TU WIEN Informatics

Design and Verification of Information Flows

DISSERTATION

zur Erlangung des akademischen Grades

Doktorin der Technischen Wissenschaften

eingereicht von

MSc Ana A. Oliveira da Costa
Matrikelnummer 01624245

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof. Dr. Ezio Bartocci
Zweitbetreuung: Univ.Prof. Dr. Matteo Maffei

Diese Dissertation haben begutachtet:

Alessandro Aldini	Einar Broch Johnsen

Wien, 21. Juni 2024

Ana A. Oliveira da Costa

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

TU WIEN Informatics

Design and Verification of Information Flows

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

Doktorin der Technischen Wissenschaften

by

MSc Ana A. Oliveira da Costa
Registration Number 01624245

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Dr. Ezio Bartocci
Second advisor: Univ.Prof. Dr. Matteo Maffei

The dissertation has been reviewed by:

Alessandro Aldini	Einar Broch Johnsen

Vienna, June 21, 2024

Ana A. Oliveira da Costa

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

MSc Ana A. Oliveira da Costa

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 21. Juni 2024

Ana A. Oliveira da Costa

Acknowledgements

Bringing my ideas to paper has often been a daunting and painfully slow task to tackle. During my PhD journey, there were many moments when I thought I would not reach this milestone. This milestone was possible because of the many good and supportive people I had the luck to meet in the past years.

I want to start by expressing my deepest gratitude to my supervisor, Ezio Bartocci, for taking a chance on me and for staying by my side on all the ups and, mainly, the many downs that we faced. None of this would have been possible without his unconditional support and guidance. The feedback and continuous interest in my developments from my proficiency evaluation committee, Laura Kovacs and Martina Seidl, made me confident in my progress and helped me move forward. The same can be said of my co-advisor, Matteo Maffei.

I am also profoundly grateful to my co-authors, Thomas Henzinger, Thomas Ferrère, and Dejan Nickovic. From the beginning, they welcomed me as one of their own, making me feel valued and seen. Our insightful discussions, their constructive feedback, and their unwavering support made me grow as a researcher and a person. I am genuinely grateful for their commitment to our projects and for the opportunity to work alongside such inspiring people.

Essential for this achievement is also the environment and the people I met as part of the Doctoral Program Logical Methods in Computer Science. From all the inspiring colleagues who made my research challenges feel less heavy, I would like to give special thanks to Sanja, Emily and Andre. From our many coffee-inspired conversations, office sharing days and evenings doing some sport or maybe just trying out some new restaurant, they were a constant in this journey. From the staff, I am deeply grateful to Anna Prianichnikova and Beatrix Buhl.

During my time abroad at Michigan State University, I brought new ideas and projects and made lifelong friends. I am thankful for crossing paths with Borzoo Bonakdarpour and having the opportunity to visit him and his group. During my stay, Borzoo was a great mentor. His care and attention to his students are inspiring. From this stay, I will forever remember the Michigan family I found in Borzoo, Oyendrila, Tzu-Han, and Ritam.

The adventure that led to this day started many years ago in Lisbon when I left my software developer job and took a leap of faith to pursue a Master's in Computational Logic. This would not have been possible without the support and enthusiasm of Peter and his family, who became my first family outside of Portugal. They believed in me before I did. From my time in that Master's program, I am most grateful for the lifelong friends I found in Mariëlle, Alexandra, Elena and Isabelly. We grew together during this experience of studying and building a life abroad. I could not miss to mention some of my closest friends in Vienna: Maria, Jenny, Robert, and Serge.

From all the friendships I built over these years, I am the most thankful to have found Deba. Our friendship taught me more about myself and the world than I could have ever hoped for. With Deba, I learned how to speak out and trust my instincts. In him, I found a partner for life. Along with him and our feline babies, Bulina and Werner, I could start building a home and a family.

Finally, I could not be here without my parents' and brother's unconditional love and support. What I am today is because of all you taught me.

Kurzfassung

Immer komplexere Systeme führen zu einem erhöhten Bedarf an verantwortungsvollem Umgang mit sensiblen und kritischen Daten. Richtlinien zur Verwendung und zum Schutz von Daten legen fest, wie Informationen durch Systeme fließen und beobachtet werden dürfen. Die Durchsetzung dieser Richtlinien stellt eine Herausforderung dar, weil Informationen aus der Analyse mehrerer Systemausführungen abgeleitet werden können. Um solche Richtlinien durchzusetzen, reicht es nicht aus, jede Systemausführung einzeln zu prüfen (wie bei trace properties). Aus formaler Sicht werden Richtlinien zum Informationsfluss durch hyperproperties beschrieben. In dieser Arbeit setzen wir uns mit den mathematischen Grundlagen zur Gestaltung und Überprüfung sicherer Systeme auseinander.

Im ersten Teil der Dissertation beschäftigen wir uns mit dem Problem der „Security by Design“, indem wir die strukturelle Sicht auf Informationsflüsse (wo Informationen fließen) von der semantischen Sicht (was ein Informationsfluss ist) trennen. Danach führen wir eine Schnittstellentheorie ein, um die Gestaltung aus der Perspektive der Richtlinien zu unterstützen.

Im zweiten Teil der Dissertation konzentrieren wir uns auf die Spezifikation des Informationsflusses und beweisen, dass die erfolgreichste Logik für hyperproperties, HyperLTL, asynchrone hyperproperties nicht ausdrücken kann.

Im letzten Beitrag der Dissertation nähern wir uns dieser Beschränkung, indem wir Automaten und Logik in einem neuen Formalismus namens hypernode automata kombinieren. Die Logik regelt Asynchronität innerhalb von Spezifikationszuständen, während Automaten-Transitionen Synchronisationspunkte zwischen diesen Zuständen ermöglichen. Abschließend führen wir einen neuen lösbaren Modellprüfungsalgorithmus für asynchrone hyperproperties ein, welche mittels hypernode automata beschrieben werden.

Abstract

The ever-increasing systems' interconnectivity with the amount of information made available lead to a growing demand for responsible handling of sensitive and critical data. Data use and protection policies can be specified as *information-flow policies* that state how information can flow and be observed through systems or computations.

Enforcing information-flow policies is notoriously a challenging problem because information can be inferred by analyzing multiple executions of a system. To enforce such policies, verifying each system execution separately (as it is done for trace properties) is not enough. From a formal standpoint, information-flow policies are specified by *hyperproperties*. In this work, we dive into the mathematical foundations for designing and verifying secure systems.

Many challenges must be addressed to design frameworks that help build secure systems. Due to the complexity of the systems' development cycle, security requirements should be laid down as soon as possible. This approach, called *security by design*, is difficult to achieve in practice because hyperproperties are not closed under refinement. Moreover, it is usually unclear how to interpret security policies in terms of hyperproperties at the early stages of a system design. We address this in the first part of the thesis by separating the structural view of information flow (*where* information flows) from its semantic view (*what* is a flow of information). We then introduce an *interface theory* to support *contract-based design* for the structural perspective.

Observations of different systems' executions are often not aligned temporally, with information-flow specification often defining *asynchronous hyperproperties*. In the second part of the thesis, we focus on the specification of information-flow and prove that the most successful logic for hyperproperties, HyperLTL, cannot express asynchronous hyperproperties.

Asynchronous hyperproperties received much attention recently, with many logics proposed to specify them. They all have, however, undecidable model-checking problems, with decidability achieved only for restricted fragments. In the last contribution of this thesis, we approach this difficulty by combining automata and logic in a new formalism called *hypernode automata*. The logic handles asynchronicity within specification states, while automata transitions enable synchronization points between those states. We finish by introducing a novel decidable model-checking algorithm for asynchronous hyperproperties specified with hypernode automata.

Contents

CHAPTER 1

Introduction

In our highly connected world, the fast development of technology and its ever-increasing reach in all facets of people's lives makes it a priority to find solutions to guarantee that systems function safely and securely. Technological solutions are becoming more complex and interconnected, with systems encompassing diverse components and having sophisticated architectures. In this thesis, we delve into the problem of designing, specifying and verifying secure flow of information in systems with complex architectures.

As an example of challenging systems to formally ensure security features, we look at the case of Cyber-Physical Systems (CPS) that combine computational and physical processes by integrating functionalities from different components like sensors and actuators [Tri16, RKG+19]. Recently, CPS have become prominent in many vital areas of our society, such as transportation (e.g. advanced automotive control, avionics or traffic control) or healthcare (e.g. high-confidence medical devices or assisted living) [Lee08]. A modern car, for example, is a CPS consisting of several computational units such as the Engine Control Unit (ECU), the Electronic Stability Control (ESC), and an Electronic Brake Control Module (EBCM). Undoubtedly, decisions made by these control units do not only affect the physical hardware of the car but also its surroundings. Thus, there are rising concerns about the reliability and safety guarantees of CPS because their failures can be life-threatening. These concerns are supported by many examples of life-critical malfunctioning or unsafe CPS [AFRP15, RKG+19, Hel18].

Security policies are usually instantiated by restricting information flows in a system. The main idea is to prevent a user who can only observe the system's public behavior from learning its secrets, i.e., preventing information leakage to an unauthorized user. For example, in a modern car, an attacker may try to gain control by taking advantage of the information we can gather by observing its physical and computational behavior. The tight coupling between the cyber and the physical components enables the attacker to deduce computational properties, like encryption secrets, by exploiting side channels such as power consumption and electromagnetic radiation [SDK19]. These attacks often require

you to analyze multiple signals from such channels to deduce the necessary information. Hence, this type of security vulnerabilities cannot be characterized by properties of a single execution trace, i.e., they cannot be characterized by trace properties. Instead, security properties verification requires comparing different executions of the system. From a language point-of-view, they define *hyperproperties* [CS10]. While trace properties define sets of traces (i.e., a set with all correct executions), hyperproperties define sets of trace sets (i.e., a set with all well-behaved systems). This added level of sets brings in interesting questions. For instance, how to define refinement to support a step-wise design of systems or build formal frameworks that effectively allow reasoning about information flow requirements.

In this work, we explore three perspectives in which formal methods can aid in developing secure systems. In particular, we investigate how to add security considerations while designing a system; we explore expressivity boundaries of languages for hyperproperties; and, finally, we propose a new formalism that combines logic and automata to verify security requirements effectively. Naturally, we organize the contributions present in this thesis in three parts, namely: *(I) Information-flow Design*, *(II) Information-flow Specification* and *(III) Information-flow Verification*.

Information-flow Design

In the fist part of this thesis, we address the problem of adding security considerations while designing a system. When we look, for instance, at CPS, their heterogeneous nature often requires different teams and providers to be involved during the system's life cycle. Therefore, security requirements should be laid down as soon as possible, as it may not be possible to enforce them later without compromising the system's functionality due to earlier design or implementation decisions. This approach called *security-by-design*, is essential to address reliability and safety concerns in such complex systems. In particular, in this part, we explore how to use *contract-based design* [BCN+18] ideas and techniques to design secure systems.

A theory of contracts provides operators and relations to support both top-down design steps (i.e., starting from an abstract specification, refining it step-by-step and possibly decomposing it into smaller components) and bottom-up design steps (i.e., assembling a system by composing different components). In this work, we are interested in contract-based design theories formalized by *interface theories* [dAH01b]. Interfaces specify requirements over *open systems* where part of the system's behavior is influenced by its interaction with its environment (i.e., the context in which it will be executed) and the rest is determined by the system's implementation. Hence, an interface specification distinguishes between its requirements on the environment, called *assumptions*, and its promises on its implementation behavior, called *guarantees*.

We introduce a new interface theory tailored to support a contract-based design approach of information-flow policies, called *information-flow interfaces*. This theory focuses on the structural view of information flow, i.e., in specifying *where* information cannot flow,

instead of *what* is an information leak. Focusing on the structural aspect addresses two key points to enable secure-by-design for complex systems (like CPS): it allows delaying semantical considerations to a later design phase (effectively allowing the addition of formal security guarantees at early design stages), and it supports compositional and uniform reasoning across heterogeneous components, where a common semantic-view may not be possible.

Information-flow Specification

As we transition from the system's design to its implementation, it becomes necessary to instantiate the information-flow requirements established at design time to concrete security properties to be enforced and verified. Unfortunately, this step (from policies to properties) is not always straightforward because it may, for example, depend on how the system runs or how we observe its executions.

Let us look at the example of using the general notion of independence to instantiate information-flow requirements. Concretely, we can require that there is no flow of information from variables holding secret information to public channels by specifying that observations of the public channels must be independent of the secret values. Without formally defining independence, it is already possible to identify some considerations that will affect the specification of such a property. For example, we can adopt either a point-wise or a trace-segment view on the scope of the independence requirement. In the point-wise view, we check if each time point across all executions satisfies the independence requirement, while, in the trace-segment view, we check if all the executions observed behavior meets the intended independence constraint. When, additionally, we need to account for misalignment between observations of different executions of the system, the possible semantic interpretations (and, consequently, encoding) of independence grow considerably.

In this part of the thesis, our goal is to investigate how minor changes in a property specification can significantly affect the set of models it generates (i.e., the set with all systems that satisfy it) and, in particular, how it affects the possibility to express the property with a HyperLTL formula. To study such expressiveness boundaries, we consider the sequential information-flow policy requiring values of a variable y to be independent of a variable x until the values of a variable z are independent of x. We prove that we cannot use HyperLTL formulas to specify most of the variants of this sequential information-flow policy because they cannot deal with the specification state change that may happen asynchronously across the system executions. We can generalize these results to the inadequacy of HyperLTL to express *asynchronous hyperproperties*, which are hyperproperties where the comparison points between different executions may be arbitrarily far. Additionally, our results illustrate the struggle to define truly declarative security specifications, i.e., that are not dependent on the system to verify.

Information-flow Verification

In the final part of this thesis, we delve into the problem of verifying information-flow policies. In particular, we look into model-checking asynchronous hyperproperties. We are interested in asynchronous hyperproperties because they are less sensitive to small changes in how the system runs or is observed, offering a more general specification of our intended properties.

Verifying asynchronous hyperproperties is notoriously challenging, with many formalisms introduced recently to address this problem [GMO21, BPS21, BCB+21, BFFM23] having a highly undecidable model-checking. The undecidability stems from the analysis of asynchronous hyperproperties having two different ways to diverge: the analysis may be unbounded due to an infinite number of traces to consider or the distance between the parts to be compared in different traces may be arbitrarily far. While the formalisms introduced recently achieved decidability by introducing fragments limiting one of these two directions, we propose a novel approach to this problem. In this part, we introduce *hypernode automata* that mix automata and logic, where the automata structure specifies synchronization points between fully asynchronous state specifications. We present a novel model-checking algorithm for hypernode automata, effectively introducing the first formalism with a decidable model-checking problem for all asynchronous hyperproperties specified with it.

1.1 Organization and Contributions

The first part of this thesis, titled *Information-flow Design*, has two chapters. The first chapter, *Chapter 2*, gives an overview of the contract-based design approach, presenting its historical development and relevant state-of-art. In this introductory view, we dedicate a section to present interface theories in detail, as they are the contract-based formalism we are interested in. In the following chapter, *Chapter 3*, we present information-flow interfaces introduced in the joint collaboration below:

> [BFH+22b] Ezio Bartocci, Thomas Ferrère, Thomas A. Henzinger, Dejan Nickovic, and Ana Oliveira da Costa. Information-flow interfaces. In *International Conference on Fundamental Approaches to Software Engineering (FASE)*, volume 13241 of *LNCS*, pages 3–22, 2022.

This publication was awarded the *EASST best paper award* at ETAPS (the European Joint Conferences on Theory and Practice of Software) in 2022. An extended version of this work was later published in the journal *Formal Methods in System Design*[BFH+24].

The main contribution in this part is the novel interface theory supporting the specification of information-flow requirements, paving the way for a formal method approach to safety and security co-engineering.

The second part of the thesis, titled *Information-flow Specification*, has three chapters. The first two chapters include preliminaries related to information-flow policies and their instantiation with security properties. Chapter 4 introduces trace properties and explains how they differ from hyperproperties. This chapter introduces the most successful logics to express linear-time trace properties and hyperproperties: LTL and HyperLTL. We follow this with Chapter 5, where we introduce information-flow policies and different information-flow properties to instantiate the policies. For the information-flow properties introduced, we specify some variants with HyperLTL and set comprehension together with first-order logic formulas. In the last chapter of this part, Chapter 6, we present the results in the joint work below:

> [BFH+22a] 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, *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 1–19. Springer International Publishing, 2022.

This joint work, published as an invited paper, studies the expressiveness boundaries of the most successful logic to specify hyperproperties, HyperLTL. This chapter proves that HyperLTL can not express hyperproperties with simple sources of asynchronicity (e.g., just an asynchronous state specification change). In our proofs, we introduce general lemmas and techniques that can be used to prove that HyperLTL cannot express other properties.

In the last part of this thesis, titled *Information-flow Verification*, we present hypernode automata, a new formalism for asynchronous hyperproperties introduced in the following joint work:

> [BHNOdC23] Ezio Bartocci, Thomas A. Henzinger, Dejan Nickovic, and Ana Oliveira da Costa. Hypernode Automata. In Guillermo A. Pérez and Jean-François Raskin, editors, *International Conference on Concurrency Theory (CONCUR)*, volume 279 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 21:1–21:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023.

Hypernode automata is the first formalism for asynchronous hyperproperties mixing an asynchronous logic with an automaton specifying synchronous specification state changes. By taking advantage of the underlying synchronization provided by the automata, we present an effective algorithm to model-check sets of traces generated by Kripke structures. Hypernode automata is the first formalism for asynchronous hyperproperties with a decidable model-checking problem. In contrast, the other formalisms for asynchronous hyperproperties in the literature have a highly undecidable model-checking problem, with some fragments identified with decidable model-checking.

Part I

Information-flow Design

CHAPTER 2

Contract-based Design

This chapter introduces an overview of contract-based design, motivated by challenges from designing cyber-physical systems (CPS). Two concepts prevail in different software engineering approaches to address complex systems' designs: *abstract/refinement* to express hierarchies and *composition/decomposition* to reason about systems and their subsystems. We focus our overview on formalisms that address these foundational concepts to support a modular and hierarchical design of systems.

We start by motivating and giving a historical overview of the development of the contract-based design paradigm. We focus on dynamic reactive systems, like CPS, and, from a theoretical point of view, we limit ourselves to a discrete-time view of time progression. We then introduce the main concepts related to *interface theories*.

2.1 Contracts for Systems Design

Consider, for instance, CPS that integrate computation, networking and physical processes and naturally lead to systems which combine multiple heterogeneous sub-systems[Lee08]. Approaches like layered design and component based-design, used in the automobile and avionics industry, are employed to cope with the intricacies of CPS as systems of systems[1]. Common to all these methods is the need to split concerns clearly (between different teams or companies) during design time to allow distributing, parallelizing and re-using implementations and designs. This split can be formalized as a *contract* that specifies what is expected from each design component (its *guarantee*) and, at the same time, it states what each component needs from its environment to meet its requirements (its *assumption*). In Figure 2.1, we depict a timeline of foundational work on contract-based reasoning in different communities that we elaborate on next.

[1]Surveys on approaches and techniques developed to address the increasing complexity of systems (in particular, CPS) can be found in the survey by Sangiovanni-Vincentelli et al. [SVDP], and in the monograph about contract-based design by Benviste et al. [BCN+18].

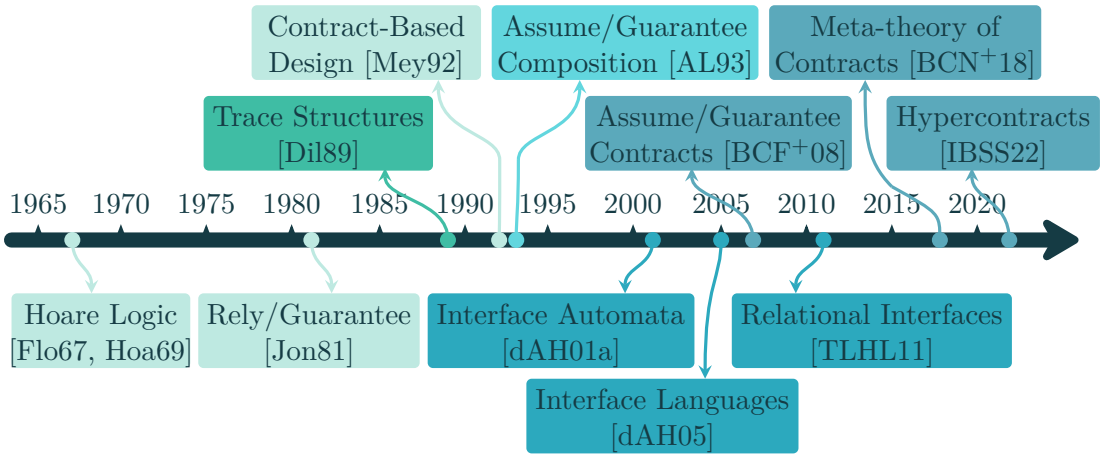


Figure 2.1: Timeline of foundational results in contract-based verification and design.

Side note: Graph vs Game view on open systems

While designing a complex system, it is necessary to consider how its different parts interact to reach a common goal. We work then with *open systems*, where their output behavior (implementations) is influenced by their inputs (environment). There are two natural views on the role of inputs and outputs in formalisms to verify or design systems: the graph and the game view [CDAHM02]. The graph view treats inputs and outputs equally, while they assume opposite polarities under the game view (environment and implementations are interpreted as the opposing Input and Output players [dA03]). This distinction results in different interpretations of refinement and composition.

Refinement: In the graph view, refinement corresponds to behavior containment, usually defined as trace inclusion or simulation. If we treat inputs and outputs equally under this refinement interpretation, a refined system can refuse inputs accepted by the original specification. Then, the specification can not be safely substituted by the refined system.

The game view adopts an alternating view of refinement, usually defined as alternating trace refinement or alternating simulation [AHKV98], to support safe substitutability under refinement. We say that a system P alternate refines a system Q if P accepts at least the same inputs as Q and, for all inputs common to P and Q, the output behavior of P under these inputs is a subset of the output behaviors of Q. Then, Q accepts a subset of the environments of P, while P produces a subset of the behaviors of Q.

Composition: The graph view interprets composition as conjunction because there is no distinction between inputs and outputs. This definition fails, however,

to capture the causality between input and output actions. Formalisms, like Input/Output (I/O) automata [LT87], capture this causality but, in turn, require systems to be *input-enabled*, i.e., they must be able to accept all possible inputs from the environment.

Game-based formalisms, like interface theories [dAH01b], introduced the notion of compatibility for composition. Two systems are not required to be able to work together under all environments, but instead, it suffices that there exists one environment in which they can collaborate. Compatibility is then translated to the following game problem: *Is there a winning strategy for the Input player that guarantees that no (local) incompatibilities between the two systems are reached?*. The set of such strategies defines the environment assumption of the composite.

