Hypernode Automata

Ezio Bartocci
Technische Universität Wien, Austria

Thomas A. Henzinger
IST Austria, Klosterneuburg, Austria

Dejan Nickovic
AIT Austrian Institute of Technology, Wien, Austria

Ana Oliveira da Costa
IST Austria, Klosterneuburg, Austria

Abstract

We introduce *hypernode automata* as a new specification formalism for hyperproperties of concurrent systems. They are finite automata with nodes labeled with *hypernode logic* formulas and transitions labeled with actions. A hypernode logic formula specifies relations between sequences of variable values in different system executions. Unlike HyperLTL, hypernode logic takes an *asynchronous* view on execution traces by constraining the values and the order of value changes of each variable without correlating the timing of the changes. Different execution traces are synchronized solely through the transitions of hypernode automata. Hypernode automata naturally combine asynchronicity at the node level with synchronicity at the transition level. We show that the model-checking problem for hypernode automata is decidable over action-labeled Kripke structures, whose actions induce transitions of the specification automata. For this reason, hypernode automaton is a suitable formalism for specifying and verifying asynchronous hyperproperties, such as declassifying observational determinism in multi-threaded programs.

2012 ACM Subject Classification Security and privacy → Logic and verification

Keywords and phrases Hyperproperties, Asynchronous, Automata, Logic

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2023.21

Related Version *Full Version*: `https://arxiv.org/abs/2305.02836`

Funding This work was supported in part by the Austrian Science Fund (FWF) SFB project SpyCoDe F8502, by the FWF projects ZK-35 and W1255-N23, and by the ERC Advanced Grant VAMOS 101020093.

1 Introduction

Formalisms like Linear temporal logic (LTL) or automata are commonly used to specify and verify trace properties of concurrent systems. Security requirements such as information-flow policies require simultaneous reasoning about multiple execution traces; hence, they cannot be expressed as trace properties. Hyperproperties address this limitation by specifying properties of trace sets [11]. HyperLTL [10], an extension of LTL with trace quantifiers, has emerged as a popular formalism for both the specification and verification of an important class of hyperproperties. The temporal operators of HyperLTL and related hyperlogics progress in lockstep over all traces that are bound to a trace variable; they specify *synchronous* hyperproperties. As a consequence, HyperLTL cannot specify, for instance, an information-flow policy that changes from one system mode to another if the mode transition can occur at different times in different system executions [3]. This limitation has been observed repeatedly and independently in recent years by [12, 8, 5] all of whom have proposed asynchronous versions of hyperlogics to address the problem.

We take a different route and propose a specification language for hyperproperties, called *hypernode automata*, which combines synchronicity and asynchronicity by combining automata and logic. *Hypernode automata* are finite automata with nodes labeled with formulas from a fully asynchronous, non-temporal hyperlogic, called *hypernode logic*, and transitions labeled with actions used to synchronize different execution traces. While automata-based languages have been used before for specifying synchronous hyperproperties [7], hypernode automata are the first language that systematically separates trace synchronicity from trace asynchronicity in the specification of hyperproperties: within hypernodes (i.e., states of the hypernode automata), different execution traces proceed at independent speeds, only to "wait for each other" when they transition to the next hypernode. This separation leads to natural specifications and plays to the strengths of both automata-based and logic-based formalisms.

Hypernodes' specification adopts a maximally asynchronous view over finite trace segments: each program variable can progress independently. We introduce *hypernode logic* to specify such asynchronous hyperproperties. Hypernode logic includes quantification over finite traces and the binary relation $x(\pi) \precsim y(\pi')$, for trace variables π and π', and system (or program) variables x and y. This 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 happen at different times (stuttering), and there may be additional changes of y in π' (prefixing). The *stutter-reduced prefixing* relation $\precsim$ between finite traces is the only nonlogical operator of hypernode logic, yielding a novel, elegant, and powerful method to specify asynchronous hyperproperties over finite traces.

For each finite or infinite action sequence, a *hypernode automaton* specifies a corresponding sequence of formulas from hypernode logic. This paper's main contribution is a model-checking algorithm for hypernode automata. Our algorithm checks if a given hypernode automaton accepts the set of (possibly) infinite traces defined by an action-labeled Kripke structure. The action labels on transitions of the Kripke structure (the "model") induce equally labeled transitions of the hypernode automaton (the "specification"). The subroutine that model-checks formulas of hypernode logic is technically novel: it introduces automata-theoretic constructions on a new concept called *stutter-free automata*, which are then used in familiar logical contexts such as filtration and self-composition. While hypernode logic has existential trace quantifiers and thus can specify nonsafety hyperproperties such as the independence of variables [3], we focus in this paper on safe hypernode automata to specify hyperproperties of infinite executions. To our knowledge, this is the first decidability result for a simple but general formalism that can specify important asynchronous hyperproperties.

Perhaps the most famous asynchronous hyperproperty is *observational determinism* [17]. In Section 2, we motivate hypernode logic by providing a formal specification of observational determinism as defined by [17]. We further motivate hypernode automata by specifying *declassification* of information, which can be represented by a transition between hypernodes. Section 3 defines hypernode logic and hypernode automata. In Section 4, we first solve the model-checking problem for hypernode logic over Kripke structures using stutter-reduced automata, and then the more general model-checking problem for hypernode automata over action-labeled Kripke structures. In contrast to previously proposed specification formalisms for asynchronous hyperproperties [12, 8, 5], which are undecidable in general and decidable only for specific fragments, our model-checking algorithms are doubly exponential in the number of variables in the Kripke structure. Finally, in Section 5, we argue that our formalism is expressively incomparable to these other formalisms. For this reason, hypernode automaton is a promising specification formalism for asynchronous hyperproperties and thus significantly contributes to the automatic verification of security properties.

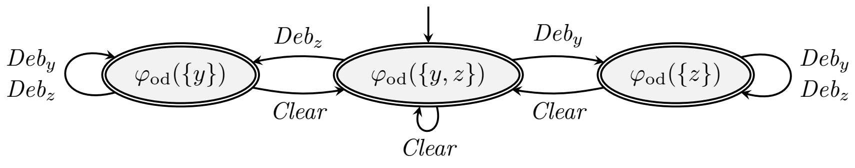
$\varphi_{\text{od}}(\{y\})$
$\varphi_{\text{od}}(\{y,z\})$
$\varphi_{\text{od}}(\{z\})$
Deb_y
Deb_z
$Clear$

Figure 1 Hypernode automaton $\mathcal{H}$ specifying the mutually exclusive declassification of secure information, where $\varphi_{\text{od}}(L) \overset{\text{def}}{=} \forall\pi\forall\pi' \bigwedge_{l\in L}(l(\pi) \precsim l(\pi') \vee l(\pi') \precsim l(\pi))$.

2 Motivating Example

The seminal work by Zdancewic and Myers [17] proposes the notion of observational determinism to specify information-flow policies of concurrent programs. Observational determinism is a noninterference property which requires publicly visible values to not depend on secret information. Noninterference specification is particularly challenging for multi-threaded programs because (i) the executions of a multi-threaded program depend on the scheduling policy, and (ii) a change from one program state to another can happen at different times in each execution of the program. According to [17], a program is observationally deterministic if, when starting from any two low-equivalent states, then any two traces of each low variable are equivalent up to *stuttering* and *prefixing*. This definition takes an asynchronous view of execution traces, so HyperLTL is not adequate to specify it [3].

