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, United Kingdom
[3] Saarland University, Saarbrücken, Germany

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 [48] (LTL) has been extended with trace quantification, resulting in Hyper Linear Temporal Logic [19] (HyperLTL). There exist several model checking algorithms for HyperLTL [19, 22, 35], but they are designed for finite-state systems and are therefore 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.

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

** Research carried out while at CISPA Helmholtz Center for Information Security.

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* [45]. 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 $\varphi_{gni} = \forall\pi\exists\pi'.\Box(i_{\pi'} = \lambda \land c_\pi = c_{\pi'})$. 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) [21]. 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 [39], 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 $assert(n = 0)\ ; n--;\ (assert(n = 0))^\omega$ [4] 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 (Sec. 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 [26], 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 (Sec. 5.1) by applying *self-composition*, a technique commonly used for the verification of hyperproperties [5, 6, 29].

Next, in Sec. 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

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

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 Sec. 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 [26] 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 [48] and was originally designed for synthesis [33]. For synthesis, the logic has been successfully applied to synthesize the FPGA game 'Syntroids' [37], and to synthesize smart contracts [32]. To advance smart contract synthesis, TSL has been extended to HyperTSL in [21]. 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 [31], 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, 24, 26, 36]. Our work builds on the automata-based LTL software model checking algorithm from [26]. 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 [50]. 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 [43].

Beyond the scope of software model checking, the verification of hyperproperties has been studied for various system models and classes of hyperproperties. There exist model checking algorithms for ω-regular properties [30,35] and asynchronous hyperproperties [7,12] in finite-state Kripke structures, as well as timed systems [41], real-valued [47] and probabilistic hyperproperties [3, 27, 28] (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) [33] extends Linear Temporal Logic (LTL) [48] 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 [31], which extends the definition [33]. The definition we present is due to [44] and it slightly differs from the definition of [31]; 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 *true*, *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 $\Box\varphi = \neg\Diamond\neg\varphi$.

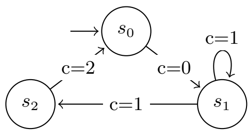
Assume a fixed initial variable assignment ζ_{-1} (e.g., setting all values to zero). The satisfaction of a TSL(T) formula with respect to a *computation* $\zeta \in \mathsf{A}^\omega$ and a time point t is defined as follows, where we define $\zeta \vDash \varphi$ as $0, \zeta \vDash \varphi$.

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

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 [21], 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 non-interference 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).

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



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

$\hat{a}_1 : \{c_{\pi} \mapsto 0, c_{\pi'} \mapsto 0\}, \hat{a}_2 : \{c_{\pi} \mapsto 1, c_{\pi'} \mapsto 1\}$

$\hat{a}_3 : \{c_{\pi} \mapsto 1, c_{\pi'} \mapsto 2\}, \hat{a}_4 : \{c_{\pi} \mapsto 1, c_{\pi'} \mapsto 0\}$

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'.

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* $\hat{\mathsf{A}} = (\mathbb{I} \cup \mathbb{C}) \times \Pi \rightarrow \mathbb{V}$ 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. 3 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:*

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

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*

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

The cases that do not involve path quantification are analogous to those of TSL(T) as defined in Sec. 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 [26] to the setting of TSL. Then, we introduce our model checking algorithm for TSL(T). In Sec 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, [39]), 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*

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

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 $\Sigma = Stmt$, that is, $\mathcal{P} = (Stmt, Q, q_0, \delta, F)$ and $\delta \subseteq Q \times Stmt \times Q$. When modeling the system we only need basic statements, thus we have $Stmt = Stmt_0$; 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 $(n := input_1); assert(n > 0); assert(n < 0);\ assert(true)^\omega$ 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 $\sigma \in Stmt_0^\omega$ at time point t, denoted by $\zeta \triangleleft_t \sigma$, if the following holds:*

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

where ζ_{-1} is the initial assignment. A computation ζ matches a trace $\sigma \in Stmt_0^\omega$, denoted by $\zeta \triangleleft \sigma$, if $\forall t \in \mathbb{N}.\ \zeta \triangleleft_t \sigma$.