Programs. The use of contracts in software development dates back to *Hoare logic* [Flo67, Hoa69] by Floyd and Hoare, which formalizes such a split of concerns (program vs environment) for sequential imperative programs with support for refinement and composition of programs. The main building block of Hoare logic is the Hoare triple $\{P\}C\{Q\}$, which defines a contract where Q (called *post-condition*) specifies the outcome of a command C given that assumption P (called *pre-condition*) holds. The software engineering technique of *design-by-contract*, popularized by Bertrand Meyer [Mey92, Mey09], extends this idea by requiring software designers to specify interfaces for classes in object-oriented programming with verifiable pre-conditions, post-conditions and invariants. Pre- and post-conditions are predicates for the methods in the class, while invariants are predicates that must hold in all class states. Class inheritance is defined in terms of refinement, where a subclass can weaken pre-conditions or strengthen post-conditions or invariants.

For concurrent programs, it is necessary to consider also interference through shared variables, i.e. when a program and its environment can update the same state. Jones in [Jon81] proposes to add rely- and guarantee-conditions to the specification of concurrent programs to handle requirements on concurrent state updates. A contract is then represented by a quadruple $(pre, rely, gar, post)$. Elements *pre* and *post* are pre- and post-conditions as defined previously by Hoare triples, i.e., they are concerned with the program outcome. The *rely* specifies how much interference on the shared variables the program can tolerate and still satisfy the functional contract specified by *pre* and *post*. While *gar* specifies how the program influences shared variables. A recent overview of the subsequent development of *Rely/Guarantee* approach can be found in [JHC15].

Trace Structures. In [Hoa80], Hoare proposes to define the behavior of a concurrent process as the set of all traces describing its communication with other processes. Later, in [Dil89], Dill extends this idea to design and verify asynchronous circuits from the observation that sequences of input and output wires describe a circuit's behavior. Dill then introduces *trace structures* as pairs of trace sets: a *success* set with all traces

that describe the correct circuit behavior; and a *failure* set identifying undesirable (but possible) circuit behaviors. The failure set defines the complement of assumptions on the environment. Note that the failure set does not restrict the circuit environment, i.e., trace structures are input-enabled. Dill defines a composition operator and a conformation relation (akin to our notion of refinement) for trace structures to support the hierarchical specification and verification of asynchronous circuits. The conformation relation identifies circuits that can safely be used instead of a given circuit. It requires the conforming circuit not to introduce new failures, i.e., it should work correctly in at least the same environments as the circuit it will substitute. Then, the conforming circuit can weaken its assumptions on the environment (fewer failures) or strengthen its guarantees (fewer successes). This work is later extended to synchronous circuits by Wolf, in [Wol96].

Transition Systems. For reactive systems interpreted over a discrete-time abstraction, transition systems are a natural choice to model and reason about them. Abadi, Lamport, and Wolper in [ALW89] were the first to introduce processes specified by a pair of transition systems (P_t, P_i) where P_t is a finite-state transition system, and P_i defines an infinitary restriction on the process communication actions (which can act as an assumption on the process environment). The behavior of a process is the intersection of the languages defined by the two machines in its definition. The authors define a refinement relation between such processes in the same paper, where a process P' refines a process P iff P' has the same or fewer behaviors with the same or fewer failures than P.

In [AL93], Abadi and Lamport propose an Assume/Guarantee specification theory where they introduce a *composition principle* for modular specifications, depicted below[2]. A system specification is given by a pair (E, M) requiring that if the assumption E on the environment holds, then the system guarantees M, formally $E \rightarrow M$. The composition principle establishes that a system specified by (E, M) abstracts the composition of two processes (E_1, M_1) and (E_2, M_2), if when its assumption is composed with each of the sub-process guarantees (for example, $E \cap M_1$), then it defines an environment allowed by the other sub-process ($E \cap M_1 \subseteq M_2$). They prove that the proof rule below is sound for assumptions that are safety properties. For this reason, they can strengthen the rule to consider guarantees closure, denoted by $[M]$, which is the smallest safety property containing M.

$$\frac{E \cap [M_1] \cap [M_2] \;\subseteq\; E_1 \cap E_2}{(E_1, M_1) \cap (E_2, M_2) \;\subseteq\; (E, (M_1 \cap M_2))} \text{ Composition Principle [AL93]}$$

Interface Automata. Lynch and Turtle introduce, in [LT87], *Input/Output (I/O) automata* to model asynchronous concurrent systems. An I/O automaton is a state machine with transitions labeled as either input, output or hidden. The labeling establishes the action ownership: input actions are generated by the automaton environment, while the

[2]For simplicity, we assume specifications to be realizable and omit realizability concerns from the rule statement.

automaton controls output and hidden actions. Interface automata (IA), introduced by de Alfaro and Henzinger in [dAH01a], adopt the syntax of I/O automaton but change its semantics. While I/O automata are assumed to be input-enabled, interface automata assume the environment to be helpful, which results in different notions of composition and refinement between these two formalisms. The main features of IA are (i) their optimistic view on composition, where two interfaces are compatible if there exists an environment where they can work together; and (ii) the alternating approach to refinement, where more detailed interfaces can be more restrictive on outputs but must not be more restrictive on inputs. IA was later enriched to incorporate extra-functional requirements like time constraints [dAHS02, DLL+10], resources management [CdAHS03], modalities [LNW07] and non-interference requirements [LD10b, LD10c]. We discuss the development of modal interface automata and interface automata for structure and security next.

Larsen, Nyman, and Wasowski introduce in [LNW07] modal I/O automata, which extend Modal Transition Systems (MTS) with a distinction between input and output actions. Modal theories define the modality of actions, i.e., whether they are required (*must*) or allowed (*may*). Later, in [LV12], Gerald Lüttgen and Walter Vogler define Modal Interface Automata (MIA). They specify first disjunctive Modal Transition Systems (dMTS) as an MTS with disjunctive *must* transitions. MIA is then defined as a dMTS with deterministic inputs transitions and the requirement that every *input-may* transition is an *input-must* transition, as well. This last constraint is necessary to guarantee that refinement defines a precongruence for composition, i.e., refinement is preserved under composition.

Interface for structure and security (ISS) [LD10b, LD10c] introduced by Lee and D'Argenio, extends IA by annotating visible actions as either public (low confidentiality) or private (high confidentiality). They then define that two ISS automata are compatible for composition if their composite satisfies the intended non-interference property. They explore two different notions of non-interference. In [LD10b], they use a bisimulation-based view of non-interference that requires the system to behave the same way when high actions are performed or hidden. Alternatively, in [LD10c], they introduce a non-interference definition based on alternating refinement, requiring that input restriction does not reveal information about high-level behavior.

Interface Languages. In [dAH01b], de Alfaro and Henzinger formalize the distinction between interface and component models, laying the groundwork for formal definitions of interface models, referred to as *interface theories*. The main difference between components and interfaces is their view of environments. Component theories adopt a pessimistic view in which all environments are allowed. In contrast, interface theories follow an optimistic view in that two interfaces are only compatible for composition if they can be part of an environment that satisfies both interface assumptions. Later, in [dAH05], the same authors define *interface languages* as a formalism that satisfies the incremental design and the independent implementability of systems. We will explore these notions more in-depth in section 2.2.

Tripakis et al. introduce *relational interfaces*, in [TLHL11], as tuples (X, Y, f) where X and Y are disjoint sets of input and output variables, and f is a contract defined as a prefix-closed set of finite sequences of input and output assignments. Then, as RI do not have assumptions and guarantees defined separately, they can explicitly relate input and output interaction in a system execution (but not across multiple executions).

Contract theories. Benviste et al. present, in [BCF+08], a contract-based formalism to support the *speculative design* of systems, where different parts of the system can be independently and concurrently designed. They introduce Assume/Guarantee (A/G) contracts, which in addition to refinement and composition, can handle components' multiple viewpoints (functional and non-functional specification) by defining new operators to fuse their different dimensions. Assume/Guarantee contracts approach the design of correct systems from a semantic perspective: they work directly with the objects we intend to design, which we refer to as *components*. Interfaces theories, on the other hand, focus on the syntactic aspect, i.e., they define languages to specify and relate the objects we are interested in during the design process.

Example: Assume/Guarantee Contracts

We present a simplified view on the A/G contract theory introduced in [BCF+08]. Contracts are defined around the notion of a *component*, which represents a system design unit. Components are characterized by a set of variables[a] equipped with a domain. A component implementation is then defined by a set of traces describing its execution runs. Components are connected by their common variables that have the same value. Then, composition is reduced to set intersection.

An *A/G-contract* is defined by a tuple $\mathcal{C} = (A, G)$ where A is a set of traces specifying its assumptions on the environment, and G is a set of traces specifying its guarantees. An implementation of a component, M, satisfies the contract $\mathcal{C}$, denoted $M \models \mathcal{C}$, if it meets the guarantees under permitted environments. In this setting, satisfaction reduces to trace containment, then formally $M \models (A, G)$ iff $A \cap M \subseteq G$. An A/G-contract $\mathcal{C} = (A, G)$ is saturated iff $G = G \cup \overline{A}$, where $\overline{A}$ is the complement of A. Note that we can interpret $G \cup \neg A$ as an implication, i.e., $A \rightarrow G$, which captures or intended meaning for the guaranteed behavior of a component (*behavior described by G holds if the assumption A is satisfied*). From now on, we work only with saturated contracts. and refer to the following contracts $\mathcal{C} = (A, G)$ and $\mathcal{C}' = (A', G')$.

Contracts *refinement* is defined as an alternating refinement. Formally, a contract $\mathcal{C}'$ refines $\mathcal{C}$, denoted $\mathcal{C}' \leq \mathcal{C}$, iff $A' \supseteq A$ (more environments) and $G' \subseteq G$ (less implementations). Refinement then induces an order on contracts with greatest-lower bound (GLB) and least-upper bounds (LUB) defining *conjunction* ($\sqcap$) and *disjunction* ($\sqcup$) of contracts, respectively, as follows: $\mathcal{C} \sqcap \mathcal{C}' = (A \cup A', G \cap G')$ and

$\mathcal{C} \sqcup \mathcal{C}' = (A \cap A', G \cup G')$. The conjunction operator can be used to aggregate in a contract its multiple-view points.

The guarantee of the composition of contracts $\mathcal{C}$ and $\mathcal{C}'$ is naturally defined by the composition of their guarantees, i.e., $G \cap G'$. For the composite assumptions, we observe that for each contract in the composition, its guarantee partially defines the environment for the other contracts being composed. So, we need to evaluate what assumptions are missing to be satisfied. Giving two contracts $\mathcal{C}$ and $\mathcal{C}'$ their composition, denoted $\mathcal{C} \otimes \mathcal{C}'$, is the contract $(A_\otimes, G_\otimes)$ where: $A_\otimes = \overline{(G \cap G')} \cup (A \cap A')$ and $G_\otimes = G \cap G'$.

[a] A component is usually defined with a set of variables and a set of ports. For simplicity, we only refer to variables and omit ports.

In [BDH+12], Bauer et al. show how to derive a contract theory from a specification theory of component behaviors. Given a class $\mathbb{S}$ of specifications, a specification theory is defined by a triple $(\mathbb{S}, \otimes, \leq)$ where $\otimes$ is a composition operator and $\leq$ is a refinement relation that defines a pre-order over the specifications. Additionally, to support compositionality and the independent design of systems, the composition operator must preserve the refinement and be a commutative and associative function over the equivalence relation defined by the refinement. Given such a specification theory, the authors show how to lift the composition and refinement operators to their counterparts in a contract theory.

Later, Benviste et al. [BCN+18] present a *meta-theory for the contract-based approach to design systems*, which builds from component algebras. The meta-theory presents a general characterization of concepts like refinement, implementations, composition, and conjunction. A drawback of this approach is that, due to its generality, it builds from unstructured sets of components, leading to too abstract definitions at times.

Incer et al. introduce in [IBSS22] *hypercontracts* as an A/G contract theory built from lattices of component sets equipped with composition and refinement. As their building blocks are structured, they can exploit it to introduce more concrete definitions of the usual operators and relations in a contract theory. Moreover, contrary to the A/G contracts by Benviste et al. [BCF+08], they support the specification of assumptions and guarantees as hyperproperties by using the lattice induced by subset inclusion over components that are trace sets.

2.2 Interface Theory

Interface theories formalize the hierarchical component-based approach for systems' design [dAH01b]. In what follows, the elements in the system design are referred to as interfaces and their respective implementations as components. Additionally, we refer to the environment part of an interface specification as the interface *assumption* and the implementation part as the interface *guarantee*. In a nutshell, an interface theory

provides relations and operators to address the following questions of component-based design [CDAHM02]:

Well-formedness Is there an environment compatible with a given interface?

Compatibility Can two interfaces interact successfully?

Verification Do all interface possible implementations satisfy a given property?

Refinement Can an interface substitute another without violating prior compatibility or verification results?

Central to the development of interface theories is the assumption that the environment is helpful, as it will be composed of components willing to meet each other's assumptions. For this reason, interfaces do not constrain the environment. An interface is *well-formed* if its specification (assumptions together with guarantees) is satisfiable. Then, an interface well-formedness is a conjunction between their environment and implementation constraints instead of an implication. Moreover, well-formedness is an *input-existential* property: it suffices that an environment exists that satisfies a given interface assumption.

Interfaces theories adopt the game view on the design of systems. Then, interface refinement is defined as alternating refinement: while refining an interface, we can weaken its assumptions or strengthen its guarantees.

Example: Assume/Guarantee Well-formedness and Refinement

Assume/Guarantee (A/G) interfaces, introduced in [dAH01b], define a simple interface theory with a strong separation between environment and implementations responsibilities. An *A/G interface* is defined by a tuple (X, Y, A, G) where X and Y are two disjoint sets of input and output variables, respectively; A is a predicate over input variables, called *assumption*; and G is a predicate over output variables, called *guarantee*.

An A/G interface is well-formed iff the first-order formula $\exists X \ A \wedge G$ holds. Due to assumptions and guarantees not sharing variables, A/G well-formedness is equivalent to requiring assumptions and guarantees to be satisfiable predicates. We depict, in Figure 2.2, and A/G interface F with the assumption that its input variable x is a natural number and its output variable y is divisible by 4.

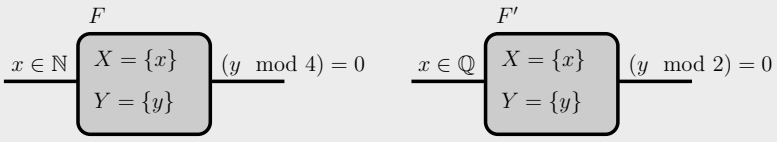


Figure 2.2: Example of A/G interfaces.

An A/G interface $F' = (X, Y, A', G')$ refines an interface $F = (X, Y, A, G)$, denoted

$F' \preceq F$, iff $(A_F \rightarrow A_{F'}) \wedge (G_{F'} \rightarrow G_F)$ holds. The interface F', in Figure 2.2, is an example of an interface that refines F in the same figure, i.e., $F' \preceq F$. As all natural numbers are also rational $(x \in \mathbb{N} \rightarrow x \in \mathbb{Q})$, then F' is compatible with all environments compatible with F. From the guarantee side, all even numbers are also divisible by 4 $((y \mod 2) = 0 \rightarrow (y \mod 4) = 0)$, which means that F' guarantees at least the same output behaviors of F.

Interface theories are equipped with a compatibility predicate to check whether the composed interfaces satisfy each other assumptions, and the resulting composite defines a well-formed interface. Compatibility check reduces to solving a game between an Input player playing to avoid incompatibilities between the interfaces being composed (i.e., avoiding violating assumptions) and an Output player, which can play any move within interfaces' guarantees. Then, two interfaces are compatible if the Input player has a winning strategy over this game, i.e., an environment must exist that works for all compositions of the two interfaces' implementations. The set of all such strategies defines the assumption of the interfaces' composite.

Example: Assume/Guarantee Compatibility and Composition

The first check we do between interfaces we want to compose is whether they are *composable*, which requires their output variables to be disjointed. We then proceed to make a semantic check: two interfaces are *compatible* whenever one provides outputs to the other, then they must satisfy the assumptions related to that output. Then, two A/G interfaces $F = (X, Y, A, G)$ and $F' = (X', Y', A', G')$ are compatible iff the following formula holds:

$$\varphi_{F,F'} \stackrel{\text{def}}{=} \forall Y, Y' : (G \wedge G') \rightarrow (A \wedge A').$$

Note that all free variables in this formula are input variables. These are the variables that are still available for future compositions. For this reason, the compatibility formula $\varphi_{F,F'}$ defines the assumption of the composition of F with F'. Then, the composition of F and F' defines the tuple $F \otimes F' = (X_\otimes, Y_\otimes, A_\otimes, G_\otimes)$ where $Y_\otimes = Y \cup Y'$, $X_\otimes = (X \cup X') \setminus Y_\otimes$, $A_\otimes = \varphi_{F,F'}$ and $G = G \wedge G'$.

Once we define refinement, compatibility and composition, we want to be sure that the defined interface theory supports a system's modular design and verification, i.e., to define an interface language. In [dAH05], de Alfaro and Henzinger propose that an interface theory must satisfy *incremental design* and *independent implementability* and to be an interface language. Independent implementability requires that compatible interfaces can be refined independently without becoming incompatible. Incremental design states that the compatibility of partially defined systems can be evaluated without further

knowledge about the complete system. Formally, for an interface theory defined by a tuple $(\mathbb{I}, \otimes, \sim, \preceq)$, where $\mathbb{I}$ is a set of interfaces, $\otimes$ is composition, $\sim$ is a compatibility relation and $\preceq$ is a refinement relation, then it satisfies:

Incremental Design: for all interfaces F, H and J in $\mathbb{I}$, if $F \sim H$ and $(F \otimes H) \sim J$, then $H \sim J$ and $F \sim (H \otimes J)$;

Independent Implementability: for all interfaces F, F' and H in $\mathbb{I}$, if $F \sim H$ and $F' \preceq F$, then $F' \sim H$ and $F' \otimes H \preceq F \otimes H$.

Example: Incremental Design and Independent Implementability

We illustrate the principles of incremental design and independent implementability with an abstract representation of a system design in 2.3. The interface F specifies the closed system we want to design and is refined as a composition between two interfaces: F_1 and F_2.

By the independent implementability principle, F_1 and F_2 can be refined independently (possibly by different teams or providers) while guaranteeing that their composition remains a refinement of F (i.e., satisfies the original requirements). In our example, F_1 is refined by F_1', defined by the composition of three other interfaces, and F_2 is refined by F_2', defined as the composition of two interfaces. Independent implementability specifies a top-down compositionality principle.

Now consider F_1', by the incremental design principle, we know that compatibility between F_{11} and F_{12} is independent of F_{13}. Incremental design allows to work with incomplete system specifications.

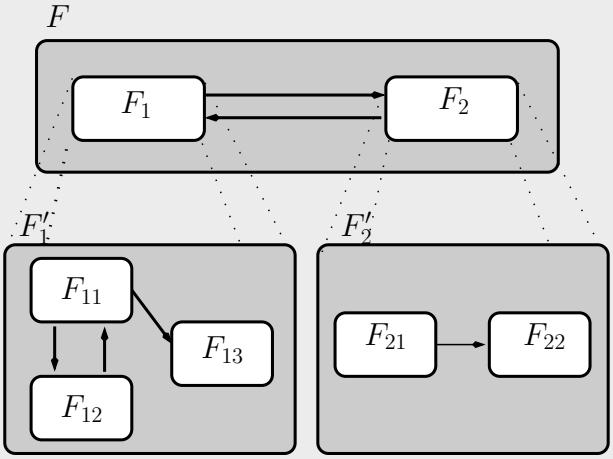


Figure 2.3: Modular design.

CHAPTER 3

Information-flow Interfaces

This chapter presents an interface theory for the structural view of information flow. Our focus is on the reachability of information within a system abstracting away from concrete trace semantics, which allows us to identify difficulties transversal to the design of information flow policies. This framework supports both the top-down design and the bottom-up verification of information-flow policies. While this view is orthogonal to work on information flow control, it can be used as a divide-and-conquer procedure for organizing information flow control tasks.

We start this chapter by introducing stateless information-flow interfaces and proving that they defined an interface language. The section 3.2 extends our theory to handle stateful specifications. In it, we introduce stateful information-flow interfaces, which are transition systems with nodes labeled with stateless interfaces, and prove that they also define an interface language.

This chapter is based on a collaboration with Ezio Bartocci, Thomas Ferrère, Thomas Henzinger and Dejan Nickovic. From this collaboration, we report here results published in the conference paper:

> [BFH+22b] Ezio Bartocci, Thomas Ferrère, Thomas A. Henzinger, Dejan Nickovic, and Ana Oliveira da Costa. Information-flow interfaces. In *International Conference on Fundamental Approaches to Software Engineering (FASE)*, volume 13241 of *LNCS*, pages 3–22, 2022.

which was later extended to a journal version:

> [BFH+24] Ezio Bartocci, Thomas Ferrère, Thomas A. Henzinger, Dejan Nickovic, and Ana Oliveira da Costa. Information-flow interfaces. *Formal Methods in System Design*, 2024.

.

Example: Integrity of a Shared Communication Infrastructure (SCI)

We will showcase some concepts of our interface theory with a running example from the automotive industry adapted from an industrial case study by Marcus Mikulcak et al. [MHGG19]. This example introduces a stepwise design of a shared communication infrastructure (a bus) from distance warners and a wheel sensor to the braking system and the odometer. Our design goal is to guarantee the integrity of a communication channel that performs a safety-critical functionality. The integrity of sensitive information is one of the main goals of information security (for more details, see Section 5.1).

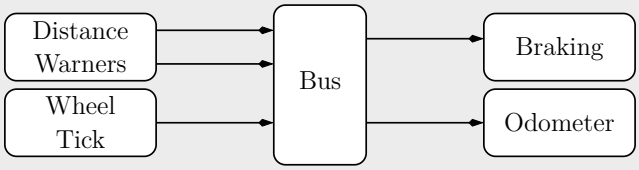


Figure 3.1: High-level view of SCI architecture.

In this example, the wheel sensors (which keep track of the wheel rotations) use the shared bus to communicate with the car's odometer. The bus is also used by two distance warners, at the front and back of the car, to send their data to the braking system. A distance warner is responsible for sensing and analyzing the proximity of the car to other objects. The communication channel to the braking system must have high integrity since it carries out safety-critical information. Hence, the distance warners and the braking system communication are classified with high integrity, while the communication between the wheel sensor and the odometer is of low integrity. We want to guarantee that data sent by the wheel sensor does not interfere with the high-integrity channel, which could lead to distance warnings sent to the braking system being delayed or lost. The main goal of our design process is to guarantee that the closed system requirement that *information from the wheel sensor does not flow to the braking system* is propagated accordingly to subsystems through successive steps of decomposition and refinement.

While in this example, we focus on the top-down design of our system, we could use our approach to do bottom-up verification. In the bottom-up direction, we start with several components (which could be implementations or mathematical models as done in [MHGG19]), and proceed by successive steps of composition and abstraction to verify the same property over the closed system.

3.1 Stateless

We introduce a stateless interface theory to reason about secure information flow. By stateless, we mean that the specification does not change over time, i.e., it remains the same during the system's execution. The interface theory we propose focuses on the

structural aspect of information flow. In other words, it captures requirements about *where* information can flow rather than specifying *what* is a flow of information.