In this section, we use hypernode logic to specify *Zdancewic-Myers observational determinism*, and use it to define a *mutually exclusive declassification policy* with a hypernode automaton. The declassification policy involves not only the asynchronous requirement of observational determinism, but also dynamic changes between different observational determinism requirements. In particular, the policy requires that during *normal* operation, two publicly visible variables y and z must not leak secret information. The policy also admits two *debugging* modes, in which either y or z can leak information, but never both. The "mode operation" is inspired by examples on declassification policies by the programming languages community (c.f. [2, 15]).

The hypernode automaton specification $\mathcal{H}$ of the declassification policy is shown in Figure 1. A hypernode automaton is interpreted over a set of action-labeled traces, which are sequences of valuations for program variables and actions. The transitions between automaton nodes are labeled with actions marking when the program changes its mode of operation, say, from normal mode to one of the two debugging modes. In the example, transitions from the normal mode to the two debugging modes are labeled with Deb_y and Deb_z actions, respectively; transitions from either debugging mode to the normal mode are labeled with *Clear*.

The automaton nodes are labelled with formulas of hypernode logic. In our example, all three formulas specify Zdancewic-Myers observational determinism but for different program variables. Observational determinism requires that for any two program executions (specified by $\forall\pi\forall\pi'$), their projections to each publicly visible program variable (l in a set L of public variables) are equivalent up to stuttering and prefixing (specified by $l(\pi) \precsim l(\pi') \vee l(\pi') \precsim l(\pi)$). The visible program variables change from mode to mode, so the different specification nodes are labeled with different instances of the same formula. For example, the hypernode with formula $\varphi_{\text{od}}(\{y\})$ requires observational determinism only for y.

Algorithm 1 defines a reactive program $\mathcal{P}_{\text{var}}$ (where var can be either y or z) which in every iteration reads the input variable x and the action status. If, for example, the action is Deb_y, then variable y is used for debugging and the program copies the content of x to y.

Algorithm 1 Program $\mathcal{P}_{\text{var}}$.

```
1 do
2     var := 0;
3     read($x$);
4     read(status);
5     if (status = $Deb_{\text{var}}$) then
6         var := $x$;
7     end
8     output(var);
9 while true;
```

Table 1 Executions of $\mathcal{P}_y \mid\mid \mathcal{P}_z$.

τ_1	x:	0	0	0	
	y:	0	0	0	
	z:	0	0	0	
	status:	ε	Deb_y	Deb_z	
τ_2	x:	1	1	1	1
	y:	0	0	1	1
	z:	0	0	0	1
	status:	ε	ε	Deb_y	Deb_z

The parallel composition $\mathcal{P}_y \mid\mid \mathcal{P}_z$ does not satisfy the specification $\mathcal{H}$. Consider, for instance, the set $T = \{\tau_1, \tau_2\}$ of traces shown in Table 1. We make two important observations on τ_1 and τ_2: (1) these traces have different lengths, and (2) they exhibit the *same* sequence of actions (Deb_y followed by Deb_z) happening at *different* times (hence the traces are asynchronous). We note that the above sequence of actions partitions each trace into a sequence of three trace segments, called *slices*, which we denote using the white, the light gray, and the dark gray color in Table 1. We map each slice to a unique node in the hypernode automaton. The white slice of T is mapped to the initial node of $\mathcal{H}$. The light-gray slice of T is mapped to the node accessible from the initial state with action Deb_y (i.e., the debugging mode for y), which is labeled by the formula $\varphi_{\text{od}}(\{z\})$. The dark-gray slice of T is mapped to the same node, because the action Deb_z triggers the self-loop transition.

A sequence of actions defines a path in the hypernode automaton. Then, a set T of traces with a sequence of actions p satisfies the hypernode automaton $\mathcal{H}$ iff the slicing of T induced by p satisfies the hypernode formulas in the path defined by p in $\mathcal{H}$. This is not the case in our example because the dark-gray slice of T violates its associated hypernode formula $\varphi_{\text{od}}(\{z\})$. More specifically, the program variable z evaluates to 0 in the dark-gray segment of the trace τ_1, while it evaluates to 1 in the dark-gray segment of τ_2. The specification violation occurs because the critical section (lines $5-7$ in Algorithm 1) is unprotected. It is possible that the action Deb_z happens (line 4 in $\mathcal{P}_z$) after Deb_y. Thereafter the input value x is copied to z (line 6 in $\mathcal{P}_z$), and both y and z are made observable (line 8 in both $\mathcal{P}_y$ and $\mathcal{P}_z$). Hence they both leak information about x, which violates the specification. Our model-checking algorithm allows us to fully automate the reasoning in this example.

3 Hypernode Automata

In this section, we define *hypernode logic* and *hypernode automata*. We represent program executions as finite or infinite sequences of finite trace segments with synchronization actions. Let X be a finite set of *program variables* over a finite domain Σ, A be a finite set of *actions* and $A_\varepsilon = A \cup \{\varepsilon\}$.

Hypernode logic is interpreted over finite trace segments. A *trace segment* τ is a finite sequence of valuations in Σ^X, where each *valuation* $v : X \to \Sigma$ maps program variables to domain values. We denote the set of trace segments over X and Σ by $(\Sigma^X)^*$. A *segment property* T is a set of trace segments, that is, $T \subseteq (\Sigma^X)^*$. A formula of hypernode logic specifies a property of a set of trace segments, which is called a *segment hyperproperty*. Formally, a segment hyperproperty $\mathbf{T}$ is a set of segment properties, that is, $\mathbf{T} \subseteq 2^{(\Sigma^X)^*}$.

Hypernode automata are interpreted over finite and infinite action-labeled traces. An *action-labeled trace* ρ 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; that is, $\rho \in (\Sigma^X \times A_\varepsilon)^*$ or $\rho \in (\Sigma^X \times A_\varepsilon)^\omega$.

We require, for technical simplicity, that for infinite action-labeled traces, infinitely many labels are non-empty. 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 trace hyperproperty*, namely, the set of all action-labeled trace properties it accepts.

3.1 Hypernode Logic

Hypernode logic, FO[$\precsim$], is a first-order formalism to specify relations between the changes of the values of program variables over a set of trace segments. The formulas of hypernode logic are defined by the grammar: $\varphi \ ::= \ \exists\pi\,\varphi \,|\, \neg\varphi \,|\, \varphi\wedge\varphi \,|\, x(\pi)\precsim x(\pi)$, where the first-order variable π ranges over the set $\mathcal{V}$ of trace variables and the unary function symbol x ranges over the set X of program variables. Hypernode logic refers to time only through the binary *stutter-reduced prefixing* predicate $\precsim$. The intended meaning of the atomic formula $x(\pi)\precsim y(\pi')$ is that x undergoes the same ordered value changes in the trace segment assigned to π as the variable y does in π', followed by possibly additional value changes of y in π'. In other words, hypernode logic adopts a fully asynchronous comparison of different trace segments in which all variables are considered separately.

