Acta Informatica (2025) 62:43
https://doi.org/10.1007/s00236-025-00509-8

RESEARCH

Hypernode automata

Ezio Bartocci[1] · Marek Chalupa[2] · Thomas A. Henzinger[2] · Dejan Nickovic[3] · Ana Oliveira da Costa[2]

Received: 6 December 2024 / Accepted: 30 September 2025 / Published online: 9 December 2025

Abstract

In this work, we present *hypernode automata* as a specification formalism for hyperproperties of systems whose executions may be misaligned among themselves, such as concurrent systems. These automata consist of nodes labeled with *hypernode logic* formulas and transitions marked with synchronizing actions. Hypernode logic formulas establish relations between sequences of variable values among different system executions. This logic enables both synchronous and asynchronous analysis of traces. In its asynchronous view on execution traces, hypernode formulas establish relations on the order of value changes for each variable without correlating their timing. In both views, the analysis of different execution traces is synchronized through the transitions of hypernode automata. By combining logic's declarative nature with automata's procedural power, hypernode automata seamlessly integrate asynchronicity requirements at the node level with synchronicity between node transitions. We show that the model-checking problem for hypernode automata is decidable for specifications where each node specifies either a synchronous or an asynchronous requirement for the system's executions, but not both.

✉ Ana Oliveira da Costa
ana.costa@ist.ac.at

Ezio Bartocci
ezio.bartocci@tuwien.ac.at

Marek Chalupa
mchalupa@ist.ac.at

Thomas A. Henzinger
tah@ist.ac.at

Dejan Nickovic
dejan.nickovic@ait.ac.at

1 TU Wien, Vienna, Austria

2 IST Austria, Klosterneuburg, Austria

3 AIT, Vienna, Austria

1 Introduction

Formalisms like linear temporal logic (LTL) or automata are commonly used to specify and verify properties of isolated system executions, also known as trace properties. Security requirements such as information-flow policies cannot be expressed as trace properties as they require simultaneous reasoning about multiple execution traces. Such requirements define *hyperproperties*. From a formal-language perspective, a hyperproperty specifies a collection of correct systems, that is, it defines a set of sets of traces [12]. HyperLTL [14], an extension of LTL with trace quantifiers, has emerged as a popular formalism for both the specification and verification of hyperproperties. The temporal operators of HyperLTL (and related hyperlogics) progress in lockstep over all traces bounded to one of the quantified trace variables. We refer to such logics as *synchronous hyperlogics*. Due to its lockstep analysis of traces, HyperLTL cannot express, for example, stateful information-flow policies, requiring that all executions of the system need to satisfy a sequence of security policies that change over their execution time [3].

In such cases, each system execution transitions through different specification states at different times, which poses a challenge for formalisms with a bound on the number of traces it can analyze simultaneously. This limitation has been observed repeatedly and independently in recent years by [1, 6, 19] all of whom have proposed asynchronous hyperlogics to address the problem. We take a different route and propose a specification language for hyperproperties, called *hypernode automata*, that mixes synchronous and asynchronous reasoning by combining automata and logic.

Hypernode automata are finite automata whose nodes are labeled with formulas from a hyperlogic called *hypernode logic*. Hypernode formulas enable the comparison of execution trace segments in two modes: *synchronous*, where all traces are analyzed in lock-step, and *asynchronous*, where each trace analysis can progress at a different speed. In the asynchronous interpretation, the focus is on how the values of individual variables evolve, regardless of the specific time at which those changes occur. The transitions between nodes (in a hypernode automaton) are labeled with actions that re-synchronize the analysis of different execution traces : while the traces can progress independently within hypernodes, they must synchronize (that is, "wait for each other") before the analysis moves to the next hypernode. Although automata-based languages have been used before to specify synchronous hyperproperties [8], hypernode automata is the first formalism to systematically distinguish between trace synchronicity and trace asynchronicity in the specification of hyperproperties, offering a clearer and more expressive framework for reasoning about complex asynchronous temporal behaviors.

In a nutshell, we address the natural misalignment of system executions in hyperproperty verification at two levels: (i) within a single (stateless) specification by using stutter-reduced representations of executions to account for differences in execution pace; and (ii) across specification states by modeling stateful specifications as automata with actions that govern the transition between specification states (effectively enabling dynamic re-synchronization of executions). We explain hypernode logic and automata in more detail below.

In the asynchronous part of hypernode logic, we adopt a maximally asynchronous view of finite traces: we focus solely on how each program variable evolves, treating the progression of its values independently from all other variables. Hypernode logic was first introduced in [2] to capture such asynchronous hyperproperties. At their core, hypernode formulas are formed by combining quantification over finite traces with a binary relation comparing them (either synchronously or asynchronously). While in both the synchronous and asynchronous view, we compare traces up to their prefixing, in the asynchronous case, we also remove their stuttering steps. Concretely, synchronous comparison is made using the common prefixing predicate $\leq$, while asynchronous comparison is done using the stutter-reduced prefixing relation $x(\pi) \precsim y(\pi')$, for trace variables π and π' and program variables x and y. This stutter-reduced prefixing relation asserts that the program variable x undergoes the same ordered value changes in the trace assigned to π as the variable y does in π', but the changes may occur at different times (stuttering), and there may be additional changes of y in π' (prefixing). Hypernode logic was recently extended [13] to allow the use of regular expressions to specify evaluation patterns.

Example: Observational Determinism (OD)

Zdancewic and Myers proposed, in [36], using *observational determinism*, to capture non-interference of secure values (i.e. **H**igh confidentiality) into public observations (i.e. **L**ow confidentiality) in concurrent programs. They argue that the challenge in verifying such properties over concurrent systems lies in separating between non-determinism naturally occurring in concurrent programs and non-determinism on the public observable behavior due to the leakage of secret values. They propose comparing pairs of program executions up to the removal of repeating memory configurations (i.e., stuttering of values) and the duration of the shortest one. Formally, for every pair of program executions ($\forall\pi\forall\pi'$), if they have the same public inputs, they should exhibit the same public observable behavior.

We define below, in Fig. 1, OD as a hypernode automata. Our goal is to verify systems that at execution time continuously receive inputs and produces outputs (i.e., each execution will undergo multiple rounds of observable input and output behavior). We represent by $\Sigma_{\mathbf{in}}$ the domain of all possible public input observations. Then, for all executions with the same $\mathrm{init} \in \Sigma_{\mathbf{in}}$, we want to verify that their public output is the same up to differences in the timing of the output.

Hypernode automata synchronize the transition between hypernodes with actions: for each action sequence, a hypernode automaton specifies a sequence of hypernode formulas. In the example above, the different input values act as re-synchronization actions. The main contribution of this paper is a model-checking algorithm for hypernode automata (the "specification") and models defined by a set of (possibly) infinite traces induced by an action-labeled Kripke structure (the "model") combining both synchronous and asynchronous analysis of traces. The subroutine that model-checks formulas of hypernode logic introduces automata-theoretic constructions – *left-aligned automata* for the synchronous case and *stutter-free automata* for the asynchronous one – that are then used in familiar logical contexts such as filtration and self-composition.

Each mode of analysis over hypernode formulas has different implicit assumptions on their models (reflecting their different use cases). For the synchronous view, different executions are compared step-wise over valuations for all program variables. For this reason, the

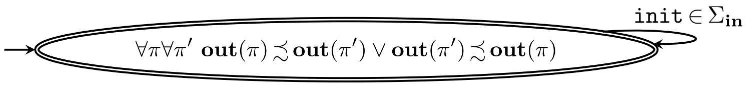

$\mathtt{init} \in \Sigma_{\mathbf{in}}$
$\forall\pi\forall\pi'\ \mathbf{out}(\pi) \precsim \mathbf{out}(\pi') \vee \mathbf{out}(\pi') \precsim \mathbf{out}(\pi)$

Fig. 1 Hypernode automaton specifying observational determinism

usual Kripke structures are an adequate abstraction for models of synchronous hypernode formulas. As for the asynchronous view, we are interested in models that capture asynchronous computation; that is, allow any possible interleaving of their variables progression. We prove in this paper that asynchronous hypernode logic interpreted over models that are not asynchronous (i.e., implicitly defining unbounded dependencies between variables) has an undecidable model-checking.

Contribution

This paper expands and improves on our preliminary manuscript published at CONCUR 2023 [2] with the following new contributions:

- We present model-checking algorithms for two decidable fragments of extended hypernode logic, introduced in [13], which extends the logic introduced in [2] with synchronous comparisons and the possibility to match traces with regular patterns.
- We identified a problem in the statement of Theorem 13 in [2], which stated that model checking (asynchronous) hypernode logic on Kripke structures is decidable, and corrected the statement to refer only to stutter-reducible Kripke structures.
- We present new results on the undecidability of model-checking of extended hypernode logic, and the asynchronous fragment of hypernode logic on generic Kripke structures.
- We include all the proofs missing in [2].

Paper organization

Section 2 introduces an example of how we can use hypernode automata to specify and detect an implicit information flow in a program that has both synchronous and asynchronous modes of operation. Section 3 defines hypernode logic and introduces its respective model-checking problem. In this section, we also present a new result on the undecidability of asynchronous hypernode logic. We present our solution for the model-checking of synchronous hypernode formulas on Kripke structures in Section 4, where we introduce left-aligned automata. Section 5 presents the model-checking of asynchronous hypernode formulas over asynchronous Kripke structures. In the same section, we present stutter-free automata, which are used in the asynchronous model-checking algorithm. Hypernode automata are introduced in Section 6, while in section 6.2, we discuss its model-checking algorithm, which uses the model-checking algorithms introduced in the previous sections.

2 Motivating example

In this example, we show how we can use hypernode automata to verify if programs supporting both synchronous and asynchronous communication modes satisfy observational determinism.

Algorithm 1 $\mathrm{Get}_{\mathrm{x,y}}$

```
s := sync;
do
    s := read(local, status);
    in := read(local, input);
    $\mathrm{in}^L$ := low(in); $\mathrm{in}^H$ := high(in);
    if ($s$ = sync) then
        if ($\mathrm{in}^H \neq 0$) then Request($x, \mathrm{in}^L$);
        Request($y, \mathrm{in}^L$);
    else if ($s$ = async) then
        Request($x, \mathrm{in}^L$) || Request($y, \mathrm{in}^L$);
while true;
```

Algorithm 2 Request(var, in)

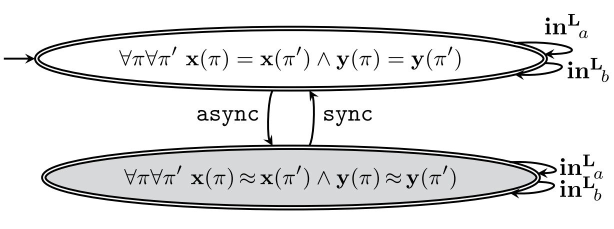
```
request(url, var, in);
response := read(url, var);
output(local, response);
```

An example of a system supporting different modes of communication is the program $\mathrm{Get}_{x,y}$ (on the left). $\mathrm{Get}_{x,y}$ reads from a local channel the current system status (either sync or async) and user input. Depending on the last received status, the program proceeds to request values for x and y, either via the synchronous or parallel composition of Request calls, with the public input values read beforehand. The program Request (on the left) sends a request to an external server, providing a variable name (var) and an input value (in). When the server returns a response through the external channel, Request forwards it to the local channel.

We observe that, in the synchronous mode, $\mathrm{Get}_{x,y}$ requests the value of x only when the secret input is different from 0. This is an example of an implicit information flow: a malicious agent can learn that the secret value changed by witnessing differences in the public observable behavior of the system for the same public inputs.

Verifying OD for systems like $\mathrm{Get}_{x,y}$ is challenging because it requires both synchronous and asynchronous analysis of multiple traces. When the analysis mode can be isolated, for example, by identifying transitions between operating modes with an action (as in this example), we can use hypernode automata.

In Fig. 2, we show a possible specification of OD that supports switching between modes of operation using a hypernode automaton. The self-loops in the automata are labeled with the two possible public input values, effectively assigning to each hypernode only the traces



$\mathbf{in^L}_a$
$\mathbf{in^L}_b$
$\forall\pi\forall\pi'\ \mathbf{x}(\pi) = \mathbf{x}(\pi') \wedge \mathbf{y}(\pi) = \mathbf{y}(\pi')$
async
sync
$\forall\pi\forall\pi'\ \mathbf{x}(\pi) \approx \mathbf{x}(\pi') \wedge \mathbf{y}(\pi) \approx \mathbf{y}(\pi')$
$\mathbf{in^L}_a$
$\mathbf{in^L}_b$

Fig. 2 Observational determinism across different synchronization modes

Table 1 Two trace execution segments of the program $\mathrm{Get}_{x,y}$ for a server (S) returning the following values: $S(x,a)=o$, $S(x,b)=p$, $S(y,a)=q$ and $S(y,b)=r$

	s:	ε	ε	ε	async	ε	ε	ε	ε	ε	ε	sync	ε	ε	ε
	$\mathbf{in^L}$:	a	ε	ε	ε	b	ε	ε	a	ε	ε	ε	b	ε	ε
τ_1	**x**:	_	o	o	_	_	_	p	_	o	o	_	_	p	p
	y:	_	_	q	_	_	r	r	_	_	q	_	_	_	r
	s:	ε	ε	ε	ε	ε	ε	async	ε	ε	ε	sync	ε	ε	
	$\mathbf{in^L}$:	a	ε	ε	b	ε	ε	ε	b	ε	ε	ε	b	ε	
τ_2	**x**:	_	o	o	_	p	p	_	_	p	p	_	_	_	
	y:	_	_	q	_	_	r	_	_	_	r	_	_	r	

Table 2 From observed trace segments to stutter-free traces grouped by input

Observed segments:		$\mathbf{in^L}$	b	ε	ε	a	ε	ε	**Hypernode View:**				
	τ_1	**x**:	_	_	p	_	o	o		_	p	_	o
		y:	_	r	r	_	_	q		_	r	_	q
		$\mathbf{in^L}$	b	ε	ε								
	τ_2	**x**:	_	p	p					_	p		
		y:	_	_	r					_	r		

that share the same input. We denote by $\approx$ the stuttering-equivalence relation; that is, the requirement that both traces are in the stutter-reduced prefix relation ($\precsim$).

The main difference between the two hypernodes in the automaton above is the relation used to compare traces: in the synchronous node, we require them to be equal, while in the asynchronous node, we require them to be equal up to the removal of their stuttering steps.

In Table 1, we show two possible executions of the program $\mathrm{Get}_{x,y}$. The traces below have four variables; while two of them (x and y) are program variables (i.e., they are updated by the program we are verifying), the other two are input actions. In our trace representation, input actions are present in the trace when the program reads them (we use ε to indicate the absence of actions). In contrast, program variables start with no assigned value (denoted by the '_') until they are assigned a value. At each iteration of the loop, we assume that program variables are reset to their default (that is, no value has been assigned).

We can see in the traces below that for the last input shown in the table, the trace τ_2 exhibits a different public observation than τ_1, even though they have the same input. This example showcases the security problem in program $\mathrm{Get}_{x,y}$, which can be detected with our model-checking algorithm.

Table 2 illustrates how nodes with asynchronous hypernode formulas interpret the trace segments assigned to them. First, note that only traces with the same input value are compared. As a result, two separate comparisons take place. We can also see in the table that when x and y values are stutter-reduced, then they define the same sequence of valuations in both traces, up to their common length.

3 Hypernode logic

In this section, we introduce *(extended) hypernode logic*. Hypernode logic was first introduced as a fully asynchronous hyperlogic [2]: traces could only be compared up to the removal of stuttering (i.e., repeating) of independent program variable valuations. The logic was later extended [13] to allow for a mix of synchronous and asynchronous comparisons together with regular expressions to specify trace patterns. From now on, when we refer to hypernode logic, we mean the extended version as in [13].

Given a finite set X of *program variables* over a finite domain Σ, a *trace segment* τ is a finite sequence of valuations in Σ^X, where each *valuation* $v : X \to \Sigma$ maps program variables to values from the domain. Given two traces τ and τ', we define their *concatenation* as $\tau.\tau'$, also written as $\tau\tau'$. For a trace $\tau = uv$, u is a *prefix of* τ, denoted $u \leq \tau$, while v is a *suffix of* τ. We denote the set of all trace segments over X and Σ by $(\Sigma^X)^*$. A *segment hyperproperty* specifies a property of sets of trace segments. Formally, a segment hyperproperty $\mathbf{T}$ is a set of sets of trace segments, that is, $\mathbf{T} \subseteq 2^{(\Sigma^X)^*}$.

Hypernode logic specifies relations between *unzipped* trace segments, which are mappings of variables to *variable traces*. Unzipping transforms each trace segment into a mapping from program variables to the sequence of values assigned to them. Formally, for a trace segment $\tau \in (\Sigma^X)^*$, with $\tau = v_0 \ldots v_n$, its unzipped trace segment is the mapping $\mathrm{unzip}(\tau) = \{\mathbf{x} : v_0(\mathbf{x})v_1(\mathbf{x})... \mid \mathbf{x} \in X\}$. The projection of a trace τ to $\mathbf{x}$, also called the $\mathbf{x}$ -*trace* (of τ), is defined as $\tau(\mathbf{x}) = (\mathrm{unzip}(\tau))(\mathbf{x})$. For unzipped trace segments τ, their $\mathbf{x}$ -trace is just the projection of the sequence of values assigned to $\mathbf{x}$ in τ, i.e., $\tau(\mathbf{x})$. The i-position of a trace is $\tau[i] = v_i$ (for unzipped traces, $\tau(\mathbf{x})[i] = v_i(\mathbf{x})$) when $i \leq n$, and the empty word, otherwise.

Example: Unzipping Trace Segments

In our motivating example (Section 2), traces are defined over the program variables $X = \{\mathbf{x}, \mathbf{y}\}$ ranging over the domain $\Sigma = \{p, q, o, r, \_\}$. The first execution in Table 1 defines the following trace segment over Σ^X:

$$\begin{aligned}\tau_1 = &\{\mathbf{x} : \_, \mathbf{y} : \_\}\{\mathbf{x} : o, \mathbf{y} : \_\}\{\mathbf{x} : o, \mathbf{y} : q\}\\ &\{\mathbf{x} : \_, \mathbf{y} : \_\}\{\mathbf{x} : \_, \mathbf{y} : \_\}\{\mathbf{x} : \_, \mathbf{y} : r\}\{\mathbf{x} : p, \mathbf{y} : r\}\\ &\{\mathbf{x} : \_, \mathbf{y} : \_\}\{\mathbf{x} : o, \mathbf{y} : \_\}\{\mathbf{x} : o, \mathbf{y} : q\}\\ &\{\mathbf{x} : \_, \mathbf{y} : \_\}\{\mathbf{x} : \_, \mathbf{y} : \_\}\{\mathbf{x} : p, \mathbf{y} : \_\}\{\mathbf{x} : p, \mathbf{y} : r\}.\end{aligned}$$

When we unzip τ_1, we get:

$$\mathrm{unzip}(\tau_1) = \{\mathbf{x} : \_oo\_\_\_p\_oo\_\_pp, \mathbf{y} : \_\_q\_\_rr\_\_q\_\_\_r\}.$$

While the trace segment τ_1 is a sequence of single-value valuations, upon unzipping it, we get a set with two (one for each variable) sequences (or strings) of valuations.

3.1 Syntax and semantics

Hypernode logic (HL) formulas are defined by the following grammar:

$$\psi ::= \epsilon \mid c \mid \mathbf{x}(\pi) \mid \psi.\psi \mid \psi + \psi \mid \psi^* \mid \lfloor \psi \rfloor$$
$$\varphi ::= \exists \pi\, \varphi \mid \neg\varphi \mid \varphi \wedge \varphi \mid \psi \leq \psi \mid \psi \precsim \psi$$

where π is a *trace variable* from a set of trace variables $\mathcal{V}$, $\mathbf{x}$ is a *program variable* from a set X, ϵ represents the empty word, c is a constant symbol from the domain Σ, $\lfloor . \rfloor$ is the stutter-reduction operator, $\leq$ is the prefix comparison on words, and $\precsim$ defines a prefixing relation on traces up to the removal of their stuttering steps. We refer to formulas defined by ψ in the grammar above as *trace formulas*. We say that a hypernode formula is *closed* when all trace variables are quantified.

We interpret hypernode formulas over a *trace assignments* $\Pi : \mathcal{V} \to (\Sigma^*)^X$ that maps trace variables to unzipped trace segments. We start by defining the semantic interpretation of trace formulas:

$$\Pi[\![c]\!] = \{c\} \qquad \Pi[\![\psi.\psi']\!] = \Pi[\![\psi]\!].\Pi[\![\psi']\!]$$
$$\Pi[\![\epsilon]\!] = \{\epsilon\} \qquad \Pi[\![\psi + \psi']\!] = \Pi[\![\psi]\!] \cup \Pi[\![\psi']\!]$$
$$\Pi[\![\mathbf{x}(\pi)]\!] = \{(\Pi(\pi))(\mathbf{x})\} \qquad \Pi[\![\psi^*]\!] = \bigcup_{n \in \mathbb{N}} \Pi[\![\psi]\!]^n$$
$$\Pi[\![\lfloor\psi\rfloor]\!] = \{\sigma_1\sigma_2\cdots\sigma_k \mid \sigma_1^+\sigma_2^+\cdots\sigma_k^+ \in \Pi[\![\psi]\!], \sigma_i \neq \sigma_{i+1} \text{ for } 1 \leq i < k\}.$$

Observe that $\Pi(\pi)$ returns an unzipped trace segment, which is a mapping from program variables to a sequence of values over the given domain. So, for an assignment Π s.t. $\Pi(\pi) = \tau$, the expression $(\Pi(\pi))(\mathbf{x})$ returns the $\mathbf{x}$-trace of τ; that is, $(\Pi(\pi))(\mathbf{x}) = \tau(\mathbf{x})$. The semantics of trace formulas corresponds precisely to regular expressions, apart from the interpretation of trace variables and the stutter-reduction operation, which have no counterparts in regular expressions. We may abuse the notation and use the stutter-reduction operator directly on words (for example, $\lfloor ppoqq \rfloor = poq$).

Example: Semantics of Stutter-reduction

Assume that $\Pi(\pi) = \text{unzip}(\tau_1)$, where τ_1 is from our previous example. Then, $\Pi[\![\lfloor\mathbf{x}(\pi)\rfloor]\!] = \{\_o\_p\_o\_p\}$.

The satisfaction relation $\models$ for a formula of hypernode logic is standard apart from the prefix operator. Because we allow choice and iteration, the evaluation of trace formulas defines a set of words, and thus the semantics for the prefix predicate must be extended to sets of words. The intended meaning for using choice in our regular expressions is to specify alternatives for pattern matching. Concretely, a formula of the form $\psi \leq \psi'$ requires the existence of a pair of words (one generated by ψ and the other by ψ'), such that both of them agree up to the point where the word from the interpretation of ψ terminates. As usual, $\Pi[\pi \mapsto \tau]$ denotes the update of Π where π is assigned to the trace τ and all other assignments remain the same. Altogether, the satisfaction relation is defined as:

$$
\begin{aligned}
&(\Pi, T) \models \exists \pi \varphi \text{ iff } \text{ there exists } \tau \in T \text{ s.t. } (\Pi[\pi \mapsto \tau], T) \models \varphi; \\
&(\Pi, T) \models \neg \varphi \text{ iff } (\Pi, T) \not\models \varphi; \\
&(\Pi, T) \models \varphi \wedge \varphi' \text{ iff } (\Pi, T) \models \varphi \text{ and } (\Pi, T) \models \varphi'; \\
&(\Pi, T) \models \psi \leq \psi' \text{ iff } \exists \tau \in \Pi[\![\psi]\!] \; \exists \tau' \in \Pi[\![\psi']\!] \; \tau \leq \tau'; \\
&(\Pi, T) \models \psi \precsim \psi' \text{ iff } (\Pi, T) \models \lfloor \psi \rfloor \leq \lfloor \psi' \rfloor.
\end{aligned}
$$

A set of traces T is a *model* of a hypernode formula φ, denoted $T \models \varphi$, iff there exists an assignment Π such that $(\Pi, T) \models \varphi$.

