function terms evaluating to *true* or *false*; *hyper-assignments* $\hat{\mathsf{A}} = (\mathbb{I} \cup \mathbb{C}) \times \Pi \to \mathbb{V}$ as functions mapping cells and inputs of each trace to their current values; *hyper-computations* $\hat{\zeta} \in \hat{\mathsf{A}}^\omega$ as hyper-assignment sequences. See Fig. 1 for an example.

Definition 2. *Let* $c_\pi \in \mathbb{C} \times \Pi, \hat{\tau_P} \in \hat{\mathcal{T}_P}, \hat{\tau_F} \in \hat{\mathcal{T}_F}$*. A* HyperTSL(T) formula *is defined by the following grammar:*

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

To define the semantics of HyperTSL(T), we need the ability to extend a hyper-computation to new trace variables, one for each path quantifier. Let $\hat{\zeta} \in \hat{\mathsf{A}}^\omega$ be a hyper-computation, and let $\pi, \pi' \in \Pi, \zeta \in \mathsf{A}^\omega$ and $x \in (\mathbb{I} \cup \mathbb{C})$. We define the extension of $\hat{\zeta}$ by π using the computation ζ as $\hat{\zeta}[\pi, \zeta]\ (x_{\pi'}) = \hat{\zeta}(x_{\pi'})$ for $\pi' \neq \pi$, and $\hat{\zeta}[\pi, \zeta]\ (x_\pi) = \zeta(x_\pi)$ for π.

Definition 3. *The* satisfaction of a HyperTSL(T)-Formula *w.r.t. a hyper- computation* $\hat{\zeta} \in \hat{\mathsf{A}}^\omega$*, a set of computations* Z *and a time point* t *is defined by*

$$t, Z, \hat{\zeta} \vDash \forall \pi.\ \varphi \quad \Leftrightarrow \forall \zeta \in Z.\ t,\ Z,\ \hat{\zeta}[\pi, \zeta] \vDash \varphi$$
$$t, Z, \hat{\zeta} \vDash \exists \pi.\ \varphi \quad \Leftrightarrow \exists \zeta \in Z.\ t,\ Z,\ \hat{\zeta}[\pi, \zeta] \vDash \varphi$$

The cases that do not involve path quantification are analogous to those of TSL(T) as defined in Sect. 2.1. We define $Z \vDash \varphi$ *as* $0, Z, \varnothing^\omega \vDash \varphi$.

4 Büchi Product Programs and TSL Model Checking

We now describe how we model the system and specification as Büchi automata, adapting the automata of [27] to the setting of TSL. Then, we introduce our model checking algorithm for TSL(T). In Sect. 5.2 we build on this algorithm to propose an algorithm for HyperTSL(T) model checking.

We use a symbolic representation of the system (see, for example, [41]), where transitions are labeled with program statements, and all states are accepting.

Definition 4. *Let* $c \in \mathbb{C}, \tau_P \in \mathcal{T}_P$ *and* $\tau_F \in \mathcal{T}_\mathcal{F}$. *We define the set of* (basic) program statements *as*

$$s_0 ::= \mathit{assert}(\tau_P) \mid c := \tau_F \mid c := *$$
$$s ::= s_0 \mid s; s$$

We call statements of the type s_0 basic program statements*, denoted by* Stmt_0*; statements of type* s *are denoted by* Stmt. *The assignment* $c := *$ *means that any value could be assigned to* c.

A *program automaton* $\mathcal{P}$ is a Büchi automaton with $\Sigma = \mathit{Stmt}$, that is, $\mathcal{P} = (\mathit{Stmt}, Q, q_0, \delta, F)$ and $\delta \subseteq Q \times \mathit{Stmt} \times Q$. When modeling the system we only need basic statements, thus we have $\mathit{Stmt} = \mathit{Stmt}_0$; and $F = Q$ as all states are accepting. See Fig. 3 for an illustration.

Using a program automaton, one can model `if` statements, `while` loops, and non-deterministic choices. However, not every trace of the program automaton corresponds to a program execution. For example, the trace $(n := \mathit{input}_1); \mathit{assert}(n > 0); \mathit{assert}(n < 0); \ \mathit{assert}(\mathit{true})^\omega$ does not – the second assertion will always fail. Such a trace is called *infeasible*. We call a trace *feasible* if it corresponds to a program execution where all the assertions may succeed. We now define this formally.

Definition 5. *A computation* ζ matches *a trace* $\sigma \in \mathit{Stmt}_0^\omega$ *at time point* t*, denoted by* $\zeta \lhd_t \sigma$*, if the following holds:*

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

where ζ_{-1} *is the initial assignment. A computation* ζ *matches a trace* $\sigma \in \mathit{Stmt}_0^\omega$*, denoted by* $\zeta \lhd \sigma$*, if* $\forall t \in \mathbb{N}.\ \zeta \lhd_t \sigma$.

Definition 6. *A program automaton $\mathcal{P}$ over $Stmt_0$ satisfies a TSL(T)-formula φ, if for all traces σ of P we have $\forall \zeta \in \mathsf{A}^\omega.\ \zeta \triangleleft \sigma \Rightarrow \zeta \vDash \varphi$.*

We now present an algorithm to check whether a program automaton $\mathcal{P}$ satisfies a TSL(T) formula. It is an adaption of the automaton-based LTL software model checking approach by [27], where the basic idea is to first translate the negated specification φ into an automaton $\mathcal{A}_{\neg\varphi}$, and then combine $\mathcal{A}_{\neg\varphi}$ and $\mathcal{P}$ to a new automaton, namely the *Büchi program product*. The program satisfies the specification iff the Büchi program product accepts no feasible trace.