We therefore interpret hypernode formulas over *unzipped* trace segments, which encode the evolution of each program variable independently. An *unzipped trace segment* $\tau : X \to \Sigma^*$ is a function from the program variables to finite strings of values. The formulas of hypernode logic are interpreted over assignments of trace variables to unzipped trace segments. Given a set $T \subseteq (\Sigma^*)^X$ of unzipped trace segments, an assignment $\Pi_T : \mathcal{V} \to T$ maps each trace variable to an unzipped trace segment in T. We denote by $\Pi_T[\pi \mapsto \tau]$ the update of Π_T, where π is assigned to τ. The satisfaction relation for a formula φ of hypernode logic over an assignment Π_T is defined inductively as follows:

$$\Pi_T \models \exists\pi\varphi \text{ iff } \text{ there exists } \tau\in T: \ \Pi_T[\pi\mapsto\tau]\models\varphi;$$
$$\Pi_T \models \psi_1\wedge\psi_2 \text{ iff } \Pi_T\models\psi_1 \text{ and } \Pi_T\models\psi_2; \quad \Pi_T\models\neg\psi_1 \text{ iff } \Pi_T\not\models\psi_1;$$
$$\Pi_T\models x(\pi)\precsim y(\pi') \text{ iff } \Pi_T(\pi)(x)\in\sigma_0^+\ldots\sigma_n^+ \text{ and } \Pi_T(\pi')(y)\in\sigma_0^+\ldots\sigma_n^+\Sigma^*$$
$$\text{with } \sigma_i\neq\sigma_{i+1}, \text{ for } 0\leq i<n.$$

A set T of unzipped trace segments is a *model* of the formula φ, denoted by $T\models\varphi$, iff there exists an assignment Π_T such that $\Pi_T\models\varphi$. We adopt the usual abbreviations $\forall\pi\varphi \overset{\text{def}}{=} \neg\exists\pi\neg\varphi$ and $\varphi\vee\varphi' \overset{\text{def}}{=} \neg(\neg\varphi\wedge\neg\varphi')$. From now on, unless stated otherwise, program and trace variables are indexed by a natural number, i.e., $X=\{x_1,\ldots,x_m\}$ and $\mathcal{V}=\{\pi_1,\ldots,\pi_n\}$.

▶ **Example 1.** We illustrate how to use hypernode logic by specifying four different variants of non-interference between program variables. Zdancewic and Myers introduced in [17] the first notion of observational determinism to capture non-interference for concurrent programs. They require that in every program execution, every publicly visible variable in a set L must be stutter-equivalent up to prefixing (i.e., one of the executions can have more value changes): $\forall\pi\forall\pi' \bigwedge_{l\in L}(l(\pi)\precsim l(\pi')\vee l(\pi')\precsim l(\pi))$. Later, Huisman, Worah, and Sunesen [13] strengthened the previous definition by requiring every publicly visible variable to be stutter-equivalent in all executions: $\forall\pi\forall\pi' \bigwedge_{l\in L} l(\pi)\precsim l(\pi')$. Our third variant of observational determinism is from Terauchi[16], requiring the set of all publicly visible variables to be stutter-equivalent up to prefixing: $\forall\pi\forall\pi' \ (L(\pi)\precsim L(\pi')\vee L(\pi')\precsim L(\pi))$. Note that, we can encode the values of a finite set of variables within a single variable called L because we interpreted hypernode formulas over arbitrary finite domains. Finally, we specify independence (also known as generalized non-interference [11]) as defined in [3]. Two program variables x and y are *independent* iff whenever a sequence of value changes for x is possible in some trace π, and a sequence of value changes for y is possible in some trace π', then also their combination $(x(\pi), y(\pi'))$ is possible in some trace. The formula for independence specifies that for every two traces (π and π') there exists a third trace ($\pi_\exists$) that witnesses the combination $(x(\pi), y(\pi'))$ up to stuttering and prefixing: $\forall\pi\forall\pi'\exists\pi_\exists\ (x(\pi) \precsim x(\pi_\exists) \wedge y(\pi') \precsim y(\pi_\exists))$. $\lhd$

Stutter-reduced trace segments

We are interested in unzipped trace segments that are stutter-free, i.e., that do not repeat the same variable value in consecutive time points. For a program variable x and unzipped trace segment τ with $\tau(x) \in \sigma_0^+ \dots \sigma_n^+$ where $\sigma_i \neq \sigma_{i+1}$ for $i < n$, the *stutter-reduction* is $\lfloor \tau(x) \rfloor = \sigma_0 \dots \sigma_n$. We extend this notion naturally to the stutter-reduction of τ by $\lfloor \tau \rfloor(x) = \lfloor \tau(x) \rfloor$ for all program variables $x \in X$, and to the stutter reduction of a set T of unzipped trace segments by $\lfloor T \rfloor = \{\lfloor \tau \rfloor \mid \tau \in T\}$. We prove that formulas of hypernode logic cannot distinguish between a set of unzipped trace segments T and its stutter-reduction $\lfloor T \rfloor$.

▶ **Proposition 2.** *Let $T \subseteq (\Sigma^*)^X$ be a set of unzipped trace segments and φ a formula of hypernode logic. Then, $T \models \varphi$ iff $\lfloor T \rfloor \models \varphi$.*

3.2 Hypernode Automata

Hypernode automata are finite automata with states (called *hypernodes*) labeled with formulas of hypernode logic and transitions labeled with actions. A hypernode automaton reads a set $R \subseteq (\Sigma^X \times A_\varepsilon)^\omega$ of action-labeled traces, and accepts some of these sets.

▶ **Definition 3.** *A deterministic, finite* hypernode automaton *(HNA) 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 $r = q_0a_0\, q_1a_1\, q_2a_2 \dots$ 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 $p = a_0a_1a_2 \dots$ of actions as the *action sequence* of r. Note that each action sequence defines a unique run of $\mathcal{H}$.

The *action sequence* of an action-labeled trace $\rho = (v_0, a_0)(v_1, a_1)(v_2, a_2) \dots$, 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, $\rho[A] = a'_0a'_1 \dots$ with $a_0a_1 \dots \in a'_0\varepsilon^*a'_1\varepsilon^* \dots$ and $a'_i \in A$ for all $i \geq 0$. Given a set R of action-labeled traces, the *projection of R with respect to a finite action sequence* $p \in A^*$ is $R[p] = \{\rho \in R \mid \rho[A] = p\,p'$ for some suffix $p' \in A^* \cup A^\omega\}$.

Each step in the run of a hypernode automaton defines a new *slice* on a set of action-labeled traces. Let $p = a_0a_1 \dots a_n$ be a finite action sequence, and $\rho = (v_0, a'_0)(v_1, a'_1) \dots$ be an action-labeled trace that has prefix p, and let R be a set of such traces. We write $\rho(\varnothing, a_0)$ for the initial trace segment of ρ which ends with the action label a_0. Formally, $\rho(\varnothing, a_0) = v_0 \dots v_k$ such that $a'_k = a_0$, and $a'_i = \varepsilon$ for all $0 \leq i < k$. Furthermore, we write $\rho(a_0a_1 \dots a_i, a_{i+1})$ for the subsequent trace segments of ρ which end with the action label a_{i+1} after having seen the action sequence $a_0a_1 \dots a_i$. Inductively, if $\rho(a_0a_1 \dots a_{i-1}, a_i) = v_k \dots v_l$, $a'_m = a_{i+1}$ for $m > l$, and $a'_j = \varepsilon$ for all $l < j < m$, then $\rho(a_0a_1 \dots a_i, a_{i+1}) = v_{l+1} \dots v_m$. The slicing is extended to sets of action-labeled traces accordingly; for example, $R(\varnothing, a) = \{\rho(\varnothing, a) \mid \rho \in R\}$.

Since the formulas of hypernode logic are interpreted over unzipped trace segments, in a final step, we need to unzip each slice. The *unzipping of a trace segment* $\tau = v_0 \dots v_n$ over the set of variables $X = \{x_1, \dots, x_m\}$ is $\text{unzip}(\tau) = \{x_0 : v_0(x_0)..v_n(x_0), \dots, x_m : v_0(x_m)..v_n(x_m)\}$. We define the unzipping of trace segments sets naturally as $\text{Unzip}(T) = \{\text{unzip}(\tau) \mid \tau \in T\}$.