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 [26], 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 [26], 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 $(s, l) \in \mathit{Stmt}_0 \times 2^{AP}$, but with the program statement $(s;\ \mathit{assert}(l))$. 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.[5] 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 [26] 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 $\upsilon = \{[\![c_1 \leftarrowtail \tau_{F1}]\!], \dots, [\![c_n \leftarrowtail \tau_{Fn}]\!]\}$ 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 $\mathcal{A}_{\neg\varphi}$. Let $(\mathit{tmp}_j)_{j\in\mathbb{N}}$ be a family of fresh cells. Let $\mathbb{I} = \{i_1, \dots i_m\}$. We define the function combine $: \mathit{Stmt} \times \mathcal{P}(\mathcal{T}_P \cup \mathcal{T}_U) \to \mathit{Stmt}$ as follows. The result of $\mathit{combine}(s, l)$ is composed of the program statements in $\mathit{save\_values}_l, s, \mathit{new\_inputs}, \mathit{check\_preds}_l$ and $\mathit{check\_updates}_l$. Then we have:*

$$\mathit{save\_values} := \mathit{tmp}_1 := \tau_{F1};\ \dots; \mathit{tmp}_n := \tau_{Fn}$$

$$\mathit{new\_inputs} := i_1 := *;\ \dots\ ; i_m := *$$

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

$$\mathit{check\_updates}_l := \mathit{assert}\left(\bigwedge_{[\![c_j \leftarrowtail \tau_{Fj}]\!] \in \upsilon} \begin{cases} c_j = \mathit{tmp}_j & \mathit{if}\ [\![c_j \leftarrowtail \tau_{Fj}]\!] \in l \\ c_j \neq \mathit{tmp}_j & \mathit{else} \end{cases} \right)$$

$$\mathit{combine}(s, l) := \mathit{save\_values};\ s;\ \mathit{new\_inputs};\ \mathit{check\_preds}_l;\ \mathit{check\_updates}_l$$

We can extend this definition to combining traces instead of single transition labels. This leads to a function $\mathit{combine} : \mathit{Stmt}^\omega \times \mathcal{P}(\mathcal{T}_P \cup \mathcal{T}_U)^\omega \to \mathit{Stmt}^\omega$. Note

[5] For the translation of LTL formulas to Büchi automata, see, for example, [4, 46, 49].

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. The formal construction is given in App. A.4.

We can now apply Thm. 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 [26] as a black box. The algorithm of [26] 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 [26]. 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 Sec. 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 $Z = \{\zeta \in \mathsf{A}^\omega \mid \exists \sigma.\ \zeta \lhd \sigma$ 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*

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

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 Thm. 1 and are provided in App. A.5.

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.[6] 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 [35] with the algorithms of Sec. 4 and Sec. 5.1.

Let $\varphi = \forall^m \exists^n.\psi$. For HyperLTL model checking, [35] first constructs an automaton containing the system traces satisfying $\psi_\exists := \exists^n.\psi$, and then applies complementation to extract counterexamples for the $\forall\exists$ specification. Consider the automaton $\mathcal{P}^n \otimes \mathcal{A}_{\psi_{LTL}}$ from Sec. 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

[6] Note that the algorithms of Sec. 4 and Sec. 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.

$\mathcal{P}^n \otimes \mathcal{A}_{\psi_{LTL}}$, 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.*[7]

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*

$$\begin{aligned}
Q' :=& \{(q_1, s_1, q_2 \ldots, s_{k-1}, q_k) \mid (q_1, s_1, q_2) \in \delta \land \cdots \land (q_{k-1}, s_{k-1}, q_k) \in \delta\} \cup \\
& \{(q_0, s_0, q_1 \ldots, s_{k'-1}, q_{k'}) \mid k' < k-1 \land (q_0, s_0, q_1) \in \delta \land \ldots \\
& \qquad \land (q_{k'-1}, s_{k'-1}, q_{k'}) \in \delta\} \\
\delta' :=& \{((q_1, s_1, q_2 \ldots, s_{k-1}, q_k), s_k, (q_2, s_2, \ldots, q_k, s_k, q_{k+1})) \in Q' \times Stmt \times Q' \\
& \mid s_1 \ldots s_k \ feasible\} \cup
\end{aligned}$$

[7] 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, 25].

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

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

The proof follows directly from the construction above, see App. A.3 for details.

Removing Infeasible Accepting Cycles For removing infeasible accepting cycles, we first enumerate all simple cycles of the automaton (using, e.g., [42]), 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., [8,23,38]) to test if ϱ^ω is infeasible. If we successfully prove infeasibility, we refine the model, using the methods from [39, 40]. This refinement is formalized in the following.