Example: Semantics of $\psi \leq \psi'$

We begin by looking into using regular patterns to express prefix requirements, as exemplified by the hypernode formula:

$$(p+q).o^{+}.p \leq \psi' \tag{1}$$

specifying that there exists a word in the interpretation of ψ' (i.e., $\Pi[\![\psi']\!]$ for a given assignment Π) with a prefix starting with p or q, followed by one or more letters o until it reaches the value p. The interpretation $\Pi[\![(p+q).o^{+}.p]\!]$ includes all sequences $p.o^{n}.p$ and $q.o^{n}.p$, for all $n \geq 1$ and all assignments Π. Then, for all sets of traces T:

$$
\begin{aligned}
\text{If } \Pi[\![\psi']\!] = \{qoopq\} &\implies (\Pi, T) \models (p+q).o^{+}.p \leq \psi' \\
\text{If } \Pi[\![\psi']\!] = \{qooq\} &\implies (\Pi, T) \not\models (p+q).o^{+}.p \leq \psi'
\end{aligned}
$$

Another use of regular patterns is to specify the maximal size of traces. For a finite domain $\{p, q, o\}$, we define that there exists a word in the interpretation of ψ with size at most four, with the third and fourth value (if reached) being p as:

$$\psi \leq (p+q+o).(p+q+o).p.p \tag{2}$$

Then, for all sets of traces T:

$$
\begin{aligned}
\text{If } \Pi[\![\psi]\!] = \{pop\} &\implies (\Pi, T) \models \psi \leq (p+q+o).(p+q+o).p.p \\
\text{If } \Pi[\![\psi]\!] = \{pqqp\} &\implies (\Pi, T) \not\models \psi \leq (p+q+o).(p+q+o).p.p
\end{aligned}
$$

Observe that one corollary of the existential semantics of prefixing is that the evaluation of $\psi \leq \psi'$ is monotonic for satisfaction when adding new words to the interpretation of formulas ψ and ψ'. This means that all extensions of satisfying interpretations $\Pi[\![\psi]\!]$ and $\Pi[\![\psi']\!]$ with new words also satisfy $\psi \leq \psi'$. We can see the monotonicity at play by extending our examples:

$$
\begin{aligned}
\text{If } \Pi[\![\psi']\!] = \{qoopq, qooq\} &\implies (\Pi, T) \models (p+q).o^{+}.p \leq \psi' \\
\text{If } \Pi[\![\psi]\!] = \{pop, pqqp\} &\implies (\Pi, T) \models \psi \leq (p+q+o).(p+q+o).p.p
\end{aligned}
$$

Although this monotonicity may initially seem counterintuitive, it becomes clear why we need the existential semantics once we plug in program variables.

Interpreting a program variable under a given assignment yields a word describing the progression of the variable's valuations over time. Using trace formulas, we can specify patterns over these values by requiring them to be in the language defined by a regular expression. Looking back at formulas (1) and (2), let's consider the case that $\psi = \psi' = \mathbf{x}(\pi)$. Then, for all T:

$$\begin{aligned}
\text{If } (\Pi(\pi))(\mathbf{x}) = pop &\implies (\Pi, T) \models (p+q).o^+.p \leq \mathbf{x}(\pi) \quad \text{and} \\
& (\Pi, T) \models \mathbf{x}(\pi) \leq (p+q+o).(p+q+o).p.p \\
\text{If } (\Pi(\pi))(\mathbf{x}) = pqqp &\implies (\Pi, T) \not\models (p+q).o^+.p \leq \mathbf{x}(\pi) \quad \text{and} \\
& (\Pi, T) \not\models \mathbf{x}(\pi) \leq (p+q+o).(p+q+o).p.p
\end{aligned}$$

Clearly, in these examples, we do not intend to say that the progression of values of $\mathbf{x}(\pi)$ extends (is a prefix of, resp.) *all* the words described by the regular expression; that is, interpreting $\leq$ as language inclusion up to prefixing is not adequate. This can be seen, for example, in the fact that $(p+q).o^+.p \leq \mathbf{x}(\pi)$ could never be satisfied as $\mathbf{x}(\pi)$ cannot start with p and q at the same time. The interaction between regular expressions and values of program variables becomes more interesting when we combine program variables with choice. For example:

$$(p+q).o^+.p \leq (\mathbf{x}(\pi) + \mathbf{y}(\pi)) \tag{3}$$

$$(\mathbf{x}(\pi) + \mathbf{y}(\pi)) \leq (p+q+o).(p+q+o).p.p \tag{4}$$

Then, if $(\Pi(\pi))(\mathbf{x}) = pop$ and $(\Pi(\pi))(\mathbf{y}) = pqqp$, for all sets T:

$$\begin{aligned}
&(\Pi, T) \models (p+q).o^+.p \leq (\mathbf{x}(\pi) + \mathbf{y}(\pi)) \\
&(\Pi, T) \models (\mathbf{x}(\pi) + \mathbf{y}(\pi)) \leq (p+q+o).(p+q+o).p.p
\end{aligned}$$

matches our intuition that we only need one of the variable's values to satisfy the regular expression. In fact, we can remove the choice operator by splitting the hypernode into a disjunction; that is, the following are equivalent to (3) and (4):

$$\begin{aligned}
&(p+q).o^+.p \leq \mathbf{x}(\pi) \lor \ (p+q).o^+.p \leq \mathbf{y}(\pi) \\
&\qquad \mathbf{x}(\pi) \leq (p+q+o).(p+q+o).p.p \ \lor \ \mathbf{y}(\pi) \leq (p+q+o).(p+q+o).p.p
\end{aligned}$$

We adopt the usual abbreviations: $\forall \pi\ \varphi \overset{\text{def}}{=} \neg(\exists \pi\ \neg\varphi)$, $\varphi \land \varphi' \overset{\text{def}}{=} \neg(\neg\varphi \lor \neg\varphi')$ and $\psi^+ \overset{\text{def}}{=} \psi.\psi^*$. We also introduce the notations $(\psi \equiv \psi') \overset{\text{def}}{=} (\psi \leq \psi' \land \psi' \leq \psi)$ and $(\psi \approx \psi') \overset{\text{def}}{=} (\psi \precsim \psi' \land \psi' \precsim \psi)$. Observe that it may happen that $\psi \equiv \psi'$ and yet $\Pi[\![\psi]\!] \cap \Pi[\![\psi']\!] = \emptyset$ for all trace assignments Π (and analogously for $\approx$). For example, $a + (b.b) \equiv b + (a.a)$ is valid (i.e., all sets of traces are a model of this formula) because $a \leq aa$ and $b \leq bb$; however, $\Pi[\![a + (b.b)]\!] \cap \Pi[\![b + (a.a)]\!] = \{a, bb\} \cap \{b, aa\} = \emptyset$, for all trace assignments Π.

Example: Mixing Synchronous and Asynchronous

Hypernode formulas allow mixing synchronous and asynchronous hyperproperties with regular constraints on variable valuations, as exemplified below:

$$\forall\pi\ \big(0^*.1.0^*.1 \leq \mathbf{flag}(\pi) \rightarrow \exists\pi'(\mathbf{flag}(\pi') \precsim 0 \wedge \mathbf{x}(\pi) \approx \mathbf{x}(\pi'))\big). \tag{5}$$

This formula requires that for all program executions where **flag** is raised at least twice, there must exist another execution where **flag** was never raised (the value remains 0) s.t. these two executions coincide on values of $\mathbf{x}$ modulo its stuttering. This property can be interpreted as a quality-of-service requirement: for any run of a system that experiences two or more failure notifications, it must be possible to restart it so that it behaves as if no failures had occurred.

In (5), we can see how assumptions on different formulas can play a role in whether to use the synchronous, $\leq$, or asynchronous, $\precsim$, comparisons. The program variable **flag** is assumed to be 1 only at the time the notification is sent, and otherwise 0. Therefore, to count whether there were at least two notifications, we need to use the synchronous ($\leq$) comparison. For example, traces τ_1 and τ_2 with **flag** values $\tau_1(\mathbf{flag}) = 010$ and $\tau_2(\mathbf{flag}) = 0110$ represent a different number of failure notifications; however, when stutter-reduced, they are indistinguishable (i.e., they are both interpreted as 010). On the other hand, to compare sequences of values of the program variable $\mathbf{x}$ that can come from different runs (that can have different lengths and progress at different speeds), we want to use asynchronous comparison ($\precsim$). For example, for two unzipped traces τ and τ' with $\tau(\mathbf{x}) = aaabcbba$ and $\tau'(\mathbf{x}) = abcbaa$ assigned to π and π' in Π, we have $(\Pi, T) \models \mathbf{x}(\pi) \approx \mathbf{x}(\pi')$, for all T.

3.2 Model-checking problem

We are interested in solving the model-checking problem for hypernode logic over Kripke structures. A *Kripke structure* is a tuple $K = (W, \Sigma^X, \Delta, V)$ consisting of a finite set W of worlds, an alphabet which is a set of assignments of variables from X to a finite domain Σ, a transition relation $\Delta \subseteq W \times W$, and a value assignment $V : (W \times X) \rightarrow \Sigma$ that assigns a value from the finite domain Σ to each variable in each world.

In this work, we consider an extension of Kripke structures with sets of *entry* and *exit worlds*, specifying where paths of the model begin and end, respectively. This extension becomes relevant later (Section 6.2) when we define the model-checking of hypernode automata. Formally, an *open Kripke structure* consists of a Kripke structure $K = (W, \Sigma^X, \Delta, V)$, and a pair $\mathbb{W} = (W_{\text{in}}, W_{\text{out}})$ where $W_{\text{in}} \subseteq W$ is the set of entry worlds while $W_{\text{out}} \subseteq W$ is the set of exit worlds.

A *path* of the open Kripke structure $(K, \mathbb{W})$ is a sequence of worlds $w_0 \dots w_n$ in K that starts in an entry world, $w_0 \in W_{\text{in}}$, ends in an exit world, $w_n \in W_{\text{out}}$, and respects the transition relation, $(w_i, w_{i+1}) \in \Delta$ for all $0 \leq i < n$. We write $\text{Paths}(K, \mathbb{W})$ for the set of paths of the open Kripke structure $(K, \mathbb{W})$ and $\text{Traces}(K, \mathbb{W})$ to define all traces defined by the paths in $\text{Paths}(K, \mathbb{W})$, i.e.,

$$\text{Traces}(K, \mathbb{W}) = \{V(w_0)V(w_1)... \mid w_0w_1... \in \text{Paths}(K, \mathbb{W})\}.$$

While, the *set of unzipped trace segments generated by* $(K, \mathbb{W})$ is:

$$\mathrm{Unzip}(K, \mathbb{W}) = \{\mathbf{x} : V(w_0, x) \ldots V(w_n, x) \mid w_0 w_1 \ldots \in \mathrm{Paths}(K, \mathbb{W}) \text{ and } \mathbf{x} \in X\}.$$

Model-checking problem for hypernode logic
Let $(K, \mathbb{W})$ be an open Kripke structure, and φ a formula of hypernode logic over the same set of variables X and finite domain Σ. Is the set of unzip trace segments generated by $(K, \mathbb{W})$ a model for φ; that is, $\mathrm{Unzip}(K, \mathbb{W}) \models \varphi$?

We prove next that the model-checking problem for hypernode logic is undecidable, even when restricted to the fragment with only stutter-prefixing comparisons and without quantifier alternation.

3.2.1 Undecidability of model-checking hypernode logic

We prove that model-checking hypernode formulas over Kripke structures is undecidable by presenting a reduction from the Post Correspondence Problem (PCP) [28].

An instance $\mathcal{I}$ of the PCP is a finite collection of dominoes:

$$\mathcal{I} = \left\{ \left[\frac{u^1}{d^1}\right], \left[\frac{u^2}{d^2}\right], \ldots, \left[\frac{u^n}{d^n}\right] \right\}$$

where $n \geq 1$, and u^i and d^i are finite words over an alphabet Σ, for $i \leq n$. A solution (or match) for a given instance $\mathcal{I}$ is a non-empty sequence of indices $i_1, \ldots, i_m$ such that $u^{i_1}.u^{i_2}.\ldots.u^{i_m} = d^{i_1}.d^{i_2}.\ldots.d^{i_m}$. The problem of finding a solution for a given PCP instance is undecidable [28].

Theorem 1 *Let $\exists^2 HL(\precsim)$ be the fragment of hypernode logic supporting only asynchronous comparison of traces and having at most two existential quantifiers; that is, formulas $\exists\pi\, \exists\pi'\ \varphi_{\precsim}$ where π and π' are trace variables, and $\varphi_{\precsim}$ is defined by the following grammar:*

$$\varphi_{\precsim} ::= \mathbf{x}(\pi'') \precsim \mathbf{x}(\pi'') \mid \neg\varphi_{\precsim} \mid \varphi_{\precsim} \wedge \varphi_{\precsim}$$

where $\pi'' \in \{\pi, \pi'\}$ and $\mathbf{x}$ is a program variable. Model checking $\exists^2 HL(\precsim)$ formulas over open Kripke structures is undecidable.

Proof Given an instance of the PCP problem $\mathcal{I}$, we translate it to an equivalent instance that avoids removing letters when stutter-reducing them. In a nutshell, we define $\mathcal{I}_\$$ as the instance that adds the fresh symbol to the right of each letter. For example, given a domino $\left[\frac{aa}{abb}\right]$, in $\mathcal{I}_\$$ this domino is translated into: $\left[\frac{a\$a\$}{a\$b\$b\$}\right]$. Clearly, the instance $\mathcal{I}_\$$ has a solution iff $\mathcal{I}$ also has a solution.

The first step of our proof is to build an open Kripke structure encoding all possible combinations of the two sides of the dominoes. Given an instance $\mathcal{I}$ of the PCP problem over a domain Σ, we build a Kripke structure over the variables:

- $\mathbf{side}$ defining whether the current letter is from the upper or down side of the domino by assigning either 0 or 1 to it;

- $\mathbf{letter}$ defining the current letter from domain Σ;
- $\mathbf{domino}$ determining which domino the letter belongs to.

For an arbitrary domino $D_i \in \mathcal{I}$ where $D_i = \left[\frac{u^i}{d^i}\right]$ with $u^i = u^i_1, \dots, u^i_j$ and $d^i = d^i_1, \dots d^i_k$, we define the Kripke structure $K_{D_i} = (W_i, \Sigma_i^X, \Delta_i, V_i)$ where:

- there is a world for each possible letter in both sides: $W_i = \{u^i_1, \dots, u^i_j, d^i_1, \dots d^i_k\}$;
- $\Sigma_i = \{0,1\} \cup \{i, \bar{i}\} \cup \Sigma$ and $X = \{\mathbf{side}, \mathbf{letter}, \mathbf{domino}\}$;
- the transition relation connects the letters as specified by the instance: $\Delta_i = \{(u^i_l, u^i_{l+1}) \mid 1 \le l < j\} \cup \{(d^i_l, d^i_{l+1}) \mid 1 \le l < k\}$;
- the valuation of $\mathbf{side}$ and $\mathbf{letter}$ are:
  - for $1 \le l \le j$, $V_i(u^i_l, \mathbf{side}) = 0$, $V_i(u^i_l, \mathbf{letter}) = u^i_l$;
  - for $1 \le l \le k$, $V_i(d^i_l, \mathbf{side}) = 1$, $V_i(d^i_l, \mathbf{letter}) = d^i_l$;
- and the valuation of $\mathbf{domino}$ is the name of domino except for the last step that is assigned the overline name:
  - for $1 \le l < j$, $V_i(u^i_l, \mathbf{domino}) = i$ and $V_i(u^i_j, \mathbf{domino}) = \bar{i}$;
  - for $1 \le l < k$, $V_i(d^i_l, \mathbf{domino}) = i$ and $V_i(d^i_k, \mathbf{domino}) = \bar{i}$.

From now on $W_i, \Sigma_i^X, \Delta_i$ and V_i refer to the states, domain, transition and labels of the Kripke structure defined by the i^{th} domino in a given instance $\mathcal{I}$.

We encode a PCP instance $\mathcal{I}$ with n dominoes as an open Kripke structure $(K_{\mathcal{I}}, \mathbb{W}_{\mathcal{I}}) = ((W_{\mathcal{I}}, \Sigma_{\mathcal{I}}^X, \Delta_{\mathcal{I}}, V_{\mathcal{I}}), (W_{\text{in}}, W_{\text{out}}))$ where:

- $W_{\mathcal{I}} = \bigcup\limits_{i \le n} W_i$, $\Sigma_{\mathcal{I}} = \bigcup\limits_{i \le n} \Sigma_i$, $X = \{\mathbf{side}, \mathbf{letter}, \mathbf{domino}\}$;
- $\Delta_{\mathcal{I}} = \bigcup\limits_{i \le n} \Delta_i \cup \{(u^i_{|u^i|}, u^l_1) \mid i, l \le n\} \cup \{(d^i_{|d^i|}, d^l_1) \mid i, l \le n\}$, which additionally connects the last letters of each domino of each side with all initial letters of all dominoes of the same side;
- $V_{\mathcal{I}} = \bigcup\limits_{i \le n} V_i$;
- the set of initial worlds are all first letters of a domino: $W_{\text{in}} = \bigcup\limits_{i \le n} \{u^i_1, d^i_1\}$; and
- the set of finial worlds are all the last letters of a domino: $W_{\text{out}} = \bigcup\limits_{i \le n} \{u^i_{|d^i|}, d^i_{|u^i|}\}$.

Given a PCP instance $\mathcal{I}$ and its extension with \, the open Kripke structure derived from $\mathcal{I}_\$$, $(K_{\mathcal{I}_\$}, \mathbb{W}_{\mathcal{I}_\$})$, has the following properties:

- *Each trace has either only the upper or the down side of dominoes*: for all $\tau \in \text{Unzip}(K_{\mathcal{I}_\$}, \mathbb{W}_{\mathcal{I}_\$})$, $\lfloor \tau(\mathbf{side} \rfloor \in \{0,1\}$;
- *Each trace is well-defined*: that is, for all $\tau \in \text{Unzip}(K_{\mathcal{I}_\$}, \mathbb{W}_{\mathcal{I}_\$})$ with $\lfloor \tau(\mathbf{domino}) \rfloor = i_1 \bar{i}_1 \dots i_m \bar{i}_m$, if $\lfloor \tau(\mathbf{side}) \rfloor = 0$ then $\tau(\mathbf{letter}) = u^{i_1} \dots u^{i_m}$, and if $\lfloor \tau(\mathbf{side}) \rfloor = 1$ then $\tau(\mathbf{letter}) = d^{i_1} \dots d^{i_m}$;
- *Stuttering removal does not remove repeated dominoes*: for all $\tau, \tau' \in \text{Unzip}(K_{\mathcal{I}_\$}, \mathbb{W}_{\mathcal{I}_\$})$

with the same side, $\lfloor \tau(\mathbf{side}) \rfloor = \lfloor \tau'(\mathbf{side}) \rfloor$, if they have the same sequence of dominoes after stutter-reducing them, $\lfloor \tau(\mathbf{domino}) \rfloor = \lfloor \tau'(\mathbf{domino}) \rfloor$, then they should have had the same sequence of dominoes before the reduction, $\tau(\mathbf{domino}) = \tau'(\mathbf{domino})$;

- *There are no stuttering letters because they are all surrounded with the fresh symbol* : for all $\tau \in \mathrm{Unzip}(K_{\mathcal{I}_\$}, \mathbb{W}_{\mathcal{I}_\$})$, $\tau(\mathbf{letter}) = \lfloor \tau'(\mathbf{domino}) \rfloor$.

Using these properties, we can prove that an instance $\mathcal{I}$ has a solution iff:

$$\mathrm{Unzip}(K_{\mathcal{I}_\$}, \mathbb{W}_{\mathcal{I}_\$}) \models \exists \pi \exists \pi' \ \mathbf{side}(\pi) \not\approx \mathbf{side}(\pi') \wedge \mathbf{letter}(\pi) \approx \mathbf{letter}(\pi') \wedge \mathbf{domino}(\pi) \approx \mathbf{domino}(\pi').$$

□

Our reduction from PCP leverages the ability to mix synchronous and asynchronous reasoning within the asynchronous fragment of hypernode logic by allowing some variables to evolve fully asynchronously while others remain synchronized. Later in this work, we show that when restricting to structures where variables can be analyzed independently, the model-checking problem for asynchronous hypernode logic becomes decidable.

3.2.2 Decidable hypernode logic fragments

In what follows, we present a model-checking algorithm for a synchronous and an asynchronous fragment of hypernode logic. In both fragments, we allow only comparison between either two program variables or a program variable and a regular language defined by a regular expression over constants and the empty word. We refer to this restriction on trace formulas as the *regular fragment*, which we often denote by Reg. For both the synchronous and the asynchronous fragments, we allow for arbitrary quantifier alternation. The only difference lies in the relation used for the comparisons: the synchronous comparison is only up to prefixing, while the asynchronous one also removes repeated valuations We refer to these fragments as $\mathrm{HL}(\mathrm{Reg}, \leq)$ and $\mathrm{HL}(\mathrm{Reg}, \precsim)$ and summarize our results in Table 3.

Kripke structures are natural models for the synchronous interpretation of hypernodes. However, for the asynchronous interpretation, it is often more appropriate to allow less synchronization between variables, enabling them to evolve independently. We present an algorithm to model-check asynchronous hypernode formulas over stutter-reducible Kripke structures. Informally, a Kripke structure is stutter-reducible if it can be transformed into an automaton where variables can be stuttered independently while defining the same set of stutter-free traces as the original Kripke structure. We present, in Section 6.3, models from the literature that naturally define stutter-reducible Kripke structures.

Table 3 Summary of decidability results

	All Kripke Structures	Stutter-reducible Kripke Structures
$\mathrm{HL}(\mathrm{Reg}, \leq)$	Decidable (Thm. 12)	Decidable (Thm. 12)
$\mathrm{HL}(\mathrm{Reg}, \precsim)$	Undecidable (Thm. 1)	Decidable (Thm. 21)

4 Model checking synchronous hypernode logic

In general, model checking for hypernode formulas is undecidable. In this section, we introduce a decidable fragment of hypernode logic, referred to as the *regular synchronous* fragment. This fragment allows only synchronous comparisons between either two program variables or a program variable and a regular language defined by a regular expression over constants and the empty word. This fragment is defined by the following grammar:

$$\begin{aligned} \psi &::= \epsilon \mid c \mid \psi.\psi \mid \psi + \psi \mid \psi^* \\ \varphi &::= \exists \pi\, \varphi \mid \neg\varphi \mid \varphi \wedge \varphi \mid \mathbf{x}(\pi) \leq \psi \mid \psi \leq \mathbf{x}(\pi) \mid \mathbf{x}(\pi) \leq \mathbf{x}(\pi) \end{aligned} \tag{6}$$

We observe that, in this fragment, trace formulas do not allow composing program variables with other trace formulas. This captures the restriction in which constraints on program variables are defined by regular languages or another program variable. To simplify the grammar, we removed program variables from trace formulas and use them explicitly only within comparisons.

In the rest of this section, we show how to model-check this fragment of HL using constructions on *left-aligned (multi-tape) automata*, introduced next.

4.1 Left-aligned automata

A multi-tape automaton [18, 23] is a finite-state automaton with $n \geq 1$ heads reading *n* tapes (input words). If heads read the tapes in lock-step, we call the automaton *synchronous* or *0-synchronized* (where 0 refers to how much the reading heads diverge). Synchronous multi-tape automata are identical to *(aligned) multi-track* automata [23, 34], where an automaton reads a single tape over an *n*-track alphabet $(\Sigma \cup \{\lambda\})^n$ (with $\lambda \notin \Sigma$). Symbol λ is a special symbol that means "read nothing" and is used as a padding symbol when a tape finishes (in the case of aligned automata).

The algorithms presented in this work are based on multi-track automata. Nonetheless, we refer to them as *left-aligned automata* because we want the (left) alignment requirement to be explicit and *syntactic*. For multi-track automata, being aligned is a common convention rather than a requirement, and, mainly, the alignment is achieved by aligning the input words (it is not a condition on the automata, which is our case). Having alignment as a condition on the automata is helpful when further restricting them to *stutter-free automata*. Another difference to multi-track automata is that our automata read value assignments instead of tuples of values. However, both aligned multi-track automata and left-aligned automata are equivalent. Because most of the results about multi-track automata in the literature are hand-waved, we build the theory behind left-aligned automata in this section. The goal is to prepare the ground for stutter-free automata, which will be presented in the next section.

In the rest of the text, $\Lambda(X, \Sigma)$ denotes the alphabet which is comprised of all assignments of variables in X to values in Σ or the padding symbol $\# \notin \Sigma$ (where at least one variable is assigned a value from Σ). Formally, for $X = \{x_0, \ldots, x_m\}$:

$$\begin{aligned} \Lambda(X, \Sigma) &= (\Sigma \cup \{\#\})^X \setminus \{x_0 : \#, \ldots, x_m : \#\} \\ &= \{x_0 : \sigma_0, \ldots, x_m : \sigma_m \mid \forall 0 \leq i \leq m \text{ we have } \sigma_i \in \Sigma \cup \{\#\}\} \setminus \{x_0 : \#, \ldots, x_m : \#\}. \end{aligned}$$

If X and Σ are clear from the context or are not important, we write simply Λ instead of $\Lambda(X, \Sigma)$. We write $\Lambda(X)$ to stress that the alphabet uses variables from X. We do not consider transitions in which all variables are terminated, because later we define the accepted language of an LAA as all words accepted by it, with the padding symbol # removed. If we had included assignments of the form $\{x_0 : \#, \ldots, x_m : \#\}$ in our alphabet, the removal of #-symbols would break the one-to-one correspondence between words with and without it.

Given a nondeterministic finite automaton with the transition relation $\delta : Q \times \Lambda \times Q$, the set $In(q, x)$ is the set of all x-valuations incoming to state q, and $Out(q, x)$ is the set of all x-valuations outgoing from state q; formally:

$$
\begin{aligned}
In(q, x) &= \{v(x) \mid (q', v, q) \in \delta \textit{ for some } q' \in Q\} \\
Out(q, x) &= \{v(x) \mid (q, v, q') \in \delta \textit{ for some } q' \in Q\}.
\end{aligned}
$$

A left-aligned automaton is a nondeterministic finite state automaton over the alphabet Λ, where the # symbol on an x-trace cannot be followed by anything else than # (for any x).