▶ **Definition 4.** *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 pattern p, denoted $R \models_p \mathcal{H}$, iff for the run $\mathcal{H}[p] = q_0 a_0\, q_1 a_1 \, \dots \, q_n a_n$, all slices of R induced by p are models of the formulas that label the respective hypernodes; that is, $\text{Unzip}(R[p](\varnothing, a_0)) \models \gamma(q_0)$, and $\text{Unzip}(R[p](a_0 \dots a_{i-1}, a_i)) \models \gamma(q_i)$ 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 accepting states or, for example, infinite runs must visit accepting states infinitely often.

4 Model Checking

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

4.1 Action-labeled Kripke Structures

A *Kripke structure* is a tuple $K = (W, \Sigma^X, \Delta, V)$ consisting of a finite set W of worlds, a set X of variables over a finite domain Σ, a transition relation $\Delta \subseteq W \times W$, and a value assignment $V : W \times X \to \Sigma$ that assigns a value from the finite domain Σ to each variable in each world. Given a Kripke structure 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 *pointed Kripke structure* is a Kripke structure with one of its worlds being an initial world, denoted (K, w_0) with $w_0 \in W$.

A *path* in the Kripke structure K with action labeling $\mathbb{A}$ is a finite or infinite sequence $w_0 a_0\, w_1 a_1\, w_2 a_2 \dots$ 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 $\text{Paths}(K, \mathbb{A})$ for the set of all such paths. The path $\varrho = w_0 a_0\, w_1 a_1 \dots$ defines the action-labeled trace $\text{zip}(\varrho) = V(w_0) a_0\, V(w_1) a_1 \dots$. We write $\text{Zip}(K, \mathbb{A})$ for the set of action-labeled traces defined by paths in $\text{Paths}(K, \mathbb{A})$. By $\text{Paths}(K, \mathbb{A}, w_0)$ we denote the set of all paths in $\text{Paths}(K, \mathbb{A})$ that start at the world w_0. As before, $\text{Zip}(K, \mathbb{A}, w_0)$ refers to the set of all action-labeled traces that are defined by paths in $\text{Paths}(K, \mathbb{A}, w_0)$.

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 with action labelling. 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 with 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. 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, $\mathrm{Zip}(K, \mathbb{A}, w_0) \in \mathcal{L}(\mathcal{H})$?

4.2 Model Checking Hypernodes

We begin by formulating and solving the model-checking problem for hypernode logic (rather than automata) over Kripke structures. This algorithm constitutes the key subroutine for model-checking hypernode automata. To interpret formulas of hypernode logic over a Kripke structure, we equip the Kripke structure with two set of worlds: the *entry worlds*, where trace segments begin, and the *exit worlds*, where trace segments end. Formally, an *open Kripke structure* consists of a Kripke structure $K = (W, \Sigma^X, \Delta, V)$, and a pair $\mathbb{W} = (W_{\mathrm{in}}, W_{\mathrm{out}})$ consisting of a set $W_{\mathrm{in}} \subseteq W$ of entry worlds, and a set $W_{\mathrm{out}} \subseteq W$ of exit worlds.

A *path* of the open Kripke structure $(K, \mathbb{W})$ is path $w_0 \dots w_n$ in K that starts in a entry world, $w_0 \in W_{\mathrm{in}}$ and ends in an exit world, $w_n \in W_{\mathrm{out}}$. The set of unzipped trace segments generated by the open Kripke structure $(K, \mathbb{W})$ is $\mathrm{Unzip}(K)(x) = \{V(w_0, x) \dots V(w_n, x) \mid w_0 \dots w_n \in \mathrm{Paths}(K, \mathbb{W})\}$ for all variables $x \in X$.

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 unzipped trace segments generated by $(K, \mathbb{W})$ a model for φ; that is, $\mathrm{Unzip}(K, \mathbb{W}) \models \varphi$?

Stutter-free automata

From Proposition 2, it follows that it suffices to consider the stutter reduction of $\mathrm{Unzip}(K, \mathbb{W})$ to solve the model-checking problem for hypernode logic. We introduce *stutter-free automata* as a formalism for specifying sets of stutter-free unzipped trace segments. We use stutter-free automata boolean operators to define a filtration that, when applied to a hypernode formula φ and a stutter-free automaton over the variables in φ, returns an automaton with non-empty language iff the language of the input automaton is a model of φ. Finally, we construct, from a given open Kripke structure $(K, \mathbb{W})$, a stutter-free automaton that accepts an unzipped trace segment if the segment is the stutter reduction of a trace segment generated by $(K, \mathbb{W})$. We include a graphical overview of the algorithm in the extended version in [4].

Stutter-free automata are a restricted form of nondeterministic finite automata (NFA) that read unzipped trace segments and guarantees that, for each state, there are no repeated variable assignments on their incoming and outgoing transitions. We denote by Σ^X all assignments of variables in X to values in Σ or the termination symbol #. Formally, for $X = \{x_0, \dots, x_m\}$, let $\Sigma^X = \{x_0 : \sigma_0, \dots, x_m : \sigma_m \mid \forall 0 \le i \le m \ \sigma_i \in \Sigma \cup \{\#\}\} \setminus \{x_0 : \#, \dots, x_m : \#\}$.

▶ **Definition 5.** *Let X be a finite set of variables over Σ. A nondeterministic* stutter-free automaton *(NSFA) 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 $\delta : Q \times \Sigma^X \to 2^Q$ that satisfies the following for all states $q \in Q$ and variables $x \in X$: (i)* stutter-freedom *requiring $\mathit{In}(q, x) \cap \mathit{Out}(q, x) \subseteq \{\#\}$, and (ii)* termination *requiring that if $\# \in \mathit{In}(q, x)$,* *then* $Out(q,x) = \{\#\}$, *where* $In(q,x)$ *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,* $In(q,x) = \{v(x) \mid q \in \delta(q',v)$ *for some* $q' \in Q\}$ *and* $Out(q,x) = \{v(x) \mid \delta(q,v) \neq \emptyset\}$.

A *run* of the stutter-free automaton $\mathcal{A}$ is a finite sequence $q_0v_0q_1v_1 \ldots v_{n-1}q_n$ of alternating states and variable assignments which starts with an initial state, $q_0 \in \hat{Q}$, and satisfies the transition function, $q_{i+1} \in \delta(q_i, v_i)$ for all $i < n$. The run is *accepting* if it ends in a final state, $q_n \in F$. An unzipped trace segment τ over a set of variables X with domain Σ is *accepted* by the stutter-free automaton $\mathcal{A}$ iff there exists an accepting run $q_0v_0 \ldots v_{n-1}q_n$ such that $\tau(x) = v_0(x) \ldots v_{n-1}(x)$, for all $x \in X$. The *language* of $\mathcal{A}$, denoted $\mathcal{L}(\mathcal{A})$, is the set of all accepted unzipped trace segments accepted by $\mathcal{A}$. We sometimes refer to the language of a stutter-free automaton without the termination symbol: $\mathcal{L}(\mathcal{A})|_\# = \{\tau|_\# : X \to \Sigma^* \mid \tau \in \mathcal{L}(\mathcal{A})\}$, where $\tau|_\#$ removes all occurrences of $\#$ in a trace segment τ. Note that since $\mathcal{A}$ is stutter-free, $\mathcal{L}(\mathcal{A})|_\# = \lfloor \mathcal{L}(\mathcal{A})|_\# \rfloor$, where $\lfloor \cdot \rfloor$ is the stutter reduction of unzipped trace segments.