Definition 14. *Let $\mathcal{P} = (Stmt, Q, q_0, \delta, F)$ 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 $C \subseteq (Q \times Stmt \times Q)^\omega$ 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 $\mathcal{A}_\varrho = (Stmt, Q = \{q_0, q_1, \ldots q_n\}, q_0, \delta, Q \backslash \{q_0\})$ where

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

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 $\mathcal{P}_C := \mathcal{P} \backslash \left(\bigcup_{\varrho \in C} \mathcal{A}_\varrho\right)$.[8] 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 $\mathcal{P}$.

[8] 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 $\mathcal{L}(\mathcal{A}_1) \setminus \mathcal{L}(\mathcal{A}_2)$.

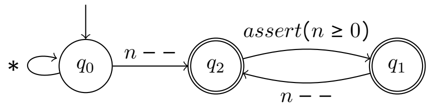


Fig. 2. Automaton $\mathcal{A}_\varrho$ for the infeasible cycle $\varrho = (q_1,\ n--,\ q_2)(q_2,\ assert(n > 0),\ q_1)$. Label $*$ denotes an edge for every (relevant) statement.

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 Def. 4. For more details on the universal projection we refer the reader to [34].

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. Thm. 5 states the soundness of our algorithm. See App. A.6 for its proof.

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 Sec. 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 Sec. 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 $k = 1$. 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

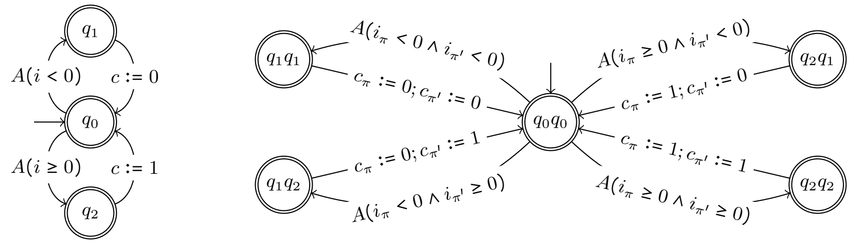


Fig. 3. Left: The program automaton $\mathcal{P}$ used in the first example. Right: The program automaton $\mathcal{P}^2$. For brevity, we use A for *assert* and join consecutive assertions.

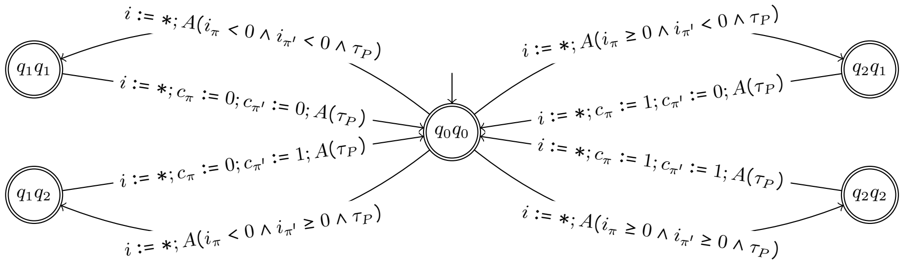


Fig. 5. The combined product $(\mathcal{P}^2 \otimes \mathcal{A}_\psi)$

transitions removed (see Fig. 5 for the combined product).

The automaton $(\mathcal{P}^2 \otimes \mathcal{A}_\psi)^\forall_k$ 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 $\mathcal{P} \setminus (\mathcal{P}^2 \otimes \mathcal{A}_\psi)^\forall_k$ 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 $i = 0$.

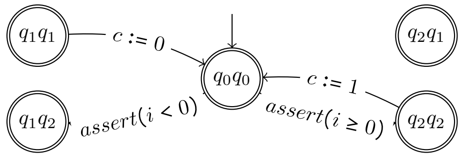


Fig. 4. program automaton $(\mathcal{P}^2 \otimes \mathcal{A}_\psi)^\forall_k$

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.