We abstract flows in systems with *flow relations* that are reflexive and transitively closed relations. A system (or subsystem) is then represented by an *information-flow component* that includes the corresponding system's flow relation. An *information-flow interface* specifies forbidden information flows in a (possibly open) system. As for classic interface theory, our interfaces specify a no-flow *assumption* on the environment and a no-flow *guarantee* for its implementations.

As flows are transitive, an interaction between flows from an interface environment and one of its implementations can define flows outside the scope of the interface's assumption and guarantee. For this reason, we introduce a third no-flow requirement over all the closed systems defined by a composition of one of the interface's environments and implementations. We refer to it as *closed-guarantee* and to the guarantees local to the system's implementations as *open-guarantee*. Each no-flow in an interface closed guarantee can be supported by a combination of no-flows from the interface's assumption and open guarantee. The main advantage of having an explicit requirement over the closed systems is that the designer can decide how to split the responsibilities between the environment and the implementations to support it. We use closed guarantees to determine which interfaces are well-formed (formalized later in Definition 3.1.4), i.e., what interfaces can be independently specified and implemented while satisfying the closed systems' no-flow requirements.

Definition 3.1.1. *Let X and Y be disjoint sets of* input *and* output *variables, respectively, with $Z = X \cup Y$ the set of all variables. A relation $\mathcal{M} \subseteq Z \times Y$ is a* flow relation *iff it is a transitive relation, and reflexive over $Y \times Y$. A* stateless information-flow component *is a tuple $(X, Y, \mathcal{M})$ where $\mathcal{M} \subseteq Z \times Y$ is a flow relation, called* flows. *A* stateless information-flow interface *is a tuple $(X, Y, \mathcal{A}, \mathcal{G}, \mathcal{P})$ where $\mathcal{A} \subseteq Z \times X$ is a relation, called* assumption*; $\mathcal{G} \subseteq Z \times Y$ is a relation, called* open-guarantee*; and $\mathcal{P} \subseteq Z \times Y$ is a relation, called* closed-guarantee*.*

Example: SCI assumption and guarantees

In Figure 3.2, we show an interface named *Bus′* specifying no-flow requirements for the shared communication infrastructure. This interface has three input variables: two for the data received (source) from the distance warners – *distw_f_s* and *distw_b_s* – and one for the wheel sensor – *wheel_tick*. It also has three output variables: two for the target of the distance warners data – *distw_f_t* and *distw_b_t* – and one for the odomoter – *odometer*. The *Bus′* assumption requires no-flow from *wheel_tick* and *odometer* to the input ports for distance warners source data. Additionally, it is required as both an open and a closed guarantee that there are no flows from *wheel_tick* to the output ports for the distance warners data.

The same figure presents the key for the graphical representation of both interfaces and components. No-flow requirements on the open system are depicted with dashed arrows, while closed system no-flows are represented as dotted arrows. To improve the readability of the drawings, it is implicit that for each drawn closed-guarantee, we have the same open-guarantee over the open system. Additionally, for components, we do not draw arrows in the reflexive and transitive closure of the arrows depicted. We may omit variable(s) names when clear from the context.

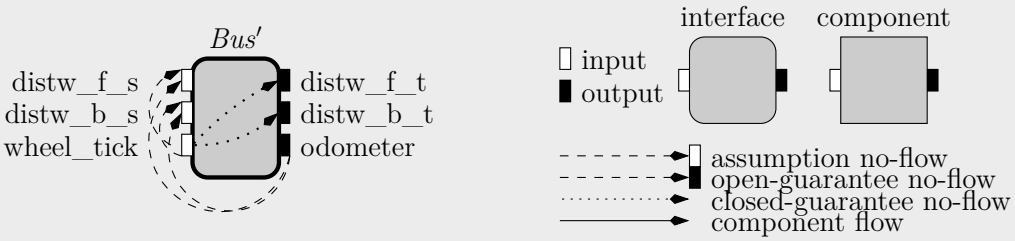


Figure 3.2: Example of an interface with the key for the graphical representation of information-flow interfaces and components.

The *composition between flow relations* is defined straightforwardly: given two flow relations $\mathcal{M}$ and $\mathcal{M}'$, their composition is the transitive closure of all their flows, formally, $(\mathcal{M} \cup \mathcal{M}')^*$. We say that a flow relation $\mathcal{M}$ *includes only flows allowed by a relation* $\mathcal{N}$ when they define disjoint sets, $\mathcal{M} \cap \mathcal{N} = \emptyset$. Given an interface, a component *implements* it when the interface's open-guarantee allows all of its flows; and a component is one of its *permissible environments* when it includes only flows allowed by the interface's assumption. Implementations of an interface F have the same inputs and outputs as F, while permissible environments have the same sets but with switched roles.

Definition 3.1.2. *Let $F = (X, Y, \mathcal{A}, \mathcal{G}, \mathcal{P})$ be an information-flow interface. A component $f_\mathcal{E} = (Y, X, \mathcal{E})$ is an* environment *of F. We say that $f_\mathcal{E}$ is a* permissible environment of *F, denoted $f_\mathcal{E} \models_\mathcal{A} F$, iff $\mathcal{E} \subseteq \overline{\mathcal{A}}$, where $\overline{\mathcal{A}} = (Z \times X) \setminus \mathcal{A}$. A component $f = (X, Y, \mathcal{M})$* implements *the interface F, denoted $f \models_\mathcal{G} F$, iff $\mathcal{M} \subseteq \overline{\mathcal{G}}$, where $\overline{\mathcal{G}} = (Z \times Y) \setminus \mathcal{G}$.*

As we explained intuitively before, closed-guarantees do not play a role in defining implementations and permissible environments. They will be, however, essential to define well-formed interfaces, which we will explain next.

Example: SCI implementations and environments

In Figure 3.3, we present a different interface from our previous example for the shared communication channel, called *Bus*. This interface requires only, as both open- and closed- guarantee, that information of the wheel sensor does not flow to the distance warners target.

The component *sending* (bottom left) specify a permissible environment for the *Bus* interface. Note, however, that the *sending* component is not a permissible environment for the *Bus*′ interface in the previous example because the flow from the wheel sensor to the source for the front distance warner is in the interface assumption. The *bus* component (bottom right) is an implementation of *Bus* because it has only a flow from *distw_f_s* to *distw_f_t*, which is not in the open-guarantee of the *Bus* interface.

As *Bus* has no assumptions, all environments are permissible. The composition of the flow relations from *sending* and *bus* includes the pair $(wheel\_tick, distw\_f\_t)$ because there is information flow from *wheel_tick* to *distw_f_s* through the environment and then to *distw_f_t* through the implementation. This flow, from *wheel_tick* to *distw_f_t*, is forbidden by the closed guarantees. We will see next that the *Bus* interface is an ill-formed interface: when we independently implement its environments and implementations, we may inadvertently violate a requirement over the closed system.

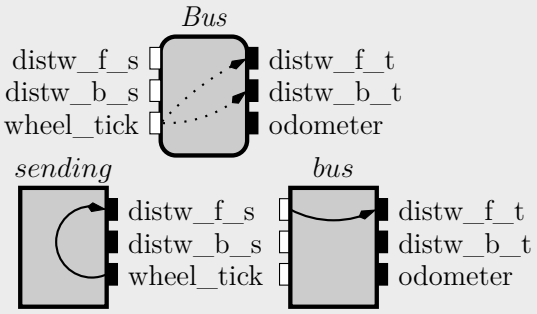


Figure 3.3: Interface *Bus* implementation – *bus* – with one of its permissible environments – *sending*.

An interface must define at least one implementation and one permissible environment to avoid vacuous solutions. Then, as flows are reflexive, assumptions and guarantees must be specified by irreflexive relations, which we will refer to as *no-flow relations*. A well-formed interface ensures, additionally, that its *closed-guarantee is consistent* with its open-guarantee and assumption, i.e., the composition of any permissible environment with any implementation does not include a flow forbidden by the closed-guarantee. To formalize this notion of consistency, we need first to define *composition of no-flow relations*, which is naturally defined as the union of all compositions between flow relations that only include flows allowed by one of given no-flow relations.

Definition 3.1.3. *The* set of flows defined by the complement-flow composition of relations $\mathcal{N} \subseteq U \times V$ *and* $\mathcal{N}' \subseteq U' \times V'$ *is:*

$$\mathcal{N} \bullet \mathcal{N}' = \{(z, z') \in (\mathcal{M} \cup \mathcal{M}')^* \mid \mathcal{M} \subseteq \overline{\mathcal{N}}, \mathcal{M}' \subseteq \overline{\mathcal{N}'}, \text{ and } \mathcal{M} \text{ and } \mathcal{M}' \text{are flow relations}\}.$$

A well-formed interface is specified only with no-flow relations such that the set of all flows in one of its closed systems (i.e., $\mathcal{A} \bullet \mathcal{G}$) is disjoint from the interface's closed-guarantee.

Definition 3.1.4. *An information-flow interface* $(X, Y, \mathcal{A}, \mathcal{G}, \mathcal{P})$ *is* well-formed *iff* $\mathcal{A}$*,* $\mathcal{G}$ *and* $\mathcal{P}$ *are* no-flow relations *(i.e., irreflexive relations); and the closed-guarantee is consistent with the open-guarantee and assumption, i.e.* $(\mathcal{A} \bullet \mathcal{G}) \cap \mathcal{P} = \emptyset$*.*

We prove below that our definition of interface well-formedness captures the intended relationship between the interface's closed-guarantees with its permissible environments and implementations.

Proposition 3.1.1. *For all well-formed interfaces* $F = (X, Y, \mathcal{A}, \mathcal{G}, \mathcal{P})$*, and for all components* $f = (X, Y, \mathcal{M})$ *and* $f_{\mathcal{E}} = (Y, X, \mathcal{E})$*: if* f *implements* F*,* $f \models_F$*, and* $f_{\mathcal{E}}$ *is an permissible environment of* F*,* $f_{\mathcal{E}} \models_F$*, then their combined flows are consistent with the closed-guarantee of* F*,* $(\mathcal{M} \cup \mathcal{E})^* \cap \mathcal{P} = \emptyset$*.*

Proof. Consider an arbitrary interface F, and components $f = (X, Y, \mathcal{M})$ and $f_{\mathcal{E}} = (Y, X, \mathcal{E})$, s.t.: (i) F is a well-formed interface, (ii)$f \models_F$, and (iii) $f_{\mathcal{E}} \models_F$. Let $(z, z') \in (\mathcal{M} \cup \mathcal{E})^*$. By Definition 3.1.3 and assumptions (ii) and (iii), $(z, z') \in \mathcal{A} \bullet \mathcal{G}$, and by our initial assumption (i), $(z, z') \notin \mathcal{P}$. Hence $(\mathcal{M} \cup \mathcal{E})^* \cap \mathcal{P} = \emptyset$. □

Though intuitive, the definition of no-flows composition does not offer an easy and economical way to compute it. Note that it requires computing the reflexive and transitive closure over all possible flow relations satisfying the no-flow relations being composed. We will now discuss how to characterize the composition of no-flow relations syntactically, i.e., as a regular expression over the no-flow relations being composed. Our first challenge is that the complement of a no-flow relation does not necessarily define a flow relation because it is not necessarily transitively closed. The biggest challenge, however, is that not all no-flow relations have a maximal flow relation that satisfies it. Concretely, for an interface open-guarantee and assumption, there may not exist an implementation or environment that is maximal, i.e., all other implementations or environments are their subsets, respectively.

Example: SCI maximal implementations

In Figure 3.4, we depict two implementations for the interface *Bus* from the previous example, bus_s and bus_t. Their composition does not define an implementation of *Bus* because it includes a flow from *wheel_tick* to *distw_f_t*, violating the *Bus* open-guarantee. Therefore bus_s and bus_t are incomparable (no implementation subsumes both), so there is no maximal implementation of *Bus*.

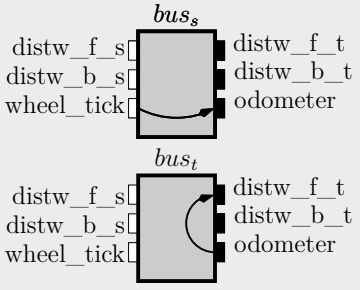


Figure 3.4: Incomparable *Bus*' implementations.

Without maximal flow relations for a given no-flow, we cannot characterize the flows in the closed systems defined by an interface F by the transitive closure of the complement of its assumption and open-guarantee, i.e., flows in the closed system defined by F are not the same as $(\overline{\mathcal{A}} \cup \overline{\mathcal{G}})^*$. This definition would yield more flows than the actual flows of the closed system defined by F. However, we know that all allowed flows of a given no-flow relation are in its complement. Then, we just need to be careful not to compose pairs in the complement of the same no-flow relation while computing the composition of two no-flow relations.

The set with all flows in the closed systems defined by an interface F includes all pairs of variables (z, z') such that there exists a path from z to z' defined by alternating between flows in the complement of the assumption, $\overline{\mathcal{A}}$, and the complement of the open-guarantee, $\overline{\mathcal{G}}$. We formalize this below and prove it precisely characterizes the composition between two no-flow relations. We provide a step-by-step example of how to evaluate no-flow relationship composition when we introduce later composite open-guarantees in Definition 3.1.5.

Lemma 3.1.2. *Let $\mathcal{N} \subseteq U \times V$ and $\mathcal{N}' \subseteq U' \times V'$ be no-flow relations, with V and V' being disjoint sets, $V \cap V' = \emptyset$, and the set of all variables being $Z = U \cup U' \cup V \cup V'$. Then, $\mathcal{N} \bullet \mathcal{N}' = (\mathrm{Id}_Z \cup \overline{\mathcal{N}}) \circ (\overline{\mathcal{N}'} \circ \overline{\mathcal{N}})^* \circ (\mathrm{Id}_{V \cup V'} \cup \overline{\mathcal{N}'})$, where Id_Z is the identity relation over all variables in $\mathcal{N}$ and $\mathcal{N}'$, $\mathrm{Id}_{V \cup V'}$ is the identity relation over $V \cup V'$ and $R \circ R' = \{(z, z'') \mid (z, z') \in R \text{ and } (z', z'') \in R'\}$ is the usual composition between relations.*

Proof. Consider arbitrary no-flow relations $\mathcal{N} \subseteq U \times V$ and $\mathcal{N}' \subseteq U' \times V'$ where V and V' are disjoint sets.

We start by proving that $\mathcal{N} \bullet \mathcal{N}' \subseteq (\mathrm{Id}_Z \cup \overline{\mathcal{N}}) \circ (\overline{\mathcal{N}'} \circ \overline{\mathcal{N}})^* \circ (\mathrm{Id}_{V \cup V'} \cup \overline{\mathcal{N}'})$. Let $(z, z') \in \mathcal{N} \bullet \mathcal{N}'$. Then, there exists two flow relations $\mathcal{M} \subseteq U \times V$ and $\mathcal{M}' \subseteq U' \times V'$ s.t. (i) $\mathcal{M} \subseteq \overline{\mathcal{N}}$, (ii) $\mathcal{M}' \subseteq \overline{\mathcal{N}'}$ and (iii) $(z, z') \in (\mathcal{M} \cup \mathcal{M}')^*$ or, equivalently, $(z, z') \in (\mathrm{Id}_Z \cup \mathcal{M}^+) \circ (\mathcal{M}'^+ \circ \mathcal{M}^+)^* \circ (\mathrm{Id}_{V \cup V'} \cup \mathcal{M}'^+)$. The identity relation in the rightmost side of the expression is defined over the domain $V \cup V'$ (i.e., $\mathrm{Id}_{V \cup V'}$) because $z' \in V \cup V'$. By flow relations being transitively closed, it follows: $(z, z') \in (\mathrm{Id}_Z \cup \mathcal{M}) \circ (\mathcal{M}' \circ \mathcal{M})^* \circ (\mathrm{Id}_{V \cup V'} \cup \mathcal{M}')$. By the initial assumptions (i) and (ii), $(z, z') \in (\mathrm{Id}_Z \cup \overline{\mathcal{N}}) \circ (\overline{\mathcal{N}'} \circ \overline{\mathcal{N}})^* \circ (\mathrm{Id}_{V \cup V'} \cup \overline{\mathcal{N}'})$.

We prove the other direction: $(\mathrm{Id}_Z \cup \overline{\mathcal{N}}) \circ (\overline{\mathcal{N}'} \circ \overline{\mathcal{N}})^* \circ (\mathrm{Id}_{V \cup V'} \cup \overline{\mathcal{N}'}) \subseteq \mathcal{N} \bullet \mathcal{N}'$. We start with the case $\overline{\mathcal{N}'} \circ (\overline{\mathcal{N}} \circ \overline{\mathcal{N}'})^* \subseteq \mathcal{N} \bullet \mathcal{N}'$. We remark that all sequences defined by $\overline{\mathcal{N}'} \circ (\overline{\mathcal{N}} \circ \overline{\mathcal{N}'})^*$ have elements of $\overline{\mathcal{N}'}$ in the odd positions and elements of $\overline{\mathcal{N}}$ in the even positions. We choose this to simplify the presentation of the proof. The other cases can be proved analogously. We prove this case by proving first the stronger property below for all sequences defined by $\overline{\mathcal{N}'} \circ (\overline{\mathcal{N}} \circ \overline{\mathcal{N}'})^*$:

$$(\star) \text{ for all } n \in \mathbb{N} \text{ and all sequences } (z_1, z_2) \cdot (z_2, z_3) \cdot \ldots \cdot (z_n, z_{n+1}) \\ \text{where } (z_{2i-1}, z_{2i}) \in \overline{\mathcal{N}'} \text{ and } (z_{2i}, z_{2i+1}) \in \overline{\mathcal{N}}, \text{ with } 1 \leq i \leq \lceil n/2 \rceil,$$

$$\text{there is } 1 \leq m \leq n \text{ s.t. } z_1 = z_m \text{ and } (z_1, z_{n+1}) \in (\mathcal{M} \cup \mathcal{M}')^*$$
$$\text{for flow relations } \mathcal{M}' = \{(z_j, z_{j+1}) \mid m \leq j \leq n \text{ and } j \text{ is odd}\}^+ \cup \mathrm{Id}_{B'} \text{ and}$$
$$\mathcal{M} = \{(z_j, z_{j+1}) \mid m \leq j \leq n \text{ and } j \text{ is even}\}^+ \cup \mathrm{Id}_B, \text{ with } \mathcal{M}' \subseteq \overline{\mathcal{N}'} \text{ and } \mathcal{M} \subseteq \overline{\mathcal{N}}.$$

The property above tell us that for any alternating sequence between elements in the complement of $\mathcal{N}$ and $\mathcal{N}'$ defining a path from z_1 to z_{n+1}, we can use this sequence to define two flow relations that are allowed by either $\mathcal{N}$ or $\mathcal{N}'$ such that (z_1, z_{n+1}) is in composition of the defined flow relations. From this property, it follows that for all pair of variables $(z, z') \in \overline{\mathcal{N}'} \circ (\overline{\mathcal{N}} \circ \overline{\mathcal{N}'})^*$, we can define two flow relations $\mathcal{M}$ and $\mathcal{M}'$ that witness $(z, z') \in \mathcal{N} \bullet \mathcal{N}'$.

We prove now $(\star)$ by natural induction on the size of the sequences. We start with the *base case*, $n = 1$, i.e., we consider a sequence of the form (z_1, z_2) where $(z_1, z_2) \in \overline{\mathcal{N}'}$. Then, for $k = 1$, we have $(\mathcal{M}' = \{(z_1, z_2)\} \cup \mathrm{Id}_{U'}) \subseteq \overline{\mathcal{N}'}$ and $(\mathcal{M} = \mathrm{Id}_U) \subseteq \overline{\mathcal{N}}$ (no-flow relations are irreflexive).

For the *induction step*, we assume by induction hypothesis (IH) that $(\star)$ holds for an arbitrary $n \in \mathbb{N}$. Consider arbitrary sequence $\sigma = (z_1, z_2) \cdot \ldots \cdot (z_{n+1}, z_{n+2})$ of size $n + 1$. By (IH), there exists $1 \leq m_n \leq n$ defining $\mathcal{M}_n$ and $\mathcal{M}'_n$ over the σ's sub-sequence $(z_1, z_2) \cdot (z_2, z_3) \cdot \ldots \cdot (z_n, z_{n+1})$ of size n, as specified in $(\star)$, s.t. $\mathcal{M}'_n \subseteq \overline{\mathcal{N}'}$ and $\mathcal{M}_n \subseteq \overline{\mathcal{N}}$. We proceed by cases on the parity of $n + 1$.

Let $n + 1$ *be an odd number*, then $(z_{n+1}, z_{n+2}) \in \overline{\mathcal{N}'}$. Consider the case that the last pair in the sequence, (z_{n+1}, z_{n+2}), together with $\mathcal{M}'_n$ defines a flow relation that is a subset of $\overline{\mathcal{N}'}$, i.e., $(\mathcal{M}'_n \cup \{(z_{n+1}, z_{n+2})\})^* \subseteq \overline{\mathcal{N}'}$. Then, by (IH) there exists $m = m_n$ that satisfies $(\star)$ for the sequence of size $n + 1$. Otherwise, there exists a sequence using (z_{n+1}, z_{n+2}) and elements of $\mathcal{M}'_n$ defining a path between a pair of variables in $\mathcal{N}'$. Formally, there exists a sequence for $(z'_1, z'_{k+1}) \in \mathcal{N}'$:

$$(z'_1, z'_2) \cdot \ldots \cdot (z_{n+1}, z_{n+2}) \cdot \ldots \cdot (z'_k, z'_{k+1})$$

where $k \in \mathbb{N}$ and $\{(z'_1, z'_2) \cdots (z'_k, z'_{k+1})\} \subseteq \{(z_j, z_{j+1}) \mid m \leq j \leq n \text{ and } j \text{ is odd}\}$. Note that both the sequence before and the sequence after (z_{n+1}, z_{n+2}) may be empty, i.e., the path may start or end with (z_{n+1}, z_{n+2}). As $\mathcal{M}'_n$ is transitively closed, the sequence above can be simplified to:

$$(\dagger)\ (z'_1, z_{n+1}) \cdot (z_{n+1}, z_{n+2}) \cdot (z_{n+2}, z'_2)$$

where $\{(z'_1, z_{n+1}), (z_{n+2}, z'_2)\} \in \mathcal{M}'_n$. Recall that, $\overline{\mathcal{N}} \subseteq U \times V$ and $\overline{\mathcal{N}'} \subseteq U' \times V'$. By V and V' being disjoint sets and $(z_n, z_{n+1}) \in \overline{\mathcal{N}}$, it follows that $z_{n+1} \in V$ and $z_{n+1} \notin V'$. Then, for all variables z, $(z, z_{n+1}) \notin \mathcal{M}'_n$. Hence the sequence $(\dagger)$ must start with the pair (z_{n+1}, z_{n+2}). As $(z_{n+2}, z'_2) \in \mathcal{M}'_n$ then there exists a pair in the original sequence that starts with z_{n+2} and another pair that ends with z'_2. In particular, for

$m_n \leq h \leq h' \leq (n+1)$, we have the following sub-sequence of σ:

$$(z_{m_n}, z_{m_n+1}) \cdot \ldots \cdot (z_{h-1}, z_h) \cdot (\overset{z_{n+2}}{\overset{=}{z_h}}, z_{h+1}) \cdot \ldots \cdot (z_{h'}, \overset{z'_2}{\overset{=}{z_{h'+1}}}) \cdot \ldots \cdot (z_{n+1}, z_{n+2}).$$

By $(z_{n+1}, z_{n+2}) \in \overline{\mathcal{N}'}$ and V disjoint from V', then $z_{n+2} \in V'$ and $z_{n+2} \notin V$. Then, as $z_h = z_{n+2}$, there is no pair $(z_{h-1}, z_h) \in \overline{\mathcal{N}}$ and the sequence above must start with (z_h, z_{h+1}). So, $m_n = h$ and $z_{m_n} = z_h = z_{n+2}$ and the previous sequence simplifies as follows:

$$(\overset{z_{n+2}}{\overset{=}{z_{m_n}}}, z_{m_n+1}) \cdot \ldots \cdot (z_{h'}, \overset{z'_2}{\overset{=}{z_{h'+1}}}) \cdot \ldots \cdot (z_{n+1}, z_{n+2}).$$

By (IH), $z_1 = z_{m_n}$. Let $m = n+1$. As $n+1$ is an odd number, it defines the flow relations $\mathcal{M}'_{n+1} = \{(z_{n+1}, z_{n+2})\} \cup \mathrm{Id}_{V'}$ and $\mathcal{M}_{n+1} = \mathrm{Id}_V$. Then, $(z_1, z_{n+2}) \in \mathcal{M}'_{n+1}$ because $z_1 = z_{m_n} = z_{n+2}$ and $(z_{n+2}, z_{n+2}) \in \mathrm{Id}_{V'}$. By both $\mathcal{N}$ and $\mathcal{N}'$ being no-flow relations and $(z_{n+1}, z_{n+2}) \in \overline{\mathcal{N}'}$, then $\mathcal{M}'_{n+1} \subseteq \overline{\mathcal{N}'}$ and $\mathcal{M}_{n+1} \subseteq \overline{\mathcal{N}}$.