The union, intersection, and determinization for NSFA are defined as usual for NFA; we omit the formal definitions for reasons of space. The complementation of a stutter-free automaton follows the same approach as for NFA: we first determinize the automaton, then complete it, and lastly swap the final and nonfinal states. The only operation that requires special attention for NSFA is *completion*, as we need to be careful to statisfy the condition of stutter-freedom.

A stutter-free automaton $\mathcal{A} = (Q, \hat{Q}, F, \delta)$ over Σ^X is *complete* iff $In(q) \cup Out(q)$ is a maximal subset of Σ^X according to the conditions in Definition 5, where $In(q) = \{v(x) \mid x \in X \text{ and } v(x) \in In(q,x)\}$ and $Out(q) = \{v(x) \mid x \in X \text{ and } v(x) \in Out(q,x)\}$. The *universal* stutter-free automaton $\mathcal{U}_{\Sigma^X}$ over Σ^X, defined next, is a deterministic and complete automaton with language $\mathcal{L}(\mathcal{U}_{\Sigma^X})|_\# = \lfloor (\Sigma^*)^X \rfloor$, i.e., it contains all stutter-free unzipped traces over Σ^X. We use the universal stutter-free automaton as a "sink" area when completing other automata.

▶ **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* Σ^X *is* $\mathcal{U}_{\Sigma^X} = (Q_\mathcal{U}, Q_\mathcal{U}, Q_\mathcal{U}, \delta_\mathcal{U})$*, where* $Q_\mathcal{U} = \Sigma^X$ *and*

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

We use the states and transitions of the universal automaton to complete other stutter-free automata. The details are in the extended version in [4]. The complement of a deterministic and complete stutter-free automaton $\mathcal{A} = (Q, \hat{Q}, F, \delta)$ over Σ^X is $\overline{\mathcal{A}} = (Q, \hat{Q}, Q \setminus F, \delta)$, with the final and nonfinal states interchanged.

▶ **Proposition 7.** *Let* $\mathcal{A}$ *be a deterministic and complete stutter-free automaton over* Σ^X*. Then,* $\overline{\mathcal{A}}$ *is a stutter-free automaton and* $\mathcal{L}(\overline{\mathcal{A}}) = \lfloor (\Sigma^*)^X \rfloor \setminus \mathcal{L}(\mathcal{A})$*.*

From formulas of hypernode logic to stutter-free automata

Having prepared the ground by defining stutter-free automata, which are closed under union, intersection, and complement, we now turn to the model-checking problem for hypernode logic. Given a stutter-free automaton and a hypernode formula, we define an inductive filtration (i.e., in each step we get produce a sub-automaton) over the hypernode formula structure to apply to the automaton we want to model-check. The input automaton is a model of the input formula if the language of the automaton returned by the filtration is non-empty. The inductive filtration for boolean operators translates naturally to automata operators. For atomic hypernode formulas (i.e., the predicate $\precsim$), we define a stutter-free automaton that captures the meaning of $\precsim$.

▶ **Definition 8.** *Let* $\mathcal{U}_X = (Q_\mathcal{U}, Q_\mathcal{U}, Q_\mathcal{U}, \delta_\mathcal{U})$ *be the universal stutter-free automaton over* Σ^X*, and let* $x, y \in X$*. The* stutter-free automaton for the atomic formula $x \precsim y$ of hypernode logic *is the stutter-free automaton* $\mathcal{A}_{x \precsim y} = (Q, Q, Q, \delta)$ *over the same variables and domain, where* $Q = \{v \in Q_\mathcal{U} \mid v(x) = v(y) \text{ or } v(x) = \#\}$*, and* $\delta(q, v) = \delta_\mathcal{U}(q, v)$ *for all* $q \in Q$ *and* $v \in \Sigma^X$*.*

To cope with trace variables in hypernode formulas, we extend the set of variables X with a reference to trace variables in $\mathcal{V}$, by $X_\mathcal{V} = \{x_\pi \mid x \in X \text{ and } \pi \in \mathcal{V}\}$. From an unzipped trace segment τ over the set of variables $X_\mathcal{V}$ we derive the trace assignment $\Pi_\tau(\pi, x) = \tau(x_\pi)$ for all $x \in X$ and $\pi \in \mathcal{V}$. We prove now that all words accepted by the stutter-free automaton for $x(\pi) \precsim y(\pi')$ define assignments that satisfy that hypernode atomic formula.

▶ **Lemma 9.** *An unzipped trace segment* τ *over* $(\Sigma^{X_\mathcal{V}})^*$ *is accepted by* $\mathcal{A}_{x_\pi \precsim y_{\pi'}}$ *over the same variables and domain,* $\tau \in \mathcal{L}(\mathcal{A}_{x_\pi \precsim y_{\pi'}})|_\#$*, iff* $\Pi_\tau \models x(\pi) \precsim y(\pi')$*.*

The inductive filtration defined next is the main element of the model-checking algorithm for formulas of hypernode logic.

▶ **Definition 10.** *Let* $\mathcal{A}$ *be a stutter-free automaton, and* φ *a formula of hypernode logic. 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:*

$$\begin{aligned}
(x(\pi) \precsim y(\pi'))^+[\mathcal{A}] &= \mathcal{A} \cap \mathcal{A}_{x_\pi \precsim y_{\pi'}} & (x(\pi) \precsim y(\pi'))^-[\mathcal{A}] &= \mathcal{A} \cap \overline{\mathcal{A}_{x_\pi \precsim y_{\pi'}}} \\
(\varphi_1 \wedge \varphi_2)^+[\mathcal{A}] &= \varphi_1^+[\mathcal{A}] \cap \varphi_2^+[\mathcal{A}] & (\varphi_1 \wedge \varphi_2)^-[\mathcal{A}] &= \varphi_1^-[\mathcal{A}] \cup \varphi_2^-[\mathcal{A}] \\
(\neg\varphi)^+[\mathcal{A}] &= \varphi^-[\mathcal{A}] & (\neg\varphi)^-[\mathcal{A}] &= \varphi^+[\mathcal{A}] \\
(\exists\pi\varphi)^+[\mathcal{A}] &= \varphi^+[\mathcal{A}] & (\exists\pi\varphi)^-[\mathcal{A}] &= \mathcal{A} \setminus \varphi^+[\mathcal{A}].
\end{aligned}$$

We reduce the problem of model checking a stutter-free automaton $\mathcal{A}$ over a formula φ with n trace variables to filtering the n-self-composition of $\mathcal{A}$ by φ. The stutter-free automaton $\mathcal{A}^n$ is the result of composing n copies of $\mathcal{A}$ under a standard synchronous product construction, where for each copy $\mathcal{A}_i$, with $i \leq n$, all program variables $x \in X$ are renamed to x_{π_i}. Note that, the assignment derived by an unzipped trace segment τ accepted by $\mathcal{A}^n$ defines a trace assignment from $\{\pi_1, \ldots, \pi_n\}$ to traces accepted by $\mathcal{A}$.