Definition 1 (Left-aligned automaton) Let X be a finite set of variables over Σ. A nondeterministic *left-aligned automaton* (LAA) over the alphabet Λ is a tuple $\mathcal{A} = (Q, \hat{Q}, F, \delta)$ with a finite set Q of states, a set $\hat{Q} \subseteq Q$ of initial states, a set $F \subseteq Q$ of final states, and a transition relation $\delta \subseteq Q \times \Lambda \times Q$ that for all states $q \in Q$ and variables $x \in X$ satisfies the *termination* condition:

$$
\text{if } \# \in In(q, x) \text{ then } Out(q, x) = \{\#\}.
$$

The condition on the transition relation ensures that once a variable x is assigned the termination symbol # on a transition entering a state q, every transition leaving q must assign # to x. This prevents assigning non-termination values to x on states reachable from q and thereby guarantees the left-alignedness of variable assignments.

A *run* of an LAA $\mathcal{A}$ is a finite non-empty sequence $q_0 v_0 q_1 v_1 \ldots v_{n-1} q_n$ alternating between states and variable assignments starting with an initial state, $q_0 \in \hat{Q}$, and following $\mathcal{A}$'s transition function δ (i.e., $(q_i, v_i, q_{i+1}) \in \delta$ for all $i < n$). As usual, a run is *accepting* if it ends in a final state. A *path* induced by a run $r = q_0 v_0 q_1 v_1 \ldots v_{n-1} q_n$ is the sequence of states $q_0 q_1 \ldots q_n$. The *trace* of the run r, denoted $trace(r)$, is the sequence of variable assignments $v_0 v_1 \ldots v_{n-1}$.

Given variables $X = \{x_0, \ldots, x_m\}$ with domain Σ and an unzipped trace segment $\tau = \{x_0 : w_0, \ldots, x_m : w_m\}$ with $w_0, \ldots, w_m \in \Sigma^*$, an LAA $\mathcal{A}$ over $\Lambda(X, \Sigma)$ accepts this unzipped trace segment if there exists an accepting run $q_0 v_0 q_1 v_1 \ldots v_{n-1} q_n$ of $\mathcal{A}$ s.t.

$$
v_i(x) = \begin{cases} \tau(x)[i] & \text{if } i < |\tau(x)| \\ \# & \text{else} \end{cases}
$$

In other words, an unzipped trace segment is accepted by $\mathcal{A}$ if there exists an accepting run in $\mathcal{A}$ where for all x, the sequence of assignments for x on the run defines the x-trace in τ, modulo possible padding with #. The *language* of $\mathcal{A}$, denoted $\mathcal{L}(\mathcal{A})$, is the set of all unzipped trace segments accepted by $\mathcal{A}$.

Example: Left-aligned Automaton

In Fig. 3 below, we depict a left-aligned automaton that accepts all unzipped trace segments for the boolean variables $\{x, y, z\}$, where all traces start with 0 and the *x*-trace is a prefix of the *y*-trace, which is a prefix of the *z*-trace. The language accepted by the automaton $\mathcal{A}$ from Fig. 3 is:

$$\mathcal{L}(\mathcal{A})=\{x:\tau_x, y:\tau_y, z:\tau_z \mid \tau_x, \tau_y, \tau_z \in 0.\{0,1\}^* \text{ and } \tau_x \leq \tau_y \leq \tau_z\}.$$

The union, intersection, and determinization for LAA are as usual for NFA. Let $\mathcal{A}_1=(Q_1, \hat{Q}_1, F_1, \delta_1)$ and $\mathcal{A}_2=(Q_2, \hat{Q}_2, F_2, \delta_2)$ be left-aligned automata over the same alphabet Λ. We define:

- their *union* as $\mathcal{A}_1 \cup \mathcal{A}_2 = (Q_1 \uplus Q_2, \hat{Q}_1 \uplus \hat{Q}_2, F_1 \uplus F_2, \delta_\cup)$ over the same alphabet Λ where $\uplus$ is the disjoint union, and $\delta_\cup(q) = \delta_i(q)$ when $q \in Q_i$ with $i \in \{1, 2\}$;
- their *intersection* as $\mathcal{A}_1 \cap \mathcal{A}_2 = (Q_1 \times Q_2, \hat{Q}_\cap, F_\cap, \delta_\cap)$ over the same alphabet Λ where $\hat{Q}_\cap = \{(q_1, q_2) \mid q_1 \in \hat{Q}_1 \text{ and } q_2 \in \hat{Q}_2\}$, $F_\cap = \{(q_1, q_2) \mid q_1 \in F_1 \text{ and } q_2 \in F_2\}$ and $\delta_\cap((q_1, q_2), l) = (q_1', q_2')$ iff $\delta_1(q_1, l) = q_1'$ and $\delta_2(q_2, l) = q_2'$; and
- the *determinization* of $\mathcal{A}_1$ as $\det(\mathcal{A}_1) = (2^{Q_1}, \hat{Q}_1, F_d, \delta_d)$ where $F_d=\{S \in 2^{Q_1} \mid S \cap F_1 \neq \emptyset\}$ and $\delta_d(S, v) = \bigcup_{q \in S} \delta_1(q, v)$ with $v \in \Lambda$.

We prove below that LAA are closed under the operators defined above.

Proposition 2 *Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be two LAA over the same alphabet Λ. Both $\mathcal{A}_1 \cup \mathcal{A}_2$ and $\mathcal{A}_1 \cap \mathcal{A}_2$ are LAA over Λ with $\mathcal{L}(\mathcal{A}_1 \cup \mathcal{A}_2)=\mathcal{L}(\mathcal{A}_1) \cup \mathcal{L}(\mathcal{A}_2)$ and $\mathcal{L}(\mathcal{A}_1 \cap \mathcal{A}_2)=\mathcal{L}(\mathcal{A}_1) \cap \mathcal{L}(\mathcal{A}_2)$, resp. Moreover, for a nondeterministic LAA $\mathcal{A}$, its determinization, $det(\mathcal{A})$, defines a deterministic LAA automaton accepting the same language, that is, $\mathcal{L}(det(\mathcal{A}))=\mathcal{L}(\mathcal{A})$.*

Proof We observe that union and intersection do not affect the termination condition. Then, the result follows from the fact that LAA are NFA.

Let $\mathcal{A}_1$ be an arbitrary LAA.

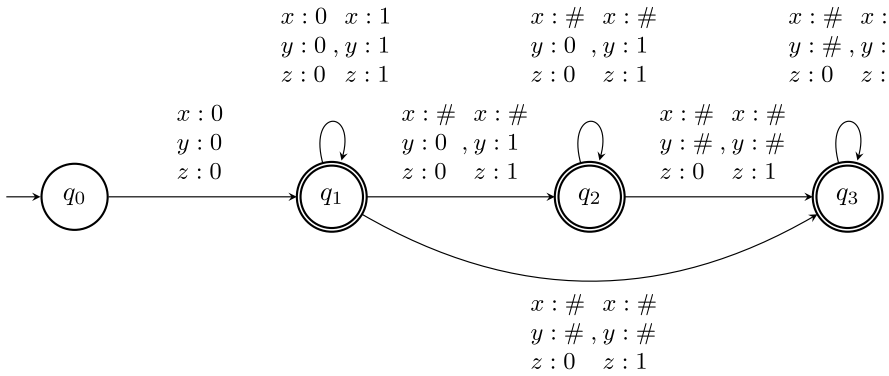


Fig. 3 Left-aligned automaton $\mathcal{A}$ with three traces over the boolean domain. The automaton accepts traces that all start with 0, x-trace is a prefix of *y*-trace, and *y*-trace is a prefix of z-trace

We show that $\det(\mathcal{A}_1)$ satisfies the termination condition and thus is also LAA. For any state in $\det(\mathcal{A}_1)$ with no predecessors, the termination condition is trivially satisfied. Now assume a state S' that has a predecessor and $\# \in In(S', x)$ (otherwise the implication is satisfied). $\# \in In(S', x)$ only if there exists a state S and $v \in \Lambda$ with $v(x) = \#$, s.t. $\delta_d(S, v) = S'$. By definition, $S' = \bigcup_{q \in S} \delta_1(q, v)$. Because $\mathcal{A}_1$ is LAA and $v(x) = \#$, then for any $q' \in \delta_1(q, v)$ of any $q \in S$ it holds that $Out(q', v) = \{\#\}$. Therefore $Out(q_s, v) = \{\#\}$ for any $q_s \in S'$ and consequently $Out(S', v) = \{\#\}$.

The fact that $\mathcal{L}(\det(\mathcal{A}_1)) = \mathcal{L}(\mathcal{A}_1)$ follows directly from the same result for NFA. □

To complement LAA, we follow the same approach as for NFA: we start by transforming it into an equivalent deterministic automaton, followed by completing it, and, in a final step, we swap the role of final and non-final states. The only challenging step is completing the automaton, as we must ensure that the completed automaton satisfies the termination condition. The termination condition prevents us from adding a non-final sink state with a self-loop to which we connect all missing transitions. The solution we present here introduces *universal left-aligned automata* to serve as such a sink element.

4.1.1 Completing and complementing left-aligned automata

For any LAA $(Q, \hat{Q}, F, \delta)$ and a state $q \in Q$, we define two functions $allowed : Q \to 2^{\Lambda(X)}$ returning all letters from the alphabet allowed to be on outgoing transitions of a given state, and $\Lambda_{out}(q)$ returning all letters that are actually on outgoing transitions of a given state:

$$\begin{aligned} allowed(q) &= \{v \in \Lambda \mid \forall x \in X : \# \in In(q, x) \implies v(x) = \#\} \\ \Lambda_{out}(q) &= \{v \in \Lambda \mid (q, v, q') \in \delta \text{ for some } q' \in Q\}. \end{aligned}$$

A left-aligned automaton $\mathcal{A} = (Q, \hat{Q}, F, \delta)$ over Λ is *complete* iff for each $q \in Q$ it holds that $allowed(q) = \Lambda_{out}(q)$. The *universal* left-aligned automaton $\mathcal{U}_\Lambda$ over Λ is a deterministic and complete automaton, accepting all left-aligned unzipped traces over Λ. That is, the $\mathcal{U}_{\Lambda(X,\Sigma)}$ accepts the language $\mathcal{L}(\mathcal{U}_{\Lambda(X,\Sigma)}) = \{\{x_0 : u_0, \ldots, x_k : u_k\} \mid u_0, \ldots, u_k \in \Sigma^*\}$ where $\{x_0, \ldots, x_k\} = X$, as proved in Proposition 4. In a nutshell, states of $\mathcal{U}_{\Lambda(X,\Sigma)}$ are tuples where each variable in X is assigned either the label q or $q_\#$, signaling for each variable whether it can still be assigned to values from the domain (label q) or the trace for the variable is already terminated and the variable can only be assigned to the value $\#$ (label $q_\#$).

Definition 2 Let $X = \{x_0, \ldots, x_m\}$ be a set of variables over the finite domain Σ. The *universal left-aligned automaton* over $\Lambda(X, \Sigma)$ is $\mathcal{U}_\Lambda = (Q_\mathcal{U}, Q_\mathcal{U}, Q_\mathcal{U}, \delta_\mathcal{U})$, where $Q_\mathcal{U} = \{q, q_\#\}^X \setminus \{\{x_0 : q_\#, \ldots, x_m : q_\#\}\}$, and $s \xrightarrow{v} s' \in \delta_\mathcal{U}$ iff $v \in \Lambda(X, \Sigma)$ and, for all $0 \le i \le m$:

1) if $s(x_i) = q_\#$ then $v(x_i) = \#$, and
2) if $v(x_i) = \#$ then $s'(x_i) = q_\#$.

Proposition 3 *The automaton $\mathcal{U}_\Lambda$ is left-aligned and complete.*

Proof Assume there is a state s' with a predecessor state s, s.t. $(s, v, s') \in \delta_{\mathcal{U}}$ and $v(x) = \#$. Then by point 2) in the definition of universal LAA, $s'(x) = q_{\#}$. This implies that for any outgoing edge with valuation v', it holds that $v'(x) = \#$ (point 1) in the definition. So $\mathcal{U}_\Lambda$ is left-aligned.

The automaton is complete. Condition 1) defines all outgoing edges of a state s. If $s(x_i) = q_{\#}$, then by condition 2), it means that $\# \in In(s, x_i)$. At the same time $\sigma'_i = \#$, so the set of outgoing transitions is a subset of $allowed(s)$. It is also a superset, because if $\# \in In(s, x_i)$, then $s(x_i) = q_{\#}$ which implies that $\sigma'_i = \#$. □

Proposition 4 *The automaton* $\mathcal{U}_{\Lambda(X,\Sigma)}$ *accepts the universal language, i.e.,* $\mathcal{L}(\mathcal{U}_{\Lambda(X,\Sigma)}) = \{\{x_0 : w_0, \ldots, x_m : w_m\} \mid x_0, \ldots, x_m \in X, w_0, \ldots, w_m \in \Sigma^*\}$.

Proof Follows from the completeness and the definition of the transition function. □

Example: Universal Left-aligned Automaton

In Fig. 4 below, we depict the universal left-aligned automaton $\mathcal{U}_X$ for the set of boolean variables $X = \{x, y\}$.

To define complete left-aligned automata, we define a helper function $\mu : \Lambda \to \{q, q_{\#}\}^X$ that takes an assignment from Λ and maps it into an assignment from $\{q, q_{\#}\}^X$ s.t. $x : \sigma$ gets mapped to $x : q_{\#}$ if and only if $\sigma = \#$:

$$\mu(s)(x) = \begin{cases} q & \text{if } s(x) \neq \# \\ q_{\#} & \text{if } s(x) = \# \end{cases}$$

Note that the transition function $\delta_{\mathcal{U}}$ of the universal LAA with variables X then can be defined as $\delta_{\mathcal{U}}(s, \sigma) = \mu(\sigma)$ for s and σ s.t.: if $s(x_i) = q_{\#}$ then $\sigma = \#$.

Definition 3 Let $\mathcal{A} = (Q, \hat{Q}, F, \delta)$ be a left-aligned automaton over the set of variables X with domain Σ.

The *completion* of $\mathcal{A}$ defines the left-aligned automaton $\text{complete}(\mathcal{A}) = (Q \uplus Q_{\mathcal{U}}, \hat{Q}, F, \delta')$ over the same variables, where $Q_{\mathcal{U}}$ are the states of the universal left-aligned automaton $\mathcal{U}_\Lambda$, and δ' is defined as

$$\delta' = \delta \cup \delta_{\mathcal{U}} \cup \{(q \xrightarrow{v} \mu(v)) \mid \text{if } v \in (allowed(q) \setminus \Lambda_{out}(q)), q \in Q\}$$

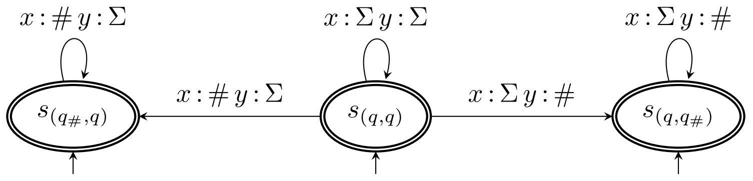


Fig. 4 The universal left-aligned automaton $\mathcal{U}_\Lambda$ over the boolean variables $X = \{x, y\}$. Σ instead of a letter is a shortcut for an arbitrary symbol from $\{0, 1\}$. The automaton accepts all left-aligned unzipped trace segments over $\{x, y\}$. All states are both initial and final

where $\delta_{\mathcal{U}}$ is the transition relation of the universal left-aligned automaton $\mathcal{U}_\Lambda$.
We can now prove that our completion procedure returns an automaton accepting the same language as the input automaton.

Proposition 5 *Let $\mathcal{A}$ be an LAA. Then,* $\mathrm{complete}(\mathcal{A})$ *is a complete LAA with the same language as $\mathcal{A}$, i.e., $\mathcal{L}(\mathrm{complete}(\mathcal{A})) = \mathcal{L}(\mathcal{A})$.*

Proof Consider an arbitrary left-aligned automaton $\mathcal{A}$. Both $\mathcal{A}$ and the universal LAA are left-aligned automata. We have to prove that the extension of the $\mathcal{A}$ transition relation preserves the termination requirement and that it completes $\mathcal{A}$.

For each state $q \in Q$, we extend the transition relation with edges $q \xrightarrow{v} \mu(v)$ s.t. $v \in (\mathit{allowed}(q) \setminus \Lambda_{out}(q))$. By the definition of $\mathit{allowed}$, adding such an edge preserves the termination requirement.

We now prove that $\mathrm{complete}(\mathcal{A})$ is complete. By the definition of universal automaton, it follows that universal automata are complete. For all non-complete transitions in $\mathcal{A}$ (i.e., variables that are not terminated yet), we add the missing transition pointing to the matching universal automaton state; hence, $\mathrm{complete}(\mathcal{A})$ is complete.

Finally, we prove that both automata define the same language. As we kept all $\mathcal{A}$ transitions and final states, it follows that $\mathcal{L}(\mathrm{complete}(\mathcal{A})) \supseteq \mathcal{L}(\mathcal{A})$. We now prove that $\mathcal{L}(\mathrm{complete}(\mathcal{A})) \subseteq \mathcal{L}(\mathcal{A})$. First, we note that no transition connects states from the universal automaton to states from $\mathcal{A}$. Then, when a run reaches a state from the universal automaton, all the subsequent steps are within the universal automaton. Additionally, the final states do not include states from the universal automaton. Thus, accepting runs include only transitions in $\mathcal{A}$. □

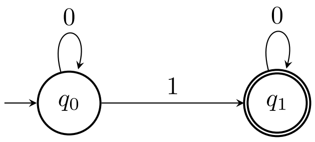
Finally, to complement a deterministic and complete left-aligned automaton, we just flip final with non-final states, i.e., the complement of $\mathcal{A} = (Q, \hat{Q}, F, \delta)$ is $\overline{\mathcal{A}} = (Q, \hat{Q}, Q \setminus F, \delta)$.

Proposition 6 *Let $\mathcal{A}$ be a deterministic and complete left-aligned automaton over Λ. Then, $\overline{\mathcal{A}}$ is a left-aligned automaton and $\mathcal{L}(\overline{\mathcal{A}}) = \mathcal{L}(\mathcal{U}_\Lambda) \setminus \mathcal{L}(\mathcal{A})$.*

Proof Let $\mathcal{A}$ be an arbitrary left-aligned automaton. It follows directly from $\mathcal{A}$ being a left-aligned automaton that $\overline{\mathcal{A}}$ is also a left-aligned automaton (complementing an automaton does not change its transition function). For the same reason, $\overline{\mathcal{A}}$ is a deterministic and complete left-aligned automaton, as well. Then, for all unzipped traces $\tau \in (\Sigma^*)^X$ there exists a run r in $\overline{\mathcal{A}}$ for τ (from completeness); and this is the only run for τ (from determinism). As $\overline{\mathcal{A}}$ and $\mathcal{A}$ share the same transition function, then r is also the only run in $\mathcal{A}$ for τ. Finally, as the final states are flipped, it follows that r is an accepting run for τ in $\overline{\mathcal{A}}$ iff r is not an accepting run for τ in $\mathcal{A}$. Hence $\mathcal{L}(\overline{\mathcal{A}}) = \mathcal{L}(\mathcal{U}_\Lambda) \setminus \mathcal{L}(\mathcal{A})$. □

4.1.2 From formulas of hypernode logic to left-aligned automata

We show how to translate atomic formulas of the synchronous fragment of the hypernode logic into left-aligned automata. These automata are the building blocks for the model checking algorithm described in the next section. Examples of the translation of atomic formulas to automata can be found in Figs. 5 and 6, later in this section. In the subsequent



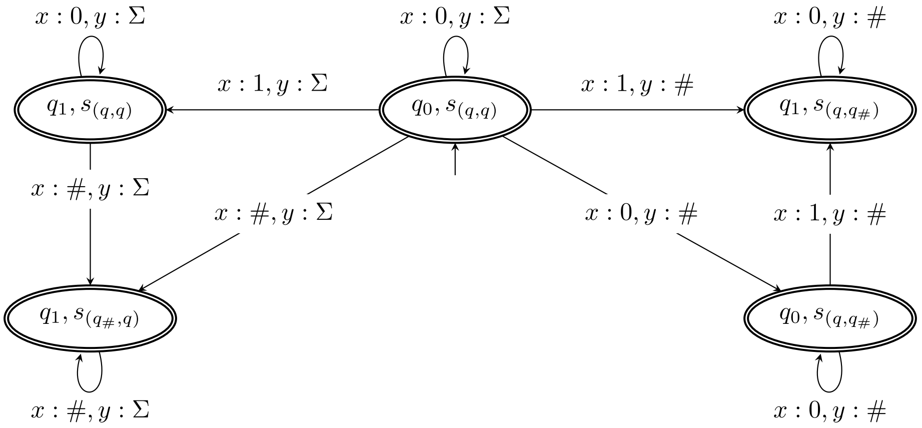


Fig. 5 The top automaton is the finite state automaton recognizing 0^*10^* and below is the LAA for $\mathbf{x}(\pi) \leq 0^*10^*$ over the alphabet $\Lambda(\{x, y\}, \{0, 1\})$

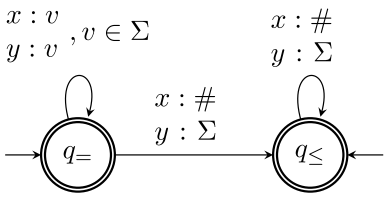


Fig. 6 Left-aligned automaton for the atomic formula $\mathbf{x}(\pi) \leq \mathbf{y}(\pi')$ over the alphabet $\Lambda(\{x, y\}, \Sigma)$

text, we assume an alphabet $\Lambda(X, \Sigma)$ and that $\mathcal{U}_\Lambda$, the universal left-aligned automaton with alphabet Λ, is the tuple $(Q_\mathcal{U}, Q_\mathcal{U}, Q_\mathcal{U}, \delta_\mathcal{U})$.

Automaton for $\mathbf{x}(\pi) \leq \psi$

For the term $\mathbf{x}(\pi) \leq \psi$ where ψ is a regular expression over Σ, we first create the finite state automaton $\mathcal{A}_\psi = (Q, \hat{Q}, F, \delta)$ accepting the language of ψ (using any of the known methods, e.g. [9, 11, 31, 33]), and we strip it off of useless states (states from which no final state is reachable). Notice that from the syntax of hypernode logic, the language must be non-empty. Let $x_\pi \in X$ be the program variable that represents $\mathbf{x}(\pi)$. We define the automaton $\mathcal{A}_{x_\pi \leq_\Lambda \psi} = (Q_\psi \cup Q_\#, \hat{Q} \times Q_\mathcal{U}, Q_\psi \cup Q_\#, \delta_{x_\pi})$ where

- $Q_\psi = Q \times \{s \in Q_\mathcal{U} \mid s(x_\pi) \neq q_\#\}$ and $Q_\# = \{s \in Q_\mathcal{U} \mid s(x_\pi) = q_\#\}$
- δ_{x_π} is defined as the smallest set s.t.:
  1) if $q \xrightarrow{\sigma} q' \in \delta$ and $s \xrightarrow{v} s' \in \delta_\mathcal{U}$ and $v(x_\pi) = \sigma$, then $(q, s) \xrightarrow{v} (q', s') \in \delta_{x_\pi}$
  2) if $s \xrightarrow{v} s' \in \delta_\mathcal{U}$ and $s(x_\pi) \neq q_\#$ and $s'(x_\pi) = q_\#$, then $(q, s) \xrightarrow{v} s' \in \delta_{x_\pi}$

3) if $s \xrightarrow{v} s' \in \delta_{\mathcal{U}}$ and $s(x_\pi) = s'(x_\pi) = q_\#$, then $s \xrightarrow{v} s' \in \delta_{x_\pi}$

We show that the automaton $\mathcal{A}_{x_\pi \leq_\Lambda \psi}$ over the alphabet Λ recognizes unzipped trace segments where variable x_π maps to words from $\mathcal{L}(\psi)$ and other variables map to arbitrary words.

Proposition 7 *Let $\Lambda(X, \Sigma)$ be an alphabet, x_π a variable from X and ψ a regular expression over Σ. $\tau \in \mathcal{L}(\mathcal{A}_{x_\pi \leq_\Lambda \psi})$ iff exists $v \in \Sigma^*$ s.t. $\tau(x_\pi) = u$, $uv \in \mathcal{L}(\psi)$ and, for all variables $y \in (X \setminus \{x_\pi\})$, $\tau(y) \in \Sigma^*$.*

Proof First, let us make two observations. (1) Every run has a prefix of states in Q_ψ and then possibly an empty suffix of states in $Q_\#$. (2) For any run, all transitions between states in Q_ψ are transitions 1), transitions between Q_ψ and $Q_\#$ are transitions 2), and transitions between states in $Q_\#$ are transitions 3).