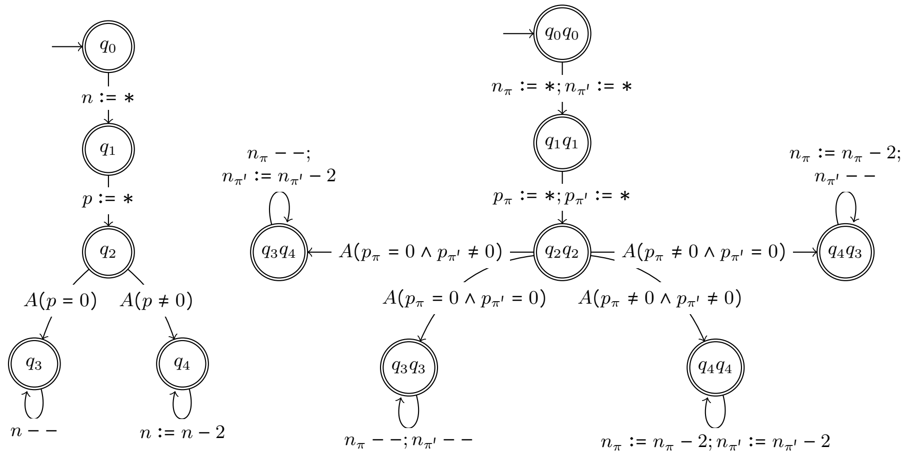


Fig. 6. Left: The program automaton $\mathcal{P}_{cy}$, Right: The program automaton $\mathcal{P}_{cy}^2$.

The automaton $\mathcal{P}_{cy}^2$ 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. 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.
2. B. Alpern and F.B. Schneider. Defining liveness. *Information Processing Letters*, pages 181–185, 1985.
3. Shiraj Arora, René Rydhof Hansen, Kim Guldstrand Larsen, Axel Legay, and Danny Bøgsted Poulsen. Statistical model checking for probabilistic hyperproperties of real-valued signals. In Owolabi Legunsen and Grigore Rosu, editors, *Model Checking Software - 28th International Symposium, SPIN 2022, Virtual Event, May 21, 2022, Proceedings*, volume 13255 of *Lecture Notes in Computer Science*, pages 61–78. Springer, 2022.
4. 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.
5. 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.
6. Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. Secure information flow by self-composition. *Math. Struct. Comput. Sci.*, 21(6):1207–1252, 2011.
7. Jan Baumeister, Norine Coenen, Borzoo Bonakdarpour, Bernd Finkbeiner, and César Sánchez. A temporal logic for asynchronous hyperproperties. In Alexandra Silva and K. Rustan M. Leino, editors, *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part I*, volume 12759 of *Lecture Notes in Computer Science*, pages 694–717. Springer, 2021.
8. 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.
9. 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.
10. 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.
11. Borzoo Bonakdarpour, César Sánchez, and Gerardo Schneider. Monitoring hyperproperties by combining static analysis and runtime verification. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods,*

Verification and Validation. Verification - 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November 5-9, 2018, Proceedings, Part II, volume 11245 of *Lecture Notes in Computer Science*, pages 8–27. Springer, 2018.
12. Laura Bozzelli, Adriano Peron, and César Sánchez. Expressiveness and decidability of temporal logics for asynchronous hyperproperties. In Bartek Klin, Slawomir Lasota, and Anca Muscholl, editors, *33rd International Conference on Concurrency Theory, CONCUR 2022, September 12-16, 2022, Warsaw, Poland*, volume 243 of *LIPIcs*, pages 27:1–27:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.
13. 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.
14. 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.
15. 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.
16. 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.
17. 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.
18. 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.
19. 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.

20. Michael R. Clarkson and Fred B. Schneider. Hyperproperties. *J. Comput. Secur.*, 18(6):1157–1210, 2010.
21. 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.
22. 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.
23. 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.
24. 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.
25. 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.
26. 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.
27. Rayna Dimitrova, Bernd Finkbeiner, and Hazem Torfah. Probabilistic hyperproperties of markov decision processes. In Dang Van Hung and Oleg Sokolsky, editors, *Automated Technology for Verification and Analysis - 18th International Symposium, ATVA 2020, Hanoi, Vietnam, October 19-23, 2020, Proceedings*, volume 12302 of *Lecture Notes in Computer Science*, pages 484–500. Springer, 2020.
28. Oyendrila Dobe, Erika Ábrahám, Ezio Bartocci, and Borzoo Bonakdarpour. Hyperprob: A model checker for probabilistic hyperproperties. In Marieke Huisman, Corina S. Pasareanu, and Naijun Zhan, editors, *Formal Methods - 24th International Symposium, FM 2021, Virtual Event, November 20-26, 2021, Proceedings*, volume 13047 of *Lecture Notes in Computer Science*, pages 657–666. Springer, 2021.
29. Marco Eilers, Peter Müller, and Samuel Hitz. Modular product programs. *ACM Trans. Program. Lang. Syst.*, 42(1):3:1–3:37, 2020.
30. Bernd Finkbeiner. Model checking algorithms for hyperproperties (invited paper). In Fritz Henglein, Sharon Shoham, and Yakir Vizel, editors, *Verification, Model Checking, and Abstract Interpretation - 22nd International Conference, VMCAI*