▶ **Theorem 11.** *Let* $\mathcal{A}$ *be a stutter-free automaton, and* φ *a formula of hypernode logic with* n *trace variables. 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* $\mathcal{L}(\mathcal{A}) \not\models \varphi$*.*

Model checking hypernode logic over Kripke structures

We are only missing to translate an open Kripke structure $(K, \mathbb{W})$ to a stutter-free automaton $\mathcal{A}_{K,\mathbb{W}}$ defining the same unzipped trace segments; i.e., $\mathcal{L}(\mathcal{A}_{K,\mathbb{W}})|_\# = \lfloor \text{Unzip}(K, \mathbb{W}) \rfloor$. We present the details in the appendix, but, in a nutshell, we represent the progression of each variable valuation along the Kripke structure independently (to allow skipping stuttering states) in the derived stutter-free automaton. Having this translation, we can apply the filtration from Definition 10 to the stutter-free automaton derived by an open Kripke structure to solve the model-checking problem for hypernode logic.

▶ **Theorem 12.** *Let $(K, \mathbb{W})$ be an open Kripke structure, and φ a formula of hypernode logic over the same set of variables. Let n be the number of trace variables in φ. Then,* $\mathrm{Unzip}(K, \mathbb{W}) \models \varphi$ *iff* $\mathcal{L}(\varphi^+[\mathcal{A}^n_{K,\mathbb{W}}]) \neq \emptyset$*.*

The proof follows from Proposition 7, Theorem 11, and $\mathcal{L}(\mathcal{A})|_\# = \lfloor \mathcal{L}(\mathcal{A})|_\# \rfloor$. This gives us our main result.

▶ **Theorem 13.** *Model checking of hypernode logic over open Kripke structures is decidable.*

Using our algorithm, the running time of model checking a formula of hypernode logic over an open Kripke structure depends doubly exponentially on the number of variables, singly exponentially on the number of worlds of the Kripke structure, and singly exponentially on the length of the formula.

▶ **Corollary 14.** *The time complexity of model checking a formula φ of hypernode logic with n trace variables and m variables, over an open Kripke structure with k worlds, is $\mathcal{O}(2^{n \cdot k^m})$.*

Proof. The encoding of the open Kripke Structure by a stutter-free automaton has $\mathcal{O}(k^m)$ states. The determinized stutter-free automaton has $\mathcal{O}(2^{k^m})$ states. After completing the deterministic stutter-free automaton there are 2^m states. We observe that the size of the domain Σ only affects the step of completing a stutter-free automaton, which adds $|\Sigma|^{|X|}$ states. This addition is dominated by the more expensive step of determinizing the automaton. Finally, the n-self-composition of the resulting automaton has $\mathcal{O}(2^{n \cdot k^m})$ states. ◀

4.3 Model Checking Hypernode Automata

We defined the run of a hypernode automaton for a given action sequence p, with each run inducing a slicing of a set of action-labeled traces consistent with p. To model-check a hypernode automaton $\mathcal{H}$ against a pointed Kripke structure $(K, \mathbb{A}, w_0)$ with an action labeling, we build a finite automaton, called $\mathrm{Slice}(K, \mathbb{A}, w_0)$, which encodes all slicings of action-labeled traces generated by $(K, \mathbb{A}, w_0)$. We then reduce the model-checking problem to checking whether the language defined by the composition of $\mathrm{Slice}(K, \mathbb{A}, w_0)$ with the specification automaton $\mathcal{H}$, called $\mathrm{Join}(\mathcal{H}, K, \mathbb{A}, w_0)$, is non-empty. We include an overview of this process in the extended version in [4].

We start by defining the *slicing* of a given Kripke structure $K = (W, \Sigma^X, \Delta, V)$ for a given action labeling $\mathbb{A}$. The building blocks of the slicing are Kripke substructures. A Kripke structure $K' = (W', \Sigma^X, \Delta', V')$ is a *substructure* of K, denoted $K' \leq K$, iff $W' \subseteq W$, and for all worlds $w \in W'$ we have $\Delta'(w) \subseteq \Delta(w)$ and $V'(w) = V(w)$. The *substructure induced by a transition relation* $\Delta' \subseteq \Delta$ is $K[\Delta'] = (W', X, \Delta', V(W'))$, where $W' = \{w, w' \mid (w, w') \in \Delta'\}$. The transition relation defined by *all transitions in a path* of the action-labeled Kripke structure $(K, \mathbb{A})$ from an entry world in $W_{\mathrm{in}} \subseteq W$ to the first step labeled with $a \in A$ is:

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

The *open substructure induced by* $(K, \mathbb{A}, W_{\mathrm{in}}) \downarrow a$, written $\mathbb{W}[(K, \mathbb{A}, W_{\mathrm{in}}) \downarrow a]$, is the open Kripke structure where the Kripke structure is $K[(K, \mathbb{A}, W_{\mathrm{in}}) \downarrow a]$, the set W_{in} are the entry worlds, and the set $\{w \mid w \in W \text{ and } \mathbb{A}(w, a) \neq \emptyset\}$ are the exit worlds, containing all possible exit points for action a.

We define the finite automaton $\mathrm{Slice}(K, \mathbb{A}, w_0)$ and prove, in Lemma 16 below, that every finite action sequence p defines a unique path in this automaton, and the slices of this path contain the same trace segments that are obtained when the action sequence p is applied directly to the original pointed, action-labeled Kripke structure. The states of the automaton

$\text{Slice}(K, \mathbb{A}, w_0)$ are all open substructures induced by paths from any choice of entry worlds to an action $a \in \mathbb{A}$. Note that there are only finitely many such states. The transition relation of $\text{Slice}(K, \mathbb{A}, w_0)$ connects, for all actions a, open substructures with exit a and open substructures with matching entry worlds.

▶ **Definition 15.** *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 with states $Q = \{\mathbb{W}[(K, \mathbb{A}, W_{\text{in}}) \downarrow a] \mid a \in A \text{ and } W_{\text{in}} \subseteq W\}$; initial states $\hat{Q} = \{\mathbb{W} \in Q \mid \text{entry}(\mathbb{W}) = \{w_0\}\}$; transition function $\delta : Q \times A \to Q$, where $\delta(\mathbb{W}, a) = \mathbb{W}'$ iff $\mathbb{W}$ exits with action a, that is, for all $w \in \text{exit}(\mathbb{W})$ there exists $w' \in \mathbb{W}'$ such that $a \in \mathbb{A}(w, w')$, and the entry worlds of $\mathbb{W}'$ define a maximal subset of the worlds accessible with action a from the exit worlds in $\mathbb{W}$, that is, for all $\mathbb{W}'' \in Q$ that are not $\mathbb{W}'$, if $\text{entry}(\mathbb{W}'') \subseteq \{w \mid a \in \mathbb{A}(w', w) \text{ for some } w' \in \text{exit}(\mathbb{W})\}$, then $\text{entry}(\mathbb{W}') \not\subseteq \text{entry}(\mathbb{W}'')$. Here, $\text{entry}(\mathbb{W})$ and $\text{exit}(\mathbb{W})$ refer to the sets of entry and exit worlds of the open Kripke structure $\mathbb{W}$, respectively.*

We remark that the transition function $\delta : Q \times A \to Q$ is well-defined, because there is a unique maximal subset for the next entry worlds, given an action a. For every two open Kripke substructures, $\mathbb{W}_1$ and $\mathbb{W}_2$, their union defines $\mathbb{W}[(K, \mathbb{A}, \text{entry}(\mathbb{W}_1) \cup \text{entry}(\mathbb{W}_2)) \downarrow a]$, which is again a state of the slicing.