In [27], the Büchi program product is constructed similarly to the standard product automata construction. To ensure that the result is again a program automaton, the transitions are not labeled with pairs $(s, l) \in Stmt_0 \times 2^{AP}$, but 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 violates the specification. In the following, we discuss how we adapt the construction of the Büchi program product for TSL(T) such that this property – a feasible trace corresponds to a counterexample – remains true for TSL(T).

Let φ be a TSL(T) specification. For the construction of $\mathcal{A}_{\neg\varphi}$, we treat all update and predicate terms as atomic propositions, resulting in an LTL formula $\neg\varphi_{LTL}$, which is translated to a Büchi automaton.[2] For our version of the Büchi program product, we need to merge a transition label s from $\mathcal{P}$ with a transition label l from $\mathcal{A}_{\neg\varphi_{LTL}}$ into a single program statement such that the assertion of the combined statement succeeds iff l holds for the statement s. Note that l is a set of update and predicate terms. For the update terms $[\![c \leftarrowtail \tau_F]\!]$ we cannot just use an assertion to check if they are true, as we need to 'save' the value of τ_F before the statement s is executed.

Our setting differs from [27] also in the fact that their program statements do not reason over input streams. We model the behavior of input streams by using fresh memory cells that are assigned a new value at every time step. In the following, we define a function *combine* that combines a program statement s and a transition label l to a new program statement as described above.

Definition 7. *Let $\upsilon = \{[\![c_1 \leftarrowtail \tau_{F1}]\!], \ldots, [\![c_n \leftarrowtail \tau_{Fn}]\!]\}$ 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 $\mathcal{A}_{\neg\varphi}$. Let $(tmp_j)_{j\in\mathbb{N}}$ be a family of fresh cells. Let $\mathbb{I} = \{i_1, \ldots i_m\}$. We define the function combine : $Stmt \times \mathcal{P}(\mathcal{T}_P \cup \mathcal{T}_U) \to Stmt$ as follows. The result of combine(s, l) is composed of the program statements in $save\_values_l, s, new\_inputs, check\_preds_l$ and $check\_updates_l$. Then we have:*

$$save\_values := tmp_1 := \tau_{F1};\ \ldots; tmp_n := \tau_{Fn}$$

$$new\_inputs := i_1 := *;\ \ldots\ ; i_m := *$$

$$check\_preds_l := assert\left(\bigwedge_{\tau_P \in l} \tau_P \wedge \bigwedge_{\tau_P \in \rho \setminus l} \neg\tau_P\right)$$

[2] For the translation of LTL formulas to Büchi automata, see, for example, [4,48,51].

$$check\_updates_l := assert\left(\bigwedge_{[\![c_j \leftarrowtail \tau_{Fj}]\!] \in v} \begin{cases} c_j = tmp_j & \text{if } [\![c_j \leftarrowtail \tau_{Fj}]\!] \in l \\ c_j \neq tmp_j & \text{else} \end{cases}\right)$$

$$combine(s, l) := save\_values;\ s;\ new\_inputs;\ check\_preds_l;\ check\_updates_l$$

We can extend this definition to combining traces instead of single transition labels. This leads to a function $combine : Stmt^\omega \times \mathcal{P}(\mathcal{T}_P \cup \mathcal{T}_U)^\omega \to Stmt^\omega$. 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 1. Let $\mathbb{I} = \{i\}$. Then the result of $combine(n := 42, \{[\![n \leftarrowtail n + 7]\!], n > 0\})$ is $tmp_0 := n + 7;\ n := 42;\ i := *;\ assert(n > 0);\ assert(n = tmp_0)$.

As *combine* leads to composed program statements, we now need to extend the definition of feasibility to all traces. To do so, we define a function *flatten* : $Stmt^\omega \to {Stmt_0}^\omega$ that 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.

Definition 8. *A trace* $\sigma \in Stmt^\omega$ matches *a computation* ζ*, denoted by* $\zeta \triangleleft \sigma$ *if* $\zeta \triangleleft flatten(\sigma)$*. A trace* σ *is* feasible *if there is a computation* ζ *such that* $\zeta \triangleleft \sigma$*.*

Definition 9 ***(Combined Product).*** *Let* $\mathcal{P} = (Stmt, Q, q_0, \delta, Q)$ *be a program automaton and* $\mathcal{A} = (\mathcal{P}(\mathcal{T}_P \cup \mathcal{T}_U), Q', q_0', \delta', F')$ *be a Büchi automaton (for example, the automaton* $\mathcal{A}_{\neg\varphi_{LTL}}$*). The combined product* $\mathcal{P} \otimes \mathcal{A}$ *is an automaton* $\mathcal{B} = (Stmt, Q \times Q', (q_0, q_0'), \delta_B, F_B)$*, where*

$$F_B = \{(q, q') \mid q \in Q \wedge q' \in F'\}$$

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

Theorem 1. *Let* $\mathcal{P}$ *be a program automaton over* $Stmt_0$*. Let* φ *be a TSL(T) formula. Then* $\mathcal{P}$ *satisfies* φ *if and only if* $\mathcal{P} \otimes \mathcal{A}_{\neg\varphi_{LTL}}$ *has no feasible trace.*

Proof (sketch). If $\zeta \triangleleft \sigma$ is a counterexample, we can construct a computation $\tilde{\zeta}$ that matches the corresponding combined trace in $\mathcal{P} \otimes \mathcal{A}_{\neg\varphi_{LTL}}$, and vice versa. See the full version [32] for the formal construction.

We can now apply Theorem 1 to solve the model checking problem by testing whether $\mathcal{P} \otimes \mathcal{A}_{\neg\varphi_{LTL}}$ does not accept any feasible trace, using the feasibility check in [27] as a black box. The algorithm of [27] is based on counterexample-guided abstraction refinement (CEGAR [18]). Accepted traces are checked for feasibility. First, finite prefixes of the trace are checked using an SMT-solver. If they are feasible, a ranking function synthesizer is used to check whether the whole trace eventually terminates. If the trace is feasible, it serves as a counterexample. If not, the automaton is refined such that it now does not include the spurious counterexample trace anymore, and the process is repeated. For more details, we refer to [27]. The limitations of SMT-solvers and ranking function synthesizers also limit the functions and predicates that can be used in both the program and in the TSL(T) formula.