($\Rightarrow$) By observation 1, any run of $\mathcal{A}_{x_\pi \leq_\Lambda \psi}$ has a prefix *p* in Q_ψ, and by observation 2, valuations on this prefix (if any) are from transitions 1). These transitions are synchronized with A_ψ and therefore there is a run r_ψ in A_ψ corresponding to *p* that is labeled with $\mathit{trace}(p)(x_\pi)$. The run r_ψ in A_ψ can be extended to an accepting run because A_ψ has no useless states and accepts a non-empty language. Therefore, A_ψ accepts *uv* for some *v* where $u = \mathit{trace}(p)(x_\pi)$.

Finally, $\tau(y) \in \Sigma^*$ for $y \in (X \setminus \{x_\pi\})$ is trivially satisfied.

($\Leftarrow$) Let us fix an arbitrary unzipped trace segment τ with $\tau(x_\pi) = u$ s.t. $uv \in \mathcal{L}(\psi)$. W.l.o.g assume that $|u| = k$ and *m* is the length of the longest word in τ, $m \geq k \geq 0$. Because $uv \in \mathcal{L}(\mathcal{A}_\psi)$, there is a run of $\mathcal{A}_\psi$ that accepts *uv* and a prefix $q_0\sigma_0q_1\sigma_1...q_k$ of this run that is labeled with *u* ($u = \sigma_0\sigma_1...\sigma_{k-1}$). Now consider an arbitrary run $r = (q0, s_0)v_0(q_1, s_1)v_1...(q_k, s_k)v_ks_{k+1}v_{k+1}s_{k+2}...s_m$ of $\mathcal{A}_{x_\pi \leq_\Lambda \psi}$. We have that $v_i(x_\pi) = \sigma_i$, for each $0 \leq i < k$, as the first *k* transitions are transitions 1), and $v_i(x_\pi) = \#$ for $k \leq i < m$ as these come from transitions 2) and 3). Therefore, $\mathit{trace}(r)(x_\pi) = u$. At the same time, there are no constraints on other variables, so $\mathit{trace}(r)(y)$ for $y \in (X \setminus \{x_\pi\})$ is arbitrary. In particular, we can pick the run so that $\mathit{trace}(r)(y) = \tau(y)$ for all $y \in (X \setminus \{x_\pi\})$. Therefore, there is an accepting run for τ in $\mathcal{A}_{x_\pi \leq_\Lambda \psi}$. □

Example: Automaton For Atomic Formula With Regular Expression

Automaton for $\psi \leq \mathbf{x}(\pi)$

For terms $\psi \leq \mathbf{x}(\pi)$ where ψ is a regular expression over Σ, we again first create the finite state automaton $\mathcal{A}_\psi = (Q, \hat{Q}, F, \delta)$ accepting the language of ψ. Let $x_\pi \in X$ be the program variable that instantiates $\mathbf{x}(\pi)$. We define the automaton $\mathcal{A}_{\psi \leq_\Lambda x_\pi} = (Q_\psi \cup Q_{\mathcal{U}}, \hat{Q} \times Q_{\mathcal{U}}, Q_F, \delta_{x_\pi})$ where

- $Q_\psi = Q \times \{s \in Q_{\mathcal{U}} \mid s(x_\pi) \neq q_\#\}$
- $Q_F = Q_{\mathcal{U}} \cup \{(q, s) \in Q_\psi \mid q \in F\}$
- δ_{x_π} is defined as the smallest set s.t.:

1) if $q \xrightarrow{\sigma} q' \in \delta$ and $s \xrightarrow{v} s' \in \delta_{\mathcal{U}}$ and $v(x_\pi) = \sigma$, then $(q, s) \xrightarrow{v} (q', s') \in \delta_{x_\pi}$

2) if $s \xrightarrow{v} s' \in \delta_{\mathcal{U}}$ and $q \in F$, then $(q, s) \xrightarrow{v} s' \in \delta_{x_\pi}$

3) if $s \xrightarrow{v} s' \in \delta_{\mathcal{U}}$, then $s \xrightarrow{v} s' \in \delta_{x_\pi}$

Proposition 8 *Let $\Lambda(X, \Sigma)$ be an alphabet, x_π a variable from X and ψ a regular expression over Σ. $\tau \in \mathcal{L}(\mathcal{A}_{\psi \leq_\Lambda x_\pi})$ iff exists $v \in \Sigma^*$ s.t. $u \in \mathcal{L}(\psi)$, $\tau(x_\pi) = uv$, and, for all variables $y \in (X \setminus \{x_\pi\})$, $\tau(y) \in \Sigma^*$.*

Proof $(\Rightarrow)$ If $\tau \in \mathcal{L}(\mathcal{A}_{\psi \leq_\Lambda x_\pi})$, then there is an accepting run for τ. This run has the shape $(q_0, s_0)v_0...(q_k, s_k)v_k s_{k+1} v_{k+1} s_{k+2}...s_n$ where the prefix is in Q_ψ and the suffix is in $Q_{\mathcal{U}}$. Because this run is accepting, states $(q_k, s_k), s_{k+1}, s_{k+2}$ are final (follows from definition of Q_F and transitions 2). Therefore, $q_k \in F$. For the prefix in Q_ψ of this run, there is a corresponding run in $\mathcal{A}_\psi$ $q_0\sigma_0....\sigma_{k-1}q_k$ which is accepting, because $q_k \in F$. Therefore, $\sigma_0\sigma_1...\sigma_{k-1} \in \mathcal{L}(\psi)$. From transitions 1, we know that $v_i(x_\pi) = \sigma_i$ for $0 \leq i < k$ and therefore $\tau(x_\pi) = \sigma_0\sigma_1...\sigma_k$. If we let $u = \sigma_0\sigma_1...\sigma_k$, then we get that $\tau(x_\pi) = uv$ s.t. $u \in \mathcal{L}(\psi)$ for some $v \in \Sigma^*$. The rest of the right-hand side condition is trivial.

$(\Leftarrow)$ Assume an unzipped trace segment τ s.t. $\tau(x_\pi) = uv$ s.t. $u \in \mathcal{L}(\psi)$ and $\tau(y) \in \Sigma^*$ are arbitrary (but fixed) for $y \in (X \setminus \{x_\pi\})$. We construct a run *r* of $\mathcal{A}_{\psi \leq_\Lambda x_\pi}$ accepting τ. The run has the shape $(q_0, s_0)v_0...(q_k, s_k)v_k s_{k+1} v_{k+1} s_{k+2}...s_n$ where $q_0 v_0(x_\pi) q_1 v_1(x_\pi)...q_k$ is an accepting run of *u* in $\mathcal{A}_\psi$ and $s_k v_k s_{k+1}...s_n$ is a run of a universal automaton. By definition of transitions, this run exists in $\mathcal{A}_{\psi \leq_\Lambda x_\pi}$ for arbitrary $s_0, s_1, \ldots, s_n$, and therefore we can pick them to match τ. □

Automaton for $\mathbf{x}(\pi) \leq \mathbf{y}(\pi')$

We construct the automaton for a term $\mathbf{x}(\pi) \leq \mathbf{y}(\pi')$ by restricting the universal automaton. The automaton $\mathcal{A}_{x \leq_\Lambda y}$ is defined as $\mathcal{A}_{x \leq_\Lambda y} = (Q, Q, Q, \delta)$ where $Q = \{s \in Q_{\mathcal{U}} \mid s(x) = s(y) = q \text{ or } s(x) = q_\#\}$. The transition relation is defined by the relation $(q, v, q') \in \delta$ iff $(q, v, q') \in \delta_{\mathcal{U}}$ and $v(x) = v(y)$ or $v(x) = \#$. The automaton for $\mathbf{x}(\pi) \leq \mathbf{y}(\pi')$ is depicted in Fig. 6.

Proposition 9 *Let $\Lambda(X, \Sigma)$ be an alphabet and x and y variables from X. $\tau \in \mathcal{L}(\mathcal{A}_{x \leq_\Lambda y})$ iff $\tau(x) \leq \tau(y)$.*

Proof Consider any run *r* of $\mathcal{A}_{x \leq_\Lambda y}$ that accepts a segment τ. There are two options. Either the run starts in a state *s* with $s(x) = q_\#$, and in that case, any future state of the run s' has $s'(x) = q_\#$ and all valuations *v* of the run have $v(x) = \#$. Therefore, $\tau(x) = \epsilon$ and $\tau(x) \leq \tau(y)$.

The other option is that *r* starts in a state with $s(x) \neq q_\#$. Then there is a maximal prefix p of *r* where every state *s* has $s(x) = s(y) = q$, and by definition of δ and $\delta_{\mathcal{U}}$, also every valuation *v* between these states satisfies $v(x) = v(y)$. That is, for p it holds that $trace(p)(x) = trace(p)(y)$. Since this prefix was maximal, next state *s* from *r* after this prefix has $s(x) = q_\#$ and any valuation *v* on the rest of the path has $v(x) = \#$. As a result, $\tau(x) = trace(p)(x)$. But because more letters can be still added to $trace(p)(y)$, we get that $\tau(x) \leq \tau(y)$.

The other direction is analogous. □

Example: Automaton For $\mathbf{x}(\pi) \leq \mathbf{y}(\pi')$

4.2 Model-checking algorithm

We start by showing how to model-check hypernode formulas of the regular synchronous fragment over left-aligned automata. Solving the model-checking problem of left-aligned automata also gives us a model-checking algorithm for Kripke structures, because Kripke structures can be easily translated into automata that recognize the traces of the Kripke structure (this is a classical construction). That is, for each open Kripke structure $(K, \mathbb{W})$, there is a left-aligned automaton such that the automaton accepts the same unzipped trace segments as generated by $(K, \mathbb{W})$.

Let us define the set $X_{\mathcal{V}}$ of propositional variables as $X_{\mathcal{V}} = \{x_\pi \mid x \in X \text{ and } \pi \in \mathcal{V}\}$. For quantifier-free formulas, the model-checking procedure of left-aligned automata is defined using operations on left-aligned automata defined in previous sections. For trace quantifiers, we introduce projection by erasure of the transitions associated with the quantified variable. That is, given a left-aligned automaton $\mathcal{A} = (Q, \hat{Q}, F, \delta)$ over $\Lambda(X_{\mathcal{V}}, \Sigma)$, the erasure of a trace variable $\pi \in \mathcal{V}$ defines the left-aligned automaton $\mathcal{A}_{-\pi} = (Q, \hat{Q}, F, \delta')$ over $\Lambda(X_{(\mathcal{V}\setminus\{\pi\})}, \Sigma)$ where, for all $v \in \Lambda(X_{(\mathcal{V}\setminus\{\pi\})}, \Sigma)$:

$$\delta'(q, v) = \{q' \mid \exists v' \in \Lambda(X, \Sigma) \text{ such that } (q, v \cup v', q') \in \delta\}.$$

As it holds that $(\mathcal{A}_{-\pi})_{-\pi'} = (\mathcal{A}_{-\pi'})_{-\pi}$, we naturally extend the definition of erasure to sets of trace variables $\mathcal{A}_{-V}$. Now we can define the main component of the model-checking algorithm: the filtration of left-aligned automata over hypernode formulas.

Definition 4 Let $\mathcal{A}$ be a left-aligned automaton over $\Lambda(X_{\mathcal{V}}, \Sigma)$, and φ a formula of hypernode logic in prenex normal form. We define the positive and negative filtration of $\mathcal{A}$ by φ, denoted $\varphi^+[\mathcal{A}]$ and $\varphi^-[\mathcal{A}]$, respectively, inductively over the structure of φ as follows:

$$\begin{aligned}
(x(\pi) \leq y(\pi'))^+[\mathcal{A}] &= \mathcal{A} \cap \mathcal{A}_{x_\pi \leq_\Lambda y_{\pi'}} & (x(\pi) \leq y(\pi'))^-[\mathcal{A}] &= \mathcal{A} \cap \overline{\mathcal{A}_{x_\pi \leq_\Lambda y_{\pi'}}} \\
(x(\pi) \leq \psi)^+[\mathcal{A}] &= \mathcal{A} \cap \mathcal{A}_{x_\pi \leq_\Lambda \psi} & (x(\pi) \leq \psi)^-[\mathcal{A}] &= \mathcal{A} \cap \overline{\mathcal{A}_{x_\pi \leq_\Lambda \psi}} \\
(\psi \leq x(\pi))^+[\mathcal{A}] &= \mathcal{A} \cap \mathcal{A}_{\psi \leq_\Lambda x_\pi} & (\psi \leq x(\pi))^-[\mathcal{A}] &= \mathcal{A} \cap \overline{\mathcal{A}_{\psi \leq_\Lambda x_\pi}}
\end{aligned}$$

$$\begin{aligned}
(\varphi_1 \wedge \varphi_2)^+[\mathcal{A}] &= \varphi_1^+[\mathcal{A}] \cap \varphi_2^+[\mathcal{A}] & (\varphi_1 \wedge \varphi_2)^-[\mathcal{A}] &= \varphi_1^-[\mathcal{A}] \cup \varphi_2^-[\mathcal{A}] \\
(\neg\varphi)^+[\mathcal{A}] &= \varphi^-[\mathcal{A}] & (\neg\varphi)^-[\mathcal{A}] &= \varphi^+[\mathcal{A}] \\
(\exists\pi\varphi)^+[\mathcal{A}] &= (\varphi^+[\mathcal{A}])_{-\pi} & (\exists\pi\varphi)^-[\mathcal{A}] &= \mathcal{A} \ominus (\exists\pi\varphi)^+[\mathcal{A}].
\end{aligned}$$

with $\mathcal{A}_1 \ominus \mathcal{A}_2 = (\mathcal{A}_{1-(V_1\setminus V_2)}) \cap \overline{\mathcal{A}_2}$ where V_1 and V_2 are trace variables of $\mathcal{A}_1$ and $\mathcal{A}_2$, resp.

The meaning of filtration functions is that $\varphi^+[\mathcal{A}]$ is the automaton that recognizes trace segments that are in $\mathcal{L}(\mathcal{A})$ and satisfy φ, and $\varphi^-[\mathcal{A}]$ is the automaton that recognizes trace segments in $\mathcal{L}(\mathcal{A})$ that do not satisfy φ. The operation $\ominus$ is simply a difference of two automata that are made compatible by erasing trace variables from $\mathcal{A}_1$ that are not in $\mathcal{A}_2$. Notice that this operation is always applied with $\mathcal{A}_1 = \mathcal{A}$, so it holds that $V_1 \supseteq V_2$. Because we require the input formula to be in prenex normal form, erasing traces happens as the lasts steps and thus this is the only place in the definition where automata can have different trace variables.

Given a formula φ with n trace variables $\mathcal{V}$ and program variables X, we reduce the problem of model checking a left-aligned automaton $\mathcal{A}$ with the alphabet $\Lambda(X, \Sigma)$ to filtering the n-self-composition of $\mathcal{A}$ by φ. W.l.o.g assume that $\mathcal{V} = \{\pi_1, \ldots, \pi_n\}$. The *$n$-self-composition of $\mathcal{A}$*, denoted by $\mathcal{A}^n = \mathcal{A}_1 \parallel ... \parallel \mathcal{A}_n$, is the parallel synchronous (lock-step) product construction of $\mathcal{A}$ with itself n times, where in every $\mathcal{A}_i$ we rename each variable x to x_{π_i}. This means that $\mathcal{A}^n$ has the alphabet $\Lambda(X_{\mathcal{V}}, \Sigma)$.

Given an unzipped trace segment τ over the alphabet $\Lambda(X_{\mathcal{V}}, \Sigma)$, the trace assignment generated by τ is defined as $\Pi_\tau(\pi)(x) = \tau(x_\pi)$, for all $\pi \in \mathcal{V}$ and $x \in X$. Notice that if τ is accepted by $\mathcal{A}^n$, then Π_τ defines a trace assignment from $\{\pi_1, \ldots, \pi_n\}$ to trace segments accepted by $\mathcal{A}$. Now we can formulate the following theorem.

Theorem 10 *Let $\mathcal{A}$ be a left-aligned automaton, and φ a closed regular synchronous hypernode logic formula with n trace variables $\pi_1, \ldots, \pi_n$. Then, $\mathcal{L}(\varphi^+[\mathcal{A}^n]) \neq \emptyset$ iff $\mathcal{L}(\mathcal{A}) \models \varphi$, and $\mathcal{L}(\varphi^-[\mathcal{A}^n]) \neq \emptyset$ iff $\mathcal{L}(\mathcal{A}) \not\models \varphi$.*

Proof Follows from the lemma we prove below for open formulas:

Let φ be a regular synchronous hypernode formula over program variables X and trace variables $\mathcal{V}$ with free variables (not bound to a quantifier) $\mathcal{V}'$. Let τ be an unzipped trace segment over $(\Sigma^{X_{\mathcal{V}'}})^*$.

Then, $\tau \in \mathcal{L}(\varphi^+[\mathcal{A}^n])$ iff $(\Pi_\tau, \mathcal{L}(\mathcal{A})) \models \varphi$; and $\tau \in \mathcal{L}(\varphi^-[\mathcal{A}^n])$ iff $(\Pi_\tau, \mathcal{L}(\mathcal{A})) \not\models \varphi$.

We prove the lemma by induction on the structure of the hypernode formula.

To prove the base case for atomic formulas $\psi \leq x(\pi)$, $x(\pi) \leq \psi$, and $x(\pi) \leq y(\pi')$, we use propositions 7, 8, and 20. In particular, for the case of $x(\pi) \leq y(\pi')$, $\tau \in \mathcal{L}((x(\pi) \leq y(\pi'))^+[\mathcal{A}^n])$ iff $\tau \in \mathcal{L}(\mathcal{A}^n)$ and $\tau \in \mathcal{L}(\mathcal{A}_{x_\pi \leq_\Lambda y_{\pi'}})$. As $\tau \in \mathcal{L}(\mathcal{A}^n)$, then Π_τ defines a trace assignment from $\{\pi_1, \ldots \pi_n\}$ to traces accepted by $\mathcal{A}$ and, because $\tau \in \mathcal{L}(\mathcal{A}_{x_\pi \leq_\Lambda y_{\pi'}})$, then by Proposition 20, $(\Pi_\tau, \mathcal{L}(\mathcal{A})) \models x(\pi) \leq y(\pi')$. We prove analogously for $\tau \in \mathcal{L}((x(\pi) \leq y(\pi')^-[\mathcal{A}^n])$ and other atomic formulas and negative filtrations.

We proceed to the inductive steps and assume as IH that the property holds for arbitrary hypernode formulas φ and φ'. The inductive case $\varphi \wedge \varphi'$ follows from IH and Proposition 2. While the inductive case $\neg\varphi$ follows from IH and Proposition 6.

We now move to the case of the existential quantifier – $\exists\pi\varphi$. We start with the positive filtration by proving the $\Rightarrow$-direction of the statement. Consider arbitrary $\tau \in \mathcal{L}((\exists\pi\varphi)^+[\mathcal{A}^n])$ over $(\Sigma^{X_{\mathcal{V}'}})^*$. By definition of filtration, this is equivalent to $\tau \in \mathcal{L}(\varphi^+[\mathcal{A}^n]_{-\pi})$. Then, by definition of erasure of a trace variable and accepted word of a left-aligned automata, there exists $\tau_\exists$ over $(\Sigma^{X_{\mathcal{V}}})^*$ such that for all variables $x \in X_{\mathcal{V}'}$: $\tau_\exists(x) = \tau(x)$ and $\tau_\exists \in \mathcal{L}(\varphi^+[\mathcal{A}^n])$. Recall that, $\mathcal{V} = \mathcal{V}' \cup \{\pi\}$. By IH, $\Pi_{\tau_\exists} \models \varphi$, and $\Pi_\tau \models \exists\pi\varphi$.

We now prove the $\Leftarrow$-direction. Assume a trace assignment exists s.t. $\Pi \models \exists\pi\varphi$. Then, by definition of the satisfaction relation for hypernode formulas, there exists an unzipped trace τ' over $(\Sigma^{X_{\mathcal{V}}})^*$ accepted by $\mathcal{A}$ (i.e., $\tau' \in \mathcal{L}(\mathcal{A})$) s.t. $\Pi[\pi \mapsto \tau'] \models \varphi$. Note that the function to derive a trace assignment (mapping trace variables $\mathcal{V}$ to unzipped traces over Σ^X) from an unzipped trace segment τ over $(\Sigma^{X_{\mathcal{V}}})^*$ is invertible. Then, by IH, the trace τ_π derived from the assignment extension $\Pi_\pi = \Pi[\pi \mapsto \tau']$ is in $\mathcal{L}(\varphi^+[\mathcal{A}^n])$. And, by definition of filtration, $\tau \in \mathcal{L}(\exists\pi\varphi^+[\mathcal{A}^n])$.

We now prove the negative filtering. We start with the $\Rightarrow$-direction, which we prove by contra-position. Assume that for an arbitrary trace assignment Π we have $\Pi \models \exists\pi\varphi$, then there exists an extension $\Pi_\pi = \Pi[\pi \mapsto \tau']$ with $\tau' \in \mathcal{L}(\mathcal{A})$ s.t. $\Pi_\pi \models \varphi$. By IH, the trace τ_π derived by Π_π is accepted by $\varphi^+[\mathcal{A}^n]$. Thus, τ_π is not accepted by $\mathcal{A}^n \setminus \varphi^+[\mathcal{A}^n]$, and, by definition, $\tau_\pi \notin \mathcal{L}((\exists\pi\varphi)^-[\mathcal{A}^n])$. For the $\Leftarrow$-direction, consider an arbitrary trace assignment s.t. $\Pi \models \neg\exists\pi\varphi$. Then, $\Pi \models \forall\pi\neg\varphi$. Equivalently, for all extensions of Π, i.e. $\Pi_\pi = \Pi[\pi \mapsto \tau']$ for all $\tau' \in \mathcal{L}(\mathcal{A})$, we have $\Pi_\pi \not\models \varphi$. Consider an arbitrary of such extensions Π_π, then, by IH, for the trace τ_π derived by it, we have $\tau_\pi \in \mathcal{L}(\mathcal{A}^n)$ and $\tau_\pi \notin \mathcal{L}(\varphi^+[\mathcal{A}^n])$. Equivalently, $\tau_\pi \in \mathcal{L}(\mathcal{A}^n \setminus \varphi^+[\mathcal{A}^n])$ and, so $\tau_\pi \in \mathcal{L}((\exists\pi\varphi)^-[\mathcal{A}^n])$. □

We define the translation of an open Kripke structure $(K, \mathbb{W})$ into the LAA $\mathcal{A}_{(K,\mathbb{W})}$ recognizing all of $(K, \mathbb{W})$ finite traces as usual: by a direct translation preserving the structure of K. The set of states in $\mathcal{A}_{(K,\mathbb{W})}$ is the same as the set of worlds in K including a fresh new state, which is the only initial state, and the set of final states in $\mathcal{A}_{(K,\mathbb{W})}$ is the set of exit worlds in $\mathbb{W}$. We include in the transition function of $\mathcal{A}_{(K,\mathbb{W})}$ transitions between the initial state and all states matching the entry worlds in $\mathbb{W}$. The transition function of $\mathcal{A}_{(K,\mathbb{W})}$ is derived directly from the transition relation of K together with the new transitions added from the initial state, which are labeled as follows: transitions from state w to state w' in $\mathcal{A}_{(K,\mathbb{W})}$ are labeled by the valuation assigned to the world w' in K. By translating open Kripke structures to LAAs and the filtration from Definition 4, we have an effective way to solve the model-checking problem for hypernode logic over Kripke structures.

Corollary 11 *Let $(K, \mathbb{W})$ be an open Kripke structure, and φ a closed regular synchronous hypernode logic formula over the same set of variables. Let n be the number of trace variables in φ. Then,* $\mathrm{Traces}(K, \mathbb{W}) \models \varphi$ *iff* $\mathcal{L}(\varphi^+[\mathcal{A}^n_{(K,\mathbb{W})}]) \neq \emptyset$*.*

Proof We start by observing that, by construction, $\mathrm{Traces}(K, \mathbb{W}) = \mathcal{L}(\mathcal{A}_{(K,\mathbb{W})})$. Then, the result follows directly from Theorem 10. □

The theorem below follows from the corollary above, all operations used for filtering being decidable, and checking for non-emptiness of a finite automata also being decidable.

Theorem 12 *Model checking the regular synchronous fragment of hypernode logic over open Kripke structures is decidable.*

5 Model checking asynchronous hypernode logic

In this section, we discuss the model-checking of the *regular asynchronous* fragment of hypernode logic. The only difference from the fragment presented in the previous section is that we now allow only asynchronous comparisons; that is, our formulas can only use $\precsim$ to compare program variables and regular languages. This fragment is defined by the following grammar:

$$\psi ::= \epsilon \mid c \mid \psi.\psi \mid \psi + \psi \mid \psi^*$$
$$\varphi ::= \exists\pi\,\varphi \mid \neg\varphi \mid \varphi \wedge \varphi \mid \mathbf{x}(\pi) \precsim \psi \mid \psi \precsim \mathbf{x}(\pi) \mid \mathbf{x}(\pi) \precsim \mathbf{x}(\pi)$$

In the rest of this section, we show how to model-check this fragment of HL using constructions on *stutter-free automata*, introduced next.