If $n+1$ *is an even number*, then $(z_{n+1}, z_{n+2}) \in \overline{\mathcal{N}}$ and the argument is analogous.

We can prove analogously that $\overline{\mathcal{N}} \circ (\overline{\mathcal{N}'} \circ \overline{\mathcal{N}})^* \subseteq \mathcal{N} \bullet \mathcal{N}'$. Finally, note that $\mathrm{Id}_Z \circ \mathrm{Id}_{V \cup V'} \subseteq \mathcal{N} \bullet \mathcal{N}'$ follows directly from $\mathcal{N}$ and $\mathcal{N}'$ being no-flow relations and their domain. □

3.1.1 Composition and Incremental Design

This section introduces *component and interface composition*, together with a predicate for *interfaces' compatibility for composition*. We prove that our notions of composition and compatibility support *incremental design* of systems.

From now on, to simplify the presentation, elements of an interface or component are annotated with the interface name, for example, $F = (X_F, Y_F, \mathcal{A}_F, \mathcal{G}_F, \mathcal{P}_F)$. The different types of variables between two interfaces F and F' are $Y_{F,F'} = Y_F \cup Y_{F'}$, $X_{F,F'} = (X_F \cup X_{F'}) \setminus Y_{F,F'}$, and $Z_{F,F'} = Y_{F,F'} \cup X_{F,F'}$. The same definition applies to components f and f'.

Variables between interfaces (components) define the set of variables in the composition of interfaces (components). We will often denote $X_{F,F'}$ as $X_{F \otimes F'}$, $Y_{F,F'}$ as $Y_{F \otimes F'}$, and $Z_{F,F'}$ as $Z_{F \otimes F'}$. The *composition of components* f *and* f' defines the component $f \otimes f' = (X_{f,f'}, Y_{f,f'}, (\mathcal{M}_f \cup \mathcal{M}_{f'})^*)$. We present interface composition by defining the open- and closed-guarantee, and the assumption of the composite separately.

We compose components and interfaces through their *shared variables*, i.e., all variables that are input for one of the interfaces (or components) while being output for the other. Formally, the set of shared variables for interfaces F and F', is $\mathrm{Shared}_{F,F'} =$

$(X_F \cup X_{F'}) \cap Y_{F,F'}$. We say that two interfaces F and F' are *composable* when their set of output variables are disjoint, $Y_F \cap Y_{F'} = \emptyset$.

The *composition of components* f and f' is defined with respect to the composition between their flow relations, i.e., $f \otimes f' = (X_{f,f'}, Y_{f,f'}, (\mathcal{M}_f \cup \mathcal{M}_{f'})^*)$.

Example: Confidentiality Over a Controller Area Network (CAN)

We introduce our second running example: the design of an *electronic vehicle immobilizer* (EVI) functionality. An EVI is a security device handling a transponder key used by car manufacturers to prevent hot wiring a car [LSS05, BRLEK17]. The engine control unit (ECU) only starts the car when the transponder authentication succeeds. The ECU and the immobilizer communicate through a controller area network (CAN), which is a serial communication technology commonly used in automobile architectures. Figure 3.5 presents a high-level view of the communication architecture between the immobilizer and the ECU, adapted from the architecture described in [LSS05].

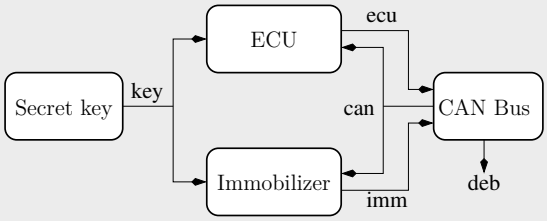


Figure 3.5: High-level view on immobilizer high-level architecture.

A CAN bus does not include native support for security-related features. So it is the responsibility of the components using the bus to enforce the confidentiality and integrity of the data sent over the CAN. In our example, the immobilizer's authentication follows a challenge-response protocol: the session starts with the ECU sending a freshly generated random number encrypted with a secret key known to both of the devices – challenge, which must be followed by the immobilizer sending an appropriate message encrypted with the same key – response. Our design goal is to guarantee the confidentiality policy: *the secret key shared by the ECU and immobilizer shall never leak to the environment via the CAN bus.*

In Figure 3.6 below, we depict an information-flow interface for a car's ECU – F_{ecu}, an immobilizer – F_{imm}, and for a CAN bus – F'_{can}. Interface F_{imm} only requires from the environment (i.e., its assumption) that there is no information flow from the secret key – key – to the input variable for the CAN bus – can; while it guarantees that none of its implementations has an information flow from key to the imm variable that connects the immobilizer to the CAN. The interface for the CAN requires that none of their implementations has a flow from both the input port for the ECU – ecu – and the immobilizer – imm– to the output debug

port – deb. In the same figure, we show the result of composing F_{imm} with F_{ecu} and F'_{can}, which we will explain in detail in this section.

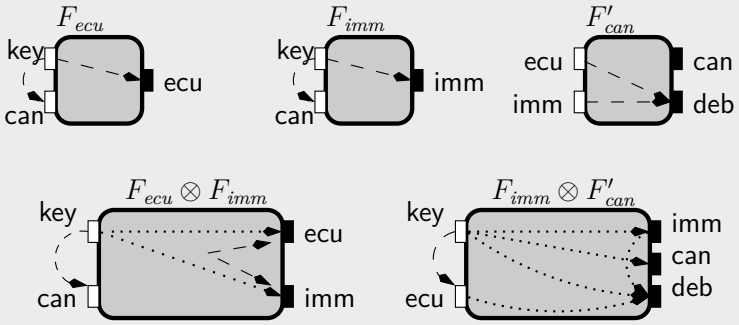


Figure 3.6: Interfaces for an ecu, an immobilizer and a CAN bus, and their composition.

Composite Open- and Closed-guarantee

The composition of two interfaces includes all information flows in the composition of any of the interfaces' implementations. Then, the composite open-guarantee of two interfaces is just the complement-flow composition of the interfaces' open-guarantees.

Definition 3.1.5. *Let F and F' be two interfaces with open-guarantees $\mathcal{G}_F$ and $\mathcal{G}_{F'}$. The* composite open-guarantee of F and F' *is $\mathcal{G}_{F\otimes F'} = (Z_{F,F'} \times Y_{F,F'}) \setminus (\mathcal{G}_F \bullet \mathcal{G}_{F'})$, also denoted $\mathcal{G}_{F\otimes F'}$.*

In the proposition below, that follows directly from definitions, we prove that a composite open-guarantee preserves all implementations of the interfaces being composed.

Proposition 3.1.3. *For all interfaces F and F' with open-guarantees $\mathcal{G}$ and $\mathcal{G}'$, and all components $f = (X, Y, \mathcal{M})$ and $f' = (X', Y', \mathcal{M}')$ that implements them, $f \models_{\mathcal{G}} F$ and $f' \models_{\mathcal{G}'} F'$, then the composition of the components satisfies the restriction imposed by the open-guarantee defined by both interfaces, i.e., $(\mathcal{M} \cup \mathcal{M}')^* \subseteq \overline{\mathcal{G}}_{F\otimes F'}$.*

Example: CAN bus Information Flows in Composite Implementations

We present a step-by-step evaluation of the composite open-guarantees defined by F_{imm} and F'_{can} depicted in Figure 3.6. From Lemma 3.1.2, we know that,

$$\mathcal{G}_{F_{\mathsf{imm}}} \bullet \mathcal{G}_{F'_{\mathsf{can}}} = (\mathrm{Id}_Z \cup \overline{\mathcal{G}}_{F_{\mathsf{imm}}}) \circ (\overline{\mathcal{G}}'_{F'_{\mathsf{can}}} \circ \overline{\mathcal{G}}_{F_{\mathsf{imm}}})^* \circ (\mathrm{Id}_Y \cup \overline{\mathcal{G}}'_{F'_{\mathsf{can}}}),$$

where $\mathrm{Id} = \{(z, z) \mid z \in \{\mathsf{key}, \mathsf{ecu}, \mathsf{can}, \mathsf{deb}, \mathsf{imm}\}\}$ and $\mathrm{Id}_{Y_{\mathsf{imm,can}}} = \{(z, z) \mid z \in \{\mathsf{can}, \mathsf{deb}, \mathsf{imm}\}\}$.

The set of flows that can be in some implementation of the immobilizer and the

can bus are:

$$\overline{\mathcal{G}}_{F_{\mathsf{imm}}} = \{(\mathsf{can}, \mathsf{imm}), (\mathsf{imm}, \mathsf{imm})\} \text{ and}$$
$$\overline{\mathcal{G}}_{F'_{\mathsf{can}}} = \{(\mathsf{ecu}, \mathsf{can}), (\mathsf{imm}, \mathsf{can}), (\mathsf{can}, \mathsf{deb}), (\mathsf{deb}, \mathsf{can}), (\mathsf{can}, \mathsf{can}), (\mathsf{deb}, \mathsf{deb})\}.$$

When we consider paths with two steps between implementations of immobilizer we get the following sets:

$$\overline{\mathcal{G}}_{F_{\mathsf{imm}}} \circ \overline{\mathcal{G}}_{F'_{\mathsf{can}}} = \{(\mathsf{can}, \mathsf{can}), (\mathsf{imm}, \mathsf{can})\} \text{ and}$$
$$\overline{\mathcal{G}}_{F'_{\mathsf{can}}} \circ \overline{\mathcal{G}}_{F_{\mathsf{imm}}} = \{\mathbf{(ecu,\ imm)}, (\mathsf{imm}, \mathsf{imm}), \mathbf{(deb,\ imm)}, (\mathsf{can}, \mathsf{imm})\}.$$

For example, the pair $(\mathsf{ecu}, \mathsf{imm})$ is defined by the path that starts with $(\mathsf{ecu}, \mathsf{can})$ from F'_{can} followed by $(\mathsf{can}, \mathsf{imm})$ in F_{imm}. The two-step paths added two new possible flows: from the ECU input port and the debug output port of the CAN bus to the imm output port in the immobilizer. We consider next three step paths and will see that there are no new flows:

$$(\overline{\mathcal{G}}_{F_{\mathsf{imm}}} \circ \overline{\mathcal{G}}_{F'_{\mathsf{can}}}) \circ \overline{\mathcal{G}}_{F_{\mathsf{imm}}} = \{(\mathsf{can}, \mathsf{imm}), (\mathsf{imm}, \mathsf{imm})\} = \overline{\mathcal{G}}_{F_{\mathsf{imm}}},$$
$$\overline{\mathcal{G}}_{F_{\mathsf{imm}}} \circ (\overline{\mathcal{G}}_{F'_{\mathsf{can}}} \circ \overline{\mathcal{G}}_{F_{\mathsf{imm}}}) = \{(\mathsf{can}, \mathsf{imm}), (\mathsf{imm}, \mathsf{imm})\} = \overline{\mathcal{G}}_{F_{\mathsf{imm}}},$$
$$(\overline{\mathcal{G}}_{F'_{\mathsf{can}}} \circ \overline{\mathcal{G}}_{F_{\mathsf{imm}}}) \circ \overline{\mathcal{G}}_{F'_{\mathsf{can}}} = \{(\mathsf{ecu}, \mathsf{can}), (\mathsf{imm}, \mathsf{can}), (\mathsf{deb}, \mathsf{can}), (\mathsf{can}, \mathsf{can})\} \subseteq \overline{\mathcal{G}}_{F'_{\mathsf{can}}} \text{ and}$$
$$\overline{\mathcal{G}}_{F'_{\mathsf{can}}} \circ (\overline{\mathcal{G}}_{F_{\mathsf{imm}}} \circ \overline{\mathcal{G}}_{F'_{\mathsf{can}}}) = \{(\mathsf{ecu}, \mathsf{can}), (\mathsf{imm}, \mathsf{can}), (\mathsf{can}, \mathsf{can})(\mathsf{deb}, \mathsf{can})\} \subseteq \overline{\mathcal{G}}_{F'_{\mathsf{can}}}.$$

We can now stop our evaluation, as considering longer paths will not define new flows. The set of flows in the composition between implementations of the immobilizer and the CAN bus are all their individual flows together with the two new ones we found:

$$\mathcal{G}_{F_{\mathsf{imm}}} \bullet \mathcal{G}_{F'_{\mathsf{can}}} = \overline{\mathcal{G}}_{F_{\mathsf{imm}}} \cup \overline{\mathcal{G}}_{F'_{\mathsf{can}}} \cup \{(\mathsf{ecu}, \mathsf{imm}), (\mathsf{deb}, \mathsf{imm})\}.$$

In Figure 3.6, we see that the composite open-guarantees are exactly the set with the pair of variables that are not in $\mathcal{G}_{F_{\mathsf{imm}}} \bullet \mathcal{G}_{F'_{\mathsf{can}}}$.

The complement of the composite open-guarantee of two interfaces F and F' and their open-guarantees no-flow composition define the same sets. Formally, $\overline{\mathcal{G}}_{F \otimes F'} = \mathcal{G}_F \bullet \mathcal{G}_{F'}$ because $\overline{\mathcal{G}}_{F \otimes F'} = (Z \times Y) \setminus ((Z \times Y) \setminus \mathcal{G}_F \bullet \mathcal{G}_{F'})$. By the definition of no-flow composition, it follows that it defines a monotonic and associative operation. Then, the complement of a composite open-guarantee is also monotonic and associative, i.e., $\overline{\mathcal{G}}_F \subseteq \overline{\mathcal{G}}_{F \otimes F'}$ and $\overline{\mathcal{G}}_{F \otimes F'} = \overline{\mathcal{G}}_{F' \otimes F}$.

We prove below that the composite open-guarantee defined by composable interfaces is also associative, i.e., the order we compute with different interfaces does not change the set of implementations their composition defines. We denote multiple iterations of evaluating composite open-guarantees as, for example, $\mathcal{G}_{(F \otimes F') \otimes F''} = (Z_{F \otimes F', F''} \times$

$Y_{F\otimes F',F''}) \setminus (\mathcal{G}_{F\otimes F'} \bullet \mathcal{G}_{F''})$ which is the composite open-guarantee from first computing it for F with F' followed by the open-guarantee of F''.

Lemma 3.1.4. *Let F, F' and F'' be composable. Then, $\mathcal{G}_{(F\otimes F')\otimes F''} = \mathcal{G}_{F\otimes(F'\otimes F'')}$.*

Proof. Consider arbitrary interfaces F, F' and F'' with pairwise disjoint set of output variables. By definition of variables between different interfaces:

$$Z_{F\otimes F',F''} \times Y_{F\otimes F',F''} = (Z_F \cup Z_{F'} \cup Z_{F''}) \times (Y_F \cup Y_{F'} \cup Y_{F''}) = Z_{F,F'\otimes F''} \times Y_{F,F'\otimes F''}.$$

In what follows, we denote the set of all variables over the three interfaces as Z (i.e., $Z = Z_F \cup Z_{F'} \cup Z_{F''}$), and the set of output variables as Y (i.e., $Y = Y_F \cup Y_{F'} \cup Y_{F''}$). By definition of composite open-guarantees:

$$\mathcal{G}_{(F\otimes F')\otimes F''} = (Z \times Y) \setminus (\mathcal{G}_{F\otimes F'} \bullet \mathcal{G}_{F''}) \text{ and}$$
$$\mathcal{G}_{F\otimes(F'\otimes F'')} = (Z \times Y) \setminus (\mathcal{G}_{F} \bullet \mathcal{G}_{F'\otimes F''}).$$

Then, what we want to prove is equivalent to proving:

$$\mathcal{G}_{F\otimes F'} \bullet \mathcal{G}_{F''} \;=\; \mathcal{G}_{F} \bullet \mathcal{G}_{F'\otimes F''}.$$

We start by proving that $\mathcal{G}_{F\otimes F'} \bullet \mathcal{G}_{F''} \subseteq \mathcal{G}_{F} \bullet \mathcal{G}_{F'\otimes F''}$. By Lemma 3.1.2, this is equivalent to prove that for all pair of variables $(z, y) \in Z \times Y$ and all $n \in \mathbb{N}$:

$$\text{if } (z,y) \in (\mathrm{Id}_Z \cup \overline{\mathcal{G}}_{F''}) \circ (\overline{\mathcal{G}}_{F\otimes F'} \circ \overline{\mathcal{G}}_{F''})^n \circ (\mathrm{Id}_Y \cup \overline{\mathcal{G}}_{F\otimes F'}),$$
$$\text{there exists } m \in \mathbb{N} \text{ s.t. } (z,y) \in (\mathrm{Id}_Z \cup \overline{\mathcal{G}}_{F'\otimes F''}) \circ (\overline{\mathcal{G}}_{F} \circ \overline{\mathcal{G}}_{F'\otimes F''})^m \circ (\mathrm{Id}_Y \cup \overline{\mathcal{G}}_{F}).$$

To simplify the presentation of the proof, we start by proving for the case that $y \in Y_{F''}$. Note that, if $y \in Y_{F''}$, by output variables being disjoint, then, for all variables $z' \in Z$, $(z', y) \notin (\overline{\mathcal{G}}_{F\otimes F'} \cup \overline{\mathcal{G}}_{F})$. Moreover, $\overline{\mathcal{G}}_{F''} \circ \mathrm{Id}_{Y_{F''}} = \overline{\mathcal{G}}_{F''}$. Then, we want to prove by induction on $n \in \mathbb{N}$ that:

$$\text{if } (z,y) \in (\mathrm{Id}_Z \cup \overline{\mathcal{G}}_{F''}) \circ (\overline{\mathcal{G}}_{F\otimes F'} \circ \overline{\mathcal{G}}_{F''})^n,$$
$$\text{there exists } m \in \mathbb{N} \text{ s.t. } (z,y) \in (\mathrm{Id}_Z \cup \overline{\mathcal{G}}_{F'\otimes F''}) \circ (\overline{\mathcal{G}}_{F} \circ \overline{\mathcal{G}}_{F'\otimes F''})^m \circ \mathrm{Id}_{Y_{F''}}.$$

For the *base case*, $n = 0$, we consider arbitrary $(z, y) \in (\mathrm{Id}_Z \cup \overline{\mathcal{G}}_{F''})$. By monotonicity of composite open-guarantees $\overline{\mathcal{G}}_{F''} \subseteq \overline{\mathcal{G}}_{F'\otimes F''}$, and by $y \in Y_{F''}$, it follows $(z, y) \in (\mathrm{Id}_Z \cup \overline{\mathcal{G}}_{F'\otimes F''}) \circ \mathrm{Id}_{Y_{F''}}$.

For the *induction step*, we assume as induction hypothesis (IH) that the property holds for n. Now, consider arbitrary (z, y) s.t. $(z,y) \in (\mathrm{Id}_Z \cup \overline{\mathcal{G}}_{F''}) \circ (\overline{\mathcal{G}}_{F\otimes F'} \circ \overline{\mathcal{G}}_{F''})^{n+1}$. Then, $(z, y) = \{(z, s)\} \circ \{(s, y)\}$ with:

$$(\star)\ (z,s) \in (\mathrm{Id}_Z \cup \overline{\mathcal{G}}_{F''}) \circ (\overline{\mathcal{G}}_{F\otimes F'} \circ \overline{\mathcal{G}}_{F''})^n \;\text{ and }\; (s,y) \in \overline{\mathcal{G}}_{F\otimes F'} \circ \overline{\mathcal{G}}_{F''}.$$

By induction hypothesis, exists $m \in \mathbb{N}$ s.t.:

$$(z, s) \in (\mathrm{Id}_Z \cup \overline{\mathcal{G}}_{F' \otimes F''}) \circ (\overline{\mathcal{G}}_F \circ \overline{\mathcal{G}}_{F' \otimes F''})^m \circ \mathrm{Id}_{Y_{F''}}.$$

We proceed by cases on (z, s).

If $(z, s) \in \mathrm{Id}_Z \circ \mathrm{Id}_{Y_{F''}}$, then $(z, y) \in \mathrm{Id}_{Y_{F''}} \circ \overline{\mathcal{G}}_{F \otimes F'} \circ \overline{\mathcal{G}}_{F''}$ and, more generally, $(z, y) \in \overline{\mathcal{G}}_{F \otimes F'} \circ \overline{\mathcal{G}}_{F''}$. By definition of composite open-guarantees, $(z, y) \in (\mathcal{G}_F \bullet \mathcal{G}_{F'}) \circ \overline{\mathcal{G}}_{F''}$. Then, by Lemma 3.1.2:

$$(z, y) \in (\mathrm{Id}_Z \cup \overline{\mathcal{G}}_F) \circ (\overline{\mathcal{G}}_{F'} \circ \overline{\mathcal{G}}_F)^* \circ (\mathrm{Id}_Y \cup \overline{\mathcal{G}}_{F'}) \circ \overline{\mathcal{G}}_{F''}.$$

By monotonicity of open-guarantees, $\overline{\mathcal{G}}_{F'} \subseteq \overline{\mathcal{G}}_{F' \otimes F''}$, $\overline{\mathcal{G}}_{F''} \subseteq \overline{\mathcal{G}}_{F' \otimes F''}$ and $\overline{\mathcal{G}}_{F'} \circ \overline{\mathcal{G}}_{F''} \subseteq \overline{\mathcal{G}}_{F' \otimes F''}$, so $(z, y) \in (\mathrm{Id} \cup \overline{\mathcal{G}}_F) \circ (\overline{\mathcal{G}}_{F' \otimes F''} \circ \overline{\mathcal{G}}_F)^* \circ \overline{\mathcal{G}}_{F' \otimes F''}$. Equivalently, $(z, y) \in (\mathrm{Id} \cup \overline{\mathcal{G}}_{F' \otimes F''}) \circ (\overline{\mathcal{G}}_F \circ \overline{\mathcal{G}}_{F' \otimes F''})^*$.