5 HyperTSL(T) Model Checking

We now turn to the model checking problem of HyperTSL(T). We start with alternation-free formulas and continue with $\forall^*\exists^*$ formulas.

5.1 Alternation-Free HyperTSL(T)

In this section, we apply the technique of self-composition to extend the algorithm of Sect. 4 to alternation-free HyperTSL(T). First, we define what it means for a program automaton to satisfy a HyperTSL(T) formula.

Definition 10. *Let $\mathcal{P}$ be a program automaton over $Stmt_0$, let φ be a HyperTSL(T) formula and let $Z = \{\zeta \in \mathsf{A}^\omega \mid \exists \sigma.\ \zeta \triangleleft \sigma$ and σ is a trace of $\mathcal{P}\}$. We say that $\mathcal{P}$* satisfies *φ if $Z \vDash \varphi$.*

Definition 11. *Let $\mathcal{P} = (Stmt, Q, q_0, \delta, Q)$ be a program automaton. The n-fold* self-composition *of $\mathcal{P}$ is $\mathcal{P}^n = (Stmt', Q^n, q_0^n, \delta^n, Q^n)$, where $Stmt'$ are program statements over the set of inputs $\mathbb{I} \times \Pi$ and the set of cells $\mathbb{C} \times \Pi$ and where $Q^n = Q \times \cdots \times Q$, $q_0^n = (q_0, \ldots, q_0)$ and*

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

where $(s)_\pi$ renames every cell c used in s to c_π and every input i to i_π.

Theorem 2. *A program automaton $\mathcal{P}$ over $Stmt_0$ satisfies a universal HyperTSL(T) formula $\varphi = \forall \pi_1.\ \ldots \forall \pi_n.\ \psi$ iff $\mathcal{P}^n \otimes \mathcal{A}_{\neg \psi_{LTL}}$ has no feasible trace.*

Theorem 3. *A program automaton P over $Stmt_0$ satisfies an existential HyperTSL(T) formula $\varphi = \exists \pi_1.\ \ldots \exists \pi_n.\ \psi$ iff $\mathcal{P}^n \otimes \mathcal{A}_{\psi_{LTL}}$ has some feasible trace.*

The proofs of are analogous to the proof of Theorem 1, see the full version of this paper [32] for details.

5.2 $\forall^*\exists^*$ HyperTSL(T)

In this section, we present a sound but necessarily incomplete algorithm for finding counterexamples for $\forall^*\exists^*$ HyperTSL(T) formulas.[3] Such an algorithm can also provide witnesses $\exists^*\forall^*$ formulas. As HyperTSL(T) is built on top of HyperLTL, we combine ideas from finite-state HyperLTL model checking [37] with the algorithms of Sect. 4 and Sect. 5.1.

Let $\varphi = \forall^m \exists^n . \psi$. For HyperLTL model checking, [37] first constructs an automaton containing the system traces satisfying $\psi_\exists := \exists^n . \psi$, and then applies

[3] Note that the algorithms of Sect. 4 and Sect. 5.1 are also incomplete, due to the feasibility test. However, the incompleteness of the algorithm we provide in this section is inherent to the quantifier alternation of the formula.

complementation to extract counterexamples for the $\forall\exists$ specification. Consider the automaton $\mathcal{P}^n \otimes \mathcal{A}_{\psi_{LTL}}$ from Sect. 4, whose feasible traces correspond to the system traces satisfying $\psi_\exists$. If we would be able to remove all infeasible traces, we could apply the finite-state HyperLTL model checking construction. Unfortunately, removing all infeasibilities is impossible in general, as the result would be a finite-state system describing exactly an infinite-state system. Therefore, the main idea of this section is to remove parts of the infeasible traces from $\mathcal{P}^n \otimes \mathcal{A}_{\psi_{LTL}}$, constructing an over-approximation of the system traces satisfying $\psi_\exists$. A counterexample disproving φ is then a combination of system traces that is not contained in the over-approximation.

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 trade-off: if k is larger, more counterexamples can be identified, but the automaton construction gets exponentially larger.

The second technique removes *infeasible accepting cycles* from the automaton. It might not be possible to remove all of them, thus we bound the number of iterations. We present an example and then elaborate on these two methods.

Example 2. The trace t_1 below is 3-infeasible, because regardless of the value of n prior to the second time step, the assertion in the fourth time step will fail.

$$t_1 = (n--;\ assert(n >= 0))\ (n := 1;\ assert(n >= 0))\ (n--;\ assert(n >= 0))^\omega$$

In contrast, the trace $t_2 = (n := *)\ (n--;\ assert(n >= 0))^\omega$ is not k-infeasible for any k, because the value of n can always be large enough to pass the first k assertions. 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.