5.1 Stutter-free automata

Definition 5 Let X be a finite set of variables over Σ. A left-aligned automaton $\mathcal{A}=(Q,\hat{Q},F,\delta)$ is a *stutter-free automaton* (NSFA) if its transition relation satisfies the *stutter-freedom* condition for any $q \in Q$ and $x \in X$:

$$In(q,x) \cap Out(q,x) \subseteq \{\#\}.$$

In simple terms, an LAA is stutter-free if, for every state and every variable that is not yet terminated at that state, no variable's valuation appears in both the incoming and outgoing transitions of that state. We observe that all languages accepted by some NSFA are stutter-reduced, i.e., for any NSFA $\mathcal{A}$, we have that $\mathcal{L}(\mathcal{A}) = \lfloor \mathcal{L}(\mathcal{A}) \rfloor$ (in this and following sections, we freely use the stutter-reduction operation $\lfloor \cdot \rfloor$ on words and languages with the obvious meaning).

Example: Stutter-free Automata

In Fig. 7 below, we depict a stutter-free automaton that accepts all unzipped trace segments for the boolean variables $\{x,y\}$, where all x-trace segments are of odd size, while all y-trace segments are of even size, and the first value for both x and y is 0. The language accepted by the automaton $\mathcal{A}$ from Fig. 7 is:

$$\mathcal{L}(\mathcal{A})=\{x:\tau_x, y:\tau_y \mid \tau_x \in (01)^*0 \text{ and } \tau_y \in (01)^*01\}.$$

The union, intersection, and determinization for NSFA work the same as for LAAs.

Proposition 13 *Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be two stutter-free automata over the same alphabet Λ. Both $\mathcal{A}_1 \cup \mathcal{A}_2$ and $\mathcal{A}_1 \cap \mathcal{A}_2$ are stutter-free automata over Λ with $\mathcal{L}(\mathcal{A}_1 \cup \mathcal{A}_2) = \mathcal{L}(\mathcal{A}_1) \cup \mathcal{L}(\mathcal{A}_2)$ and $\mathcal{L}(\mathcal{A}_1 \cap \mathcal{A}_2) = \mathcal{L}(\mathcal{A}_1) \cap \mathcal{L}(\mathcal{A}_2)$. Moreover, for a nondeterministic stutter-free automa-*

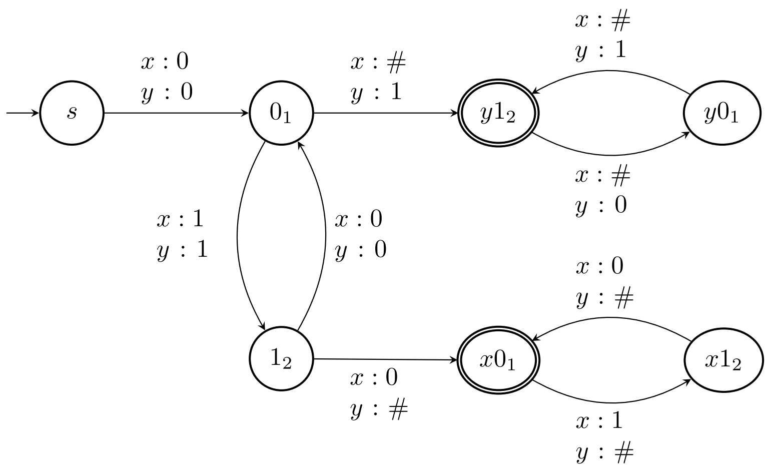


Fig. 7 Stutter-free automaton $\mathcal{A}$ where x-traces are of odd size while y-traces are of even size, and the first valuation for both x and y is 0

ton $\mathcal{A}$, its determinization, $det(\mathcal{A})$, defines a deterministic stutter-free automaton accepting the same language, that is, $\mathcal{L}(det(\mathcal{A})) = \mathcal{L}(\mathcal{A})$.

Proof We observe that the union and intersection do not affect the stutter-free condition; therefore, the result follows from NSFA being LAA.

Let $\mathcal{A}$ be an arbitrary stutter-free automaton. We must prove that $\det(\mathcal{A})$ satisfies stutter-freedom, i.e., $In(S,x) \cap Out(S,x) \subseteq \{\#\}$, for all states S of $\det(\mathcal{A})$ and variables $x \in X$. By definition:

$$\begin{aligned}
In(S,x) &= \{v(x) \mid S \in \delta_d(S', v) for some S' \in Q_d\} \Leftrightarrow \\
In(S,x) &= \{v(x) \mid S = \bigcup_{q_1 \in S_1} \delta(q_1, v) for some S' \in Q_d\} \Leftrightarrow \\
In(S,x) &= \{v(x) \mid \forall q \in S\ \exists q_1 \in S' \text{ s.t. } q \in \delta(q_1, v) for some S' \in Q_d\}.
\end{aligned}$$

Thus, $(\star)$ for all $q \in S$, $In(q,x) = In(S,x)$. From $\mathcal{A}$ being a stutter-free automaton, we know that, for all $q \in S$, $In(q,x) \cap Out(q,x) \subseteq \{\#\}$. Assume towards a contradiction that there exists a value (different from the termination symbol $\#$) that is in both the incoming and outgoing transitions of S for a variable x, i.e., $l \in In(S,x) \cap Out(S,x)$ and $l \neq \#$. Then, by definition of $Out(S,x)$, there exists a state in S, $q \in S$, s.t. $\delta(q, x:l) \neq \emptyset$. This contradicts our conclusion $(\star)$.

Finally, $\mathcal{L}(\det(\mathcal{A})) = \mathcal{L}(\mathcal{A})$ follows directly from the same result for NFA. □

To make a stutter-free automaton complete, we can use the same procedure as for LAA, but we must make sure to also fulfill the stutter-freedom requirement: instead of using universal LAA, we must use a universal stutter-free automaton. Also, to avoid adding new edges that would violate the stutter-freedom requirement, we must re-define the function *allowed*:

$$allowed_{sf}(q) = \{v \in \Lambda \mid v \in allowed(q) \text{ and } \forall x \in X\ In(q,x) \cap \{v(x)\} \subseteq \{\#\}\}.$$

A stutter-free automaton $\mathcal{A} = (Q, \hat{Q}, F, \delta)$ over Λ is complete if for each $q \in Q$ it holds that $allowed_{sf}(q) = \Lambda_{out}(q)$. The *universal* stutter-free automaton $\mathcal{U}_\Lambda$ over Λ is a deterministic and complete automaton, accepting all stutter-free unzipped traces over Λ. This means that $\mathcal{U}_\Lambda$ accepts the language $\mathcal{L}(\mathcal{U}_\Lambda) = \lfloor (\Sigma^*)^X \rfloor$.

Definition 6 Let $X = \{x_0, \ldots, x_m\}$ be a set of variables over the finite domain Σ.

The *universal stutter-free automaton* over $\Lambda(X, \Sigma)$ is $\mathcal{U}_\Lambda = (Q_\mathcal{U}, Q_\mathcal{U}, Q_\mathcal{U}, \delta_\mathcal{U})$, where $Q_\mathcal{U} = \Lambda$ and

$$\delta_\mathcal{U}(\{x_i : \sigma_i\}_{i \in [0,m]}, \{x_i : \sigma'_i\}_{i \in [0,m]}) = \begin{cases} \{x_i : \sigma'_i\}_{i \in [0,m]} & \text{if } \forall 0 \leq i \leq m, \text{ if } \sigma_i = \# \\ & \quad \text{then } \sigma'_i = \# \text{ else } \sigma_i \neq \sigma'_i; \\ \emptyset & \text{otherwise.} \end{cases}$$

Proposition 14 *The universal stutter-free automaton $\mathcal{U}_{\Lambda(X,\Sigma)}$ accepts the universal language, i.e., $\mathcal{L}(\mathcal{U}_{\Lambda(X,\Sigma)}) = \{\{x_0 : w_0, \ldots, x_m : w_m\} \mid for\, all i \in [0, m]\ x_i \in X, w_i \in \Sigma^*, and w_i = \lfloor w_i \rfloor\}$.*

Proof Assume any $\tau \in \mathcal{L}(\mathcal{U}_{\Lambda(X,\Sigma)})$. The accepting run of τ is a sequence $s_0 v_0 s_1 v_1 ... v_{k-1} s_k$ where, by definition of $\delta_{\mathcal{U}}$, $val_i = s_{i+1}$ for $0 \leq i < k$. Therefore $\tau = \text{unzip}(v_0 \cdot ... \cdot v_{k-1})$ s.t. this sequence is stutter-reduced (again by definition of $\delta_{\mathcal{U}}$).

For the other direction, let τ be an arbitrary unzipped trace segment over $\Lambda(X)$ s.t. for all $x \in X$ $\tau(x) = \lfloor \tau(x) \rfloor$. Then, for all $x \in X, \tau(x) = \sigma_0 \sigma_1 ... \sigma_k$ s.t. for all $i \in [0, k-1]$ it holds that $\sigma_i \neq \sigma_{i+1}$ Let $v_0 v_1 ... v_m$ be a sequence of valuations s.t. $\tau = \text{unzip}(v_0 v_1 ... v_m)$ (note that this sequence is unique) and for each two valuations v_i, v_{i+1} for $i \in [0, m-1]$ and $x \in X$ it holds that either $v_i(x) = \#$ and then $v_{i+1}(x) = \#$, or $v_i(x) \neq v_{i+1}(x)$. By the definition of $\delta_{\mathcal{U}}$, there is a run $s_0 v_0 s_1 v_1 ... v_m s_{m+1}$ of $\mathcal{U}_\Lambda$ and this run is accepting (as any run of $\mathcal{U}_\Lambda$ is accepting. □

Example: Universal Stutter-free Automaton

In Fig. 8 below, we depict the universal stutter-free automaton $\mathcal{U}_X$ for the set of boolean variables $X = \{x, y\}$.

Definition 7 Let $\mathcal{A} = (Q, \hat{Q}, F, \delta)$ be a stutter-free automaton over the set of variables X with domain Σ. The *completion* of $\mathcal{A}$ defines the stutter-free automaton $\text{complete}(\mathcal{A}) = (Q \cup Q_{\mathcal{U}}, \hat{Q}, F, \delta')$ over the same variables, where $Q_{\mathcal{U}}$ are the states of the universal stutter-free automaton $\mathcal{U}_\Lambda$, and the transition relation is defined as

$$\delta' = \delta \cup \delta_{\mathcal{U}} \cup \{(q \xrightarrow{v} v) \mid \text{if } v \in (allowed_{sf}(q) \setminus \Lambda_{out}(q)), q \in Q\}.$$

where $\delta_{\mathcal{U}}$ is the transition relation of the universal left-aligned automaton $\mathcal{U}_\Lambda$.
We can now prove that our completion procedure returns an automaton accepting the same language as the input automaton.

Proposition 15 *Let $\mathcal{A}$ be a stutter-free automaton. Then,* $\text{complete}(\mathcal{A})$ *is a complete stutter-free automaton with the same language as $\mathcal{A}$, i.e.,* $\mathcal{L}(\text{complete}(\mathcal{A})) = \mathcal{L}(\mathcal{A})$.

Proof Consider an arbitrary stutter-free automaton $\mathcal{A}$. Both $\mathcal{A}$ and the universal automaton are stutter-free automata, satisfying the stutter-free and termination requirements. To prove that $\text{complete}(\mathcal{A})$ is a stutter-free automaton, we are only missing to prove that the extension of the $\mathcal{A}$ transition relation (pointing to states in the universal automaton) preserves both requirements. By definition of $allowed_{sf}$, if a variable trace is terminated, it will remain terminated, so termination is satisfied; and if it is not terminated, we only add the transitions labeled with values not seen in the incoming and outgoing transitions of a state, hence it satisfies the stutter-free requirement. We now prove that $\text{complete}(\mathcal{A})$ is complete. By the definition of universal automaton, it follows that universal automata are complete. For all non-complete transitions in $\mathcal{A}$ (i.e., variables that are not terminated yet), we add the

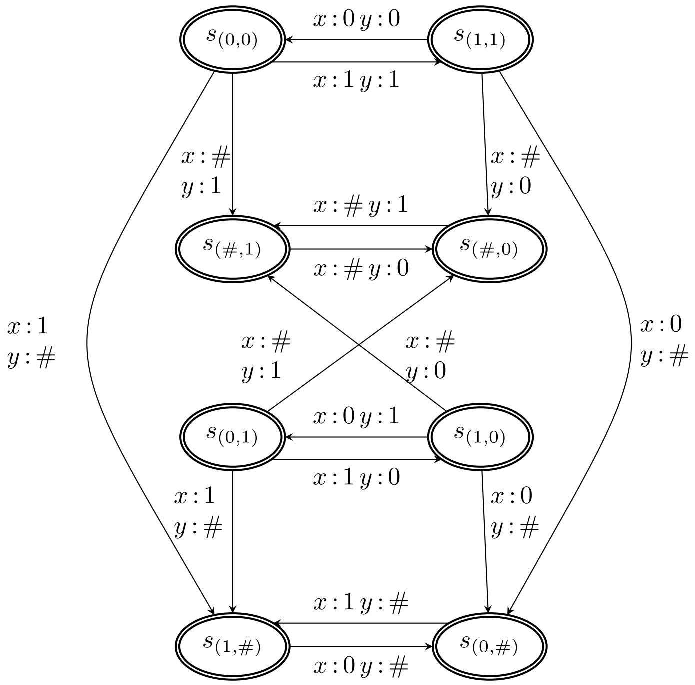


Fig. 8 The universal stutter-free automaton $\mathcal{U}_{\{x,y\}}$ over the boolean variables $\{x, y\}$. It accepts all stutter-free unzipped trace segments over $\{x, y\}$. All states are both initial and final

missing transition pointing to the matching universal automaton state; hence, $\text{complete}(\mathcal{A})$ is complete.

Finally, we prove that both automata define the same language. As we kept all $\mathcal{A}$ transitions, it follows that $\mathcal{L}(\text{complete}(\mathcal{A})) \supseteq \mathcal{L}(\mathcal{A})$. We now prove that $\mathcal{L}(\text{complete}(\mathcal{A})) \subseteq \mathcal{L}(\mathcal{A})$. First, we note that no transition connects states from the universal automaton to states from $\mathcal{A}$. Then, when a run reaches a state from the universal automaton, all the following steps are within the universal automaton. Additionally, the final states do not include states from the universal automaton. Thus, accepting runs include only transitions in $\mathcal{A}$. □

Finally, to complement a deterministic and complete stutter-free automaton, we just flip final with non-final states, i.e., the complement of $\mathcal{A} = (Q, \hat{Q}, F, \delta)$ is $\overline{\mathcal{A}} = (Q, \hat{Q}, Q \setminus F, \delta)$.

Proposition 16 *Let $\mathcal{A}$ be a deterministic and complete stutter-free automaton over Λ.*

Then, $\overline{\mathcal{A}}$ is a stutter-free automaton and $\mathcal{L}(\overline{\mathcal{A}}) = \lfloor (\Sigma^*)^X \rfloor \setminus \mathcal{L}(\mathcal{A})$.

Proof The proof is analogous to the same proof for left-aligned automata. □

5.1.1 From formulas of hypernode logic to stutter-free automata

Similar to what we did before for the synchronous fragment, we show how to translate atomic formulas of the asynchronous fragment of the extended hypernode logic into stutter-free automata. The translation is basically the same as in the synchronous case, but in the constructions, we must use the universal stutter-free automaton instead of the left-aligned universal automaton. Also, we have to stutter-reduce automata for regular expressions, for which we use a procedure described below. In the subsequent text, we assume an alphabet $\Lambda(X, \Sigma)$ and that $\mathcal{U}_\Lambda$, the universal stutter-free automaton with alphabet Λ, is the tuple $(Q_\mathcal{U}, Q_\mathcal{U}, Q_\mathcal{U}, \delta_\mathcal{U})$.

From deterministic finite automata to stutter-free automata

We start by defining how to translate a DFA, $\mathcal{A}$, into an NFA with no stuttering transitions, $\mathcal{A}_{\mathrm{st}}$, accepting all words that are a stutter-reduction of words accepted by $\mathcal{A}$. Formally, the resulting stutter-free NFA should satisfy: $\mathcal{L}(\mathcal{A}_{\mathrm{st}}) = \lfloor \mathcal{L}(\mathcal{A}) \rfloor$. The states of the stutter-free NFA are named using a letter from the domain of $\mathcal{A}$ and a set of states from $\mathcal{A}$. The letter encodes the label of the incoming transition to the current state, ensuring that no stuttering steps are introduced during the translation. Additionally, we use the auxiliary function $\mathrm{stReach}(q, \sigma)$, defining all states reachable from q only with transitions labeled with σ.

Definition 8 The *stutter-free NFA induced by the DFA* $\mathcal{A} = (Q, \hat{Q}, F, \delta)$ is $\mathcal{A}_{\mathrm{st}} = (\Sigma \times 2^Q, \Sigma \times \hat{Q}, \Sigma \times 2^F, \delta_{\mathrm{st}})$ where:

$$\delta_{\mathrm{st}}((\sigma, S), \sigma') = \{(\sigma', S') \mid q \in S, \sigma \neq \sigma' \text{ and } S' \in \mathrm{stNext}(q, \sigma')\} \text{ with}$$

$$\mathrm{stNext}(q, \sigma) = \{q' \in \mathrm{stReach}(q, \sigma) \mid \delta(q', \sigma') = q'' \text{ and } \sigma \neq \sigma'\} \cup \bigcup_{q' \in \mathrm{stReach}(q,\sigma) \cap F} \{\{q'\}\}$$

$$\mathrm{stReach}(q, \sigma) = \{q' \mid q\sigma_0 q_1 \sigma_1 \ldots \sigma_n q' \text{ is a path of } \mathcal{A} \text{ and } \forall 0 \leq i \leq n\ \sigma_i = \sigma\}.$$

Example: Stutter-reducing DFA

The DFA $\mathcal{A}$ in Fig. 9 defines the language $\mathcal{L}(\mathcal{A}) = \{pq, pp, pppo\}$. We will show how to build its induced stutter-free automaton $\mathcal{A}_{\mathrm{st}}$, which accepts the language $\mathcal{L}(\mathcal{A}_{\mathrm{st}}) = \{pq, p, po\}$.

We start by evaluating the function stReach for each state:

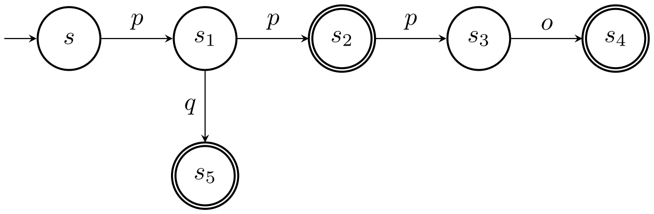


Fig. 9 DFA $\mathcal{A}$ for stutter-reduction

stReach	s	s_1	s_2	s_3	s_4	s_5
p	$\{s_1, s_2, s_3\}$	$\{s_2, s_3\}$	$\{s_3\}$	$\emptyset$	$\emptyset$	$\emptyset$
o	$\emptyset$	$\emptyset$	$\emptyset$	$\{s_4\}$	$\emptyset$	$\emptyset$
q	$\emptyset$	$\{s_5\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

We look now at the transition relation of $\mathcal{A}_{\text{st}}$, focusing only on the reachable states. We start with the initial state, which has only transitions with p: $\text{stNext}(\{s\}, p) = \{\{s_1, s_2, s_3\}, \{s_2\}\}$. From the definition of the transition relation, $\delta_{\text{st}}((o, \{s\}), p) = \delta_{\text{st}}((q, \{s\}), p) = \{(p, \{s_1, s_2, s_3\}), (p, \{s_2\})\}$ while $\delta_{\text{st}}((p, \{s\}), p) = \emptyset$. To derive the transitions from $(p, \{s_1, s_2, s_3\})$ and $(p, \{s_2\})$ we first evaluate their stNext:

$$\text{stNext}(\{s_1\}), q) = \{\{s_5\}\} \quad \text{stNext}(\{s_1\}), o) = \emptyset$$
$$\text{stNext}(\{s_2\}), q) = \text{stNext}(\{s_2\}), o) = \emptyset$$
$$\text{stNext}(\{s_3\}), q) = \emptyset \quad \text{stNext}(\{s_3\}), o) = \{\{s_4\}\}$$

As there are no transitions from s_4 and s_5, we have all we need to draw $\mathcal{A}_{\text{st}}$, which is depicted in Fig. 10.

Proposition 17 *Let $\mathcal{A}$ be a DFA and $\mathcal{A}_{st}$ be the stutter-free NFA induced from $\mathcal{A}$. $\mathcal{L}(\mathcal{A}_{st}) = \lfloor \mathcal{L}(\mathcal{A}) \rfloor$.*

Proof We start by observing that by $\mathcal{A}$ being a DFA and the construction of $\mathcal{A}_{\text{st}}$, $(\star)$ for all states (σ, S) from $\mathcal{A}_{\text{st}}$ the set S (of states of $\mathcal{A}$) defines a path in $\mathcal{A}$ with steps connected with transitions labeled with σ. This is, there exists a path of $\mathcal{A}$ s.t. $q_0 \sigma q_1 \sigma \ldots \sigma q_n$ with $\{q_0, \ldots, q_n\} = S$. Moreover, $(\star\star)$ given a state in the transition function $(\sigma', S') \in \delta_{\text{st}}((\sigma, S), \sigma')$, where S' is not a single set with a final state, by construction, we have that there exists $q' \in S'$ s.t. $\text{stReach}(q', \sigma) = S'$.

We prove $\mathcal{L}(\mathcal{A}_{\text{st}}) \subseteq \lfloor \mathcal{L}(\mathcal{A}) \rfloor$ by induction on the size of paths of $\mathcal{L}(\mathcal{A}_{\text{st}})$. We explore the induction case, as it is the most interesting. Consider an arbitrary accepting path of $\mathcal{A}_{\text{st}}$: $(\sigma, S_0)\sigma_0(\sigma_0, S_1)\sigma_1 \ldots (\sigma_{n-1}, S_n)\sigma_n(\sigma_n, S_{n+1})$. By the induction hypothesis, we know there exists a path $q'_0 \sigma'_0 q'_1 \sigma'_1 \ldots \sigma'_{m-1} q'_m$ of $\mathcal{A}$ s.t. $\sigma_0 \ldots \sigma_{n-1} = \lfloor \sigma'_0 \ldots \sigma'_{m-1} \rfloor$ and $q'_m \in S_n$. Additionally, we know that after the last occurrence of σ_{n-2} the tail of the path has only transitions with σ_{n-1}. Formally, there exists $k > 1$ s.t. $\sigma_{m-(k+1)} = \sigma_{n-2}$ and $\sigma_{m-k} \ldots \sigma_{m-1} = \sigma^k_{n-1}$. If S_{n+1} is not a single set with a final state, let $q_{\text{pre}} \in S_n$ be the state satisfying $(\star\star)$ (i.e., $\text{stReach}(q_{\text{pre}}, \sigma_n) = S_{n+1}$) and $q_{\text{pos}} \in \delta(q_{\text{pre}}, \sigma_n)$. Let $\text{path}(q, q', \sigma)$ be a path defined from q to q' reading only the letter σ. Using the induction hypothesis, we can now define the following path of $\mathcal{A}$: $q'_0 \sigma'_0 q'_1 \sigma'_1 \ldots \sigma'_{m-k} \text{path}(q'_{m-k-1}, q_{\text{pre}}, \sigma_{n-1}) \sigma_n q_{\text{pos}}$,

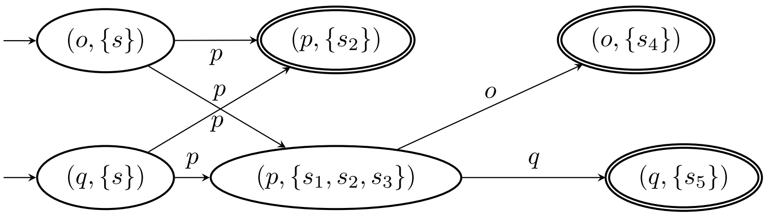


Fig. 10 Induced stutter-free NFA from $\mathcal{A}$

which is a path of $\mathcal{A}$ reading a word $w \in q'_0 \dots (q'_{m-k-1})^+\sigma_n$. By our induction hypothesis, $\lfloor w \rfloor = \sigma_0 \dots \sigma_n$. For the case that, $S_{n+1} = \{q_F\}$ where $q_F \in F$, we have that there exists $q_{\text{pre}} \in S_n$ s.t. $q_F \in \text{stReach}(q', \sigma)$. The rest is analogous.