If $(z, s) \notin \mathrm{Id}_Z$ then, by $(\star)$, $s \in Y_{F''}$. By F' and F'' having disjoint sets of output variables, $\overline{\mathcal{G}}_{F' \otimes F''} = \mathcal{G}_{F'} \bullet \mathcal{G}_{F''}$ and Lemma 3.1.2:

$$(z, s) \in (\mathrm{Id}_Z \cup \overline{\mathcal{G}}_{F' \otimes F''}) \circ (\overline{\mathcal{G}}_F \circ \overline{\mathcal{G}}_{F' \otimes F''})^{m-1} \circ \overline{\mathcal{G}}_F \circ (\mathrm{Id}_Z \cup \overline{\mathcal{G}}_{F'}) \circ (\overline{\mathcal{G}}_{F''} \circ \overline{\mathcal{G}}_{F'})^* \circ \overline{\mathcal{G}}_{F''}.$$

By $(\star)$, $\overline{\mathcal{G}}_{F \otimes F'} = \mathcal{G}_F \bullet \mathcal{G}_{F'}$ and Lemma 3.1.2:

$$(s, y) \in (\mathrm{Id}_Z \cup \overline{\mathcal{G}}_F) \circ (\overline{\mathcal{G}}_{F'} \circ \overline{\mathcal{G}}_F)^* \circ (\mathrm{Id}_Z \cup \overline{\mathcal{G}}_{F'}) \circ \overline{\mathcal{G}}_{F''}.$$

Then,

$$\begin{aligned}(z, y) \in &(\mathrm{Id}_Z \cup \overline{\mathcal{G}}_{F' \otimes F''}) \circ (\overline{\mathcal{G}}_F \circ \overline{\mathcal{G}}_{F' \otimes F''})^{m-1} \circ \overline{\mathcal{G}}_F \circ (\mathrm{Id} \cup \overline{\mathcal{G}}_{F'}) \circ (\overline{\mathcal{G}}_{F''} \circ \overline{\mathcal{G}}_{F'})^* \circ \overline{\mathcal{G}}_{F''} \\ &\circ (\mathrm{Id}_Z \cup \overline{\mathcal{G}}_F) \circ (\overline{\mathcal{G}}_{F'} \circ \overline{\mathcal{G}}_F)^* \circ (\mathrm{Id} \cup \overline{\mathcal{G}}_{F'}) \circ \overline{\mathcal{G}}_{F''}\end{aligned}$$

Equivalently, $(z, y) \in (\mathrm{Id} \cup \overline{\mathcal{G}}_{F' \otimes F''}) \circ (\overline{\mathcal{G}}_F \circ \overline{\mathcal{G}}_{F' \otimes F''})^{m'}$ for some $m' > m$.

The case $y \in Y_{F,F'}$ is analogous. We prove by a similar inductive argument that $\mathcal{G}_{F \otimes F'} \bullet \mathcal{G}_{F''} \supseteq \mathcal{G}_F \bullet \mathcal{G}_{F' \otimes F''}$. □

We define composite closed-guarantee by introducing *derived closed-guarantee*: the set with all pairs of variables with no information flow in all closed systems defined by a given interface. Formally, given an interface, we remove from its open-guarantee all pairs that define a flow in some of the closed systems defined by that interface. Note that, in a well-formed interface, all pairs in the closed-guarantee must be in the open-guarantee.

Definition 3.1.6. *Let F be an interface with assumption $\mathcal{A}$ and open-guarantee $\mathcal{G}$. The* derived closed-guarantee *from $\mathcal{A}$ and $\mathcal{G}$ is $\hat{\mathcal{P}}_{\mathcal{A},\mathcal{G}} = \mathcal{G} \setminus (\mathcal{A} \bullet \mathcal{G})$.*

Example: CAN bus Composite Closed-guarantee

We consider the composition between F_{ecu} and F_{imm}, depicted in Figure 3.6. The composite open-guarantee of $F_{ecu} \otimes F_{imm}$ has four no-flow requirements. Note that there are no shared variables between the interfaces being composed, so there are no new flows in their composition. In particular, all no-flows in the open-guarantee of F_{ecu} and F_{imm} will be in their open-guarantee composite, too. Additionally, there is no way for information to flow between ecu and imm only through implementations of F_{ecu} and F_{imm}, formally, $\{(\mathsf{ecu}, \mathsf{imm}), (\mathsf{imm}, \mathsf{ecu})\} \cap (\mathcal{G}_{F_{ecu}} \bullet \mathcal{G}_{F_{imm}}) = \emptyset$.

There can be, however, flows between ecu and imm if we consider the environment. For example, information from ecu can flow to can through the environment and then from can can flow to imm by an implementation of F_{imm}. Hence $\{(\mathsf{ecu}, \mathsf{imm}), (\mathsf{imm}, \mathsf{ecu})\} \subseteq \mathcal{A}_{F_{ecu} \otimes F_{imm}} \bullet \mathcal{G}_{F_{ecu} \otimes F_{imm}}$, and so, it is not a requirement of the composite closed-guarantee. The other no-flows in the open-guarantee – from key to ecu and imm – cannot be defined by flow paths going through the environment because we assume that the environment does not include a flow from key to can. Then, they are in the derived closed-guarantee.

Composite Assumption

We use interfaces' assumptions to determine whether interfaces are compatible for composition. Intuitively, we must check that the composite interface fulfills all no-flows requirements over the environment pointing to shared variables. Note that, as shared variables will be output variables of the composite interface, all pairs in the assumptions pointing to a shared variable will not be in the assumption of the composite.

Given two interfaces, their composite assumption is the weakest condition in the environment, allowing the interfaces to work together. In particular, we can add new assumptions during composition as we assume that the environment is helpful. However, we need to be careful not to unnecessarily restrict the environment because we want to support the incremental design of systems (i.e., interfaces' compatibility for composition needs to be independent of the order they are composed). We introduce next *derived assumptions* as the set of all no-flow requirements necessary to guarantee that no assumption to a shared variable goes unfulfilled after composition.

Example: CAN bus Derived Assumptions

The interface F_{imm} in Figure 3.6 has as it only no-flow requirement in the environment that there should be no information flow from the input variable key to the output variable can. The variable can is a shared variable with F'_{can}, in the same figure. Note that, F'_{can} cannot guarantee that no information flow from key to can because key is not one of its output variables. Additionally, key will remain

an input variable of the composition of F_{imm} and F'_{can}. Hence we need to add a new no-flow requirement to the composite's assumption to guarantee that F_{imm}'s assumption is satisfied by the environment of the composite. In particular, we need to require that key's information does not flow to an input port in F'_{can} that allow flows to can. In this example, in F'_{can} information in ecu can flow to can and so the composite interface must include the assumption that key does not flow to ecu. This new requirement in the environment is called a *derived assumption*.

Definition 3.1.7. *Let F and F' be two interfaces with assumptions $\mathcal{A}_F$ and $\mathcal{A}_{F'}$, respectively. The* assumption derived from F and F' *is:*

$$\hat{\mathcal{A}}_{F,F'} = \{(z, z') \mid \exists s \in \text{Shared}_{F,F'} \text{ s.t. } (z, s) \in \mathcal{A}_F \cup \mathcal{A}_{F'} \text{ and } (z', s) \in \overline{\mathcal{G}}_{F \otimes F'}\}$$

where $\text{Shared}_{F,F'} = (X_F \cup X_{F'}) \cap Y_{F,F'}$. *The* composite assumption *between F and F' is* $\mathcal{A}_{F \otimes F'} = (\mathcal{A}_F \cup \mathcal{A}_{F'} \cup \hat{\mathcal{A}}_{F,F'}) \cap (Z_{F,F'} \times X_{F,F'})$.

As composite open-guarantees are commutative, then composite assumptions are also commutative. We prove below that derived assumptions between composable interfaces are also monotonic and associative.

Lemma 3.1.5. *Let F, F' and F'' be information-flow interfaces that are pairwise composable.* (a) *If $(z, z') \in \hat{\mathcal{A}}_{F',F''}$, then $(z, z') \in \hat{\mathcal{A}}_{F \otimes F',F''}$.* (b) *If $(z, z') \in \hat{\mathcal{A}}_{F,F' \otimes F''}$, then $(z, z') \in \hat{\mathcal{A}}_{F \otimes F',F''} \cup \hat{\mathcal{A}}_{F,F'}$.*

Proof. Consider arbitrary information-flow interfaces F, F' and F'' that are pairwise composable, i.e., all three interfaces have pairwise disjoint sets of output variables.

We start by proving item (a). Consider arbitrary $(z, z') \in \hat{\mathcal{A}}_{F',F''}$. By definition of derived assumptions, there exists a variable s s.t.:

$$(z, s) \in \mathcal{A}_{F'} \cup \mathcal{A}_{F''} \text{ and } (z', s) \in \overline{\mathcal{G}}_{F' \otimes F''}.$$

Note that, by definition of interface and domain of composite open-guarantee, s is a shared variable, i.e., $s \in \text{Shared}_{F',F''}$, where $\text{Shared}_{F',F''} = (X_{F'} \cup X_{F''}) \cap (Y_{F'} \cup Y_{F''})$. Then, by $s \in \text{Shared}_{F',F''}$, $s \in Y_{F',F''}$ and, by output variables being disjoint, s cannot be an output variable of F (i.e., $s \notin Y_F$). We proceed by cases on the domain of s. If $s \in Y_{F'}$, then, by $s \in X_{F'} \cup X_{F''}$ and definition of information-flow interfaces, s must an input variable of F''; hence, $(z, s) \in \mathcal{A}_{F''}$. By monotonicity, associativity (Lemma 3.1.4) and definition of of composite open-guarantee, $(z,' s) \in \mathcal{G}_{F \otimes F'} \bullet \mathcal{G}_{F''}$. So, by definition of derived assumptions, $(z, z') \in \hat{\mathcal{A}}_{F \otimes F',F''}$. If $s \in Y_{F''}$, then, by $s \in X_{F'} \cup X_{F''}$ and definition of information-flow interfaces, s must an input variable of F' and $(z, s) \in \mathcal{A}_{F'}$. As for the previous case, it follows that $(z', s) \in \mathcal{G}_{F \otimes F'} \bullet \mathcal{G}_{F''}$ and consequently $(z, z') \in \hat{\mathcal{A}}_{F \otimes F',F''}$.

We now prove item (b). Assume that $(z, z') \in \hat{\mathcal{A}}_{F,F'\otimes F''}$. By definition of derived assumptions, there exists a variable s s.t.:

$$(z, s) \in \mathcal{A}_F \cup \mathcal{A}_{F'\otimes F''} \text{ and } (z', s) \in \overline{\mathcal{G}}_{F,F'\otimes F''}, \text{ where } s \in \text{Shared}_{F,F'\otimes F''}.$$

By $(z', s) \in \overline{\mathcal{G}}_{F\otimes(F'\otimes F'')}$ and associativity of composite open-guarantees (Lemma 3.1.4), it follows (b1) $(z', s) \in \overline{\mathcal{G}}_{(F\otimes F')\otimes F''}$. We proceed now by cases on the domain of the shared variable $s \in (X_F \cup X_{F'\otimes F''}) \cap Y_{F,F'\otimes F''}$.

If $s \in X_F \cap Y_{F''}$, by all three interfaces having disjoint sets of output variables, then $s \notin Y_{F'}$, $s \in X_{F,F'}$ and s is a shared variable between $F \otimes F'$ and F''. Moreover, by $(z, s) \in \mathcal{A}_F \cup \mathcal{A}_{F'\otimes F''}$ and $s \in Y_{F''}$, then $s \notin X_{F'\otimes F''}$ and the only relevant case for (z, s) is it being in the assumption of F, i.e., $(z, s) \in \mathcal{A}_{F\otimes F'}$. Then, by (b1) and definition of derived assumptions, $(z, z') \in \hat{\mathcal{A}}_{F\otimes F',F''}$.

If $s \in X_F \cap Y_{F'}$, then, s is a shared variable between F and F'. By $s \in Y_{F'}$, then $s \notin X_{F'\otimes F''}$. Then, we can assume (z, s) to be in the assumption of F, i.e., $(z, s) \in \mathcal{A}_F$. By associativity of composite open-guarantees, $(z', s) \in \mathcal{G}_{F\otimes F'} \bullet \mathcal{G}_{F''}$. By output variables of all three interfaces being disjoint, $s \in Y_{F'}$ and Lemma 3.1.2, then the last flow from any path from z' to s must be in $\overline{\mathcal{G}}_{F\otimes F'}$. Formally, $(z', s) = (\text{Id}_{Z_{F\otimes F',F''}} \cup \overline{\mathcal{G}}_{F''}) \circ (\overline{\mathcal{G}}_{F\otimes F'} \circ \overline{\mathcal{G}}_{F''})^* \circ \overline{\mathcal{G}}_{F\otimes F'}$. Then, there exists a variable s' s.t. $(z', s) = (z', s') \cdot (s', s)$ with:

$$(z', s') \in (\text{Id}_{Z_{F\otimes F',F''}} \cup \overline{\mathcal{G}}_{F''}) \circ (\overline{\mathcal{G}}_{F\otimes F'} \circ \overline{\mathcal{G}}_{F''})^* \text{ and } (s', s) \in \overline{\mathcal{G}}_{F\otimes F'}.$$

By $(z, s) \in \mathcal{A}_F$, $s \in \text{Shared}_{F,F'}$ and $(s', s) \in \overline{\mathcal{G}}_{F\otimes F'}$, then $(z, s') \in \hat{\mathcal{A}}_{F,F'}$. If $(z', s') \in \text{Id}_{Z_{F\otimes F',F''}}$, then $(z', s) = (s', s)$. Hence $(z', s) \in \overline{\mathcal{G}}_{F\otimes F'}$ and, by $(z, s) \in \mathcal{A}_F$, $(z, z') \in \hat{\mathcal{A}}_{F,F'}$. Otherwise, s' must be an output variable of F'' ($s' \in Y_{F''}$) and an input variable of the other interface ($s' \in X_{F\otimes F'}$). Then, $(z, s') \in \mathcal{A}_{F\otimes F'}$. By $(z', s') \in \overline{\mathcal{G}}_{(F\otimes F')\otimes F''}$ and definition of derived assumptions, $(z, z') \in \hat{\mathcal{A}}_{F\otimes F',F''}$.

If $s \in X_{F'\otimes F''} \cap Y_F$, then it can only be the case that $(z, s) \in \mathcal{A}_{F'\otimes F''}$ and we proceed by cases on $\mathcal{A}_{F'\otimes F''}$ definition. If $(z, s) \in \mathcal{A}_{F'}$, then $s \in X_{F'} \cap Y_F$. The rest is analogous to the previous case where $s \in X_F \cap Y_{F'}$. If $(z, s) \in \mathcal{A}_{F''}$, then, by $(z', s) \in \overline{\mathcal{G}}_{F\otimes F',F''}$ and definition of derived assumptions, $(z, z') \in \hat{\mathcal{A}}_{F\otimes F',F''}$. Lastly, if $(z, s) \in \hat{\mathcal{A}}_{F',F''}$, by definition of derived assumptions: $(z, s') \in \mathcal{A}_{F'} \cup \mathcal{A}_{F''}$ and $(s, s') \in \overline{\mathcal{G}}_{F',F''}$, with $s' \in \text{Shared}_{F',F''}$. By s being an output variable of F, $(z', s) \in \overline{\mathcal{G}}_{F,F'\otimes F''}$ and Lemma 3.1.2, $(z', s) \in (\text{Id}_{Z_{F,F'\otimes F''}} \cup \overline{\mathcal{G}}_{F'\otimes F''}) \circ (\overline{\mathcal{G}}_F \circ \overline{\mathcal{G}}_{F'\otimes F''})^* \circ \overline{\mathcal{G}}_F$. As $(s, s') \in \overline{\mathcal{G}}_{F',F''}$ then we know that a flow from s to s' can be defined by an alternating composition of elements of $\overline{\mathcal{G}}_{F'}$ and $\overline{\mathcal{G}}_{F''}$, i.e. without using elements of $\overline{\mathcal{G}}_F$. Hence $(z', s) \cdot (s, s') \in \overline{\mathcal{G}}_{F,F'\otimes F''}$ and, by associativity of composite open-guarantees (Lemma 3.1.4), $(z', s') \in \overline{\mathcal{G}}_{F\otimes F',F''}$. If $(z, s') \in \mathcal{A}_{F'}$, then $s' \in Y_{F''}$ and, by (ii), $s' \notin Y_F$. Then, $s' \in X_{F\otimes F'}$ and so $(z, s') \in \mathcal{A}_{F\otimes F'}$. Hence, by $(z', s') \in \overline{\mathcal{G}}_{F\otimes F',F''}$ and $(z, s') \in \mathcal{A}_{F\otimes F'}$, $(z, z') \in \hat{\mathcal{A}}_{F\otimes F',F''}$. If $(z, s') \in \mathcal{A}_{F''}$, then, by $(z, s') \in \overline{\mathcal{G}}_{F\otimes F',F''}$, $(z, z') \in \hat{\mathcal{A}}_{F\otimes F',F''}$. □

Composition and Compatibility

We now have all the ingredients to define interface composition, which we do below. Not all compositions will result in a well-formed interface, for this reason, we introduce the concept of two interfaces being *compatible* for composition: all requirements on the environment concerning the interfaces' shared variables must be satisfied by their composite. In particular, we require that for all pairs in the interfaces' assumptions that are not in the assumption of the composite (they point to shared variables), the composite open-guarantee includes them (the flows are not in the composition of any of the interfaces' implementations). Note that our notion of derived assumptions adds environmental requirements that affect future compatibility checks. We add these requirements for all pairs in the interfaces' assumptions, for which we cannot guarantee that the current composite fully satisfies them, ensuring that no no-flow requirement on the environment is lost during composition.

Definition 3.1.8. *The* composition *of two interfaces F and F' is the interface $F \otimes F' = (X_{F,F'}, Y_{F,F'}, \mathcal{A}_{F\otimes F'}, \mathcal{G}_{F\otimes F'}, \mathcal{P}_{F\otimes F'})$, where $\mathcal{A}_{F\otimes F}$ is as in Definition 3.1.7, $\mathcal{G}_{F\otimes F}$ is as in Definition 3.1.5, and $\mathcal{P}_{F\otimes F'} = \mathcal{P}_F \cup \mathcal{P}_{F'} \cup \hat{\mathcal{P}}_{\mathcal{A}_{F\otimes F'}, \mathcal{G}_{F\otimes F'}}$. The interfaces F and F' are* composable *iff $Y_F \cap Y_{F'} = \emptyset$; and they are* compatible*, denoted $F \sim F'$ iff they are composable and $((\mathcal{A}_F \cup \mathcal{A}_{F'}) \cap (Z_{F,F'} \times Y_{F,F'})) \subseteq \mathcal{G}_{F\otimes F'}$.*

Example: SCI Compatibility

In Figure 3.7, we depict two possible specifications for the components sending data through the shared bus together with the *Bus′* interface from Figure 3.2. The interface *Sending* has no guarantees, so it is not compatible for composition with *Bus′*. An interface is compatible for composition with *Bus′* if their composite open-guarantee is sufficient to satisfy the *Bus′* assumption concerning their shared variables. In particular, it needs to guarantee that there are no information flows from the wheel sensor – *wheel_tick* – to both distance warners source – *distw_b_s* and *distw_f_s*.

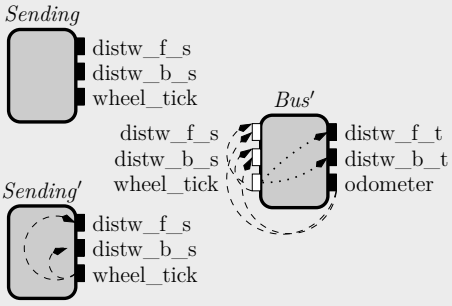


Figure 3.7: Two interfaces – *Sending* and *Sending′* –to specify components sending data to the shared bus – *Bus′*.

A system designer can use our theory to identify the source of incompatibility and change the design accordingly. In this example, the designer refined the *Sending* interface by adding to the open-guarantee the missing requirements (depicted by *Sending′*).

Incremental Design of Systems

It is clear from our definitions that both the composition operator and the compatibility relation are commutative. The first property we prove about our composition operator is that it preserves well-formedness, i.e., the composite of two well-formed interfaces is a well-formed interface.

Theorem 3.1.6. *Let F and F' be well-formed interfaces. If they are compatible, $F \sim F'$, then their composition, $F \otimes F'$, defines a well-formed interface.*

Proof. Consider arbitrary well-formed interfaces F and F', and assume they are compatible, $F \sim F'$. By definition of composition, it follows that the composite assumption, $\mathcal{A}_{F\otimes F'}$, and both the composite open- and closed-guarantee, $\mathcal{G}_{F\otimes F'}$ and $\mathcal{P}_{F\otimes F'}$, are no-flow relations (i.e. irreflexive). We are only missing to prove that the closed-guarantee is consistent with the assumption and the open-guarantee, i.e., $(\mathcal{A}_{F\otimes F} \bullet \mathcal{G}_{F\otimes F'}) \cap \overline{\mathcal{P}}_{F\otimes F'} = \emptyset$.

Before we proceed, we observe that any path alternating between elements in $\overline{\mathcal{A}}_{F\otimes F'}$ and $\overline{\mathcal{G}}_{F\otimes F'}$ from any variable of F (or F') to an output variable of the same interface can be translated to a path using only the complements of assumptions and open-guarantees of F (or F'). Formally, for compatible interfaces F and F':

$$\begin{gathered}(\star) \text{ if } (z, z') \in Z_F \times Y_F \text{ and,} \\ \text{for some } n \in \mathbb{N},\ (z, z') \in (\mathrm{Id}_{F,F'} \cup \overline{\mathcal{A}}_{F\otimes F'}) \circ (\overline{\mathcal{G}}_{F\otimes F'} \circ \overline{\mathcal{A}}_{F\otimes F'})^n \circ \overline{\mathcal{G}}_{F\otimes F'}, \\ \text{then exists } m \in \mathbb{N} \text{ s.t. } (z, z') \in (\mathrm{Id}_{Z_F} \cup \overline{\mathcal{A}}_F) \circ (\overline{\mathcal{G}}_F \circ \overline{\mathcal{A}}_F)^m \circ \overline{\mathcal{G}}_F.\end{gathered}$$

This property relies on the interfaces' compatibility requirement that assumptions to their shared variables are covered by flows allowed by their composite open-guarantee. We prove $(\star)$ by natural induction on n for compatible interfaces F and F'. Let $(z, z') \in Z_F \times Y_F$.