Removing k-Infeasibility. To remove k-infeasibility from an automaton, we construct a new program automaton that 'remembers' the $k-1$ previous statements. The states of the new automaton correspond to paths of length k in the original automaton. We add a transition labeled with l between two states p and q if we can extend the trace represented by p with l such that the resulting trace is k-feasible. Formally, we get:

Definition 12. *Let $k \in \mathbb{N}$, $\sigma \in Stmt^\omega$. We say that σ is* k-infeasible *if there exists $j \in \mathbb{N}$ such that $\sigma_j\sigma_{j+1}\ldots\sigma_{j+k-1}; assert(true)^\omega$ is infeasible for all possible initial assignments ζ_{-1}. We then also call the subsequence $\sigma_j\sigma_{j+1}\ldots\sigma_{j+k-1}$ infeasible. If a trace is not k-infeasible, we call it k-feasible.*[4]

Definition 13. *Let $\mathcal{P} = (Stmt, Q, q_0, \delta, F)$ be a program automaton. Let $k \in \mathbb{N}$. We define $\mathcal{P}$ without k-infeasibility, as $\mathcal{P}_k = (Stmt, Q', q_0, \delta', F')$ where*

$$Q' := \{(q_1, s_1, q_2 \ldots, s_{k-1}, q_k) \mid (q_1, s_1, q_2) \in \delta \wedge \cdots \wedge (q_{k-1}, s_{k-1}, q_k) \in \delta\} \cup$$

[4] Whether a subsequence $\sigma_j\sigma_{j+1}\ldots\sigma_{j+k-1}$ is a witness of k-infeasibility can be checked using an SMT-solver, e.g., [14,15,17,26].

$$\{(q_0, s_0, q_1 \ldots, s_{k'-1}, q_{k'}) \mid k' < k-1 \wedge (q_0, s_0, q_1) \in \delta \wedge \ldots \wedge (q_{k'-1}, s_{k'-1}, q_{k'}) \in \delta\}$$

$$\delta' := \{((q_1, s_1, q_2 \ldots, s_{k-1}, q_k), s_k, (q_2, s_2, \ldots, q_k, s_k, q_{k+1})) \in Q' \times \mathit{Stmt} \times Q' \mid s_1 \ldots s_k \text{ feasible}\} \cup$$

$$\{((q_0, s_0, q_1 \ldots, s_{k'-1}, q_{k'}), s_{k'}, (q_0, s_0, \ldots, q_{k'}, s_{k'}, q_{k'+1})) \in Q' \times \mathit{Stmt} \times Q' \mid k' < k-1 \wedge s_0 \ldots s_{k'} \text{ feasible}\}$$

$$F' := \{(q_1, s_1, q_2 \ldots, s_{k-1}, q_k) \in Q' \mid q_k \in F\} \cup \{(q_0, s_0, q_1 \ldots, s_{k'-1}, q_{k'}) \in Q' \mid k' < k-1 \wedge q_{k'} \in F\}$$

Theorem 4. *$\mathcal{P}_k$ accepts exactly the k-feasible traces of $\mathcal{P}$.*

The proof follows directly from the construction above. For more details, see full version [32].

Removing Infeasible Accepting Cycles. For removing infeasible accepting cycles, we first enumerate all simple cycles of the automaton (using, e.g., [44]), adding also cycles induced by self-loops. For each cycle ϱ that contains at least one accepting state, we test its feasibility: first, using an SMT-solver to test if ϱ is locally infeasible; then, using a ranking function synthesizer (e.g., [8,24,40]) to test if ϱ^ω is infeasible. If we successfully prove infeasibility, we refine the model, using the methods from [41,42]. This refinement is formalized in the following.

Definition 14. *Let $\mathcal{P} = (\mathit{Stmt}, Q, q_0, \delta, F)$ be a program automaton. Let $\varrho = (q_1, s_1, q_2)(q_2, s_2, q_3) \ldots (q_n, s_n, q_1)$ be a sequence of transitions of $\mathcal{P}$. We say that ϱ is an* infeasible accepting cycle *if there is a $1 \leq j \leq n$ with $q_j \in F$ and $(s_1 s_2 \ldots s_{n-1})^\omega$ is infeasible for all possible initial assignments ζ_{-1}.*

Definition 15. *Let $\mathcal{P}$ be a program automaton and $C \subseteq (Q \times \mathit{Stmt} \times Q)^\omega$ be a set of infeasible accepting cycles of $\mathcal{P}$. Furthermore, let*

$$\varrho = (q_1, s_1, q_2)(q_2, s_2, q_3) \ldots (q_{n-1}, s_{n-1}, q_n) \in C.$$

The automaton $\mathcal{A}_\varrho$ for ϱ is $\mathcal{A}_\varrho = (\mathit{Stmt}, Q = \{q_0, q_1, \ldots q_n\}, q_0, \delta, Q \backslash \{q_0\})$ where

$$\delta = \{(q_0, s, q_0) \mid s \in \mathit{Stmt}\} \cup \{(q_j, s_j, q_{j+1}) \mid 1 \leq j < n\} \cup \{(q_0, s_1, q_2), (q_n, s_n, q_1)\}.$$

Then, $\mathcal{A}_\varrho$ accepts exactly the traces that end with ϱ^ω, without any restriction on the prefix. See Fig. 2 for an example. To exclude the traces of $\mathcal{A}_\varrho$ from $\mathcal{P}$, we define $\mathcal{P}_C := \mathcal{P} \backslash \left(\bigcup_{\varrho \in C} \mathcal{A}_\varrho\right)$.[5] This construction can be repeated to exclude infeasible accepted cycles that are newly created in $\mathcal{P}_C$. We denote the result of iterating this process k' times by $\mathcal{P}_{C(k')}$.

Finding Counterexamples for $\forall^*\exists^*$ HyperTSL(T)-Formulas. Consider now a HyperTSL(T) formula $\varphi = \forall^{1 \cdots m} \exists^{m+1 \cdots n}.\psi$ and a program automaton $\mathcal{P}$.

[5] For two automata $\mathcal{A}_1, \mathcal{A}_2$ we use $\mathcal{A}_1 \backslash \mathcal{A}_2$ to denote the intersection of $\mathcal{A}_1$ with the complement of $\mathcal{A}_2$, resulting in the language $\mathcal{L}(\mathcal{A}_1) \setminus \mathcal{L}(\mathcal{A}_2)$.