We sketch the proof for $\mathcal{L}(\mathcal{A}_{\text{st}}) \supseteq \lfloor \mathcal{L}(\mathcal{A}) \rfloor$, which is also done by induction on the size of paths of $\mathcal{L}(\mathcal{A})$. Given a path of $\mathcal{A}$, $q_0\sigma_0 q_1\sigma_1 \dots \sigma_n q_{n+1}$, we can define the matching path in $\mathcal{L}(\mathcal{A}_{\text{st}})$ by merging all states connected by the same letter (for example, a path $p\ s_1\ p\ s_2\ p\ s_3\ q$ becomes $p\ \{s_1, s_2, s_3\}$). We can then match the merged states to states in $\mathcal{A}_{\text{st}}$ with the same letter and possibly a superset of the states of $\mathcal{A}_{\text{st}}$ (for example, we match the previous merged step to the state $(p, \{s_1, s_2, s_3\})$). □

***Automaton for* $\mathbf{x}(\pi) \precsim \psi$**

We first create the finite state automaton $\mathcal{A}_{\lfloor \psi \rfloor} = (Q, \hat{Q}, F, \delta)$ accepting the language of the stutter-reduction of the regular expression ψ. Let $x_\pi \in X$ be the program variable that instantiates $\mathbf{x}(\pi)$. We define the automaton $\mathcal{A}_{\psi \precsim_\Lambda x_\pi} = (Q_\psi \cup Q_\#, \hat{Q} \times Q_\mathcal{U}, Q_\psi \cup Q_\#, \delta_{x_\pi})$ where

- $Q_\psi = Q \times \{s \in Q_\mathcal{U} \mid s(x_\pi) \neq \#\}$ and $Q_\# = \{s \in Q_\mathcal{U} \mid s(x_\pi) = \#\}$
- δ_{x_π} is defined as the smallest set s.t.:
  1) if $q \xrightarrow{\sigma} q' \in \delta$ and $s \xrightarrow{v} s' \in \delta_\mathcal{U}$ and $v(x_\pi) = \sigma$, then $(q, s) \xrightarrow{v} (q', s') \in \delta_{x_\pi}$
  2) if $s \xrightarrow{v} s' \in \delta_\mathcal{U}$ and $s(x_\pi) \neq \#$ and $s'(x_\pi) = \#$, then $(q, s) \xrightarrow{v} s' \in \delta_{x_\pi}$
  3) if $s \xrightarrow{v} s' \in \delta_\mathcal{U}$ and $s(x_\pi) = s'(x_\pi) = \#$, then $s \xrightarrow{v} s' \in \delta_{x_\pi}$

Proposition 18 *Let $\Lambda(X, \Sigma)$ be an alphabet, x_π a variable from X and ψ a regular expression over Σ. $\tau \in \mathcal{L}(\mathcal{A}_{\psi \precsim_\Lambda x_\pi})$ iff exists $v \in \Sigma^*$ s.t. $\tau(x_\pi) = u$ s.t. $uv \in \lfloor \mathcal{L}(\psi) \rfloor$ and, for all variables $y \in (X \setminus \{x_\pi\})$, $\tau(y) \in \lfloor \Sigma^* \rfloor$.*

Proof The proof is analogous the synchronous case. Because all automata that we use are stutter-free, all the involved words and unzipped trace segments are stutter-reduced. □

***Automaton for* $\psi \precsim \mathbf{x}(\pi)$**

We first create the finite state automaton $\mathcal{A}_{\lfloor}\psi\rfloor = (Q, \hat{Q}, F, \delta)$ as in the previous case. Let $x_\pi \in X$ be the program variable that instantiates $\mathbf{x}(\pi)$. We define the automaton $\mathcal{A}_{\psi \precsim_\Lambda x_\pi} = (Q_\psi \cup Q_\mathcal{U}, \hat{Q} \times Q_\mathcal{U}, Q_F, \delta_{x_\pi})$ where

- $Q_\psi = Q \times \{s \in Q_\mathcal{U} \mid s(x_\pi) \neq \#\}$
- $Q_F = Q_\mathcal{U} \cup \{(q, s) \in Q_\psi \mid q \in F\}$
- δ_{x_π} is defined as the smallest set s.t.:
  1) if $q \xrightarrow{\sigma} q' \in \delta$ and $s \xrightarrow{v} s' \in \delta_\mathcal{U}$ and $v(x_\pi) = \sigma$, then $(q, s) \xrightarrow{v} (q', s') \in \delta_{x_\pi}$
  2) if $s \xrightarrow{v} s' \in \delta_\mathcal{U}$ and $q \in F$, then $(q, s) \xrightarrow{v} s' \in \delta_{x_\pi}$
  3) if $s \xrightarrow{v} s' \in \delta_\mathcal{U}$, then $s \xrightarrow{v} s' \in \delta_{x_\pi}$

Proposition 19 *Let $\Lambda(X, \Sigma)$ be an alphabet, x_π a variable from X and ψ a regular expression over Σ. $\tau \in \mathcal{L}(\mathcal{A}_{\psi \precsim_\Lambda x_\pi})$ iff, exists $v \in \Sigma^*$ s.t. $\tau(x_\pi) = uv$, $u \in \lfloor \mathcal{L}(\psi) \rfloor$, and, for all variables $y \in (X \setminus \{x_\pi\})$, $\tau(y) \in \lfloor \Sigma^* \rfloor$.*

Proof The proof is analogous the synchronous case. Because all automata that we use are stutter-free, all the involved words and unzipped trace segments are stutter-reduced. □

Automaton for $\mathbf{x}(\pi) \precsim \mathbf{y}(\pi')$

As in the synchronous case, we construct the automaton for a term $\mathbf{x}(\pi) \precsim \mathbf{y}(\pi')$ by restricting the (now stutter-free) universal automaton. The automaton $\mathcal{A}_{x \precsim_\Lambda y}$ is defined as $\mathcal{A}_{x \precsim_\Lambda y} = (Q, Q, Q, \delta)$ where $Q = \{s \in Q_\mathcal{U} \mid s(x) = s(y) \text{ or } s(x) = \#\}$. The transition relation is defined by the relation $(q, v, q') \in \delta$ iff $(q, v, q') \in \delta_\mathcal{U}$ and $v(x) = v(y)$ or $v(x) = \#$.

Proposition 20 *Let $\Lambda(X, \Sigma)$ be an alphabet and x and y variables from X. $\tau \in \mathcal{L}(\mathcal{A}_{x \precsim_\Lambda y})$ iff $\tau(x) \leq \tau(y) and \tau(x), \tau(y) \in \lfloor \Sigma^* \rfloor$.*

Proof The proof is the same as for the synchronous case. Because all trace segments accepted by the universal stutter-free automaton are stutter-reduced, we also have $\tau(x), \tau(y) \in \lfloor \Sigma^* \rfloor$. □

5.2 Model-checking algorithm

The model checking procedure for the regular asynchronous fragment of hypernode logic against stutter-free automata is exactly the same as model checking synchronous fragment against left-aligned automata (Definition 4), except we must do stutter reduction of all automata that enter the procedure. In particular, the input automaton must be stutter-free and for building the comparison automata we must use constructions discussed in this section. The correctness of this model-checking procedure follows from the fact that stutter-free automata are closed under all operations used in Definition 4.

Theorem 21 *Model checking the regular asynchronous fragment of hypernode logic over stutter-free automata is decidable.*

For model-checking asynchronous hypernode logic against an open Kripke structure, we need to translate it to a NSFA. This cannot be done in general, otherwise we could decide the model-checking problem for asynchronous hypernode formulas and open Kripke structures, which is undecidable for this fragment (see Section 3.2). However, given an open Kripke structure K with the alphabet Σ^X, we can often find a stutter-reduced automaton $\mathcal{A}_K$, recognizing the stutter-reduced unzipped trace segments generated by K. That is, an NSFA $\mathcal{A}_K$ such that:

$$\tau \in \mathcal{L}(\mathcal{A}_K) \text{ iff } \exists \tau' \in \text{Traces}(K, \mathbb{W}) \text{ s.t. } \tau(x) = \lfloor \text{unzip}(\tau')(x) \rfloor \text{ for all } x \in X. \tag{7}$$

We call Kripke structures that can be translated into stutter-free automata *stutter-reducible Kripke structures*.

Definition 9 A partial computable function $\mathcal{S}$ is a *stutter reduction function* for Kripke structures iff for all K in its domain it computes a stutter-free automaton A_K s.t. K and A_K satisfy condition (7). An open Kripke structure K is *stutter-reducible* if there exists a stutter reduction function that has K in its domain.

Assuming the existence of a stutter-reduction function for a particular class of Kripke structures, the model-checking problem for the regular asynchronous fragment of hypernode logic becomes solvable for open Kripke structures within this class.

Corollary 22 *Assume $\mathcal{S}$ is a stutter reduction function for open Kripke structures. If $\mathcal{S}$ has a domain $\mathcal{K}$, then model checking the regular asynchronous fragment of hypernode logic is decidable for Kripke structures from $\mathcal{K}$.*

5.3 From open Kripke structures to stutter-free automata

Unfortunately, it is impossible to decide, in general, if a given Kripke structure can be transformed into a stutter-free automaton. This claim can be derived from the fact that it is undecidable if a 2-tape automaton can be synchronized [23]. If were possible, we could first transform the Kripke structure into a multi-tape automaton, remove stuttering symbols from transitions (possibly creating copies of states to preserve the language) and then check if we can synchronize the resulting automaton.

In what follows, we describe several ways in which one can stutter-reduce specific Kripke structures.

5.3.1 Partial translation of Kripke structures to stutter-free automata

We give a *partial* algorithm that takes an open Kripke structure and returns an NSFA recognizing stutter-reduced unzipped paths of the Kripke structure, or it does not terminate (this case can be turned into a failure by setting a bound on the run of the algorithm as we suggest later).

The core of the algorithm is translating the Kripke structure into a *transducer* (i.e., an automaton with outputs on the transitions). In a nutshell, the transducer reads valuations on the traces of the Kripke structure, remembers them in the states, and uses the remembered valuations to output valuations after stutter reduction. If this construction is finite (i.e., it yields a finite-state transducer) we can derive the goal stutter-free automaton as the projection to the output of the transducer.

Let $K = ((W, \Sigma^X, \Delta, V), (W_{in}, W_{out}))$ be an open Kripke structure, $W^o = \{w^o \mid w \in W\}$ be o-marked copies of states W (o as *output*), and $W^{\#} = \{w^{\#} \mid w \in W\}$ be #-marked copies of states W. A transducer is an automaton with outputs (at this moment not necessarily finite-state) that, in our case, reads valuations from Σ^X and outputs also valuations from Σ^X. We denote a constant valuation $v(x) = c$ for all $x \in X$ simply as c, namely for $c = \epsilon$ and $c = \bot$. A transition of a transducer that reads a valuation v_i and outputs v_o is labeled with v_i/v_o. If $v_o = \epsilon$, we write $v_i/$. Assume that $\bullet \notin W$ and $\bot \notin \Sigma$. The transducer T_K that outputs stutter-reduced traces of K has states $Q \subseteq (W \cup W^o \cup W^{\#} \cup \{\bullet\}) \times (\Sigma^X \cup \{\bot\}) \times (\Sigma^*)^X$ and a transition function δ defined inductively as follows:

1. The initial state $q_0 \in Q$ is $q_0 = (\bullet, \bot, \epsilon)$, and for any initial world $w_i \in W_{in}$, $q' = (w_i, \bot, V(w_i)) \in Q$ and $q_0 \xrightarrow{V(w_i)/} q' \in \delta$.

2. If $q = (w^{\#}, v, u) \in Q$ with $u \neq \epsilon$, then $q' = (w^{\#}, u_0, u_{(1..)}) \in Q$ and $q \xrightarrow{\epsilon/u_0} q' \in \delta$.

Let $q \in Q$, $q = (w, v, v')$, and *u* be an unzipped trace segment s.t. $u(x) = \lfloor (v \cdot v')(x) \rfloor$ for all $x \in X$. Then we have:

3. If $|u(x)| < 2$ for some $x \in X$ and $w \in (W \cup W^o)$. Let $z = w$ if $w \in W$ or z is such that $z^o = w \in W^o$. Then for any $z \to w' \in \Delta$ and $q' = (w', v, v' \cdot V(w'))$: $q \in Q$ and $q \xrightarrow{V(w')/} q' \in \delta$.
4. If $|u(x)| \geq 2$ for all $x \in X$ and $w \in W$, then $q' \in Q$ for $q' = (w'^o, u_1, u_{(2..)})$ and $q \xrightarrow{\epsilon/u_1} q'$.
5. If $w \in W_{out}$, then $q' \in Q$ for $q' = (w^{\#}, u_1, u_{(2..)})$ and $q \xrightarrow{\epsilon/u_1} q' \in \delta$.

Final states of T_K are $(w^{\#}, v, \epsilon) \in Q$. The concatenation of valuations and unzipped trace segments works as expected variable-wise. For $i \geq 0$, the *i*-th element τ_i of an unzipped trace segment $\tau \in \Sigma^X$ is defined as in Section 6, i.e., it is either $\tau_i(x) = \tau(x)_i$ if $i < |\tau(x)|$ or $\tau_i(x) = \#$ otherwise. The suffix operation $\tau_{(i..)}$ is defined as $\mathrm{unzip}(\tau_i \tau_{i+1} ... \tau_{j-1})$ where $\tau_j(x) = \#$ for all $x \in X$ (notice that using unzip also has the effect that the $\#$ symbols are removed).

Proposition 23 *Given an open Kripke structure K with alphabet Σ^X, for the transducer T_K it holds that $\tau \in \mathrm{Traces}(K)$ iff τ is accepted by T_K and $T_K(\tau)(x) = \lfloor \tau(x) \rfloor$ for all $x \in X$.*

Proof First, we make an obervation (*): when applying rule 3, *u* gets increased at most by 1 symbol on each variable. Therefore, whenever rule 4 is applied, it cannot be applied again immediately (from q'), because this rule decreases the length of words in *u*, and initially these words have a length smaller than 2.

Now, for showing that $\tau \in \mathrm{Traces}(K)$ iff τ is accepted by T_K, the output of T_K is irrelevant. We show that the relation $R = \{(w, (w, v, v')) \mid w \in W\}$ is a weak bisimulation between *K* and T_K where we take W^o- and $W^{\#}$-states as silent states. Let $(w, (w, v, v')) \in R$. First, if *w* is an entry world, then from q_0 we reach $(w, \perp, V(w))$ in one step (not passing non-silent states) and $(w, (w, \perp, V(w))) \in R$ and analogously in the other direction. If *w* is an exit world, then from (w, v, v') we reach a final state $(w^{\#}, \perp, \epsilon)$ only through silent states (rule 5 and then repeated 2).

Now, assume that $w \to w' \in \Delta$. Then, in T_K, we can apply either rule 3 or 4 (or 5, but this one will not get us anywhere). Assume rule 3. Then there is a direct transition $(w, v, v') \xrightarrow{V(w')/} (w', \cdot, \cdot)$ into a related state. If rule 4 applied, then (using observation (*)) rule 3 can be applied again and we have the path: $(w, v, v') \xrightarrow{\epsilon/\cdot} (w^o, \cdot, \cdot)) \xrightarrow{V(w')/} (w', \cdot, \cdot)$ which ends in a related state.

For the reverse simulation, assume that there is a path *p* from (w, v_1, v_2) to (w', v'_1, v'_2) through only silent states. Then it must have been through a sequence of transitions from rules 3 and 4 (since after 5 there will be only silent states, and 1 and 2 do not apply). By the same reasoning as above, we get that path *p* was built either only by rule 3 or through the application of rule 3 then 4 and then 3. In both cases, we get that $w \to w' \in Delta$.

Because there is a weak bisimulation between K and T_K on states from W, and T_K reads the valuation Vw precisely when entering a state $(w, \cdot, \cdot)$, T_K accepts exactly the traces of K.

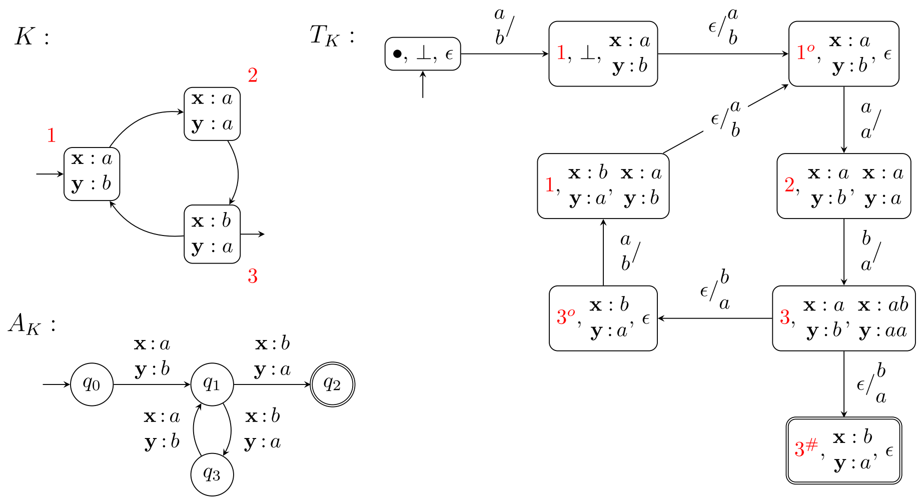
We are left to argue that if T_K accepts a trace τ, the output of T_K on τ is a trace τ' such that $\lfloor \tau(x) \rfloor = \tau'(x)$. First, notice that the second element of a state keeps the last valuation that was output. The third one gathers read valuations (rule 3) until rule 4 or 5 can be applied. If rule 4 was applied, the last output, together with the gathered valuations, has at least two symbols after stutter reduction on each variable (trace segment u), which means a new non-stuttering symbol was found on each variable. These symbols are output, and the rest of the gathered valuations (after variable-wise stutter reduction) are passed to the next state to be used in the future. In the case of rule 5 that handles final states, T_K outputs $u_{(1..)}$, (through rule 5 and then rules 2), which are the stutter-reduced gathered valuations (without the last output) that have not been output yet. □

With the definition of the transducer T_K for an open Kripke structure K, we can define our translation algorithm:

- Remove self-loops from K by replacing every transition $w \to w \in \Delta$ by a transition $w \to w'$ where w' is a copy of w. That is, we also add transitions $w' \to w''$ for any transition $w \to w''$ ($w'' \neq w$) and we set $V(w') = V(w)$.
- Compute T_K.
- Project T_K to the output automaton (this operation turns a transducer into an automaton by replacing any label x/y with y).

An example of a transducer T_K for a Kripke structure K is shown in Fig. 11.

Proposition 24 *Given an open Kripke structure K, under the condition that T_K is finite, the NSFA A_K as defined above satisfies condition (7).*



K :
1
2
3
$\mathbf{x}:a$ $\mathbf{y}:b$
$\mathbf{x}:a$ $\mathbf{y}:a$
$\mathbf{x}:b$ $\mathbf{y}:a$
A_K :
q_0
q_1
q_2
q_3
T_K :
$\bullet, \bot, \epsilon$
$1, \bot, \mathbf{x}:a\ \mathbf{y}:b$
$1^o, \mathbf{x}:a\ \mathbf{y}:b, \epsilon$
$2, \mathbf{x}:a\ \mathbf{y}:b, \mathbf{x}:a\ \mathbf{y}:a$
$3, \mathbf{x}:a\ \mathbf{y}:b, \mathbf{x}:ab\ \mathbf{y}:aa$
$3^o, \mathbf{x}:b\ \mathbf{y}:a, \epsilon$
$1, \mathbf{x}:b\ \mathbf{y}:a, \mathbf{x}:a\ \mathbf{y}:b$
$3^\#, \mathbf{x}:b\ \mathbf{y}:a, \epsilon$

Fig. 11 An open Kripke structure K with entry world 1 and exit world 3, the transducer T_K that computes the stutter reduction of K and the resulting NSFA A_K

Proof Removing self-loops models taking the self-loop just once and it is clear that it does not change the set of stutter-reduced traces as taking a self-loop once is the same as taking it any (non-zero) number of times after removing stuttering. The rest follows from Proposition 23. □

Example: From Open Kripke Structure to Stutter-free Automata

In practice, we have to put a bound on the computation of T_K as it may be infinite. A good bound can be limiting the length of u in rules 3–5. Also, notice that the algorithm is structure-based: when it fails, it does not mean that there does not exists an NSFA that accepts the stutter-reduced traces of the Kripke structure, but only that this particular Kripke structure could not be stutter-reduced with this procedure.

One particular type of open Kripke structures that can be stutter-reduced by the algorithm above are those that generate only finitely-many paths, i.e., *acyclic* (finite) open Kripke structures. These Kripke structures still cover many interesting examples of systems like some one-round cryptographic protocols, or non-recursive *multi-party session types* [21]. In model-checking of concurrent systems, it is also common to *unroll* programs or the transition relation to obtain acyclic system [5, 17]. This approach is not sound, but it is widely used to verify a system at least partially rather than not at all in cases where analyzing the entire system would be infeasible in practice.

5.3.2 Other stutter-reduction functions

Independent Structures To show another situation when we can stutter-reduce a Kripke structure, consider a case when we *know* that the Kripke structure corresponds to a composition of independently running systems – for example, a network of automata without synchronization. In such a case, we can create a projection of the Kripke structure to each variable, stutter-reduce these projections independently (which is always possible as there is only a single variable), and than compose the Kripke structures back together using the parallel lock-step composition.

This construction may succeed also when the partial algorithm fails. Consider, for example, two systems that can have runs of arbitrary length and that run independently. It is easy to do their lock-step composition which will be a stutter-free automaton, but if we first compose them using interleaving composition and then apply the partial algorithm, it will fail as the interleaving composition will contain also paths where the two system diverge arbitrarily (e.g., when one runs after the other).

Synchronized Multi-tape Automata To conclude this section, we remark that there exist an algorithm [24, Theorem 6] to transform an s-synchronized multi-tape automaton (i.e., a multi-tape automaton where any two reading heads never diverge more than s steps unless one of them finished reading) to a synchronous multi-tape automaton. If we transform an open Kripke structure to an automaton and then replace stuttering symbols on transitions by the empty symbol (meaning read nothing), we get an (asynchronous) multi-tape automaton (it may be necessary to duplicate some states to preserve the language while removing stuttering symbols). If it happens that this multi-tape automaton is s-synchronized, we can use the algorithm from [24] to obtain synchronous multi-tape automaton which is also stutter-free.

Although we described three ways how to get stutter-free automata from some open Kripke structures, the class of stutter-reducible Kripke structures is still restricted. In Section 6, we argue that hypernode automata mitigate this limitation to some extent.

6 Hypernode automata

Hypernode automata are finite automata where states are labeled with formulas from hypernode logic, and transitions are labeled with synchronizing actions.

6.1 Syntax and semantics

Hypernode automata are interpreted over action-labeled traces. Let A be a finite set of *actions*, Σ be a finite domain disjoint from A (that is, $A \cap \Sigma = \emptyset$) and X a set of program variables. We denote by $A_\varepsilon = A \cup \{\varepsilon\}$ the set with actions from A together with the empty label ε ($\varepsilon \notin A$).

An *action-labeled trace* ρ over actions A and Σ^X is a finite or infinite sequence of pairs, each consisting of a valuation and either an action label from A, or the empty label ε. Formally, $\rho \in (\Sigma^X \times A_\varepsilon)^*$ or $\rho \in (\Sigma^X \times A_\varepsilon)^\omega$. We use $(\Sigma^X \times A_\varepsilon)^\infty$ to refer to the set with finite and infinite traces; that is, $(\Sigma^X \times A_\varepsilon)^\infty \overset{\text{def}}{=} (\Sigma^X \times A_\varepsilon)^* \cup (\Sigma^X \times A_\varepsilon)^\omega$. We require, for technical simplicity, that for infinite action-labeled traces, infinitely many labels are nonempty (and, thus, that every trace segment is finite).

The *action sequence* $act(\rho)$ of an action-labeled trace $\rho = (v_0, a_0)(v_1, a_1)(v_2, a_2) \ldots$, where $v_i \in \Sigma^X$ and $a_i \in A_\varepsilon$ for all $i \geq 0$, is the projection of the trace to its actions, with all empty labels ε removed; that is, $act(\rho) = a'_0 a'_1 \ldots$ with $a_0 a_1 \ldots \in a'_0 \varepsilon^* a'_1 \varepsilon^* \ldots$ and $a'_i \in A$ for all $i \geq 0$. Given a set of action-labeled traces $R \subseteq (\Sigma^X \times A_\varepsilon)^\infty$, the *projection of R with respect to a finite action sequence* $p \in A^*$ is $R[p] = \{\rho \in R \mid p \leq act(\rho)\}$.

An *action-labeled trace property* is a set of action-labeled traces. A hypernode automaton accepts action-labeled trace properties, and thus specifies an *action-labeled hyperproperty*.

Definition 10 A deterministic, finite *hypernode automaton* (HNA) over a set of actions A and a set of program variables X is a tuple $\mathcal{H} = (Q, \hat{q}, \gamma, \delta)$, where Q is a finite set of states with $\hat{q} \in Q$ being the initial state, the state labeling function γ assigns a closed formula of hypernode logic over the program variables X to each state in Q, and the transition function $\delta : Q \times A \to Q$ is a total function assigning to each state and action a unique successor state.