▶ **Lemma 16.** *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 $\text{Zip}(K, \mathbb{A}, w_0)[p] \neq \emptyset$, then p defines a unique run $\mathbb{W}_0 a_0 \cdots \mathbb{W}_n a_n$ of $\text{Slice}(K, \mathbb{A}, w_0)$ such that for all $0 \leq i \leq n$, $\text{Paths}(\mathbb{W}_i) = \text{Paths}(K, \mathbb{A}, w_0)(a_0 \dots a_{i-1}, a_i)$.*

In a final step, we define a synchronous composition of the slicing automaton defined above and the given hypernode automaton $\mathcal{H}$. The states of this composition are pairs consisting of open Kripke substructures (stemming from the given pointed, action-labeled Kripke structure) and formulas of hypernode logic (stemming from the hypernode labels of $\mathcal{H}$). We mark as final states all pairs where the open Kripke substructure is not a model of the hypernode formula.

▶ **Definition 17.** *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 *$(K, \mathbb{A}, w_0)$,* $\text{Slice}(K, \mathbb{A}, w_0) =$ *$(Q_s, \hat{Q}_s, \delta_s)$, is the finite automaton $\text{Join}(\mathcal{H}, K, \mathbb{A}, w_0) = (Q, \hat{Q}, F, A, \delta)$ with set of states $Q = \{(\mathbb{W}, q) \mid \mathbb{W} \in Q_s, q \in Q_h \text{ and } \mathbb{W} \models \gamma(q)\} \cup \{(\mathbb{W}, \overline{q}) \mid \mathbb{W} \in Q_s, q \in Q_h \text{ and } \mathbb{W} \not\models \gamma(q)\}$; initial states $\hat{Q} = \{(\mathbb{W}, \hat{q}) \in Q \mid \mathbb{W} \in \hat{Q}_s\} \cup \{(\mathbb{W}, \overline{\hat{q}}) \in Q \mid \mathbb{W} \in \hat{Q}_s\}$; final state $F = \{(\mathbb{W}, \overline{q}) \mid (\mathbb{W}, \overline{q}) \in Q\}$; transition function $\delta : Q \times A \to Q$, where for all $(\mathbb{W}, q) \in Q$, we have $\delta((\mathbb{W}, q), a) = \{(\mathbb{W}', q') \in Q \mid \delta_h(q) = (q', a) \text{ and } \mathbb{W}' \in \delta_s(\mathbb{W}, a)\}$.*

The finite automaton $\text{Join}(\mathcal{H}, K, \mathbb{A}, w_0)$ reads sequences of actions. The notion of run is defined as usual, and a run is accepting if it ends in a final state. The language of the automaton is empty iff it has no accepting run.

▶ **Theorem 18.** *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, $\text{Zip}(K, \mathbb{A}, w_0) \in \mathcal{L}(\mathcal{H})$ iff the language of the finite automaton $\text{Join}(\mathcal{H}, K, \mathbb{A}, w_0)$ is empty.*

The following theorem puts all results from this section together.

▶ **Theorem 19.** *Model checking of hypernode automata over pointed Kripke structures with action labelings is decidable.*

Proof. We have seen that the model checking of hypernode logic over open Kripke structures is decidable (Theorem 13). Evaluating $\mathrm{Join}(\mathcal{H}, K, \mathbb{A}, w_0)$ is also decidable. The main challenge is the slicing of $\mathbb{A}(K, w_0)$. Note that there is a finite number of states that can be in $\mathbb{A}(K, w_0)$, as they are all substructures of the Kripke structure K. ◀

The hardest part of model checking a hypernode automaton over an action-labeled Kripke structure is checking the formulas of all hypernodes. Therefore, also the running time for model checking hypernode automata is dominated, as with hypernode logic, by a doubly exponential dependency on the number of program variables. Furthermore, our model-checking algorithm depends singly exponentially on both the size of the Kripke structure and the size of the hypernode automaton.

▶ **Corollary 20.** *Let A be a set of actions and X a set of m program variables. Let (K, w_0) be a pointed Kripke structure over X, and $\mathbb{A}$ an action labeling for K over A. Let $\mathcal{H}$ be a hypernode automaton over X and A. The time complexity of checking whether $\mathrm{Zip}(K, \mathbb{A}, w_0) \in \mathcal{L}(\mathcal{H})$ is $\mathcal{O}(|\mathcal{H}| \cdot 2^{|A|+n\cdot|K|^m})$, where n is the largest number of trace quantifiers that occurs in any hypernode formula in $\mathcal{H}$.*

5 Related Work

The first logic studied to express asynchronous hyperproperties was an extension of μ-calculus with explicit quantification over traces, called Hμ [12]. The trace-quantifier free formulas of Hμ are expressively equivalent to the parity multi-tape Alternating Asynchronous Word Automata (AAWA) introduced in [12]. Both formalisms have highly undecidable model-checking problems. The undecidability stems from comparing positions in different traces that are arbitrarily far apart, over an unbounded number of traces. When one of the two dimensions (the distance between positions, or the number of traces) is given an explicit finite bound, model checking becomes decidable [12]. In comparison, we achieve decidability by an entirely different means: we decouple the progress of different program variables (asynchronicity), while allowing resynchronization through automaton-level transitions.

In Hμ formulas, trace quantifiers always precede time operators, while hypernode automata allow a restricted form of quantifier alternation between time operators and trace quantifiers. In particular, the automaton-level transitions correspond to outermost time operators, which precede the trace quantifiers of hypernode logic formulas, whose stutter-reduced prefixing relations correspond to innermost time quantifiers. We conjecture that Hμ and hypernode automata have incomparable expressive powers. Consider, for example, the hypernode automaton shown in Figure 2, which 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. Each new slice can have a different trace π witnessing the asynchronous progress of p. The length of the traces in each slice is unbounded, and as we do not know how many times a repeats, the number of slices is also unbounded. Hence we do not know how many outermost existential trace quantifiers would be needed in order to guarantee a different trace witness for each slice. Therefore we conjecture that the hyperproperty that is specified by the hypernode automaton of Figure 2 cannot be expressed in Hμ.

Also various extensions of HyperLTL were explored recently in order to support asynchronous hyperproperties. *Stuttering* HyperLTL (HyperLTL$_S$) and *context* HyperLTL (HyperLTL$_C$), both introduced in [8], extend HyperLTL with new operators. *Asynchronous* HyperLTL (A-HyperLTL) [5] extends HyperLTL with quantification over *trajectories*. A trajectory specifies the traces that progress in each evaluation step. While the model-checking

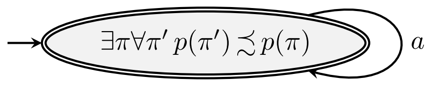

Figure 2 Hypernode automaton specifying that within each slice of a trace set induced by the repeated action a, there exists a trace that describes the asynchronous progress of the propositional variable p within the current slice.

problems for all of these extensions of HyperLTL are undecidable, the authors identify syntactic fragments that support certain asynchronous hyperproperties. These decidable fragments adopt restrictions akin to the decidable parts of Hμ. All of HyperLTL$_S$, HyperLTL$_C$, and A-HyperLTL are subsumed by Hμ [9]. As we argued that the hypernode automaton from Figure 2 cannot be expressed in Hμ, it would neither be expressible in any of the three proposed asynchronous extensions of HyperLTL.

Krebs et al. [14] propose to reinterpret LTL under a so-called *team* semantics. Team semantics works with sets of variable assignments, and the authors introduce both synchronous and asynchronous varieties. They prove that under asynchronous team semantics, LTL is as expressive as universal HyperLTL (where all trace quantifiers are universal quantifiers). As hypernode logic allows existential quantification over traces, again, our approach is orthogonal and expressively incomparable.