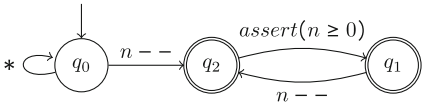


Fig. 2. Automaton $\mathcal{A}_\varrho$ for the infeasible cycle $\varrho = (q_1,\ n--,\ q_2)(q_2,\ assert(n > 0),\ q_1)$. Label $*$ denotes an edge for every (relevant) statement.

For finding a counterexample, we first construct the combined product $\mathcal{P}^n \otimes \mathcal{A}_\psi$. Each feasible accepted trace of $\mathcal{P}^n \otimes \mathcal{A}_\psi$ corresponds to a combination of n feasible program traces that satisfy ψ. Next, we eliminate k-infeasibility and remove k'-times infeasible accepting cycles from the combined product, resulting in the automaton $(\mathcal{P}^n \otimes \mathcal{A}_\psi)_{k,C(k')}$. Using this modified combined product, we obtain an over-approximation of the program execution combinations satisfying the existential part of the specification. Each trace of the combined product is a combination of n program executions and a predicate/update term sequence. We then project the m universally quantified program executions from a feasible trace, obtaining a tuple of m program executions that satisfy the existential part of the formula. Applying this projection to all traces of $(\mathcal{P}^n \otimes \mathcal{A}_\psi)_{k,C(k')}$ leads to an over-approximation of the program executions satisfying the existential part of the specification. Formally:

Definition 16. *Let $\mathcal{P}$ be a program automaton, let $m \leq n \in \mathbb{N}$, and let $\mathcal{A}_\psi$ be the automaton for the formula ψ. Let $(\mathcal{P}^n \otimes \mathcal{A})_{k,C(k')} = (Stmt, Q, q_0, \delta, F)$. We define the* projected automaton *$(\mathcal{P}^m \otimes \mathcal{A})^\forall_{k,C(k')} = (Stmt, Q, q_0, \delta^\forall, F)$ where $\delta^\forall = \{(q, (s_1; \ldots;\ s_m), q') \mid \exists s_{m+1}, \ldots s_n, l.\ (q, combine(s_1; \ldots;\ s_n, l), q') \in \delta\}$.*

The notation $s_1; s_2$ refers to a sequence of statements, as given in Definition 4. For more details on the universal projection we refer the reader to [36].

Now, it only remains to check whether the over-approximation contains all tuples of m feasible program executions. If not, a counterexample is found. This boils down to testing if $\mathcal{P}^m \backslash (\mathcal{P}^n \otimes \mathcal{A}_\psi)^\forall_{k,C(k')}$ has some feasible trace. Theorem 5 states the soundness of our algorithm. For the proof, see full version [32].

Theorem 5. *Let $\varphi = \forall^{1\cdots m} \exists^{m+1\cdots n}.\psi$ be a HyperTSL(T) formula. If the automaton $\mathcal{P}^m \backslash (\mathcal{P}^n \otimes \mathcal{A}_\psi)^\forall_{k,C(k')}$ has a feasible trace, then $\mathcal{P}$ does not satisfy φ.*

6 Demonstration of the Algorithm

In this section, we apply the algorithm of Sect. 5.2 to two simple examples, demonstrating that removing some infeasibilities can already be sufficient for identifying counterexamples.

Generalized Noninterference Recall the formula $\varphi_{gni} = \forall\pi.\ \exists\pi'.\ \square(i_{\pi'} = \lambda \wedge c_\pi = c_{\pi'})$ introduced in Sect. 1, specifying generalized noninterference. We model-check φ_{gni} on the program automaton $\mathcal{P}$ of Fig. 3 (left), setting $\lambda = 0$. The program $\mathcal{P}$ violates φ_{gni} since for the trace $(assert(i < 0)\ c := 0)^\omega$ there is no other trace where on which c is equal, but $i = 0$. The automaton for $\psi = \square(i_{\pi'} = 0 \wedge c_\pi = c_{\pi'})$ consists of a single accepting state with the self-loop labeled with $\tau_P = (i_{\pi'} = 0 \wedge c_\pi = c_{\pi'})$. For this example, it suffices to choose $k = 1$.

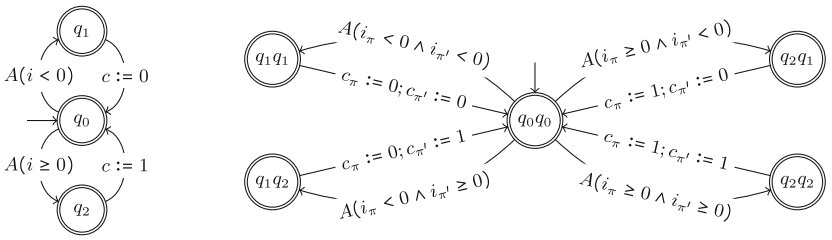


Fig. 3. Left: The program automaton $\mathcal{P}$ used in the first example. Right: The program automaton $\mathcal{P}^2$. For brevity, we use A for $assert$ and join consecutive assertions.

To detect 1-inconsistencies we construct $\mathcal{P}^2$ (Fig 3, right). Then, $(\mathcal{P}^2 \otimes \mathcal{A}_\psi)_k$ is the combined product with all 1-inconsistent transitions removed (see Fig. 5 for the combined product).

The automaton $(\mathcal{P}^2 \otimes \mathcal{A}_\psi)_k^\forall$ is shown in Fig. 4. It does not contain the trace $\sigma = assert(i < 0)\ (c := 0)^\omega$ which is a feasible trace of $\mathcal{P}$. Therefore, σ is a feasible trace accepted by $\mathcal{P} \backslash (\mathcal{P}^2 \otimes \mathcal{A}_\psi)_k^\forall$ and is a counterexample proving that $\mathcal{P}$ does not satisfy generalized noninterference – there is no feasible trace that agrees on the value of the cell c but has always $i = 0$.