2021, Copenhagen, Denmark, January 17-19, 2021, Proceedings, volume 12597 of *Lecture Notes in Computer Science*, pages 3–16. Springer, 2021.

31. 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.
32. Bernd Finkbeiner, Jana Hofmann, Florian Kohn, and Noemi Passing. Reactive synthesis of smart contract control flows. *CoRR*, abs/2205.06039, 2022.
33. 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.
34. Bernd Finkbeiner and Noemi Passing. Synthesizing dominant strategies for liveness. In Anuj Dawar and Venkatesan Guruswami, editors, *42nd IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2022, December 18-20, 2022, IIT Madras, Chennai, India*, volume 250 of *LIPIcs*, pages 37:1–37:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.
35. Bernd Finkbeiner, Markus N. Rabe, and César Sánchez. Algorithms for model checking HyperLTL and HyperCTL*. 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.
36. Hadar Frenkel, Orna Grumberg, and Sarai Sheinvald. An automata-theoretic approach to model-checking systems and specifications over infinite data domains. *J. Autom. Reason.*, 63(4):1077–1101, 2019.
37. 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.
38. 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.
39. 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.
40. 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.

41. Hsi-Ming Ho, Ruoyu Zhou, and Timothy M. Jones. On verifying timed hyperproperties. In Johann Gamper, Sophie Pinchinat, and Guido Sciavicco, editors, *26th International Symposium on Temporal Representation and Reasoning, TIME 2019, October 16-19, 2019, Málaga, Spain*, volume 147 of *LIPIcs*, pages 20:1–20:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.
42. Donald B. Johnson. Finding all the elementary circuits of a directed graph. *SIAM J. Comput.*, 4(1):77–84, 1975.
43. Leslie Lamport and Fred B. Schneider. Verifying hyperproperties with TLA. In *34th IEEE Computer Security Foundations Symposium, CSF 2021, Dubrovnik, Croatia, June 21-25, 2021*, pages 1–16. IEEE, 2021.
44. Benedikt Maderbacher and Roderick Bloem. Reactive synthesis modulo theories using abstraction refinement. *CoRR*, abs/2108.00090, 2021.
45. 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.
46. 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.
47. Luan Viet Nguyen, James Kapinski, Xiaoqing Jin, Jyotirmoy V. Deshmukh, and Taylor T. Johnson. Hyperproperties of real-valued signals. In Jean-Pierre Talpin, Patricia Derler, and Klaus Schneider, editors, *Proceedings of the 15th ACM-IEEE International Conference on Formal Methods and Models for System Design, MEMOCODE 2017, Vienna, Austria, September 29 - October 02, 2017*, pages 104–113. ACM, 2017.
48. 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.
49. 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.
50. Hiroshi Unno, Tachio Terauchi, and Eric Koskinen. Constraint-based relational verification. In Alexandra Silva and K. Rustan M. Leino, editors, *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part I*, volume 12759 of *Lecture Notes in Computer Science*, pages 742–766. Springer, 2021.

A Detailed Correctness Proofs

A.1 Hyper Linear Temporal Logic (HyperLTL)

Let AP be a finite set of atomic propositions and let Π be a finite set of trace variables. Then, a HyperLTL formula is defined by the grammar

$$\varphi ::= a_\pi \mid \neg\varphi \mid \varphi \land \varphi \mid \bigcirc\varphi \mid \varphi\,\mathcal{U}\,\varphi \mid \forall\pi.\ \varphi \mid \exists\pi.\ \varphi$$

where $a \in AP$ 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

$$m[\pi \to s](\pi) = s$$
$$m[\pi \to s](\pi') = m(\pi') \qquad \text{for } \pi \neq \pi'$$

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

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

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

For the sepcial case in which there is only one trace quantifier, and this is a universal quantifier, we are in the fragment of LTL.