We assume the totality and determinism of the transition function only for the simplicity of the technical presentation. A *run* of the HNA $\mathcal{H}$ is a finite or infinite sequence $r = q_0 a_0\, q_1 a_1\, q_2 a_2 \ldots$ of alternating hypernodes and actions which starts in the initial hypernode $q_0 = \hat{q}$ and follows the transition function, i.e., $\delta(q_i, a_i) = q_{i+1}$ for all $i \geq 0$. We refer to the corresponding sequence $p = a_0 a_1 a_2 \ldots$ of actions as the *action sequence* of r. Note that each action sequence defines a unique run of $\mathcal{H}$.

Each step in the run of a hypernode automaton defines a new *slice* on a set of action-labeled traces. Let R be a set of action-labeled traces, $p = a_0 a_1 \ldots a_n$ be a finite action sequence, and $\rho = (v_0, a'_0)(v_1, a'_1)\ldots$ be an action-labeled trace s.t. $\rho \in R[p]$. We write $\rho(\varnothing, a_0)$ for the initial trace segment of ρ which ends with the action label a_0. Formally, $\rho(\varnothing, a_0) = v_0 \ldots v_k$ such that $a'_k = a_0$, and $a'_i = \varepsilon$ for all $0 \leq i < k$. Furthermore, we write $\rho(a_0 a_1 \ldots a_i, a_{i+1})$ for the subsequent trace segment of ρ which ends with the action label a_{i+1} after having seen the action sequence $a_0 a_1 \ldots a_i$. Inductively, if:

- $\rho(a_0 a_1 \ldots a_{i-1}, a_i) = v_l \ldots v_k$ where $a'_{l-1} = a_{i-1}$ and $a'_k = a_i$,
- $a_{i+1} = a'_m$ for some $m > k$, and
- for all $k < j < m$, $a'_j = \varepsilon$

then $\rho(a_0 a_1 \ldots a_i, a_{i+1}) = v_{k+1} \ldots v_m$. We extend slicing to sets of action-labeled traces accordingly:

$$R(\varnothing, a) = \{\rho(\varnothing, a) \mid \rho \in R\} \text{ and } R[p](a_0 a_1 \ldots a_i, a_{i+1}) = \{\rho(a_0 a_1 \ldots a_i, a_{i+1}) \mid \rho \in R[p]\}.$$

Definition 11 Let $\mathcal{H} = (Q, \hat{q}, \gamma, \delta)$ be an HNA, and R a set of action-labeled traces. Let p be a finite action sequence in A^*. The set R is *accepted* by $\mathcal{H}$ with respect to the action sequence $p = a_0 \ldots a_n$, denoted $R \models_p \mathcal{H}$, iff for the run $q_0 a_0\, q_1 a_1 \ldots q_n a_n$, all slices of R induced by p are models of the formulas that label the respective hypernodes; that is, $R[p](\varnothing, a_0) \models \gamma(q_0)$, and $R[p](a_0 \ldots a_{i-1}, a_i) \models \gamma(q_i)$ for all $0 < i \leq n$.

A set R of action-labeled traces is *accepted* by the HNA $\mathcal{H}$ iff for all finite action sequences $p \in A^*$, if $R[p] \neq \emptyset$, then $R \models_p \mathcal{H}$. The *language* accepted by $\mathcal{H}$ is the set of all sets of action-labeled traces that are accepted by $\mathcal{H}$, denoted $\mathcal{L}(\mathcal{H})$. Note that this definition assumes that all finite and infinite runs of HNA are feasible; such automata are often called *safety automata*. Refinements are possible where finite runs must end in final states or, for example, infinite runs must visit final states infinitely often.

6.2 Model-checking hypernode automata

We study the model-checking problem for hypernode automata over Kripke structures whose transitions are labeled with actions.

Given a Kripke structure K with a transition relation Δ, and given a set A of actions, an *action labeling* for K over A is a function $\mathbb{A} : \Delta \to 2^{A_\varepsilon}$ that assigns a set of action labels (including possibly the empty label ε) to each transition.

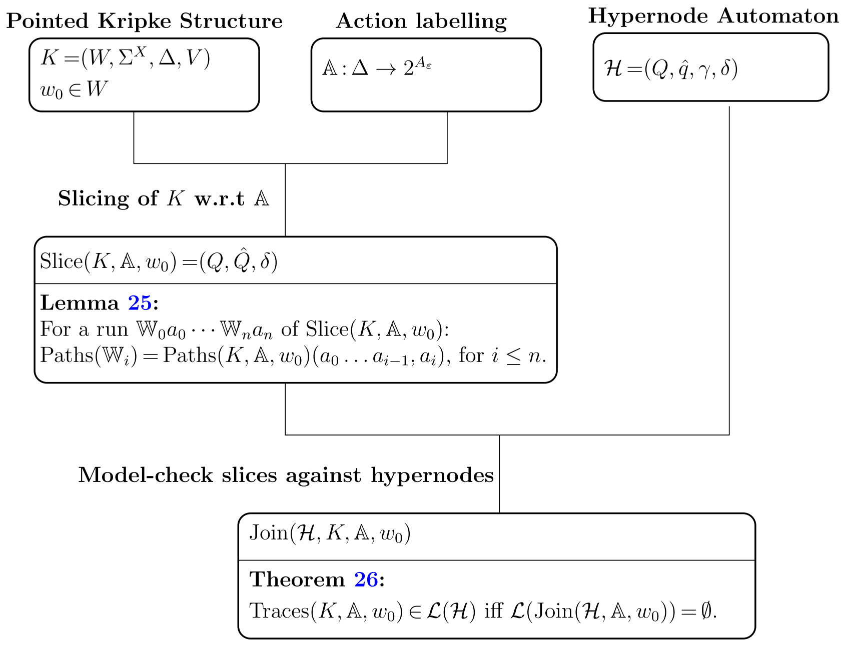
A *path* in the Kripke structure K with action labeling $\mathbb{A}$ is a finite or infinite sequence $w_0 a_0\, w_1 a_1\, w_2 a_2 \ldots$ of alternating worlds and actions which respects both the transition relation, $(w_i, w_{i+1}) \in \Delta$, and the action labeling, $a_i \in \mathbb{A}(w_i, w_{i+1})$, for all $i \geq 0$. We write $\mathrm{Paths}(K, \mathbb{A})$ for the set of all such paths. The path $\varrho = w_0 a_0\, w_1 a_1 \ldots$ defines the action-labeled trace $V(w_0) a_0\, V(w_1) a_1 \ldots$. We write $\mathrm{Traces}(K, \mathbb{A})$ for the set of action-labeled traces defined by paths in $\mathrm{Paths}(K, \mathbb{A})$. By $\mathrm{Paths}(K, \mathbb{A}, w_0)$ we denote the set of all paths in $\mathrm{Paths}(K, \mathbb{A})$ that start at the world w_0. As before, $\mathrm{Traces}(K, \mathbb{A}, w_0)$ refers to the set of all action-labeled traces that are defined by paths in $\mathrm{Paths}(K, \mathbb{A}, w_0)$.

We are now ready to formally define the central verification question solved in this paper, namely, the model-checking problem for specifications given as hypernode automata over models given as pointed Kripke structures (i.e., Kripke structures paired with a designated initial world) with action labeling. The conversion of concurrent programs, such as those from Section II, into a pointed Kripke structure with action labeling is straightforward; its formalization is omitted here for space reasons.

Model-checking problem for hypernode automata Let (K, w_0) be a pointed Kripke structure for set of variables X over a finite domain Σ, and let $\mathbb{A}$ be an action labeling for K over a set A of actions disjoint from the domain. Let $\mathcal{H}$ be a hypernode automaton over the same set X of variables, domain Σ and set A of actions. Is the set of action-labeled traces generated by $(K, \mathbb{A}, w_0)$ accepted by $\mathcal{H}$; that is, $\text{Traces}(K, \mathbb{A}, w_0) \in \mathcal{L}(\mathcal{H})$?

Model-checking hypernode automata requires slicing the model according to its action-labeled steps and matching the slices to their respective hypernode specification. We introduce the slicing of a pointed Kripke structure $(K, \mathbb{A}, w_0)$ with action labeling $\mathbb{A}$, as a finite automaton, called $\text{Slice}(K, \mathbb{A}, w_0)$. We then solve the model-checking problem by composing the slicing of our model, $(K, \mathbb{A}, w_0)$, with the specification automaton, $\mathcal{H}$, and checking whether the result, $\text{Join}(\mathcal{H}, K, \mathbb{A}, w_0)$, is non-empty. In a nutshell, the composite automaton accepts all sequences of actions that witness a violation of the specification of the hypernode automaton by the Kripke structure and action label given as a model. We depict an overview of the model-checking algorithm in Fig. 12.

We start by defining the *slicing* of a given Kripke structure $K = (W, X, \Delta, V)$ and action labeling $\mathbb{A}$. The building blocks of this slicing are Kripke substructures induced by a subset of the transition relation in the input model for the model-checking algorithm. Formally, for a Kripke structure $K = (W, X, \Delta, V)$ and action labeling $\mathbb{A}$, the *substructure induced by a transition relation* $\Delta' \subseteq \Delta$ is $K[\Delta'] = (W', X, \Delta', V|_{W'})$, where



Pointed Kripke Structure
$K = (W, \Sigma^X, \Delta, V)$
$w_0 \in W$
Action labelling
$\mathbb{A} : \Delta \to 2^{A_\varepsilon}$
Hypernode Automaton
$\mathcal{H} = (Q, \hat{q}, \gamma, \delta)$
Slicing of K w.r.t $\mathbb{A}$
$\text{Slice}(K, \mathbb{A}, w_0) = (Q, \hat{Q}, \delta)$
Lemma 25:
For a run $\mathbb{W}_0 a_0 \cdots \mathbb{W}_n a_n$ of $\text{Slice}(K, \mathbb{A}, w_0)$:
$\text{Paths}(\mathbb{W}_i) = \text{Paths}(K, \mathbb{A}, w_0)(a_0 \ldots a_{i-1}, a_i)$, for $i \le n$.
Model-check slices against hypernodes
$\text{Join}(\mathcal{H}, K, \mathbb{A}, w_0)$
Theorem 26:
$\text{Traces}(K, \mathbb{A}, w_0) \in \mathcal{L}(\mathcal{H})$ iff $\mathcal{L}(\text{Join}(\mathcal{H}, \mathbb{A}, w_0)) = \emptyset$.

Fig. 12 Model-checking algorithm for hypernode automata with relevant results

$W' = \{w \mid (w, w') \in \Delta' \text{or } (w', w) \in \Delta' \text{ for some } w' \in W\}$ and $V|_{W'} : (W' \times X \to \Sigma)$ s.t., for all worlds $w \in W'$, $V|_{W'}(w) = V(w)$.

We are interested in open sub-structures that, starting from a given entry set of worlds, include all transitions required to reach the next transition labeled with action a, using only transitions labeled with the empty action. The transition relation defined by *all transitions in a path* of the action-labeled Kripke structure $(K, \mathbb{A})$ from a world in W_{in} to the first step labeled with action $a \in A$ is defined as:

$$(K, \mathbb{A}, W_{\text{in}}) \downharpoonleft a = \{(w_j, w_{j+1}) \mid w_0\varepsilon \ldots w_{n-1}\varepsilon\, w_n a \in \text{Paths}(K, \mathbb{A}), w_0 \in W_{\text{in}} \text{ and } j < n\}.$$

Once we have all transitions from a given initial set of worlds to the next step labeled with action a, we can straightforwardly define the *open substructure induced by* $(K, \mathbb{A}, W_{\text{in}}) \downharpoonleft a$, which we denote by $\mathbb{K}[(K, \mathbb{A}, W_{\text{in}}) \downharpoonleft a]$. Formally, $\mathbb{K}[(K, \mathbb{A}, W_{\text{in}}) \downharpoonleft a] = (K_s, (W_{\text{in}}, W_{\text{out}}))$ where $K_s = K[(K, \mathbb{A}, W_{\text{in}}) \downharpoonleft a]$ and the set $W_{\text{out}} = \{w \mid (w', w) \in (K, \mathbb{A}, W_{\text{in}}) \downharpoonleft a \text{ and } (w, w'') \in \Delta \text{ s.t. } a \in \mathbb{A}(w, w'')\}$ contains all possible exit points with action a.

We define now the finite automaton $\text{Slice}(K, \mathbb{A}, w_0)$ encoding all possible slicings of the pointed action-labeled Kripke structure $(K, \mathbb{A}, w_0)$. The states of the automaton $\text{Slice}(K, \mathbb{A}, w_0)$ are all open substructures induced by paths from any set of entry worlds to the next step with a matching action $a \in \mathbb{A}$. Then, the transition relation of $\text{Slice}(K, \mathbb{A}, w_0)$ connects, for all actions a, open substructures where the exit worlds of the source open structure can transition with action *a* (in the original Kripke structure) to the entry worlds of the target structure.

Definition 12 Let (K, w_0) be a pointed Kripke structure with worlds *W*, and let $\mathbb{A}$ be an action labeling for K with actions A. The slicing $\text{Slice}(K, \mathbb{A}, w_0) = (Q, \hat{Q}, \delta)$ is a finite automaton where:

- $Q = \{\mathbb{K}[(K, \mathbb{A}, W_{\text{in}}) \downharpoonleft a] \mid a \in A \text{ and } W_{\text{in}} \subseteq W\}$ is a set of states and $\hat{Q} = \{(K, (\{w_0\}, W_{\text{out}})) \in Q \mid W_{\text{out}} \subseteq W\}$ is the set of initial states;
- $\delta : Q \times A \to Q$ is a transition function s.t. $\delta((K, (W_{\text{in}}, W_{\text{out}})), a) = (K', (W'_{\text{in}}, W'_{\text{out}}))$ iff:

  – $(K, (W_{\text{in}}, W_{\text{out}}))$ exits with action a, that is, for all $w \in W_{\text{out}}$ there exists $w' \in W$ such that $a \in \mathbb{A}(w, w')$; and
  – the set of entry worlds W'_{in} define a maximal subset of the worlds accessible with action a from the exit worlds in $\mathbb{W}$; that is, for all $(K'', (W''_{\text{in}}, W''_{\text{out}})) \in Q$ that are different from $(K', (W'_{\text{in}}, W'_{\text{out}}))$:

$$\text{if } W''_{\text{in}} \subseteq \{w \mid a \in \mathbb{A}(w', w) \text{ for some } w' \in W_{\text{out}}\}, \text{ then } W'_{\text{in}} \not\subseteq W''_{\text{in}}.$$

We remark that, for all open Kripke structures $\mathbb{K}$ and actions a, there is a unique maximal set of worlds accessible from the exit worlds of $\mathbb{K}$ through a transition labeled with a. Note that, for every open Kripke substructure defined as $\mathbb{K}[(K, \mathbb{A}, W_{\text{in}}) \downarrow a])$ and $\mathbb{K}[(K, \mathbb{A}, W'_{\text{in}}) \downarrow a]$, their union defines $\mathbb{K}[(K, \mathbb{A}, W_{\text{in}} \cup W'_{\text{in}}) \downarrow a]$, which is a state of the slicing.

We prove, in Lemma 25 below, that every finite action sequence p defines a unique path in this automaton, and the slices in each path contain the same trace segments as the traces derived from the action-labeled Kripke structure with action pattern p.

Lemma 25 *Let (K, w_0) be a pointed Kripke structure, and $\mathbb{A}$ an action labeling for K with actions A. For every finite action sequence $p = a_0 \dots a_n$ in A^*, if $\text{Traces}(K, \mathbb{A}, w_0)[p] \neq \emptyset$, then p defines a unique run $\mathbb{K}_0\, a_0 \cdots \mathbb{K}_n a_n$ of $\text{Slice}(K, \mathbb{A}, w_0)$ such that for all $0 \leq i \leq n$, $\text{Paths}(\mathbb{K}_i) = \text{Paths}(K, \mathbb{A}, w_0)(a_0 \dots a_{i-1}, a_i)$.*

Proof Consider an arbitrary Kripke structure $K = (W, \Sigma^X, \Delta, V)$, world $w_0 \in K$ and action labeling $\mathbb{A} : (W \times A) \to W$. From the transition function of $\text{Slice}(\mathbb{A}(K, w_0))$ being deterministic, it follows that all action sequences $p \in A^*$ in (K, w_0) with labeling $\mathbb{A}$, i.e. $\text{Traces}(\mathbb{A}(K, w_0))[p] \neq \emptyset$, define a unique path in $\text{Slice}(\mathbb{A}(K, w_0))$.

We still need to prove that paths defined by a slice are the same as slicing the paths generated by $\mathbb{A}(K, w_0)$, which we prove by induction on the size of the sequence. For the base case, for all sequence actions of size 1, a_0, the induced path in $\text{Slice}(\mathbb{A}(K, w_0))$ is $\mathbb{K}_0$, then we need to prove that $\text{Paths}(\mathbb{K}_0) = \text{Paths}(\mathbb{A}(K))(\varnothing, a_0)$. By Definition 12, $\text{Paths}(\mathbb{K}_0) = \text{Paths}((\mathbb{K}[(K, \{w_0\}) \downharpoonleft a_0]))$. And, by definition open substructure induced by a, $\text{Paths}(\mathbb{K}[(K, \{w_0\}) \downharpoonleft a_0]) = \{w_0 \dots w_n \mid (w_0, \varepsilon) \dots (w_{n-1}, \varepsilon)(w_n, a_0) \in \text{Paths}(K)\}$. Thus, $\text{Paths}(\mathbb{K}[(K, \{w_0\}) \downharpoonleft a_0]) = \text{Paths}(\mathbb{A}(K))(\varnothing, a_0)$.

Now for the induction step, we assume as induction hypothesis (IH) that the statement holds for sequences of size n. Consider now a sequence of size $n + 1$, $a_0 \dots a_n$. By IH, we know that $\text{Paths}(\mathbb{K}_i) = \text{Paths}(\mathbb{A}(K))(a_0 \dots a_{i-1}, a_i)$ for all $0 \leq i < n$. We are only missing to prove that $\text{Paths}(\mathbb{K}_n) = \text{Paths}(\mathbb{A}(K))(a_0 \dots a_{n-1}, a_n)$. By IH, we know that $\text{Paths}(\mathbb{K}_{n-1})$ and $\text{Paths}(\mathbb{A}(\mathbb{K}))(a_0 \dots a_{n-2}, a_{n-1})$ have the same terminal states. Then, it follows that $\text{Paths}(\mathbb{K}_n)$ and $\text{Paths}(\mathbb{A}(\mathbb{K}))(a_0 \dots a_{n-1}, a_n)$ have the same initial states W_{in}. And, from an analogous reasoning from the base case, $\text{Paths}(\mathbb{K}_n) = \text{Paths}(\mathbb{K}[(K, W_{\text{in}}) \downharpoonleft a_n]) = \text{Paths}(\mathbb{A}(K))(a_0 \dots a_{n-1}, a_n)$ $\square$

The final step in the model-checking procedure is the synchronous composition of the slicing automaton (derived from our model) with the hypernode automaton given as our specification. Naturally, the states of this composition are pairs of open Kripke substructures and hypernode logic formulas. Our goal is to reduce the model-checking problem to check if the composite automaton is non-empty. In particular, we say that the Kripke structure is not a model of the specification automaton $\mathcal{H}$, if we can reach a final state in the composite automaton. For this reason, in the composition defined below, the final states are all the states defined by a pair with an open Kripke substructure that is not a model of its paired hypernode formula.

Definition 13 Let $\mathcal{H} = (Q_h, \hat{q}, \gamma, \delta_h)$ be a hypernode automaton. The *intersection of $\mathcal{H}$ with the slicing of a pointed, action-labeled Kripke structure* $(K, \mathbb{A}, w_0)$, $\text{Slice}(K, \mathbb{A}, w_0) = (Q_s, \hat{Q}_s, F_s, \delta_s)$, is the finite automaton $\text{Join}(\mathcal{H}, K, \mathbb{A}, w_0) = (Q, \hat{Q}, F, A, \delta)$ where:

- $Q = \{(\mathbb{K}, q) \mid \mathbb{K} \in Q_s, q \in Q_h \text{ and } \mathbb{W} \models \gamma(q)\} \cup \{(\mathbb{K}, \overline{q}) \mid \mathbb{K} \in Q_s, q \in Q_h \text{ and } \mathbb{K} \not\models \gamma(q)\}$ is the set of states;

- initial state $\hat{Q} = \{(\mathbb{K}, \hat{q}) \in Q \mid \mathbb{K} \in \hat{Q}_s\} \cup \{(\mathbb{K}, \overline{\hat{q}}) \in Q \mid \mathbb{K} \in \hat{Q}_s\}$;
- final state $F = \{(\mathbb{K}, \overline{q}) \mid (\mathbb{K}, \overline{q}) \in Q\}$; and
- transition function $\delta : Q \times A \rightarrow Q$, where for all $(\mathbb{K}, q) \in Q$, we have $\delta((\mathbb{K}, q), a) = \{(\mathbb{K}', q') \in Q \mid \delta(q) = (q', a) \text{ and } \mathbb{K}' \in \delta_s(\mathbb{W}, a)\}$.

The finite automaton $\mathrm{Join}(\mathcal{H}, K, \mathbb{A}, w_0)$ reads sequences of actions. We define its runs and accepting runs as usual. The language of the automaton is empty, if there are no accepting runs.

Theorem 26 *Let (K, w_0) be a pointed Kripke structure with action labeling $\mathbb{A}$. Let $\mathcal{H}$ be a hypernode automaton over the same set of propositions and actions as $(K, \mathbb{A})$. Then, $\mathrm{Traces}(K, \mathbb{A}, w_0) \in \mathcal{L}(\mathcal{H})$ iff the language of the finite automaton $\mathrm{Join}(\mathcal{H}, K, \mathbb{A}, w_0)$ is empty.*

Proof Consider arbitrary $K = (W, \Sigma^X, \Delta, V)$ over a domain Λ, a world $w_0 \in W$ and a action labeling $\mathbb{A} : (W \times A) \rightarrow W$. We want to prove that $\mathrm{Traces}(\mathbb{A}(K, w_0)) \notin \mathcal{L}(\mathcal{H})$ iff $\mathcal{L}(\mathrm{Join}(\mathcal{H}, \mathbb{A}(K, w_0))) \neq \emptyset$.

$\mathrm{Traces}(\mathbb{A}(K, w_0)) \notin \mathcal{L}(\mathcal{H})$ iff there exists a sequence of actions $p = a_0 \ldots a_n$ that is in $\mathrm{Traces}(\mathbb{A}(K, w_0))$, i.e. $\mathrm{Traces}(\mathbb{A}(K, w_0))[p] \neq \emptyset$, and $\mathrm{Traces}(\mathbb{A}(K, w_0))[p] \not\models \mathcal{H}[p]$. Wlog, we can assume that only the last slice does not satisfy the corresponding node in $\mathcal{H}$. Let $\mathcal{H}[p] = q_0 a_0 \ldots q_n a_n$. Then, $\mathrm{Traces}(\mathbb{A}(K, w_0))[p] \not\models \mathcal{H}[p]$ iff $(\star)$ for all $0 \leq j < n$, $\mathrm{Traces}(\mathbb{A}(K, w_0))(a_0 \ldots a_{j-1}, a_j) \models q_j$ while $\mathrm{Traces}(\mathbb{A}(K, w_0))(a_0 \ldots a_{n-1}, a_n) \not\models q_n$. By Lemma 25, $\mathrm{Traces}(\mathbb{A}(K, w_0))[p] \neq \emptyset$ defines a unique path in $\mathrm{Slice}(\mathbb{A}(K, w_0))$, $\mathbb{K}_0 a_0 \ldots \mathbb{K}_n a_n$ that preserves the path slicing defined by $\mathbb{A}(K, w_0)$. Thus, from $(\star)$, definition of $\mathrm{Join}(\mathcal{H}, \mathbb{A}(K, w_0))$ and Lemma 25, $(\mathbb{K}_0, q_0) a_0 \ldots (\mathbb{K}_n, \overline{q_n}) a_n$ defines an accepting run in $\mathrm{Join}(\mathcal{H}, \mathbb{A}(K, w_0))$. Note that $(\mathbb{K}_n, \overline{q_n})$ is in $\mathrm{Join}(\mathcal{H}, \mathbb{A}(K, w_0))$ (and it is final) because $\mathrm{Traces}(\mathbb{A}(K, w_0))(a_0 \ldots a_{n-1}, a_n) \not\models q_n$. □

Theorem 27 *Let (K, w_0) be a pointed Kripke structure over the set of variables X and domain Σ, let $\mathbb{A}$ be an action labeling for K over a set A of actions disjoint from the domain, and let $\mathcal{H}$ be a hypernode automaton over the same set X of variables, domain Σ and set A of actions. Let $\mathrm{Join}(\mathcal{H}, K, \mathbb{A}, w_0) = (Q, \hat{Q}, F, A, \delta)$ be the intersection of the Kripke structure (K, w_0) and the hypernode automaton $\mathcal{H}$ w.r.t the action labeling $\mathbb{A}$. The model checking problem defined over (K, w_0), $\mathbb{A}$ and $\mathcal{H}$ is decidable if, for all states $(\mathbb{K}, q) \in Q$ or $(\mathbb{K}, \overline{q}) \in Q$ in $\mathrm{Join}(\mathcal{H}, K, \mathbb{A}, w_0)$ (for some open Kripke structure $\mathbb{K}$ and state q of $\mathcal{H}$) that are reachable from the initial states in $\hat{Q}$, it is decidable to decide whether $\mathbb{K}$ is a model of $\gamma(q)$, where γ is the state labeling function in $\mathcal{H}$.*

We observe that by combining the theorem above with Theorem 12, which establishes decidability for regular synchronous hypernode formulas, and Theorem 21, which does the same for stutter-reducible Kripke structures and regular asynchronous hypernode formulas, we obtain decidability for hypernode automata and Kripke structures whose intersection involves hypernode formulas and open Kripke structures lying within these decidable fragments.