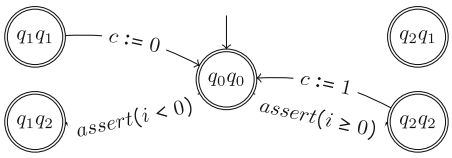


Fig. 4. program automaton $(\mathcal{P}^2 \otimes \mathcal{A}_\psi)_k^\forall$

The Need of Removing Cycles. We now present an example in which removing k-infeasibility is not sufficient, but removing infeasible accepting cycles leads to a counterexample. Consider the specification $\varphi = \forall\pi\exists\pi'.\square(p_\pi \neq p_{\pi'} \wedge n_\pi < n_{\pi'})$ and the program automaton $\mathcal{P}_{cy}$ of Fig. 6. The formula φ states that for every trace π, there is another trace π' which differs from π on p, but in which n is always greater. The trace $\pi = (n := *); (p := *); assert(p = 0); (n - -)^\omega$ is a counterexample for φ in $\mathcal{P}_{cy}$ as any trace π' which differs on p will decrease its n by 2 in every time step, and thus $n_{\pi'}$ will eventually drop below n_π.

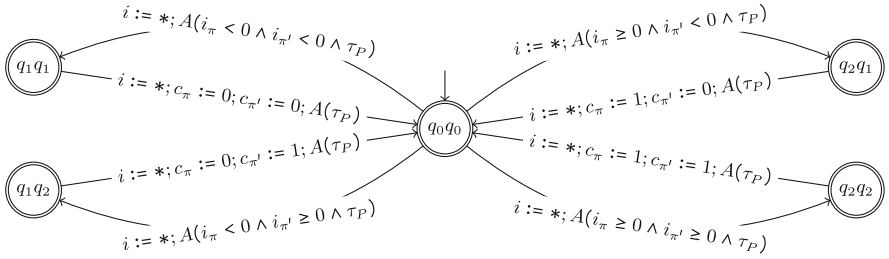


Fig. 5. The combined product $(\mathcal{P}^2 \otimes \mathcal{A}_\psi)$

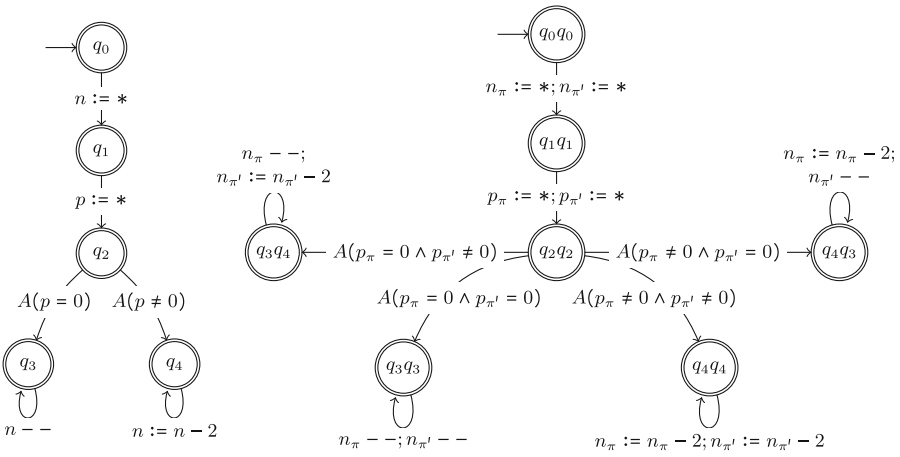


Fig. 6. Left: The program automaton $\mathcal{P}_{cy}$, Right: The program automaton $\mathcal{P}^2_{cy}$.

The automaton $\mathcal{P}^2_{cy}$ is shown in Fig. 6. In the combined product, the structure of the automaton stays the same, and $assert(p_\pi \neq p_{\pi'} \wedge n_\pi < n'_\pi)$ is added to every state. Removing local k-infeasibilities is not sufficient here; assume $k = 1$. The only 1-infeasible transition is the transition from q_2q_2 to q_3q_3, and this does not eliminate the counterexample π. Greater k's do not work as well, as the remaining traces of the combined product are not k infeasible for any k.

However, the self-loop at q_3q_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 $k' = 1$ removing all traces ending with this cycle. Next, we project the automaton to the universal part. The trace π is not accepted by the automaton $(\mathcal{P}^2 \otimes \mathcal{A}_\psi)^\forall_{1,C(1)}$. But since π is in $\mathcal{P}$ and feasible, it is identified as a counterexample.

7 Conclusions

We have extended HyperTSL with theories, resulting in HyperTSL(T), and provided the first infinite-state model checking algorithms for both TSL(T) and HyperTSL(T). As this is the first work to study (Hyper)TSL model checking, these are also the first algorithms for *finite-state* model checking for (Hyper)TSL. For TSL(T), we have adapted known software model checking algorithm for LTL to the setting of TSL(T). We then used the technique of self-composition to generalize this algorithm to the alternation-free fragment of HyperTSL(T).

We have furthermore described a sound but necessarily incomplete algorithm for finding counterexamples for $\forall^*\exists^*$-HyperTSL(T) formulas (and witnesses proving $\exists^*\forall^*$ formulas). Our algorithm makes it possible to find program executions violating properties like generalized noninterference, which is only expressible by using a combination of universal and existential quantifiers.

Finding model checking algorithms for other fragments of HyperTSL(T), and implementing our approach, remains as future work.