For the *base case*, $n = 0$, $(z, z') \in \overline{\mathcal{G}}_{F\otimes F'} \cup (\overline{\mathcal{A}}_{F\otimes F'} \circ \overline{\mathcal{G}}_{F\otimes F'})$. We proceed by cases on the (z, z') domain.

If $(z, z') \in \overline{\mathcal{G}}_{F\otimes F'}$ and, by $z' \in Y_F$ and Lemma 3.1.2, then the last flow of any path from z to z' must be in $\overline{\mathcal{G}}_F$, i.e., $(z, z') \in (\mathrm{Id}_{F,F'} \cup \overline{\mathcal{G}}_{F'}) \circ (\overline{\mathcal{G}}_F \circ \overline{\mathcal{G}}_{F'})^* \circ \overline{\mathcal{G}}_F$. Let $(z, s) \in (\mathrm{Id}_{F,F'} \cup \overline{\mathcal{G}}_{F'}) \circ (\overline{\mathcal{G}}_F \circ \overline{\mathcal{G}}_{F'})^*$ and $(s, z') \in \overline{\mathcal{G}}_F$. If $(z, s) = (s, s)$, then $(z, z') \in \overline{\mathcal{G}}_F$. Otherwise, by the interfaces being compatible, their set of output variables are disjoint, and $s \in Y_{F'}$ and $s \in X_F$. Again by interfaces compatibility, $(z, s) \in Z_{F,F'} \times Y_{F,F'}$ and $(z, s) \in \overline{\mathcal{G}}_{F\otimes F'}$, then $(z, s) \notin ((\mathcal{A}_F \cup \mathcal{A}_{F'}) \cap Z_{F\otimes F'} \times Y_{F\otimes F'})$. So, in particular, $(z, s) \notin \mathcal{A}_F$ and, by $s \in X_F$, $(z, s) \in \overline{\mathcal{A}}_F$ Hence, $(z, z') \in \overline{\mathcal{A}}_F \circ \overline{\mathcal{G}}_F$.

If $(z, z') \in \overline{\mathcal{A}}_{F\otimes F'} \circ \overline{\mathcal{G}}_{F\otimes F'}$, then, by $z' \in Y_F$ and Lemma 3.1.2, $(z, z') \in \overline{\mathcal{A}}_{F\otimes F'} \circ (\mathrm{Id}_{F,F'} \cup \overline{\mathcal{G}}_{F'}) \circ (\overline{\mathcal{G}}_F \circ \overline{\mathcal{G}}_{F'})^* \circ \overline{\mathcal{G}}_F$. Consider arbitrary:

$$(z, s) \in \overline{\mathcal{A}}_{F\otimes F'},\ (s, s') \in (\mathrm{Id}_{F,F'} \cup \overline{\mathcal{G}}_{F'}) \circ (\overline{\mathcal{G}}_F \circ \overline{\mathcal{G}}_{F'})^* \text{ and } (s', y) \in \overline{\mathcal{G}}_F.$$

If $(s, s') = (s', s')$, then $(z, s') \in \overline{\mathcal{A}}_{F\otimes F'}$ and $(s', y) \in \overline{\mathcal{G}}_F$. As $(z, s') \in \overline{\mathcal{A}}_{F\otimes F'}$, then s' is an input variable of both interfaces ($s' \in X_{F\otimes F'}$) and, by $(s', y) \in \overline{\mathcal{G}}_F$ and definition of interface, s' must be an input variable of F ($s' \in X_F$). Then, by $(z, s') \in \overline{\mathcal{A}}_{F\otimes F'}$ and definition of composite assumptions, $(z, s') \notin \mathcal{A}_F$. So, by $s' \in X_F$, $(z, s') \in \overline{\mathcal{A}}_F$. If $(s, s') \neq (s', s')$, by interfaces compatibility, $s' \in Y_{F'}$ and $s' \in Z_F$, then $s' \in X_F$. Assume towards a contradiction that $(z, s') \in \mathcal{A}_F$. Then, by definition of derived assumptions and $(s, s') \in \overline{\mathcal{G}}_{F\otimes F'}$, $(z, s) \in \hat{\mathcal{A}}_{F,F'}$. As $(z, s) \in \overline{\mathcal{A}}_{F\otimes F'}$, then $s \in X_{F\otimes F'}$ and $(z, s) \in \mathcal{A}_{F\otimes F'}$. This contradicts $(z, s) \in \overline{\mathcal{A}}_{F\otimes F'}$. Hence $(z, s') \notin \mathcal{A}_F$ and so $(z, z') \in \overline{\mathcal{A}}_F \circ \overline{\mathcal{G}}_F$.

The *induction step* is proved by applying the induction hypothesis followed by a proof analogous to the base case.

With $(\star)$ proved, we proceed to prove $(\mathcal{A}_{F\otimes F} \bullet \mathcal{G}_{F\otimes F'}) \cap \overline{\mathcal{P}}_{F\otimes F'} = \emptyset$. Consider arbitrary $(z, z') \in \mathcal{A}_{F\otimes F} \bullet \mathcal{G}_{F\otimes F'}$. By definition of derived closed-guarantee, $(z, z') \notin \hat{\mathcal{P}}_{\mathcal{A}_{F\otimes F'},\mathcal{G}_{F\otimes F'}}$. We are only missing to prove that $(z, z') \notin \mathcal{P}_F \cup \mathcal{P}_{F'}$. Note that the domain of $\mathcal{P}_F$ and $\mathcal{P}_{F'}$ is $Z_F \times Y_F$ and $Z_{F'} \times Y_{F'}$, respectively. If (z, z') is not in the union of these domains, then $(z, z') \notin \mathcal{P}_F \cup \mathcal{P}_{F'}$. Let $(z, z') \in (Z_F \times Y_F) \cup (Z_{F'} \times Y_{F'})$. Note that, by Lemma 3.1.2, $(z, z') \in (\mathrm{Id}_{Z_{F,F}} \cup \overline{\mathcal{A}}_{F\otimes F'}) \circ (\overline{\mathcal{G}}_{F\otimes F'} \circ \overline{\mathcal{A}}_{F\otimes F'})^* \circ \overline{\mathcal{G}}_{F\otimes F'}$.

Now, if we consider the case that $z' \in Y_F$, then, by the interfaces being compatible, $z' \notin Y_{F'}$. Thus, $(z, z') \in Z_F \times Y_F$, $(z, z') \notin Z_{F'} \times Y_{F'}$ and, by definition of interface, $(z, z') \notin \mathcal{P}_{F'}$. By $(z, z') \in Z_F \times Y_F$, $F \sim F$ and $(\star)$, $(z, z') \in (\mathrm{Id}_{Z_{F,F'}} \cup \overline{\mathcal{A}}_F) \circ (\overline{\mathcal{G}}_F \circ \overline{\mathcal{A}}_F)^* \circ \overline{\mathcal{G}}_F$, i.e., $(z, z') \in \mathcal{A}_F \bullet \mathcal{G}_F$. Hence, by F being well-formed, $(z, z') \notin \mathcal{P}_F$, as well. The case for $z' \in Y_{F'}$ is analogous. □

We prove next that well-formed interfaces support the incremental design of systems.

Theorem 3.1.7 (Incremental design)**.** *Let F, F' and F'' be interfaces. If $F \sim F'$ and $(F \otimes F') \sim F''$, then $F' \sim F''$ and $F \sim (F' \otimes F'')$.*

Proof. Consider arbitrary interfaces F, F' and F'', such that (i) $F \sim F'$; and (ii) $F \otimes F' \sim F''$. Note that from our initial assumptions (iii) all three interfaces have disjoint sets of output variables.

We start by proving that $F' \sim F''$. As noted in (iii), $Y_{F'} \cap Y_{F''} = \emptyset$, i.e. F' and F'' are composable. We are missing to prove that their assumptions are supported by the composite open-guarantees, i.e., $((\mathcal{A}_{F'} \cup \mathcal{A}_{F''}) \cap (Z_{F',F''} \times Y_{F',F''})) \subseteq \mathcal{G}_{F'\otimes F''}$. Let $(z, s) \in X_{F',F''} \times Y_{F',F''}$ and $(z, s) \in \mathcal{A}_{F'} \cup \mathcal{A}_{F''}$. Note that by (i) and (ii), $s \notin Y_F$. We want to prove that (z, s) is in the composite open-guarantee. As $s \notin Y_F$ and, by definition of composite assumptions, if $(z, s) \in \mathcal{A}_{F'}$, then $(z, s) \in \mathcal{A}_{F\otimes F'}$. Then, $(z, s) \in \mathcal{A}_{F\otimes F'} \cup \mathcal{A}_{F''}$ and, by the compatibility between the interfaces (ii), $(z, s) \in \mathcal{G}_{F\otimes F',F''}$. By monotonicity of composite open-guarantees, $\overline{\mathcal{G}}_{F'\otimes F''} \subseteq \overline{\mathcal{G}}_{F,F'\otimes F''}$ and, by their associativity (Lemma 3.1.4), $\overline{\mathcal{G}}_{F'\otimes F''} \subseteq \overline{\mathcal{G}}_{F\otimes F',F''}$. Then, by $(z, s) \notin \overline{\mathcal{G}}_{F\otimes F',F''}$, we have $(z, s) \notin \overline{\mathcal{G}}_{F'\otimes F''}$ and, so, $(z, s) \in \mathcal{G}_{F'\otimes F''}$.

We prove now that $F \sim F' \otimes F''$. From (iii), $Y_F \cap Y_{F' \otimes F''} = \emptyset$, i.e. F and $F' \otimes F''$ are composable. We are missing to prove that $((\mathcal{A}_F \cup \mathcal{A}_{F' \otimes F''}) \cap (X_{F,F' \otimes F''} \times Y_{F,F' \otimes F''})) \subseteq \mathcal{G}_{F,F' \otimes F''}$. Consider arbitrary $(z,s) \in X_{F,F' \otimes F''} \times Y_{F,F' \otimes F''}$ s.t. $(z,s) \in \mathcal{A}_F \cup \mathcal{A}_{F' \otimes F''}$. We prove that $(z,s) \in \mathcal{G}_{F,F' \otimes F''}$ by cases in the (z,s) domain.

We start with the case that $(z,s) \in \mathcal{A}_F$. Then, s is an input variable of F, $s \in X_F$, and, by definition of information-flow interfaces, $s \notin Y_F$. If $s \in Y_{F'}$, then, s is a shared variable between F and F' and, by their compatibility (assumption (i)), $(\star)$ $(z,s) \in \mathcal{G}_{F \otimes F'}$. Assume towards a contradiction that $(z,s) \notin \mathcal{G}_{F,F' \otimes F''}$. So, $(z,s) \in \overline{\mathcal{G}}_{F,F' \otimes F''}$. By associativity of composite open guarantees (Lemma 3.1.4), $(z,s) \in \overline{\mathcal{G}}_{F \otimes F',F''}$. By $s \in Y_{F \otimes F'}$ and Lemma 3.1.2, there exists a variable s' s.t. $(z,s) = (z,s') \cdot (s',s)$ with:

$$(\star\star)\ (z,s') \in (\mathrm{Id}_{Z_{F \otimes F',F''}} \cup \overline{\mathcal{G}}_{F''}) \circ (\overline{\mathcal{G}}_{F \otimes F'} \circ \overline{\mathcal{G}}_{F''})^* \text{ and } (s',s) \in \overline{\mathcal{G}}_{F \otimes F'}.$$

By $(z,s) \in \mathcal{A}_F$ and $(s',s) \in \mathcal{G}_{F \otimes F'}$, then $(z,s') \in \hat{\mathcal{A}}_{F,F'}$. If $(z,s') \in \mathrm{Id}_{Z_{F \otimes F',F''}}$, then $z = s'$ and $(z,s) \in \overline{\mathcal{G}}_{F \otimes F'}$, which contradicts $(\star)$. Otherwise, $s' \in Y_{F''}$ and, by the interfaces compatibility (assumption (ii)), then $s' \in X_{F \otimes F'}$. Then, by $(z,s') \in \hat{\mathcal{A}}_{F,F'}$ and definition of composition, $(z,s') \in \mathcal{A}_{F \otimes F'}$. As $s' \in X_{F \otimes F'} \cap Y_{F''}$ and by (ii), $(z,s') \in \mathcal{G}_{F \otimes F',F''}$, which contradicts $(\star\star)$. Hence $(z,s) \notin \overline{\mathcal{G}}_{F,F' \otimes F''}$, i.e. $(z,s) \in \mathcal{G}_{F,F' \otimes F''}$. For the case that $(z,s) \in \mathcal{A}_F$ and $s \in Y_{F''}$, by (ii), $s \in X_{F \otimes F'}$, $(z,s) \in \mathcal{A}_{F \otimes F'}$ and $(z,s) \in \mathcal{G}_{F \otimes F',F''}$. Then, by associativity of composite flows (Lemma 3.1.4), $(z,s) \in \mathcal{G}_{F,F' \otimes F''}$.

When $(z,s) \in \mathcal{A}_{F' \otimes F''}$, then, $s \in X_{F' \otimes F''}$ and we proceed by cases on the definition of composite assumption. Note that, by the sets of output variables being disjoint, we are only interested in the cases where $s \in Y_F$. If $(z,s) \in \mathcal{A}_{F'}$, then it is analogous to the previous case where $(z,s) \in \mathcal{A}_F$ and $s \in Y_{F'}$. If $(z,s) \in \mathcal{A}_{F''}$, then previous case where $(z,s) \in \mathcal{A}_F$ and $s \in Y_{F''}$. Otherwise, $(z,s) \in \hat{\mathcal{A}}_{F',F''}$, and, by monotonicity of derived assumptions (Lemma 3.1.5), $(z,s) \in \hat{\mathcal{A}}_{F \otimes F',F''}$. Then, by (ii) and $s \in Y_{F \otimes F',F''}$, $(z,s) \notin \overline{\mathcal{G}}_{F \otimes F',F''}$, i.e., $(z,s) \notin \overline{\mathcal{G}}_{F,F' \otimes F''}$ (Lemma 3.1.4). □

We go now a step further from incremental design of systems and prove that composition between well-formed interfaces is also associative. Note that incremental design only requires interfaces' compatibility for composition to be independent of the order we check for compatibility. By satisfying associativity, we guarantee additionally that we always get the same outcome no matter the order in which we compose the interfaces.

Theorem 3.1.8. *If $F \sim F'$ and $F \otimes F' \sim F''$, then $(F \otimes F') \otimes F'' = F \otimes (F' \otimes F'')$.*

Proof. Consider arbitrary interfaces F, F' and F''. Assume that $F \sim F'$ and $F \otimes F' \sim F''$. Then, by Theorem 3.1.7, $(\star)$ $F' \sim F''$ and $F \sim F' \otimes F''$. By definition of composition, $(\dagger)$ $X_{F \otimes F',F''} = X_{F,F' \otimes F''}$, $Y_{F \otimes F',F''} = Y_{F,F' \otimes F''}$, and $Z_{F \otimes F',F''} = Z_{F,F' \otimes F''}$. And, by Lemma 3.1.4, $\mathcal{G}_{F \otimes F',F''} = \mathcal{G}_{F,F' \otimes F''}$. Using our initial assumptions, $(\star)$, Lemma 3.1.5 and $(\dagger)$, it follows: $\mathcal{A}_{F \otimes F',F''} = \mathcal{A}_{F,F' \otimes F''}$. Then, $(\star\star)$ $\hat{\mathcal{P}}_{\mathcal{A}_{F \otimes F',F''},\mathcal{G}_{F \otimes F',F''}} = \hat{\mathcal{P}}_{\mathcal{A}_{F,F' \otimes F''},\mathcal{G}_{F,F' \otimes F''}}$.

And, by our initial assumptions and ($\star$), it follows:

$$\mathcal{P}_{F\otimes F',F''} = \mathcal{P}_F \cup \mathcal{P}_{F'} \cup \mathcal{P}_{F''} \cup \hat{\mathcal{P}}_{\mathcal{A}_{F'\otimes F''},\mathcal{P}_{F'\otimes F''}} \cup \hat{\mathcal{P}}_{\mathcal{A}_{F,F'\otimes F''},\mathcal{P}_{F,F'\otimes F''}} = \mathcal{P}_{F,F'\otimes F''}.$$

Note that by ($\star\star$) and definition of derived closed-guarantees:

$$\hat{\mathcal{P}}_{\mathcal{A}_{F'\otimes F''},\mathcal{P}_{F'\otimes F''}} \subseteq \hat{\mathcal{P}}_{\mathcal{A}_{F,F'\otimes F''},\mathcal{P}_{F,F'\otimes F''}}.$$ □

Example: SCI Top-down Design

We present a step-by-step top-down design of our first running example – the shared communication infrastructure – depicted in the Figure 3.8 below.

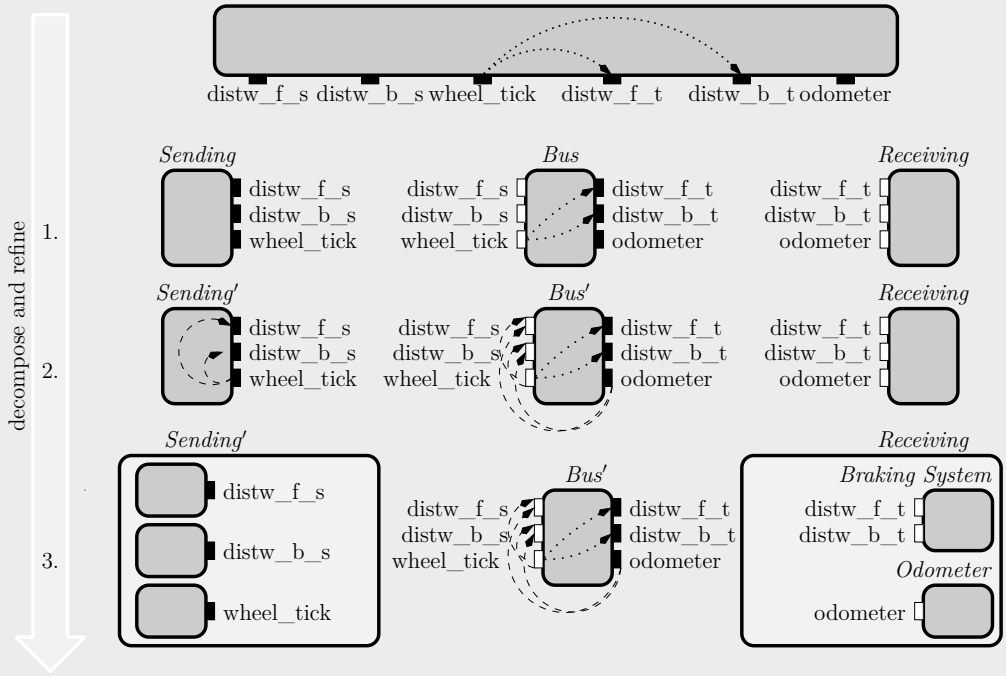


Figure 3.8: Top-down design of a shared communication infrastructure.

The top-most interface is a closed-interface (i.e., all of its variables are output variables) that specifies our main design goal: *information from the wheel sensor – wheel_tick– should not flow to the target of the distance warners – distw_b_t and distw_b_s.*

In the first design step, the designer splits the closed-system into three interfaces: one for the shared communication infrastructure – *Bus*; one for the components

that send information to the bus – *Sending*; and one for the components that receive information from the bus – *Receiving*. This is a naive split, as the designer just keeps the (open- and closed-) guarantees from the closed-system specification. However, because *Bus* interface specifies an open-system, assumptions are missing to support its closed-guarantee. Note that information from the wheel sensor can flow from one of the distance warners' targets by a flow from the wheel sensor to one of the distance warners' sources through the environment. Hence *Bus* is not a well-formed interface.

To fix this issue, identified by our framework, the designer adds new requirements to the *Bus* assumption, as shown in Bus' in the second refinement step. Now, for the *Sending* interface to be compatible with Bus', it needs to guarantee that there are no flows from the wheel sensor to the source of the distance warners. In the second design step, we add this requirement as an open-guarantee, specified in the interface $Sending'$.

The final step splits the $Sending'$ and *Receiving* into their functional roles. Note that if we compose the three interfaces at the sending side, our composition operator derives that there are no information flows between all output variables. Using the refinement relation that we will define in the next section, we infer that the composition of the interfaces inside the $Sending'$ box in the third step is indeed a refinement of $Sending'$ in the second step.

3.1.2 Refinement and Independent Implementability

Intuitively, an interface F' refines an interface F, denoted $F' \preceq F$, when F' admits all environments of F (and maybe more) while guaranteeing a subset of F's implementations. As we specify information-flow interfaces with no-flow relations and components with flow relations, then F' admits a superset of F's environments if the assumption of F' is a subset of F's assumption (i.e., the subset relation direction is reversed).

Definition 3.1.9. *Interface* $F' = (X, Y, \mathcal{A}', \mathcal{G}', \mathcal{P}')$ refines $F = (X, Y, \mathcal{A}, \mathcal{G}, \mathcal{P})$*, written* $F' \preceq F$*, when* $\mathcal{A}_{F'} \subseteq \mathcal{A}_F$*,* $\mathcal{G}_F \subseteq \mathcal{G}_{F'}$ *and* $\mathcal{P}_F \subseteq \mathcal{P}_{F'}$*.*

By definition of implementations and refinement, it follows directly that for all components f that implement refinements of F, $F' \preceq F$ and $f \models'_F$, then they are an implementation of F, $f \models_F$, too. And, likewise, for permissible environments: for all components $f_\mathcal{E}$ that are permissible environments of F, $f_\mathcal{E} \models_F$, and all of F's refinements, $F' \preceq F$, then the component $f_\mathcal{E}$ is permissible environment of F', $f_\mathcal{E} \models'_F$, too.

We prove below that the refinement relation is preserved during composition, i.e., they support independent implementability of systems.

Theorem 3.1.9 (Independent implementability)**.** *For all well-formed interfaces* F'_1*,* F_1 *and* F_2*, if* $F'_1 \preceq F_1$ *and* $F_1 \sim F_2$*, then* $F'_1 \sim F_2$ *and* $F'_1 \otimes F_2 \preceq F_1 \otimes F_2$*.*

Proof. Consider arbitrary interfaces F_1, F_1' and F_2. Assume that $F_1' \preceq F_1$ and $F_1 \sim F_2$. By $F_1' \preceq F_1$: (i) $\mathcal{A}_{F_1'} \subseteq \mathcal{A}_{F_1}$; (ii) $\mathcal{G}_{F_1} \subseteq \mathcal{G}_{F_1'}$; and (iii) $\mathcal{P}_{F_1} \subseteq \mathcal{P}_{F_1'}$.