A.2 Similiarity of LTL and TSL

The following lemma 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 17. *Let $\hat{\zeta} \in \hat{\mathrm{A}}^\omega, \rho \subseteq \hat{\mathcal{T}_P}, \upsilon \subseteq \hat{\mathcal{T}_U}$. We define*

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

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

Lemma 1. *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. (Lemma 1) 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 $\varphi = \psi \mathcal{U} \psi'$

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

A.3 Proof of Theorem 4

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

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

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

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

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

$\Leftarrow$ Let σ be a trace of P accepted by P_k. Then, there exist states of P $q_0, q_1 \ldots$ 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, $\sigma_j \ldots \sigma_{j+k-1}$ is feasible. Thus, σ is k-feasible. □

A.4 Proof of Theorem 1

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 17). 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 2 we show that if $\zeta \lhd combine(\sigma, X)$ for some X, then $\zeta_{|\sigma} \lhd \sigma$. In Lemma 3 we show that then, we also have that $X = Seq(\zeta_{|\sigma})$. Lemma 4 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 1 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 $combine(\sigma_t, Seq(\zeta)_t) =$
$save\_values_{Seq(\zeta)_t}$; σ_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 18. *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.*

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

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

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 19. *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.*

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

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 $tmp_1, tmp_2 \ldots$

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 2. *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

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

- 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 $\zeta_{\iota(t-1)} \ldots \zeta_{\iota(t)-|\mathbb{I}|-1}$ and in $\zeta_{\iota(t)-|\mathbb{I}|}, \ldots \zeta_{\iota(t)}$ are cells from $\mathbb{C}^* \backslash \mathbb{C}$.
- The two remaining cases are analogous.

Lemma 3. *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 $\tau_P \in \rho$

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

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 18 Therefore, for every $[\![c_j \leftarrowtail \tau_{Fj}]\!] \in \upsilon$,

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

Lemma 4. *If $\zeta \triangleleft \sigma$, then $\tilde{\zeta} \triangleleft combine(\sigma, Seq(\zeta))$*

Proof. We have to show that for all t, $\tilde{\zeta} \triangleleft_t combine(\sigma, Seq(\zeta))$. 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 $\forall t, \tilde{\zeta} \triangleleft_{\iota(t)+1} combine(\sigma, Seq(\zeta))$. This boils down to

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

As the temporary variables $tmp_1, tmp_2 \ldots$ are not used in any $\tau_P \in \rho$, this is by definition 18 equivalent to

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

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

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

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

Which is equivalent to

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

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

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

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

Now, we have all the lemmas needed to prove Theorem 1

Proof. (Theorem 1)
$\Rightarrow$ Assume that $P \otimes A_{\neg\varphi}$ has a feasible trace. Then, this is a trace $combine(\sigma, X)$ for some $\sigma \in \mathcal{L}(P)$ and $X \in \mathcal{L}(A_{\neg\varphi})$. Moreover, $\zeta \triangleleft combine(\sigma, X)$ for some $\zeta \in \mathsf{A}^{\omega}$. By Lemma 3, we know that $X = Seq(\zeta)$ and by Lemma 2 we know that $\zeta \triangleleft \sigma$. By the correctness of A_{φ}, we know that $Seq(\zeta) \models_{LTL} \neg\varphi$, which by Lemma 1 means that $\zeta \models \neg\varphi$. 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 $\zeta \triangleleft \sigma$ and $\zeta \models \neg\varphi$. This means by Lemma 1 that $Seq(\zeta) \models_{LTL} \neg\varphi$, so $Seq(\zeta)$ is accepted by $A_{\neg\varphi}$. Then, $combine(\sigma, Seq(\zeta))$ is a trace of $P \otimes A_{\neg\varphi}$. By Lemma 4, $\tilde{\zeta} \triangleleft combine(\sigma, Seq(\zeta))$, so this is also a feasible trace.

A.5 Proof of Theorems 2 and 3

As the two theorems are dual, it suffices to give the proof for Theorem 2.