References

1. Aguirre, A., Barthe, G., Gaboardi, M., Garg, D., Strub, P.-Y.: A relational logic for higher-order programs. Proc. ACM Program. Lang. **1**(ICFP):21:1–21:29 (2017)
2. Alpern, B., Schneider, F.B.: Defining liveness. Inf. Process. Lett. **21**, 181–185 (1985)
3. Arora, S., Hansen, R.R., Larsen, K.G., Legay, A., Poulsen, D.B.: Statistical model checking for probabilistic hyperproperties of real-valued signals. In: Legunsen, O., Rosu, G. (eds.) SPIN 2022. LNCS, vol. 13255, pp. 61–78. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-15077-7_4
4. Babiak, T., Křetínský, M., Řehák, V., Strejček, J.: LTL to Büchi automata translation: fast and more deterministic. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 95–109. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28756-5_8
5. Barthe, G., Crespo, J.M., Kunz, C.: Beyond 2-safety: asymmetric product programs for relational program verification. In: Artemov, S., Nerode, A. (eds.) LFCS 2013. LNCS, vol. 7734, pp. 29–43. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-35722-0_3
6. Barthe, G., D'Argenio, P.R., Rezk, T.: Secure information flow by self-composition. Math. Struct. Comput. Sci. **21**(6), 1207–1252 (2011)
7. Baumeister, J., Coenen, N., Bonakdarpour, B., Finkbeiner, B., Sánchez, C.: A temporal logic for asynchronous hyperproperties. In: Silva, A., Leino, K.R.M. (eds.) CAV 2021, Part I. LNCS, vol. 12759, pp. 694–717. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-81685-8_33
8. Ben-Amram, A.M., Genaim, S.: On the linear ranking problem for integer linear-constraint loops. In: Giacobazzi, R., Cousot, R. (eds.) The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2013, Rome, Italy, 23–25 January 2013, pp. 51–62. ACM (2013)
9. Benton, N.: Simple relational correctness proofs for static analyses and program transformations. In: Jones, N.D., Leroy, X. (eds.) Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, 14–16 January 2004, pp. 14–25. ACM (2004)
10. Beutner, R., Finkbeiner, B.: Software verification of hyperproperties beyond k-safety. In: Shoham, S., Vizel, Y. (eds.) CAV 2022, Part I. LNCS, vol. 13371, pp. 341–362. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-13185-1_17
11. Bonakdarpour, B., Sanchez, C., Schneider, G.: Monitoring hyperproperties by combining static analysis and runtime verification. In: Margaria, T., Steffen, B. (eds.) ISoLA 2018, Part II. LNCS, vol. 11245, pp. 8–27. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-03421-4_2
12. Bozzelli, L., Peron, A., Sánchez, C.: Expressiveness and decidability of temporal logics for asynchronous hyperproperties. In: Klin, B., Lasota, S., Muscholl, A. (eds.) 33rd International Conference on Concurrency Theory, CONCUR 2022, 12–16 September 2022, Warsaw, Poland. LIPIcs, vol. 243, pp. 27:1–27:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2022)
13. Bradley, A.R.: SAT-based model checking without unrolling. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 70–87. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-18275-4_7

14. Bruttomesso, R., Pek, E., Sharygina, N., Tsitovich, A.: The OpenSMT solver. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 150–153. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-12002-2_12
15. Christ, J., Hoenicke, J., Nutz, A.: SMTInterpol: an interpolating SMT solver. In: Donaldson, A., Parker, D. (eds.) SPIN 2012. LNCS, vol. 7385, pp. 248–254. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31759-0_19
16. Cimatti, A., Griggio, A.: Software model checking via IC3. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 277–293. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31424-7_23
17. Cimatti, A., Griggio, A., Schaafsma, B.J., Sebastiani, R.: The MathSAT5 SMT solver. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 93–107. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36742-7_7
18. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 154–169. Springer, Heidelberg (2000). https://doi.org/10.1007/10722167_15
19. Clarkson, M.R., Finkbeiner, B., Koleini, M., Micinski, K.K., Rabe, M.N., Sánchez, C.: Temporal logics for hyperproperties. In: Abadi, M., Kremer, S. (eds.) POST 2014. LNCS, vol. 8414, pp. 265–284. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54792-8_15
20. Clarkson, M.R., Schneider, F.B.: Hyperproperties. J. Comput. Secur. **18**(6), 1157–1210 (2010)
21. Coenen, N., Finkbeiner, B., Hahn, C., Hofmann, J.: The hierarchy of hyperlogics. In: 34th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2019, Vancouver, BC, Canada, 24–27 June 2019, pp. 1–13. IEEE (2019)
22. Coenen, N., Finkbeiner, B., Hofmann, J., Tillman, J.: Smart contract synthesis modulo hyperproperties. In: 36th IEEE Computer Security Foundations Symposium (CSF 2023) (2023, to appear)
23. Coenen, N., Finkbeiner, B., Sánchez, C., Tentrup, L.: Verifying hyperliveness. In: Dillig, I., Tasiran, S. (eds.) CAV 2019, Part I. LNCS, vol. 11561, pp. 121–139. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25540-4_7
24. Colón, M.A., Sipma, H.B.: Practical methods for proving program termination. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 442–454. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45657-0_36
25. Daniel, J., Cimatti, A., Griggio, A., Tonetta, S., Mover, S.: Infinite-state liveness-to-safety via implicit abstraction and well-founded relations. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016, Part I. LNCS, vol. 9779, pp. 271–291. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41528-4_15
26. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
27. Dietsch, D., Heizmann, M., Langenfeld, V., Podelski, A.: Fairness modulo theory: a new approach to LTL software model checking. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015, Part I. LNCS, vol. 9206, pp. 49–66. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_4
28. Dimitrova, R., Finkbeiner, B., Torfah, H.: Probabilistic hyperproperties of Markov decision processes. In: Hung, D.V., Sokolsky, O. (eds.) ATVA 2020. LNCS, vol. 12302, pp. 484–500. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-59152-6_27