By (i), $(\mathcal{A}_{F_1'} \cup \mathcal{A}_{F_2}) \subseteq (\mathcal{A}_{F_1} \cup \mathcal{A}_{F_2})$. By (ii), $\overline{\mathcal{G}}_{F_1'} \subseteq \overline{\mathcal{G}}_{F_1}$ and, by definition of no-flows composition, $\mathcal{G}_{F_1'} \bullet \mathcal{G}_{F_2} \subseteq \mathcal{G}_{F_1} \bullet \mathcal{G}_{F_2}$. So by definition of composite open-guarantees, $\mathcal{G}_{F_1 \otimes F_2} \subseteq \mathcal{G}_{F_1' \otimes F_2}$. So, by $F_1 \sim F_2$, it follows $((\mathcal{A}_{F_1'} \cup \mathcal{A}_{F_2}) \cap (X_{F_1,F_2} \times Y_{F_1,F_2})) \subseteq \mathcal{G}_{F_1' \otimes F_2}$. Hence $F_1' \sim F_2$.

We prove now that $F_1' \otimes F_2 \preceq F_1 \otimes F_2$. First, we note that above we prove that $(\star)$ $\mathcal{G}_{F_1 \otimes F_2} \subseteq \mathcal{G}_{F_1' \otimes F_2}$. Then, by (i) and definition of derived assumptions, $\hat{\mathcal{A}}_{F_1',F_2} \subseteq \hat{\mathcal{A}}_{F_1,F_2}$. So, $(\star\star)$ $\mathcal{A}_{F_1' \otimes F_2} \subseteq \mathcal{A}_{F_1 \otimes F_2}$. We are only missing to prove that $\mathcal{P}_{F_1 \otimes F_2} \subseteq \mathcal{P}_{F_1' \otimes F_2}$. Consider arbitrary $(z,y) \in \mathcal{P}_{F_1 \otimes F_2}$. If $(z,y) \in \mathcal{P}_{F_1} \cup \mathcal{P}_{F_2}$, then, by (iii), $(z,y) \in \mathcal{P}_{F_1'} \cup \mathcal{P}_{F_2}$. If $(z,y) \in \hat{\mathcal{P}}_{\mathcal{A}_{F_1 \otimes F_2}, \mathcal{G}_{F_1 \otimes F_2}}$, then, by definition of derived closed-guarantees: (i) $(z,y) \in \mathcal{G}_{F_1 \otimes F_2}$, and (ii) $(z,y) \notin (\mathrm{Id}_{Z_{F_1,F_2}} \cup \overline{\mathcal{A}}_{F_1 \otimes F_2}) \circ (\overline{\mathcal{G}}_{F_1 \otimes F_2} \circ \overline{\mathcal{A}}_{F_1 \otimes F_2})^* \circ \overline{\mathcal{G}}_{F_1 \otimes F_2}$. We showed earlier that $\mathcal{G}_{F_1 \otimes F_2} \subseteq \mathcal{G}_{F_1' \otimes F_2}$, then $(z,y) \in \mathcal{G}_{F_1' \otimes F_2}$, as well. We want to prove that if (z,y) is not in the flows of the closed system defined by $F_1 \otimes F_2$, then changing F_1 with one of its refinements will not change that. Formally, we prove by induction that for all $n \in \mathbb{N}$ and $(z,y) \in \mathcal{G}_{F_1 \otimes F_2}$:

$$\begin{aligned} &\text{if } (z,y) \notin (\mathrm{Id} \cup \overline{\mathcal{A}}_{F_1 \otimes F_2}) \circ (\overline{\mathcal{G}}_{F_1 \otimes F_2} \circ \overline{\mathcal{A}}_{F_1 \otimes F_2})^n \circ \overline{\mathcal{G}}_{F_1 \otimes F_2} \\ &\text{then } (z,y) \notin (\mathrm{Id} \cup \overline{\mathcal{A}}_{F_1' \otimes F_2}) \circ (\overline{\mathcal{G}}_{F_1' \otimes F_2} \circ \overline{\mathcal{A}}_{F_1' \otimes F_2})^n \circ \overline{\mathcal{G}}_{F_1' \otimes F_2}. \end{aligned}$$

We start with the *base case* $n = 0$. Consider arbitrary $(z,y) \in \mathcal{G}_{F_1 \otimes F_2}$ s.t. $(z,y) \notin (\overline{\mathcal{A}}_{F_1 \otimes F_2} \circ \overline{\mathcal{G}}_{F_1 \otimes F_2}) \cup \overline{\mathcal{G}}_{F_1 \otimes F_2}$. If $(z,y) \notin \overline{\mathcal{G}}_{F_1 \otimes F_2}$, then, by $(\star)$, the statement holds. If $(z,y) \notin (\overline{\mathcal{A}}_{F_1 \otimes F_2} \circ \overline{\mathcal{G}}_{F_1 \otimes F_2})$, then, for all $(z,s) \in \overline{\mathcal{A}}_{F_1 \otimes F_2}$ we have $(s,y) \notin \overline{\mathcal{G}}_{F_1 \otimes F_2}$ (i.e., $(s,y) \in \mathcal{G}_{F_1 \otimes F_2}$). Then, by $(\star)$ and $(\star\star)$, for all $(z,s) \in \overline{\mathcal{A}}_{F_1' \otimes F_2}$ we have $(s,y) \notin \overline{\mathcal{G}}_{F_1' \otimes F_2}$. Thus, $(z,y) \notin \overline{\mathcal{A}}_{F_1' \otimes F_2} \circ \overline{\mathcal{G}}_{F_1' \otimes F_2}$, as well.

For the *induction step*, we assume as induction hypothesis (IH) that the statement holds for n. Let $(z,y) \notin (\mathrm{Id} \cup \overline{\mathcal{A}}_{F_1 \otimes F_2}) \circ (\overline{\mathcal{G}}_{F_1 \otimes F_2} \circ \overline{\mathcal{A}}_{F_1 \otimes F_2})^{n+1} \circ \overline{\mathcal{G}}_{F_1 \otimes F_2}$. By (IH), $(z,y) \notin (\mathrm{Id} \cup \overline{\mathcal{A}}_{F_1' \otimes F_2}) \circ (\overline{\mathcal{G}}_{F_1' \otimes F_2} \circ \overline{\mathcal{A}}_{F_1' \otimes F_2})^n \circ \overline{\mathcal{G}}_{F_1' \otimes F_2} \circ \overline{\mathcal{A}}_{F_1 \otimes F_2} \circ \overline{\mathcal{G}}_{F_1 \otimes F_2}$. Then, for all $(z,s) \in (\mathrm{Id} \cup \overline{\mathcal{A}}_{F_1' \otimes F_2}) \circ (\overline{\mathcal{G}}_{F_1' \otimes F_2} \circ \overline{\mathcal{A}}_{F_1' \otimes F_2})^n \circ \overline{\mathcal{G}}_{F_1' \otimes F_2}$ we have $(s,y) \notin \overline{\mathcal{A}}_{F_1 \otimes F_2} \circ \overline{\mathcal{G}}_{F_1 \otimes F_2}$. By the same reasoning applied to the base case, then for all $(z,s) \in (\mathrm{Id} \cup \overline{\mathcal{A}}_{F_1' \otimes F_2}) \circ (\overline{\mathcal{G}}_{F_1' \otimes F_2} \circ \overline{\mathcal{A}}_{F_1' \otimes F_2})^n \circ \overline{\mathcal{G}}_{F_1' \otimes F_2}$ with $(s,y) \notin \overline{\mathcal{A}}_{F_1' \otimes F_2} \circ \overline{\mathcal{G}}_{F_1' \otimes F_2}$. Thus, $(z,y) \notin (\mathrm{Id} \cup \overline{\mathcal{A}}_{F_1' \otimes F_2}) \circ (\overline{\mathcal{G}}_{F_1' \otimes F_2} \circ \overline{\mathcal{A}}_{F_1' \otimes F_2})^{n+1} \circ \overline{\mathcal{G}}_{F_1' \otimes F_2}$. Hence by definition of derived closed-guarantees, $\mathcal{P}_{F_1 \otimes F_2} \subseteq \mathcal{P}_{F_1' \otimes F_2}$. And, by definition of refinement, $F_1' \otimes F_2 \preceq F_1 \otimes F_2$. □

Example: CAN bus Bottom-up Verification

We illustrate in Figure 3.9 how our framework can aid the bottom-up verification of the communication between the immobilizer and the ECU using a CAN bus. We start by extracting four components: for the key storage implementation – f_{key}; for the ECU functionality related to the immobilizer – f_{ecu}; for the immobilizer – f_{imm}; and for the CAN bus – f_{can}.

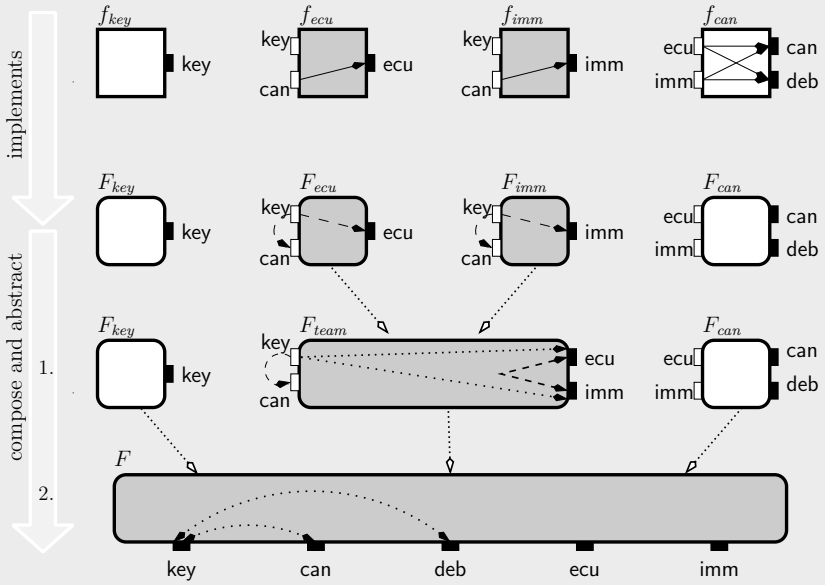


Figure 3.9: Bottom-up verification of a CAN bus implementation.

In this example, the key storage solution and the CAN bus are provided by third parties (depicted with white boxes). We assume that they have all possible flows between their variables. In contrast, both the ECU and the immobilizer are implemented in-house. We derive, using other tools like model checking or simulation, that in current implementations information only flows from the input can to both the ecu and the imm output variables. The designer specifies the interfaces F_{ecu} and F_{imm} with the respective components implementing them.

After specifying the information-flow interfaces for each component, we use our composition operator to compose the two interfaces specifying in-house components, defining the interface F_{team}. We are then able to identify four no-flows of the open system specified by F_{team}, with two of them remaining no-flows in the closed system. Finally, in the last step, we compose the remaining three interfaces using our framework to guarantee that our in-house implementation is compatible with any possible implementation of the key storage and the CAN bus, with the guarantee that there will be no information-flow from key to can and deb.

3.1.3 Shared Refinement

We introduce a *shared refinement operator* that supports the *shared implementability of interfaces*: the same interface may specify different parts of a design or even different parts from different designs. The goal is to allow the seamless reuse of implementations. Two interfaces, F and F', are *shared refinable* if an interface that refines them both exist. A shared interface between F and F' must work on all the environments they permit, while their implementations must satisfy both interfaces' guarantees. The shared refinement operator computes the greatest lower bound in the lattice defined by the refinement relation on information-flow interfaces.

To guarantee that a shared refinement between two information-flow interfaces F and F' can be placed in any of their environments, the shared interface assumption must be a subset of the intersection of F and F' assumptions. As the shared interface must satisfy all requirements on F and F' guarantees, it may be the case that the shared interface misses a pair in its assumption that is necessary to satisfy one of the no-flows in the closed-guarantee of either F or F'. For this reason, we introduce *derived open-guarantees*: for each no-flow pair (z, x) in the assumption of either F and F' that cannot be in their shared interface and is necessary to satisfy a no-flow (x, y) requirement on the closed systems, we derive the no-flow (x, y) requirement over implementations.

Definition 3.1.10. *Let F and F' be two interfaces such that $X_F = X_{F'}$ and $Y_F = Y_{F'}$. The* derived open-guarantee *of F and F' is:*

$$\hat{\mathcal{G}}_{F,F'} = \{(x, y) \mid (z, x) \notin \mathcal{A}_F \cap \mathcal{A}_{F'} \text{ and } (z, y) \in \mathcal{P}_F \cup \mathcal{P}_{F'}\}.$$

The shared refinement of F *and* F'*, denoted $F \sqcap F'$, is:*

$$F \sqcap F' = (X_F, Y_F, \mathcal{A}_F \cap \mathcal{A}_{F'}, \mathcal{G}_F \cup \mathcal{G}_{F'} \cup \hat{\mathcal{G}}_{F,F'}, \mathcal{P}_F \cup \mathcal{P}_{F'}).$$

We prove that if interfaces are well-formed, then their shared refinement is well-formed, as well. Additionally, we show that the shared refinement between two interfaces is the most abstract well-formed interface refining the given interfaces.

Theorem 3.1.10. *Let F, F' and F' be well-formed interfaces. $F \sqcap F'$ is a well-formed interface; and if $F'' \preceq F$ and $F'' \preceq F'$, then $F'' \preceq F \sqcap F'$.*

Proof. Consider arbitrary well-formed interfaces F, F' and F''. We start by proving that $F \sqcap F'$ is a well-formed interface. By definition of shared refinement:

$$(\star)\ \overline{\mathcal{G}}_{F \sqcap F'} \subseteq \overline{\mathcal{G}}_F \cap \overline{\mathcal{G}}_{F'} \text{ and } \overline{\mathcal{A}}_F \cup \overline{\mathcal{A}}_{F'} = \overline{\mathcal{A}}_{F \sqcap F'}.$$

Consider arbitrary $(z, y) \in \mathcal{P}_{F \sqcap F'}$. We start with the case that $(z, y) \in \mathcal{P}_F$. Assume towards a contradiction that: $(z, y) \in ((\mathrm{Id}_Z \cup \overline{\mathcal{A}}_{F \sqcap F'}) \circ (\overline{\mathcal{G}}_{F \sqcap F'} \circ \overline{\mathcal{A}}_{F \sqcap F'})^* \circ \overline{\mathcal{G}}_{F \sqcap F'})$. We prove next that $(z, y) \in ((\mathrm{Id}_Z \cup \overline{\mathcal{A}}_F) \circ (\overline{\mathcal{G}}_F \circ \overline{\mathcal{A}}_F)^* \circ \overline{\mathcal{G}}_F)$, which contradicts our initial

assumption that F is well-formed. Specifically, we prove the following statement about alternated paths, for all well-formed interfaces F with $(z, y) \in \mathcal{P}_F$, and for all $n \in \mathbb{N}$:

$$\begin{aligned} &\text{if } (z,y) \in ((\mathrm{Id}_Z \cup \overline{\mathcal{A}}_{F\sqcap F'}) \circ (\overline{\mathcal{G}}_{F\sqcap F'} \circ \overline{\mathcal{A}}_{F\sqcap F'})^n \circ \overline{\mathcal{G}}_{F\sqcap F'}),\\ &\text{then } (z,y) \in ((\mathrm{Id}_Z \cup \overline{\mathcal{A}}_{F}) \circ (\overline{\mathcal{G}}_{F} \circ \overline{\mathcal{A}}_{F})^n \circ \overline{\mathcal{G}}_{F}). \end{aligned}$$

For the base case ($n = 0$), we assume that $(z, y) \in \overline{\mathcal{G}}_{F\sqcap F'} \cup (\overline{\mathcal{A}}_{F\sqcap F'} \circ \overline{\mathcal{G}}_{F\sqcap F'})$.

If $(z, y) \in \overline{\mathcal{G}}_{F\sqcap F'}$, then by $(\star)$, $(z, y) \in \overline{\mathcal{G}}_F$.

If $(z, y) \in \overline{\mathcal{A}}_{F\sqcap F'} \circ \overline{\mathcal{G}}_{F\sqcap F'}$, then we consider arbitrary $(z, s) \in \overline{\mathcal{A}}_{F\sqcap F'}$ with $(s, y) \in \overline{\mathcal{G}}_{F\sqcap F'}$. Then, $(s, y) \in \overline{\mathcal{G}}_F$. By our assumption that F is well-formed and $(z, y) \in \mathcal{P}_F$, then, by definition of derived open-guarantees, $(s, y) \in \hat{\mathcal{G}}_{F,F'}$. Thus, by definition of shared refinement, it cannot be the case that $(s, y) \in \overline{\mathcal{G}}_{F\sqcap F'}$, and so $(z, y) \notin \overline{\mathcal{A}}_{F\sqcap F'} \circ \overline{\mathcal{G}}_{F\sqcap F'}$.

For the induction step, we assume as induction hypothesis the statement to hold for n. Consider arbitrary $(z, y) \in ((\mathrm{Id}_Z \cup \overline{\mathcal{A}}_{F\sqcap F'}) \circ (\overline{\mathcal{G}}_{F\sqcap F'} \circ \overline{\mathcal{A}}_{F\sqcap F'})^{n+1} \circ \overline{\mathcal{G}}_{F\sqcap F'})$. By induction hypothesis: $(z, y) \in ((\mathrm{Id}_Z \cup \overline{\mathcal{A}}) \circ (\overline{\mathcal{G}} \circ \overline{\mathcal{A}})^n \circ \overline{\mathcal{G}} \circ \overline{\mathcal{A}}_{F\sqcap F'} \circ \overline{\mathcal{G}}_{F\sqcap F'})$. Then, we can prove analogously to the base case that for all $(z, s) \in ((\mathrm{Id}_Z \cup \overline{\mathcal{A}}_F) \circ (\overline{\mathcal{G}} \circ \overline{\mathcal{A}}_F)^n \circ \overline{\mathcal{G}}_F$ and $(s, y) \in \overline{\mathcal{A}}_{F\sqcap F'} \circ \overline{\mathcal{G}}_{F\sqcap F'}$, we have $(s, y) \in \overline{\mathcal{A}}_F \circ \overline{\mathcal{G}}_F$.

We prove analogously for the case that $(z, y) \in \mathcal{P}_{F'}$.

We prove now that $F'' \preceq F \sqcap F'$. Consider arbitrary F'' s.t. $F'' \preceq F$ and $F'' \preceq F'$. Then, by definition of refinement: (i) $\mathcal{A}_{F''} \subseteq \mathcal{A}_F$ and $\mathcal{A}_{F''} \subseteq \mathcal{A}'_F$; and (ii) $\mathcal{P}_F \subseteq \mathcal{P}_{F''}$ and $\mathcal{P}_{F'} \subseteq \mathcal{P}_{F''}$. Then, $\mathcal{A}_{F''} \subseteq \mathcal{A}_F \cap \mathcal{A}_{F'}$ and $\mathcal{P}_F \cup \mathcal{P}_{F'} \subseteq \mathcal{P}_{F''}$. We are missing to prove that $\mathcal{G}_F \cup \mathcal{G}_{F'} \cup \hat{\mathcal{G}}_{F,F'} \subseteq \mathcal{G}_{F''}$.

Assume towards a contradiction that there exists $(z, y) \in \mathcal{G}_F \cup \mathcal{G}_{F'} \cup \hat{\mathcal{G}}_{F,F'}$ s.t. $(z, y) \notin \mathcal{G}_{F''}$.

If $(z, y) \in \mathcal{G}_F \cup \mathcal{G}_{F'}$ then, by $F'' \preceq F$ and $F'' \preceq F'$, $(z, y) \in \mathcal{G}_{F''}$. This is a contradiction.

Consider the case that $(z, y) \in \hat{\mathcal{G}}_{F,F'}$. Then, by definition of derived open-guarantees, there exists $(z', z) \notin \mathcal{A}_F \cap \mathcal{A}_{F'}$ s.t. $(z', y) \in \mathcal{P}_F$. By $F'' \preceq F$, $(z', y) \in \mathcal{P}_{F''}$ and $(z', z) \notin \mathcal{A}_{F''}$. Then, by $(z', z) \in \overline{\mathcal{A}}_{F''}$ and $(z, y) \in \overline{\mathcal{G}}_{F''}$, $(z', y) \in \overline{\mathcal{A}}_{F''} \circ \overline{\mathcal{G}}_{F''}$. This contradicts our assumption that F'' is well-formed because $(z', y) \in \mathcal{P}_{F''}$. Hence $F'' \preceq F \sqcap F'$. □

3.2 Stateful

In this section, we extend our theory to allow dynamic changes in the requirements of secure information flow. In particular, we introduce stateful information-flow interfaces and components, which are transition systems where each state is a stateless component or interface, respectively.

Definition 3.2.1. *Let X and Y be disjoint sets of* input *and* output *variables, respectively, with $Z = X \cup Y$ the set of all variables. Let Q be a set of states with $\hat{q} \in Q$ being the initial state and $\delta : Q \to 2^Q$ be a transition relation. A* stateful information-flow component

$\mathbb{f}$ *is a tuple* $(X, Y, Q, \hat{q}, \delta, \mathbb{M})$*, where* $\mathbb{M} : Q \to 2^{Z \times Y}$ *is a state labeling such that for all states* $q \in Q$*,* $\mathbb{M}(q)$ *defines a flow relation. We denote by* $\mathbb{f}(q) = (X, Y, \mathbb{M}(q))$ *the stateless component implied by the labeling of* q*. A* stateful information-flow interface $\mathbb{F}$ *is a tuple* $(X, Y, Q, \hat{q}, \delta, \mathbb{A}, \mathbb{G}, \mathbb{P})$*, where* $\mathbb{A} : Q \to 2^{Z \times X}$ *is called* assumption*;* $\mathbb{G} : Q \to 2^{Z \times Y}$ *is called* open-guarantee*; and* $\mathbb{P} : Q \to 2^{Z \times Y}$ *is called* closed-guarantee*. For each state* $q \in Q$ *we denote by* $\mathbb{F}(q) = (X, Y, \mathbb{A}(q), \mathbb{G}(q), \mathbb{P}(q))$ *the stateless interface defined by the assumption, open-guarantee and closed-guarantee of* q*.*

A *stateful interface* $\mathbb{F}$ is well-formed iff $\mathbb{F}(\hat{q})$ is a well-formed stateless interface, and for all states $q \in Q$ reachable from the intial state $\hat{q}$ the stateless interface defined by the state q, $\mathbb{F}(q)$, is well-formed. In what follows, $\mathbb{F} = (X, Y, Q, \hat{q}, \delta, \mathbb{A}, \mathbb{G}, \mathbb{P})$ and $\mathbb{F}' = (X', Y', Q', \hat{q}', \delta', \mathbb{A}', \mathbb{G}', \mathbb{P}')$ are stateful interfaces, and $\mathbb{f} = (X, Y, Q_{\mathbb{f}}, \hat{q}_{\mathbb{f}}, \delta_{\mathbb{f}}, \mathbb{M})$ and $\mathbb{f}_{\mathcal{E}} = (Y, X, Q_{\mathcal{E}}, \hat{q}_{\mathcal{E}}, \delta_{\mathcal{E}}, \mathbb{E})$ are stateful components.

We say that a stateful component $\mathbb{f}$ implements a stateful interface $\mathbb{F}$ if there exists a simulation relation from $\mathbb{f}$ to $\mathbb{F}$, where each stateless component implements the stateless interface they are related to by the simulation relation. As for permissible environments, we require a simulation relation from the interface they are permissible on to the stateful component we want to check if it is a permissible environment.