The proof is analogous to the proof of Theorem 1, 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 $\hat{\zeta} = \varnothing[\pi_1, \zeta_{\pi_1}] \dots [\pi_n, \zeta_{\pi_n}]$. 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 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 $\zeta_{\pi_1} \dots \zeta_{\pi_n}$, let $\hat{\zeta} = \varnothing[\pi_1, \sigma_{\pi_1}] \dots [\pi_n, \sigma_{\pi_n}]$. We define the **adapted computation** $\widetilde{(\zeta_{\pi_1}, \dots, \zeta_{\pi_n})}$.*

$$\hat{a}_t^{tmp_1} := \hat{\zeta}_{t-1}[tmp_1 \mapsto \eta(\tau_{F1}, \hat{\zeta}_{t-1})]$$
$$\hat{a}_t^{tmp_j} := \hat{a}^{tmp_{j-1}}[tmp_j \mapsto \eta(\tau_{Fj}, \hat{\zeta}_{t-1})] \qquad \text{for } 1 < j \leq m$$
$$\hat{a}_t^{\pi_j} = (\hat{\zeta}_{t-1}[\pi_1, \zeta_{\pi_1}] \dots [\pi_j, \zeta_{\pi_j}])_t[i_1 \mapsto \zeta_{t-1}(i_1), \ \dots \ , i_k \mapsto \zeta_{t-1}(i_k), \\ tmp_1 \mapsto \eta(\tau_{F1}, \hat{\zeta}_{t-1}), \dots, tmp_m \mapsto \eta(\tau_{Fm}, \hat{\zeta}_{t-1})]$$

$$\hat{a}_t^{i_1} := \hat{a}_t^{\pi_n}[i_1 \mapsto \hat{\zeta}_t(i_1)]$$
$$\hat{a}_t^{i_j} := \hat{a}^{i_{j-1}}[i_j \mapsto \hat{\zeta}_t(i_j)] \qquad \textit{for } 1 < j \le k$$
$$(\widetilde{\zeta_{\pi_1}, \dots, \zeta_{\pi_n}})^t := \hat{a}_t^{tmp_1} \dots \hat{a}_t^{tmp_n}\ \hat{a}_t^{\pi_1} \dots \hat{a}_t^{\pi_n}\ \hat{a}_t^{i_1}\ \dots\ \hat{a}_t^{i_k}\ \hat{a}_t^{i_k}\ \hat{a}_t^{i_k}$$
$$(\widetilde{\zeta_{\pi_1}, \dots, \zeta_{\pi_n}}) := (\widetilde{\zeta_{\pi_1}, \dots, \zeta_{\pi_n}})^0\ (\widetilde{\zeta_{\pi_1}, \dots, \zeta_{\pi_n}})^1 \dots$$

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

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

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

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

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

Lemma 5. *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 $1 \le j \le n$.*

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 $(\widetilde{\hat{\zeta}_{|\pi_1}, \dots, \hat{\zeta}_{|\pi_n}})$.

- 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 $\mathbb{C}^* \setminus \mathbb{C}$ or cells from other traces.

– The two remaining cases are analogous.

Lemma 6. *If* $\hat{\zeta} \triangleleft combine(\sigma, X)$*, then* $X = Seq(\varnothing[\pi_1, \hat{\zeta}_{|\pi_1}] \dots [\pi_n, \hat{\zeta}_{|\pi_n}])$

Proof. Set $\hat{\zeta}' = \varnothing[\pi_1, \hat{\zeta}_{|\pi_1}] \dots [\pi_n, \hat{\zeta}_{|\pi_n}]$. 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 20 equal to

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

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 $h = \left(\bigwedge_{[\![c_j \leftarrowtail \hat{\tau_{Fj}}]\!] \in \upsilon} \begin{cases} c_j = tmp_j & \text{if } [\![c_j \leftarrowtail \hat{\tau_{Fj}}]\!] \in X_t \\ c_j \neq tmp_j & \text{else} \end{cases}\right)$
As before, we know that $\eta(h, \hat{\zeta}'_t) = true$. Moreover, we know that for each j, $\hat{\zeta}_{\iota(t)+1}(tmp_j) = \eta(\hat{\tau_{Fj}}, \hat{\zeta}'_{t-1})$ again by Definition 20 Therefore, for every $[\![c_j \leftarrowtail \hat{\tau_{Fj}}]\!] \in \upsilon$,

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

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

Lemma 7. *If* $\zeta_{\pi_1} \triangleleft \sigma_{\pi_1} \wedge \dots \wedge \zeta_{\pi_n} \triangleleft \sigma_{\pi_n}$*, then* $(\widetilde{\zeta_{\pi_1}, \dots, \zeta_{\pi_n}}) \triangleleft combine(\sigma, Seq(\hat{\zeta}'))$*, where* $\hat{\zeta}' = \varnothing[\pi_1, \zeta_{\pi_1}] \dots [\pi_n, \zeta_{\pi_n}]$