29. Dobe, O., Ábrahám, E., Bartocci, E., Bonakdarpour, B.: HyperProb: a model checker for probabilistic hyperproperties. In: Huisman, M., Păsăreanu, C., Zhan, N. (eds.) FM 2021. LNCS, vol. 13047, pp. 657–666. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-90870-6_35
30. Eilers, M., Müller, P., Hitz, S.: Modular product programs. ACM Trans. Program. Lang. Syst. **42**(1), 3:1–3:37 (2020)
31. Finkbeiner, B.: Model checking algorithms for hyperproperties (invited paper). In: Henglein, F., Shoham, S., Vizel, Y. (eds.) VMCAI 2021. LNCS, vol. 12597, pp. 3–16. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-67067-2_1
32. Finkbeiner, B., Frenkel, H., Hofmann, J., Lohse, J.: Automata-based software model checking of hyperproperties. CoRR, abs/2303.14796 (2023)
33. Finkbeiner, B., Heim, P., Passing, N.: Temporal stream logic modulo theories. In: FoSSaCS 2022. LNCS, vol. 13242, pp. 325–346. Springer, Cham (2022). https://doi.org/10.1007/978-3-030-99253-8_17
34. Finkbeiner, B., Hofmann, J., Kohn, F., Passing, N.: Reactive synthesis of smart contract control flows. CoRR, abs/2205.06039 (2022)
35. Finkbeiner, B., Klein, F., Piskac, R., Santolucito, M.: Temporal stream logic: synthesis beyond the bools. In: Dillig, I., Tasiran, S. (eds.) CAV 2019, Part I. LNCS, vol. 11561, pp. 609–629. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25540-4_35
36. Finkbeiner, B., Passing, N.: Synthesizing dominant strategies for liveness. In: Dawar, A., Guruswami, V. (eds.) 42nd IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2022, 18–20 December 2022, IIT Madras, Chennai, India, volume 250 of LIPIcs, pp. 37:1–37:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2022)
37. Finkbeiner, B., Rabe, M.N., Sánchez, C.: Algorithms for model checking HyperLTL and HyperCTL*. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015, Part I. LNCS, vol. 9206, pp. 30–48. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_3
38. Frenkel, H., Grumberg, O., Sheinvald, S.: An automata-theoretic approach to model-checking systems and specifications over infinite data domains. J. Autom. Reason. **63**(4), 1077–1101 (2019)
39. Geier, G., Heim, P., Klein, F., Finkbeiner, B.: Syntroids: synthesizing a game for FPGAs using temporal logic specifications. In: Barrett, C.W., Yang, J. (eds.) 2019 Formal Methods in Computer Aided Design, FMCAD 2019, San Jose, CA, USA, 22–25 October 2019, pp. 138–146. IEEE (2019)
40. Heizmann, M., Hoenicke, J., Leike, J., Podelski, A.: Linear ranking for linear lasso programs. In: Van Hung, D., Ogawa, M. (eds.) ATVA 2013. LNCS, vol. 8172, pp. 365–380. Springer, Cham (2013). https://doi.org/10.1007/978-3-319-02444-8_26
41. Heizmann, M., Hoenicke, J., Podelski, A.: Software model checking for people who love automata. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 36–52. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_2
42. Heizmann, M., Hoenicke, J., Podelski, A.: Termination analysis by learning terminating programs. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 797–813. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_53
43. Ho, H.-S., Zhou, R., Jones, T.M.: On verifying timed hyperproperties. In: Gamper, J., Pinchinat, S., Sciavicco, G. (eds.) 26th International Symposium on Temporal Representation and Reasoning, TIME 2019, October 16–19, 2019, Málaga, Spain, LIPIcs, vol. 147, pp. 20:1–20:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2019)

44. Johnson, D.B.: Finding all the elementary circuits of a directed graph. SIAM J. Comput. **4**(1), 77–84 (1975)
45. Lamport, L., Schneider, F.B.: Verifying hyperproperties with TLA. In: 34th IEEE Computer Security Foundations Symposium, CSF 2021, Dubrovnik, Croatia, 21–25 June 2021, pp. 1–16. IEEE (2021)
46. Maderbacher, B., Bloem, R.: Reactive synthesis modulo theories using abstraction refinement. CoRR, abs/2108.00090 (2021)
47. McCullough, D.: Noninterference and the composability of security properties. In: Proceedings of the 1988 IEEE Symposium on Security and Privacy, Oakland, California, USA, 18–21 April 1988, pp. 177–186. IEEE Computer Society (1988)
48. Mochizuki, S., Shimakawa, M., Hagihara, S., Yonezaki, N.: Fast translation from LTL to Büchi automata via non-transition-based automata. In: Merz, S., Pang, J. (eds.) ICFEM 2014. LNCS, vol. 8829, pp. 364–379. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11737-9_24
49. Nguyen, L.V., Kapinski, J., Jin, X., Deshmukh, J.V., Johnson, T.T.: Hyperproperties of real-valued signals. In: Talpin, J.-P., Derler, P., Schneider, K. (eds.) Proceedings of the 15th ACM-IEEE International Conference on Formal Methods and Models for System Design, MEMOCODE 2017, Vienna, Austria, 29 September –02 October 2017, pp. 104–113. ACM (2017)
50. Pnueli, A.: The temporal logic of programs. In: 18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October–1 November 1977, pp. 46–57. IEEE Computer Society (1977)
51. Tsay, Y.-K., Vardi, M.Y.: From linear temporal logics to Büchi automata: the early and simple principle. In: Olderog, E.-R., Steffen, B., Yi, W. (eds.) Model Checking, Synthesis, and Learning. LNCS, vol. 13030, pp. 8–40. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-91384-7_2
52. Unno, H., Terauchi, T., Koskinen, E.: Constraint-based relational verification. In: Silva, A., Leino, K.R.M. (eds.) CAV 2021, Part I. LNCS, vol. 12759, pp. 742–766. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-81685-8_35