In [6] the authors introduce HyperATL*, which extends alternating-time temporal logic [1] with strategy quantifiers that bind strategies to trace variables, and an explicit construct to resolve games in parallel. HyperATL* enables the specification of strategic hyperproperties. This work is orthogonal to ours as we are interested in *linear-time* asynchronous hyperproperties, rather than strategic hyperproperties.

Although many logic-based specification languages have been proposed to express asynchronous hyperproperties, there is a lack of automaton-based approaches to specify such properties. Note that AAWA [12] do not support explicit quantification over trace variables. The finite-word *hyperautomata* of [7] constitute a step in this direction by prefixing finite automata with explicit quantification over traces, but they are limited to *synchronous* hyperproperties.

6 Conclusion

We presented a new formalism for specifying hyperproperties of concurrent systems. Our formalism mixes synchronization between different execution traces, expressed as action-labeled transitions of a specification automaton, with asynchronous comparisons between corresponding segments of different traces, expressed as hypernode logic formulas that label the states of the specification automaton. In this way, the specification language of hypernode automata can alternate asynchronous requirements on trace segments of possibly different lengths with synchronization points. Unlike previous formalisms for specifying asynchronous hyperproperties, hypernode automata fully support automatic verification. Our model-checking algorithm for hypernode automata is based on an entirely novel technique that introduces stutter-free automata and operations on these automata, thus providing a nice example for the power of automata-theoretic methods in verification.

Besides having a decidable verification problem, hypernode automata represent a genuinely new and useful specification language. We demonstrated this by specifying several published variations of observational determinism using hypernode logic, by specifying information declassification using hypernode automata, and by presenting a formula to support our conjecture that the formalism of hypernode automata is expressively incomparable to various hyperlogics that have been proposed recently for specifying asynchronous hyperproperties.

The boundary between asynchronicity and synchronicity of trace comparisons can be fine-tuned by introducing variables with compound types, such as boolean arrays, which can be used, for example, to couple the variables of each thread of a multi-threaded program. The ramifications of such alphabet variations on hypernode logic and hypernode automata are to be explored in future work. There is no shortage of additional topics that follow immediately from the present work but, even if straightforward, require further investigations, including the study of hypernode automata with partial and nondeterministic transition relations, and of hypernode automata with infinitary acceptance conditions (such as hypernode Büchi automata), as well as the extension of formal expressiveness studies for hyperproperty specifications in order to include hypernode logic and automata, and the presentation of algorithms for solving classical decision problems for hypernode logic and automata other than model checking (such as satisfiability and emptiness). Also the applicability of stutter-free automata in other asynchronous verification contexts (not necessarily concerning hyperproperties) is an interesting question.

References

1 Rajeev Alur, Thomas A. Henzinger, and Orna Kupferman. Alternating-time temporal logic. *J. ACM*, 49(5):672–713, 2002. doi:10.1145/585265.585270.

2 Aslan Askarov and Andrei Sabelfeld. Gradual release: Unifying declassification, encryption and key release policies. In *2007 IEEE Symposium on Security and Privacy*, pages 207–221, 2007. doi:10.1109/SP.2007.22.

3 Ezio Bartocci, Thomas Ferrère, Thomas A. Henzinger, Dejan Nickovic, and Ana Oliveira da Costa. Flavors of sequential information flow. In Bernd Finkbeiner and Thomas Wies, editors, *23rd International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 13182 of *LNCS*, pages 1–19. Springer International Publishing, 2022. doi:10.1007/978-3-030-94583-1_1.

4 Ezio Bartocci, Thomas A. Henzinger, Dejan Nickovic, and Ana Oliveira da Costa. Hypernode automata, 2023. arXiv:2305.02836.

5 Jan Baumeister, Norine Coenen, Borzoo Bonakdarpour, Bernd Finkbeiner, and César Sánchez. A temporal logic for asynchronous hyperproperties. In Alexandra Silva and K. Rustan M. Leino, editors, *Computer Aided Verification (CAV)*, pages 694–717. Springer International Publishing, 2021. doi:10.1007/978-3-030-81685-8_33.

6 Raven Beutner and Bernd Finkbeiner. A Temporal Logic for Strategic Hyperproperties. In *32nd International Conference on Concurrency Theory (CONCUR)*, volume 203 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 24:1–24:19, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.CONCUR.2021.24.

7 Borzoo Bonakdarpour and Sarai Sheinvald. Finite-word hyperlanguages. In Alberto Leporati, Carlos Martín-Vide, Dana Shapira, and Claudio Zandron, editors, *Language and Automata Theory and Applications (LATA)*. Springer International Publishing, 2021. doi:10.1007/978-3-030-68195-1.

8 Laura Bozzelli, Adriano Peron, and César Sánchez. Asynchronous extensions of HyperLTL. In *Proceedings of the 36th Annual ACM/IEEE Symposium on Logic in Computer Science*. Association for Computing Machinery, 2021. doi:10.1109/LICS52264.2021.9470583.

9 Laura Bozzelli, Adriano Peron, and César Sánchez. Expressiveness and Decidability of Temporal Logics for Asynchronous Hyperproperties. In Bartek Klin, Sławomir Lasota, and Anca Muscholl, editors, *33rd International Conference on Concurrency Theory (CONCUR)*, volume 243 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 27:1–27:16, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.CONCUR.2022.27.

10 Michael R. Clarkson, Bernd Finkbeiner, Masoud Koleini, Kristopher K. Micinski, Markus N. Rabe, and César Sánchez. Temporal logics for hyperproperties. In *Proc. of POST 2014: the Third International Conference on Principles of Security and Trust*, volume 8414 of *Lecture Notes in Computer Science*, pages 265–284. Springer, 2014. doi:10.1007/978-3-642-54792-8.

11 Michael R. Clarkson and Fred B. Schneider. Hyperproperties. *Journal of Computer Security*, 18(6):1157–1210, 2010. doi:10.3233/JCS-2009-0393.

12 Jens Oliver Gutsfeld, Markus Müller-Olm, and Christoph Ohrem. Automata and fixpoints for asynchronous hyperproperties. *Proc. ACM Program. Lang.*, 5(POPL), 2021. doi:10.1145/3434319.

13 M. Huisman, P. Worah, and K. Sunesen. A temporal logic characterisation of observational determinism. In *19th IEEE Computer Security Foundations Workshop (CSFW)*, pages 13 pp.–3, 2006. doi:10.1109/CSFW.2006.6.

14 Andreas Krebs, Arne Meier, Jonni Virtema, and Martin Zimmermann. Team Semantics for the Specification and Verification of Hyperproperties. In Igor Potapov, Paul Spirakis, and James Worrell, editors, *43rd International Symposium on Mathematical Foundations of Computer Science (MFCS 2018)*, volume 117 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 10:1–10:16. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018. doi:10.4230/LIPIcs.MFCS.2018.10.

15 Andrei Sabelfeld and David Sands. Declassification: Dimensions and principles. *Journal of Computer Security*, 17(5):517–548, 2009.

16 Tachio Terauchi. A type system for observational determinism. In *2008 21st IEEE Computer Security Foundations Symposium*, pages 287–300, 2008. doi:10.1109/CSF.2008.9.

17 S. Zdancewic and A.C. Myers. Observational determinism for concurrent program security. In *16th IEEE Computer Security Foundations Workshop, 2003. Proceedings.*, pages 29–43, 2003. doi:10.1109/CSFW.2003.1212703.