6.3 Models of asynchronous systems

As discussed previously, for model-checking asynchronous hypernode logic, we need to translate an open Kripke structure to an NSFA, which cannot be done in general. Fortunately, when model-checking hypernode automata, formulas from nodes are model-checked against the slices of the Kripke structure, not against the Kripke structure itself, and oftentimes the slices define stutter-reducible Kripke structures. Moreover, for hypernode automata that mix synchronous and asynchronous nodes, we need only the slices that are model-checked against asynchronous hypernode logic to be stutter-reducible. Therefore, the Kripke structure that we model-check with a hypernode automaton can be arbitrary as long as the automaton breaks it down into slices that can be model-checked by particular hypernodes.

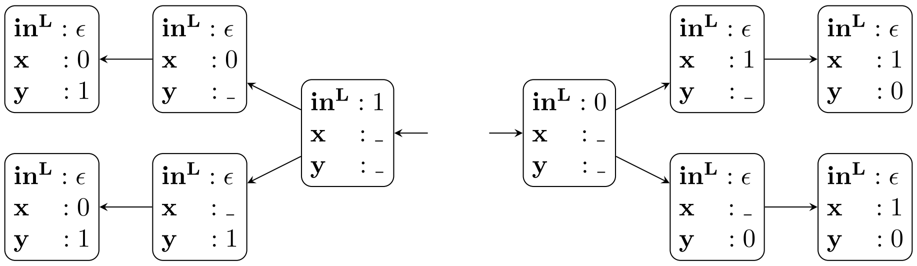
For example, consider the program in Fig. 2. We know that the system supports two communication modes (synchronous and asynchronous), and we can precisely determine when a mode change occurs, since each transition is triggered by a specific action. Given that the hypernode automaton partitions the Kripke structure derived from the program according to the mode-switch actions, the slice associated with asynchronous communication is stutter-reducible. This is because, in asynchronous mode, updates to program variables are independent of one another. This slice is shown in Fig. 13.

Example: A Slice of a Kripke Structure

Figure 13 shows the Kripke structure for the asynchronous part of the program in Fig. 2. The example assumes that the variables are boolean, and the server function S is implemented as $S(x,0)=1, S(x,1)=1, S(y,0)=0, S(y,1)=1$.

In the rest of this section, we discuss several models from the literature defining Kripke structures that are suitable for model checking against hypernode automata with asynchronous hypernodes.

Multiparty session types [22] describe the behavior of parties (components, agents) with respect to communication. For example, the following *global type* (a type describing the communication from the point of an external observer) is a variation on the classical two-buyer protocol [22]:



$\mathbf{in^L} : \epsilon$ $\mathbf{x} : 0$ $\mathbf{y} : 1$
$\mathbf{in^L} : \epsilon$ $\mathbf{x} : 0$ $\mathbf{y} : \_$
$\mathbf{in^L} : 1$ $\mathbf{x} : \_$ $\mathbf{y} : \_$
$\mathbf{in^L} : \epsilon$ $\mathbf{x} : 0$ $\mathbf{y} : 1$
$\mathbf{in^L} : \epsilon$ $\mathbf{x} : \_$ $\mathbf{y} : 1$
$\mathbf{in^L} : 0$ $\mathbf{x} : \_$ $\mathbf{y} : \_$
$\mathbf{in^L} : \epsilon$ $\mathbf{x} : 1$ $\mathbf{y} : \_$
$\mathbf{in^L} : \epsilon$ $\mathbf{x} : 1$ $\mathbf{y} : 0$
$\mathbf{in^L} : \epsilon$ $\mathbf{x} : \_$ $\mathbf{y} : 0$
$\mathbf{in^L} : \epsilon$ $\mathbf{x} : 1$ $\mathbf{y} : 0$

Fig. 13 Open acyclic Kripke structure for the asynchronous part of the program in Fig. 2

$$\mu t. + \begin{cases} \mathrm{b} \to \mathrm{s}: \mathit{query}.\, \mathrm{s} \to \mathrm{b}: \mathit{price}. + \begin{cases} \mathrm{b} \to \mathrm{s}: \mathit{buy}.\, t \\ \mathrm{b} \to \mathrm{s}: \mathit{decline}.\, t \\ \mathrm{b} \to \mathrm{a}: \mathit{assess}.\, \mathrm{a} \to \mathrm{b}: \mathit{price}. + \begin{cases} \mathrm{b} \to \mathrm{s}: \mathit{decline}.\, t \\ \mathrm{b} \to \mathrm{s}: \mathit{buy}.\, t \end{cases} \end{cases} \\ \mathrm{b} \to \mathrm{s}: \mathit{done}. \end{cases}$$

The protocol describes a communication between a seller (s), buyer (b), and a person who can assess (a) the value of goods. The buyer sends a query about an item and the seller responds with price. Then the buyer can choose to directly buy or $\mathit{decline}$ the item, or asks a to assess the price and buy or decline after that. The whole protocol then repeats via the recursion variable t and the least fixpoint operator μ until the buyer notifies the seller that the interaction is done.

Each multiparty session type can be represented by a *communicating automaton* [10] with a special structure – it has a tree-like shape with possible transitions from leaves to the root state (corresponding to recursion) [30]. Therefore, multiparty session types perfectly fit HNAs where actions coincide with the recursive transitions: slices are one round of the protocol, which yields an acyclic Kripke structure that we can stutter-reduce. Note that HNAs do not need to be single-state in this case. We may be interested in specifying different properties in different rounds of the protocol.

Communicating sequential processes (CSP) [20] is a process algebra where processes communicate through a synchronous handshake on a specified set of events. That is, a process can proceed with a synchronized event if all other processes (that synchronize over this event too) also proceed with this event. Processes proceed independently of all other events (i.e., they interleave).

The scenario when all processes synchronize on the same set of events is a good fit for HNAs: all processes run independently until they all synchronize (if they do not run independently forever) and they repeat this pattern until they terminate (if they terminate). It is straightforward to see that if we take the synchronization events as actions for an HNA, then the nodes of this HNA will analyze sub-structures where the processes progress independently. In the previous section, we saw that we can translate such (possibly cyclic) sub-structures into stutter-free automata and therefore model-check with asynchronous hypernodes.

Note that the situation is analogous for *asynchronous finite automata* [35] (also called *network of automata* [16] with asynchronous product, which is sometimes called the *mixed product* [26]) provided that all automata synchronize on the same set of events. Similarly, one can also compose Mealy machines using asynchronous product [29].

We close this section with the observation that situations where multiple systems run independently between synchronization points arise frequently, and, for these cases, it may be possible to directly derive stutter-free automata for asynchronous nodes in HNAs. In addition to the two examples explored above, parallelization libraries may execute a piece of code for different inputs by independent threads, and synchronization takes place only after all threads finish (this is, e.g., the default mode of parallelizing loops with OpenMP [27]). Other examples can be found in sessions in a web browser (where there is no synchronization), robots performing a distributed task, synchronizing once sub-tasks were done, blockchain miners that mine independently, synchronizing on the publication of a new block, voting protocols, divide-and-conquer algorithms, and so on.

7 Related work

The first general approaches to reason about hyperproperties [15] (like HyperLTL) adopted semantic interpretations with *synchronous analysis of trace sets*, evaluating temporal modalities in lock-step over the traces currently assigned to the trace variables. As proved in [3] HyperLTL cannot express asynchronous transition of specification states, which is an example of an asynchronous hyperproperty. This intrinsic limitation to synchronous traces' traversal hinders the applicability of such approaches to reason about security properties in real-world systems. Recently, many formalisms have been presented to address this limitation, which we will introduce next. The general problem of model-checking asynchronous hyperproperties turned out to be highly undecidable [19].

The first logic studied to express asynchronous hyperproperties was hyper μ-calculus [19], called Hμ. It extends the linear-time μ-calculus [32] by adding explicit quantification over traces and annotating propositional variables with trace variables (in a similar fashion as done for LTL by HyperLTL). Additionally, the next operator is parameterized by a trace variable, specifying which trace variable progresses one step at that point of the formula evaluation, thus supporting an asynchronous analysis of traces. In the same paper, the authors introduce the parity multi-tape Alternating Asynchronous Word Automata (AAWA), which they prove to be expressively equivalent to trace-quantifier free formulas of Hμ.

Both Hμ and AAWA turned out to have highly undecidable model-checking problems [19]. The undecidability of model-checking asynchronous hyperproperties is often caused by the interaction of having to compare time positions that are arbitrarily far apart with the possibility for an unbounded number of traces. In [19], the authors introduce two semantic fragments of Hμ, each limiting one of the two sources of unboundedness. The *k-synchronous* fragment imposes a distance up to k between positions of any traces being compared. In contrast, the *k-context-bound* requires the traces to be partitioned in at most k contexts (traces in the same context progress synchronously). In comparison, with hypernode automata, we achieve decidability by entirely different means: we decouple the asynchronous progress of program variables, while allowing resynchronization through automaton-level transitions. The synchronization feature reduces the problem of model-checking asynchronous hyperproperties (which may define sets of infinite traces) to the problem of model-checking sequences of sets of finite traces.

Example: Side note: Relative Expressiveness

As for HyperLTL formulas, the trace quantifiers always precede time operators in Hμ formulas. Hypernode automata, however, allow a restricted form of quantifier alternation between time operators and trace quantifiers by mixing automata and logic in the same formalism. Earlier in this document, and in [3], we showed that a change in the order between trace and time quantifiers proved to be problematic for HyperLTL, turning a hyperproperty that can be expressed in HyperLTL to not be expressible with a HyperLTL formula anymore.

Inspired by the insights of that result, we conjecture that Hμ and hypernode automata have incomparable expressive power and support our claim with the hypernode automaton in Fig. 14. The hypernode automaton specifies that the asynchronous progress of a propositional variable p is fully described by a finite trace π within each slice induced by a repeated action a.

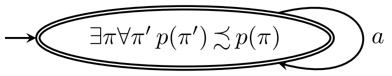


Fig. 14 Hypernode automaton specifying that within each slice of a trace set induced by observing action a, there exists a trace in each slice that describes the progress of the propositional variable p in the slice

We observe that each new slice induced by observing the action a may have a different trace τ assigned to the trace variable π, witnessing the asynchronous progress of p, with the length of the traces in each slice being finite but unbounded. Additionally, as there is no bound on how many times we will observe the action a, the number of slices is also unbounded. This means that we do not have a bound on the number of outermost existential trace quantifiers that would be necessary for the Hμ formula to guarantee we can have a different trace witness for each slice in any set of traces.

An alternative approach to specifying asynchronous hyperproperties is to enrich HyperLTL with asynchronous reasoning. *Stuttering* HyperLTL ($\mathrm{HyperLTL}_S$) and *context* HyperLTL ($\mathrm{HyperLTL}_C$), both introduced in [6], extend HyperLTL with new asynchronous operators; while, Asynchronous HyperLTL (A-HyperLTL) [1] adds quantifiers over trajectories mapping at each step which traces progress (the others will stutter). Stuttering HyperLTL introduces annotation of temporal operators with LTL formulas, describing indistinguishable time sequences (i.e., a sequence is indistinguishable as long as the LTL formula valuation does not change). Then, the next time considered while evaluating a stuttering HyperLTL formula over a set of traces is when the valuation of the annotated LTL formula changes its value, introducing asynchronous traversal of traces. Context HyperLTL follows a different route: it includes a unary modality parameterized by a set of trace variables, called *context*. Traces within a context progress together, while traces outside a context stutter. Asynchronous HyperLTL introduces a new type of quantification (instead of a new operator like in the previous two formalisms), which, together with trace quantifiers, must occur before any temporal formula. In particular, A-HLTL adds quantification over trajectories, which are sequences of sets of trace variables, with each step defining the set of trace variables progressing in that step with all the other variables stuttering. Following similar strategies as done for Hμ, the authors presented fragments with decidable model-checking for all the HyperLTL extensions mentioned above.

Bozelli et al. studied in [7] the relative expressiveness of all the mentioned formalisms and proved that all of them are subsumed by Hμ [7]. Due to trace-related quantifiers always preceding time quantification in all formalisms above, we conjecture that there are hyperproperties that hypernode automata can specify while Hμ can not. Hence, hypernode automata are not subsumed by Hμ and, due to the results in [7], the same holds for all the asynchronous HyperLTL extensions. We elaborate on this point in the *Side Note: Relative Expressiveness* above.

More recently, Beutner et al. introduced an extension of HyperLTL with second-order quantification, called Hyper^2LTL [4]. First-order quantifiers in Hyper^2LTL assign traces to trace variables (as in HyperLTL), while second-order quantifiers range over sets of traces. The full Hyper^2LTL (i.e., second-order quantifiers ranging over all possible trace sets) has a highly-undecidable model-checking problem [4]. To remedy this problem, they introduce Hyper^2 $\mathrm{LTL}_{\mathrm{fp}}$ where sets are constrained to satisfy given minimality or maximality requirements. From this fragment, they presented an algorithm to approximate solutions to the model-checking problem for the case where the only constraints allowed are minimality and guards induce least-fixed points. Hypernode automata reasoning is at the first-order level. This means that, while model-checking hypernode formulas, we only care about the set of traces generated by the model, and we do not need to reason about sets of possible trace sets to check for asynchronous properties. Hence, the Hyper^2LTL approach is, in this sense, more general than hypernode automata. However, for the same reason as in previous HyperLTL extensions, we conjecture that there are hypernode specifications that Hyper^2 LTL cannot express.

In the team semantics reinterpretations of LTL presented by Krebs et al. [25], the authors introduced an asynchronous variant. However, they prove that the asynchronous team semantics of LTL is subsumed by universal HyperLTL. (where all trace quantifiers are universal quantifiers). Hence, this is not a suitable formalism to express asynchronous hyperproperties.

8 Conclusion

In this paper, we introduced hypernode automata to specify hyperproperties on systems that can exhibit both synchronous and asynchronous executions. At the top level of our formalism, we have an automaton structure synchronizing transitions between different specification nodes. By using actions to synchronize traces, we enable the analysis of trace segments of varying lengths (as actions may occur at any point in each trace). At the node level, we support both synchronous or asynchronous formulas of hypernode logic, effectively allowing to specify a rich set of specifications. We observe that the logic used within hypernodes need not be limited to hypernode logic. Our model-checking algorithm for hypernode automata seamlessly supports the use of other specification logics within hypernodes, such as HyperLTL.

Numerous additional topics arise directly from this work, which, though straightforward, require further investigation. These include the study of hypernode automata with partial and nondeterministic transition relations and those with infinitary acceptance conditions (e.g., hypernode Büchi automata). Additionally, it will be interesting to compare the expressiveness between different formalisms to specify asynchronous hyperproperties and hypernode automata. Finally, it is also interesting to explore further which other models may be adequate to reason about asynchronous hyperproperties.

Acknowledgements This work was supported in part by the Austrian Science Fund (FWF) SFB project SpyCoDe 10.55776/F85, by the FWF projects ZK-35 and W1255-N23, and by the ERC Advanced Grant VAMOS 101020093.

Author contributions Ezio Bartocci (EB), Thomas A. Henzinger (TAH), Dejan Nickovic (DN) and Ana Oliveira da Costa (AOC) contributed to the initial conceptualization of Hypernode automata and writing of the preliminary version of this work published at CONCUR 2023. AOC led the development of the theoretical framework and writing of the main text related to that preliminary version. At the same time, EB, TAH, and DN participated in the writing and revision of it. Marek Chalupa (MC), AOC and TAH led the theoretical development related to the extensions of hypernode logic. MC and AOC contributed equally to the new results presented in this manuscript.

Funding Open access funding provided by Institute of Science and Technology (IST Austria).

Data availability No datasets were generated or analysed during the current study.

Declarations

Competing interests This work was supported in part by the Austrian Science Fund (FWF) SFB project SpyCoDe 10.55776/F85, by the FWF projects ZK-35 and W1255-N23, and by the ERC Advanced Grant VAMOS 101020093.

References

1. Baumeister, J., Coenen, N., Bonakdarpour, B., Finkbeiner, B., Sánchez, C.: A temporal logic for asynchronous hyperproperties. In: Silva, A., Leino, K.R.M. (eds.) Computer Aided Verification (CAV), pp. 694–717 (2021). https://doi.org/10.1007/978-3-030-81685-8_33
2. Bartocci, E., Henzinger, T.A., Nickovic, D., Costa, A.O.: Hypernode automata. In: Pérez, G.A., Raskin, J. (eds.) 34th International Conference on Concurrency Theory (CONCUR), pp. 21–12116 (2023). https://doi.org/10.4230/LIPICS.CONCUR.2023.21
3. Bartocci, E., Ferrère, T., Henzinger, T.A., Nickovic, D., Oliveira da Costa, A.: Flavors of sequential information flow. In: Proc. of VMCAI 2022: the 23rd International Conference on Verification, Model Checking, and Abstract Interpretation. LNCS, vol. 13182, pp. 1–19 (2022). https://doi.org/10.1007/978-3-030-94583-1_1
4. Beutner, R., Finkbeiner, B., Frenkel, H., Metzger, N.: Second-order hyperproperties. In: Enea, C., Lal, A. (eds.) Computer Aided Verification (CAV), pp. 309–332 (2023). https://doi.org/10.1007/978-3-031-37703-7_15
5. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic model checking without BDDs. In: Cleaveland, R. (ed.) Tools and Algorithms for Construction and Analysis of Systems, 5th International Conference, TACAS '99, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99, Amsterdam, The Netherlands, March 22-28, 1999,Proceedings. Lecture Notes in Computer Science, pp. 193–207 (1999). https://doi.org/10.1007/3-540-49059-0_14
6. Bozzelli, L., Peron, A., Sánchez, C.: Asynchronous extensions of HyperLTL. In: 2021 36th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), pp. 1–13 (2021). https://doi.org/10.1109/LICS52264.2021.9470583
7. Bozzelli, L., Peron, A., Sánchez, C.: Expressiveness and Decidability of Temporal Logics for Asynchronous Hyperproperties. In: Klin, B., Lasota, S., Muscholl, A. (eds.) 33rd International Conference on Concurrency Theory (CONCUR 2022). Leibniz International Proceedings in Informatics (LIPIcs), vol. 243, pp. 27–12716 (2022). https://doi.org/10.4230/LIPIcs.CONCUR.2022.27
8. Bonakdarpour, B., Sheinvald, S.: Finite-word hyperlanguages. In: Proc. of LATA 2021: the 15th International Conference on Language and Automata Theory and Applications. Lecture Notes in Computer Science, vol. 12638, pp. 173–186 (2021). https://doi.org/10.1007/978-3-030-68195-1
9. Berry, G., Sethi, R.: From regular expressions to deterministic automata. Theoret. Comput. Sci. **48**, 117–126 (1986)
10. Brand, D., Zafiropulo, P.: On communicating finite-state machines. J. ACM **30**(2), 323–342 (1983). https://doi.org/10.1145/322374.322380
11. Brzozowski, J.A.: Derivatives of regular expressions. Journal of the ACM **11**(4), 481–494 (1964)
12. Clarkson, M.R., Schneider, F.B.: Hyperproperties. J. Comput. Secur. **18**(6), 1157–1210 (2010). https://doi.org/10.3233/JCS-2009-0393

13. Chalupa, M., Henzinger, T.A., Costa, A.: Monitoring extended hypernode logic. In: Kosmatov, N., Kovács, L. (eds.) Integrated Formal Methods, pp. 151–171 (2024). https://doi.org/10.1007/978-3-031-76554-4_9
14. Clarkson, M.R., Finkbeiner, B., Koleini, M., Micinski, K.K., Rabe, M.N., nchez, C.S.: Temporal logics for hyperproperties. In: Proc. of POST 2014: the Third International Conference on Principles of Security and Trust. Lecture Notes in Computer Science, vol. 8414, pp. 265–284 (2014). https://doi.org/10.1007/978-3-642-54792-8
15. Coenen, N., Finkbeiner, B., Hahn, C., Hofmann, J.: The hierarchy of hyperlogics. In: Proc. of LICS: the 34th Annual ACM/IEEE Symposium on Logic in Computer Science, pp. 1–13 (2019). https://doi.org/10.1109/LICS.2019.8785713
16. Esparza, J., Blondin, M.: Automata Theory: An Algorithmic Approach. MIT Press, Cambridge, MA (2023)
17. Flanagan, C., Godefroid, P.: Dynamic partial-order reduction for model checking software. In: Palsberg, J., Abadi, M. (eds.) Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005, pp. 110–121 (2005). https://doi.org/10.1145/1040305.1040315
18. Furia, C.A.: A survey of multi-tape automata. CoRR **abs/1205.0178** (2012) arXiv:1205.0178
19. Gutsfeld, J.O., Müller-Olm, M., Ohrem, C.: Automata and fixpoints for asynchronous hyperproperties. Proc. ACM Program. Lang. **5**(POPL), 1–29 (2021). https://doi.org/10.1145/3434319
20. Hoare, C.A.R.: Communicating sequential processes. Commun. ACM **21**(8), 666–677 (1978). https://doi.org/10.1145/359576.359585
21. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. In: Necula, G.C., Wadler, P. (eds.) Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008, pp. 273–284 (2008). https://doi.org/10.1145/1328438.1328472
22. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. In: Necula, G.C., Wadler, P. (eds.) Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008, pp. 273–284 (2008). https://doi.org/10.1145/1328438.1328472
23. Ibarra, O.H., Trân, N.Q.: On synchronized multi-tape and multi-head automata. Theor. Comput. Sci. **449**, 74–84 (2012)
24. Ibarra, O.H., Trân, N.Q.: On synchronized multitape and multihead automata. In: Holzer, M., Kutrib, M., Pighizzini, G. (eds.) Descriptional Complexity of Formal Systems - 13th International Workshop, DCFS 2011, Gießen/Limburg, Germany, July 25-27, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6808, pp. 184–197 (2011). https://doi.org/10.1007/978-3-642-22600-7_15
25. Krebs, A., Meier, A., Virtema, J., Zimmermann, M.: Team Semantics for the Specification and Verification of Hyperproperties. In: Potapov, I., Spirakis, P., Worrell, J. (eds.) 43rd International Symposium on Mathematical Foundations of Computer Science (MFCS 2018). Leibniz International Proceedings in Informatics (LIPIcs), vol. 117, pp. 10–11016 (2018). https://doi.org/10.4230/LIPIcs.MFCS.2018.10
26. Morin, R.: Decompositions of asynchronous systems. In: Sangiorgi, D., Simone, R. (eds.) CONCUR ’98: Concurrency Theory, 9th International Conference, Nice, France, September 8-11, 1998, Proceedings. Lecture Notes in Computer Science, vol. 1466, pp. 549–564 (1998). https://doi.org/10.1007/BFB0055647
27. OpenMP. accessed 2025-06-23. https://www.openmp.org/
28. Post, E.L.: A variant of a recursively unsolvable problem. Bull. Am. Math. Soc. **52**(4), 264–268 (1946)
29. Samadi, M., Bastany, A., Hojjat, H.: Compositional learning for synchronous parallel automata. In: Boronat, A., Fraser, G. (eds.) Fundamental Approaches to Software Engineering, pp. 101–121 (2025)
30. Stutz, F.: Asynchronous multiparty session type implementability is decidable - lessons learned from message sequence charts. In: Ali, K., Salvaneschi, G. (eds.) 37th European Conference on Object-Oriented Programming, ECOOP 2023, July 17-21, 2023, Seattle, Washington, United States. LIPIcs, vol. 263, pp. 32–13231 (2023). https://doi.org/10.4230/LIPICS.ECOOP.2023.32
31. Thompson, K.: Regular expression search algorithm. Commun. ACM **11**(6), 419–422 (1968). https://doi.org/10.1145/363347.363387
32. Vardi, M.Y.: A temporal fixpoint calculus. In: Symposium on Principles of Programming Languages (POPL), pp. 250–259 (1988). https://doi.org/10.1145/73560.73582
33. Watson, B.W.: A taxonomy of finite automata construction algorithms (1993)
34. Yu, F., Bultan, T., Ibarra, O.H.: Relational string verification using multi-track automata. Int. J. Found. Comput. Sci. **22**(8), 1909–1924 (2011)

35. Zielonka, W.: Notes on finite asynchronous automata. RAIRO Theor. Informatics Appl. **21**(2), 99–135 (1987)
36. Zdancewic, S., Myers, A.C.: Observational determinism for concurrent program security. In: 16th IEEE Computer Security Foundations Workshop, 2003. Proceedings., pp. 29–43 (2003).https://doi.org/10.1109/CSFW.2003.1212703