Proof. Set $\hat{\zeta} = (\widetilde{\zeta_{\pi_1}, \dots, \zeta_{\pi_n}})$. 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 $\forall t, \hat{\zeta} \triangleleft_{\iota(t)+1} combine(\sigma, Seq(\hat{\zeta}'))$. 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 20 equal to

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

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 2

Proof. (Theorem 2)
$\Rightarrow$ 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 $\hat{\zeta} \in \hat{\mathrm{A}}^\omega$. By Lemma 5, 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, we know that $X = Seq(\hat{\zeta}')$. By the correctness of A_ψ this means that $Seq(\hat{\zeta}') \models_{LTL} \neg\varphi$ which by Lemma 1 means that $\hat{\zeta}' \models \neg\varphi$. 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 $\hat{\zeta}' = \varnothing[\pi_1, \zeta_{\pi_1}] \dots [\pi_n, \zeta_{\pi_n}] \models \neg\varphi$. This means by Lemma 1 that $Seq(\hat{\zeta}') \models_{LTL} \neg\varphi$, 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 $\sigma = \underline{\sigma_0 \sigma_1} \dots$. Then, $combine(\sigma, Seq(\zeta))$ is a trace of $P \otimes A_{\neg\varphi}$. By Lemma 7, $\widetilde{(\zeta_{\pi_1}, \dots, \zeta_{\pi_n})} \lhd combine(\sigma, Seq(\zeta'))$, so this is also a feasible trace.

A.6 Proof of Theorem 5

We now prove Theorem 5. 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 8. *Let $\sigma \in Stmt^\omega$ be feasible and $\sigma_t = (((\sigma_{\pi_1})_{\pi_1})_t, \dots, ((\sigma_{\pi_n})_{\pi_n})_t)$ for some $\sigma_{\pi_1}, \dots \sigma_{\pi_n}$. Then $\sigma_{\pi_1}, \dots, \sigma_{\pi_n}$ are also all feasible.*

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

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

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, $\zeta_{\pi_j} \triangleleft_t \sigma_{\pi_j}$

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

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

  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 $\hat{\zeta}_{(t-1) \cdot m + j - 1}, \dots \hat{\zeta}_{t \cdot m + j - 2}$ are cells from other traces.
- The remaining two cases are analogous.

We now prove Theorem 5

Proof. Assume that $P^m \backslash (P^n \otimes A_\psi)^\forall_{k, C(k')}$ has a feasible trace σ with $\hat{\zeta} \triangleleft \sigma$. By Lemma 8, this means that $\zeta_{\pi_1} \triangleleft \sigma_{\pi_1} \wedge \dots \wedge \zeta_{\pi_m} \triangleleft \sigma_{\pi_m}$. It suffices to show that $\varnothing[\pi_1, \zeta_1] \dots [\pi_m, \zeta_m] \nvDash \exists \pi_{m+1}. \ \dots \exists \pi_n. \ \psi$ as this implies that $\zeta_1, \dots \zeta_m$ are a counterexample proving that P does not satisfsy φ.

Proof by contradiction. Assume that $\varnothing[\pi_1, \zeta_1] \dots [\pi_m, \zeta_m] \vDash \exists \pi_{m+1}. \ \dots \exists \pi_n. \ \psi$. Then, there are traces $\sigma_{\pi_{m+1}}, \dots \sigma_{\pi_n}$ and computations $\zeta_{\pi_{m+1}} \dots \zeta_{\pi_n}$ such that $\zeta_{\pi_{m+1}} \triangleleft \sigma_{\pi_{m+1}} \wedge \dots \wedge \zeta_{\pi_n} \triangleleft \sigma_{\pi_n}$ and $\hat{\zeta}' = \varnothing[\pi_1, \zeta_{\pi_1}] \dots [\pi_n, \zeta_{\pi_n}] \vDash \psi$.
Set $\sigma'_t = ((\sigma_{\pi_1})_{\pi_1 t}; \dots; (\sigma_{\pi_n})_{\pi_n t})$ and $\sigma' = \sigma'_0 \ \sigma'_1 \dots$. Now, by Lemma 1 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, 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.