Definition 3.2.2. *Let* $\mathbb{F} = (X, Y, Q, \hat{q}, \delta, \mathbb{A}, \mathbb{G}, \mathbb{P})$ *be a stateful information-flow interface. A component* $\mathbb{f} = (X, Y, Q_{\mathbb{f}}, \hat{q}_{\mathbb{f}}, \delta_{\mathbb{f}}, \mathbb{M})$ implements *the interface* $\mathbb{F}$*, denoted by* $\mathbb{f} \models_{\mathbb{G}} \mathbb{F}$*, iff there exists* $H \subseteq Q_{\mathbb{f}} \times Q$ *s.t.* $(\hat{q}_{\mathbb{f}}, \hat{q}) \in H$ *and for all* $(q_{\mathbb{f}}, q) \in H$*:*

- $\mathbb{f}(q_{\mathbb{f}}) \models_{\mathbb{G}(q)} \mathbb{F}(q)$ *with* $\mathbb{F}(q) = (X, Y, \mathbb{A}(q), \mathbb{G}(q), \mathbb{P}(q))$*; and*
- *if* $q'_{\mathbb{f}} \in \delta_{\mathbb{f}}(q_{\mathbb{f}})$*, then there exists a state* $q' \in \delta(q)$ *s.t.* $(q'_{\mathbb{f}}, q') \in H$*.*

A component $\mathbb{f}_{\mathcal{E}} = (Y, X, Q_{\mathcal{E}}, \hat{q}_{\mathcal{E}}, \delta_{\mathcal{E}}, \mathbb{E})$ *is an* permissible environment *for the interface* $\mathbb{F}$*, denoted by* $\mathbb{f}_{\mathcal{E}} \models_{\mathbb{A}} \mathbb{F}$*, iff there exists a relation* $H \subseteq Q \times Q_{\mathcal{E}}$ *s.t.* $(\hat{q}, \hat{q}_{\mathcal{E}}) \in H$ *and for all* $(q, q_{\mathcal{E}}) \in H$*:*

- $\mathbb{f}(q_{\mathcal{E}}) \models_{\mathbb{A}(q)} \mathbb{F}(q)$ *with* $\mathbb{F}(q) = (X, Y, \mathbb{A}(q), \mathbb{G}(q), \mathbb{P}(q))$*; and*
- *if* $q' \in \delta_{\mathbb{F}}(q)$*, then there exists a state* $q'_{\mathcal{E}} \in \delta_{\mathcal{E}}(q_{\mathcal{E}})$ *s.t.* $(q', q'_{\mathcal{E}}) \in H$*.*

As for stateless interfaces, a well-formed stateful interface guarantees that its closed-guarantee holds under the composition between any of its implementations $\mathbb{f}$ with any of its permissible environments $\mathbb{f}_{\mathcal{E}}$.

Proposition 3.2.1. *For all well-formed interfaces* $\mathbb{F}$*, and all relations* $H \subseteq Q_{\mathbb{f}} \times Q$ *and* $H_{\mathcal{E}} \subseteq Q \times Q_{\mathcal{E}}$ *that witness* $\mathbb{f} \models_{\mathbb{G}} \mathbb{F}$ *and* $\mathbb{f}_{\mathcal{E}} \models_{\mathbb{A}} \mathbb{F}$*, respectively, it holds:*

(a) $(\mathbb{M}(\hat{q}_{\mathbb{f}}) \cup \mathbb{E}(\hat{q}_{\mathcal{E}}))^* \cap \mathbb{P}(\hat{q}) = \emptyset$*; and*

(b) for all $q \in Q$ *that are reachable from* $\hat{q}$*, if* $(q_{\mathbb{f}}, q) \in H$ *and* $(q, q_{\mathcal{E}}) \in H_{\mathcal{E}}$*, then* $(\mathbb{M}(q_{\mathbb{f}}) \cup \mathbb{E}(q_{\mathcal{E}}))^* \cap \mathbb{P}(q) = \emptyset$*.*

Proof. Consider arbitrary well-formed interface $\mathbb{F} = (X, Y, Q, \hat{q}, \delta, \mathbb{A}, \mathbb{G}, \mathbb{P})$, and components $\mathbb{f} = (X, Y, Q_\mathbb{f}, \hat{q}_\mathbb{f}, \delta_\mathbb{f}, \mathbb{M})$ and $\mathbb{f}_\mathcal{E} = (Y, X, Q_\mathcal{E}, \hat{q}_\mathcal{E}, \delta_\mathcal{E}, \mathbb{E})$. We assume that (i) $\mathbb{f} \models_\mathbb{G} \mathbb{F}$ and $H \subseteq Q_\mathbb{f} \times Q$ witnesses it; and (ii) $\mathbb{f}_\mathcal{E} \models_\mathbb{A} \mathbb{F}$ and $H_\mathcal{E} \subseteq Q \times Q_\mathcal{E}$ is a relation witnessing it. Item (a) follows from Proposition 3.1.1 for stateless interfaces. For the item (b), consider arbitrary state $q \in Q$ that is reachable from the initial state $\hat{q}$. Additionally, consider arbitrary $q_\mathbb{f}$ and $q_\mathcal{E}$ s.t. $(q_\mathbb{f}, q) \in H$ and $(q, q_\mathcal{E}) \in H_\mathcal{E}$. By our initial assumptions (i) and (ii), $\mathbb{f}(q_\mathbb{f}) \models_{\mathbb{G}(q)} \mathbb{F}(q)$ and $\mathbb{f}_\mathcal{E}(q_\mathcal{E}) \models_{\mathbb{A}(q)} \mathbb{F}(q)$. By $\mathbb{F}$ being well-formed and by q being accessible from the initial state $\hat{q}$, then $\mathbb{F}(q)$ is a well-formed (stateless) interface. Hence, by Proposition 3.1.1 for stateless interfaces, it follows that $(\mathbb{M}(q_\mathbb{f}) \cup \mathbb{E}(q_\mathcal{E}))^* \cap \mathbb{P}(q) = \emptyset$. □

3.2.1 Composition and Incremental Design

The composition between stateful interfaces or components is simply the synchronous product between their states.

Definition 3.2.3. *Let* $\mathbb{F} = (X, Y, Q, \hat{q}, \delta, \mathbb{A}, \mathbb{G}, \mathbb{P})$ *and* $\mathbb{F} = (X', Y', Q', \hat{q}', \delta', \mathbb{A}', \mathbb{G}', \mathbb{P}')$ *be stateful information-flow interfaces. Their* composition *is defined as the tuple:*

$$\mathbb{F} \otimes_{\text{full}} \mathbb{F}' = (X_{\mathbb{F},\mathbb{F}'}, Y_{\mathbb{F},\mathbb{F}'}, Q_{\mathbb{F},\mathbb{F}'}, \hat{q}_{\mathbb{F},\mathbb{F}'}, \delta_{\mathbb{F},\mathbb{F}'}, \mathbb{A}_{\mathbb{F},\mathbb{F}'}, \mathbb{G}_{\mathbb{F},\mathbb{F}'}, \mathbb{P}_{\mathbb{F},\mathbb{F}'})$$

where: $\hat{q}_{\mathbb{F},\mathbb{F}'} = (\hat{q}, \hat{q}')$ *and* $Q_{\mathbb{F},\mathbb{F}'} = \{\hat{q}_{\mathbb{F},\mathbb{F}'}\} \cup \{(q, q') \mid \mathbb{F}(q) \sim \mathbb{F}'(q')\}$*;* $(q_2, q_2') \in \delta_{\mathbb{F},\mathbb{F}'}(q_1, q_1')$ *iff* $q_2 \in \delta(q_1)$ *and* $q_2' \in \delta'(q_1')$*; assumption and guarantees are defined by the stateless composition of their respective states, formally for all* $(q, q') \in Q_{\mathbb{F},\mathbb{F}'}$ *we have* $(X_{\mathbb{F},\mathbb{F}'}, Y_{\mathbb{F},\mathbb{F}'}, \mathbb{A}(q, q'), \mathbb{G}(q, q'), \mathbb{P}(q, q')) = \mathbb{F}(q) \otimes \mathbb{F}'(q')$.

As we proved before for stateless interfaces in Proposition 3.1.3, we prove below that the composition of stateful information-flow interfaces preserves all the implementations of the interfaces being composed. The proof below exemplifies how to define the relation witnessing the composite implementation by combining the relations of the composed interfaces.

Proposition 3.2.2. *Let* $\mathbb{F}$ *and* $\mathbb{F}'$ *be stateful information-flow interfaces with open-guarantees* $\mathbb{G}$ *and* $\mathbb{G}'$*, respectively. If* $\mathbb{f} \models_\mathbb{G} \mathbb{F}$ *and* $\mathbb{f}' \models_\mathbb{G} \mathbb{F}'$*, then* $\mathbb{f} \otimes \mathbb{f}' \models_{\mathbb{G}_{\mathbb{F},\mathbb{F}'}} \mathbb{F} \otimes_{\text{full}} \mathbb{F}'$*, where* $\mathbb{G}_{\mathbb{F},\mathbb{F}'}$ *is the composite open-guarantee of* $\mathbb{F}$ *and* $\mathbb{F}'$.

Proof. Assume that: (i) $\mathbb{f} \models_\mathbb{G} \mathbb{F}$ and (ii) $\mathbb{f}' \models_\mathbb{G} \mathbb{F}'$. Then, there exists $H_\mathbb{f}$ and $H_{\mathbb{f}'}$ that witnesses (i) and (ii), respectively. Consider the relation: $H = \{((q_\mathbb{f}, q_{\mathbb{f}'}), (q_\mathbb{F}, q_{\mathbb{F}'})) \mid q_\mathbb{F} \in H_\mathbb{f}(q_\mathbb{f}) \text{ and } q_{\mathbb{F}'} \in H_{\mathbb{f}'}(q_{\mathbb{f}'})\}$. Clearly, by (i) and (ii), $((\hat{q}_\mathbb{f}, \hat{q}_{\mathbb{f}'}), (\hat{q}_\mathbb{F}, \hat{q}_{\mathbb{F}'})) \in H$. Then, $\mathbb{f}(\hat{q}_\mathbb{f}) \models_{\mathbb{G}(\hat{q}_\mathbb{F})} \mathbb{F}(\hat{q}_\mathbb{F})$ and $\mathbb{f}(\hat{q}_{\mathbb{f}'}) \models_{\mathbb{G}'(\hat{q}_{\mathbb{F}'})} \mathbb{F}'(\hat{q}_{\mathbb{F}'})$. So, by Proposition 3.1.3 for stateless interfaces, it follows that $\mathbb{f}(\hat{q}_\mathbb{f}) \otimes \mathbb{f}'(\hat{q}_{\mathbb{f}'}) \models_{\mathbb{G}_{\mathbb{F},\mathbb{F}'}(\hat{q}_{\mathbb{F},\mathbb{F}'})} \mathbb{F}(\hat{q}_\mathbb{F}) \otimes \mathbb{F}'(\hat{q}_{\mathbb{F}'})$. Consider arbitrary $((q_\mathbb{f}, q_{\mathbb{f}'}), (q_\mathbb{F}, q_{\mathbb{F}'})) \in H$. Then, by (i) and (ii), there exists $(\overline{q}_\mathbb{f}, \overline{q}_\mathbb{F}) \in H_\mathbb{f}$ s.t. $\mathbb{f}(\overline{q}_\mathbb{f}) \models_{\mathbb{G}(\overline{q}_\mathbb{F})} \mathbb{F}(\overline{q}_\mathbb{F})$, and there exists $(\overline{q}_{\mathbb{f}'}, \overline{q}_{\mathbb{F}'}) \in H_{\mathbb{f}'}$ s.t. $\mathbb{f}(\overline{q}_{\mathbb{f}'}) \models_{\mathbb{G}(\overline{q}_{\mathbb{F}'})} \mathbb{F}(\overline{q}_{\mathbb{F}'})$. Thus, by definition of H, $((\overline{q}_\mathbb{f}, \overline{q}_{\mathbb{f}'}), (\overline{q}_\mathbb{F}, \overline{q}_{\mathbb{F}'})) \in H$. And, by by Proposition 3.1.3, $\overline{q}_\mathbb{f} \otimes \overline{q}_{\mathbb{f}'} \models_{\mathbb{G}_{\mathbb{F},\mathbb{F}'}(\overline{q}_\mathbb{F}, \overline{q}_{\mathbb{F}'})} \mathbb{F}(\overline{q}_\mathbb{F}) \otimes \mathbb{F}'(\overline{q}_{\mathbb{F}'})$. Hence, H is a simulation relation for $\mathbb{f} \otimes \mathbb{f}' \models_{\mathbb{G}'} \mathbb{F} \otimes_{\text{full}} \mathbb{F}'$. □

Compatibility between two stateful interfaces boils down to checking if all stateless interfaces reachable from the initial state that will be composed together during composition are themselves compatible. Recall that we keep only states defined by the composition of compatible stateless interfaces during the composition of stateful interfaces. Hence, to check for compatibility, we only need to check if the initial state of the composite is defined by compatible stateless interfaces. Formally, two stateful information-flow interfaces, $\mathbb{F}$ and $\mathbb{F}'$, are compatible, denoted $\mathbb{F} \sim_{\text{full}} \mathbb{F}'$, iff $\mathbb{F}(\hat{q}) \sim \mathbb{F}'(\hat{q}')$, where $\hat{q}$ is the initial state of $\mathbb{F}$ and $\hat{q}'$ is the initial state of $\mathbb{F}'$.

Using the proof schema from Proposition 3.2.2, we can lift some of the stateless interface's results related to composition and compatibility to the stateful case. In particular, we can lift the results that compatibility is commutative, composition preserves well-formedness, and stateful interfaces support incremental design of systems.

Theorem 3.2.3. *Let $\mathbb{F}$, $\mathbb{F}'$ and $\mathbb{F}''$ be stateful information-flow interfaces. If $\mathbb{F} \sim_{\text{full}} \mathbb{F}'$ and $(\mathbb{F} \otimes_{\text{full}} \mathbb{F}') \sim_{\text{full}} \mathbb{F}''$, then $\mathbb{F}' \sim_{\text{full}} \mathbb{F}''$ and $\mathbb{F} \sim_{\text{full}} (\mathbb{F}' \otimes_{\text{full}} \mathbb{F}'')$.*

Proof. By definition of compatibility between stateful information-flow interfaces, we only need to prove the following statement for the intial states $\hat{q}$, $\hat{q}'$ and $\hat{q}''$ of arbitrary interfaces $\mathbb{F}$, $\mathbb{F}'$ and $\mathbb{F}''$:

$$\begin{aligned}&\text{If } \mathbb{F}(\hat{q}) \sim \mathbb{F}'(\hat{q}') \text{ and } (\mathbb{F}(\hat{q}) \otimes \mathbb{F}'(\hat{q}')) \sim \mathbb{F}''(\hat{q}''),\\ &\text{then } \mathbb{F}'(\hat{q}') \sim \mathbb{F}''(\hat{q}'') \text{ and } \mathbb{F}(\hat{q}) \sim (\mathbb{F}'(\hat{q}') \otimes \mathbb{F}''(\hat{q}'')).\end{aligned}$$

Which follows from Theorem 3.1.7 for stateless information-flow interfaces. □

3.2.2 Refinement and Independent Implementability

We define refinement between stateful information-flow interfaces as an alternating refinement relation [AHKV98]. Broadly, we say that a stateful interface $\mathbb{F}_R$ refines $\mathbb{F}_A$, if all output steps of $\mathbb{F}_R$ can be simulated by $\mathbb{F}_A$, while all input steps of $\mathbb{F}_A$ can be simulated by $\mathbb{F}_R$. To capture this definition, we need to define input and output steps. We introduce below two functions that return all assumptions and open-guarantees that can be reached in one step for each state q of a given stateful interface. Having this function in place, we can compute the set of states' sets, with one state set for each reachable assumption and reachable open- and closed-guarantee, effectively defining *input* and *output* steps, respectively.

Definition 3.2.4. *Let $\mathbb{F} = (X, Y, Q, \hat{q}, \delta, \mathbb{A}, \mathbb{G}, \mathbb{P})$ be an stateful information-flow interface.* Input steps *from a given state $q \in Q$ are defined as:*

$$\delta^X(q) = \{\delta^X(q, \mathcal{A}) \mid \mathcal{A} \subseteq Z \times X\} \text{ with } \delta^X(q, \mathcal{A}) = \{q' \in \delta(q) \mid \mathbb{A}(q') = \mathcal{A}\}.$$

While output steps *from a given state $q \in Q$ are defined as:*

$$\delta^Y(q) = \{\delta^Y(q, \mathcal{G}, \mathcal{P}) \mid \mathcal{G} \subseteq Z \times Y \text{ and } \mathcal{P} \subseteq Z \times Y\}$$
$$\text{with } \delta^Y(q, \mathcal{G}, \mathcal{P}) = \{q' \in \delta(q) \mid \mathbb{G}(q') = \mathcal{G} \text{ and } \mathbb{P}(q') = \mathcal{P}\}.$$

We can now define refinement as an alternating refinement relation, using the above functions to specify input and output steps.

Definition 3.2.5. *The stateful information-flow interface* $\mathbb{F}_R = (X, Y, Q_R, \hat{q}_R, \delta_R, \mathbb{A}_R, \mathbb{G}_R, \mathbb{P}_R)$ refines $\mathbb{F}_A = (X, Y, Q_A, \hat{q}_A, \delta_A, \mathbb{A}_A, \mathbb{G}_A, \mathbb{P}_A)$*, written* $\mathbb{F}_R \preceq_{\text{full}} \mathbb{F}_A$*, iff there exists a relation* $H \subseteq Q_R \times Q_A$ *s.t.* $(\hat{q}_R, \hat{q}_A) \in H$ *and for all* $(q_R, q_A) \in H$*:*

- $\mathbb{F}_R(q_R) \preceq \mathbb{F}_A(q_A)$*;*
- *for all set of states* $O \in \delta_R^Y(q_R)$*, there exists* $O' \in \delta_A^Y(q_A)$ *s.t. for all set of states* $I' \in \delta_A^X(q_A)$*, there exists* $I \in \delta_R^X(q_R)$ *defining a non-empty* $(O \cap I) \times (O' \cap I') \subseteq H$*.*

Example: Refinement of Stateful Information-flow Interfaces

In our first example of refinement between stateful interfaces (depicted below in Figure 3.10) we focus on the specification of dynamic information-flow requirements over a closed-system.

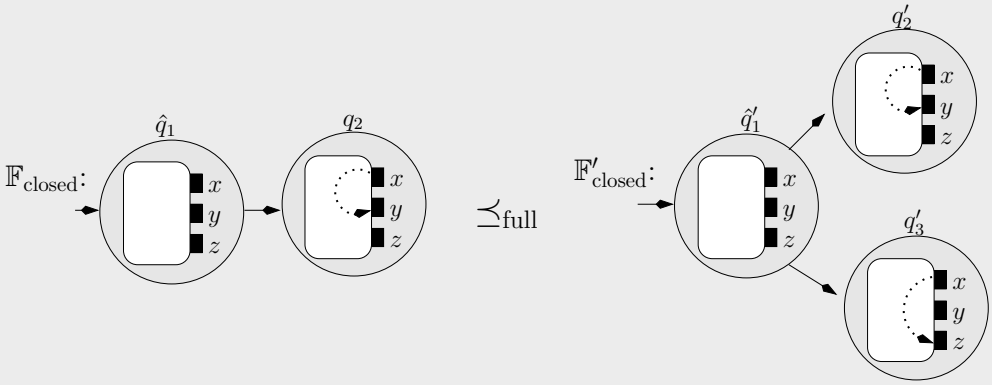


Figure 3.10: Stateful interface $\mathbb{F}_{\text{closed}}$ refines the stateful interface $\mathbb{F}'_{\text{closed}}$, witnessed by the relation $H_1 = \{(\hat{q}_1, \hat{q}'_1), (q_2, q'_2)\}$.

The stateful information-flow interface $\mathbb{F}_{\text{closed}}$ refines $\mathbb{F}'_{\text{closed}}$, by removing a nondeterministic choice on the transition function of the second interface.

Our first step to prove the refinement relation between the two interfaces in Figure 3.10, is to check whether their initial states are in the matching stateless information-flow refinement relation. This is trivial, as their initial states, $\hat{q}_1$ and $\hat{q}'_1$, define the same stateless information-flow interface. Formally, $\hat{q}_1 \preceq \hat{q}'_1$ and,

naturally, $(\hat{q}_1, \hat{q}'_1)$ is in the relation H_1 witnessing the refinement between the two interfaces. The next step is to evaluate the output and input steps from each initial state in the stateful interfaces depicted above. We present them below with δ_{closed} and $\delta_{\text{closed}'}$ being the transition relations from $\mathbb{F}_{\text{closed}}$ and $\mathbb{F}'_{\text{closed}}$, respectively.

$$
\begin{aligned}
\delta^Y_{\text{closed}}(\hat{q}_1) &= \{\delta^Y_{\text{closed}}(\hat{q}_1, \{(x,y)\}, \{(x,y)\})\} = \{\{q_2\}\} \\
\delta^Y_{\text{closed}'}(\hat{q}'_1) &= \{\delta^Y_{\text{closed}'}(\hat{q}'_1, \{(x,y)\}, \{(x,y)\}), \delta^Y_{\text{closed}'}(\hat{q}'_1, \{(x,z)\}, \{(x,z)\})\} \\
&= \{\{q'_2\}, \{\{q'_3\}\} \\
\delta^X_{\text{closed}'}(\hat{q}'_1) &= \{\delta^X_{\text{closed}'}(\hat{q}'_1, \{\})\} = \{\{q'_2, q'_3\}\} \\
\delta^X_{\text{closed}}(\hat{q}_1) &= \{\delta^X_{\text{closed}}(\hat{q}_1, \{\})\} = \{\{q_2\}\}.
\end{aligned}
$$

Then, for the only element of $\delta^Y_{\text{closed}}(\hat{q}_1)$, defining $O = \{q_2\}$, we chose the set $O' = \{q'_2\}$ from $\delta^Y_{\text{closed}'}(\hat{q}'_1)$, as it has exactly the same open- and closed-guarantees (i.e., q'_2 simulates the output requirements of q_2). As for the only element of $\delta^X_{\text{closed}'}(\hat{q}'_1)$ (we define $I' = \{q'_2, q'_3\}$), there is also only one option for I in $\delta^X_{\text{closed}}(\hat{q}_1)$, i.e., $I = \{q_2\}$. With no assumptions in any of the interfaces, this choice trivially satisfies the requirement for a refinement between the stateless interfaces in each state. Hence $(O \cap I) \times (O' \cap I') = \{q_2\} \times \{q'_2\} = \{(q_2, q'_2)\}$. With no more transitions in the stateful interfaces, we have that $H_1 = \{(\hat{q}_1, \hat{q}'_1), (q_2, q'_2)\}$ witnesses the refinement between the two interfaces.

We present a second example of the stateful information-flow interface's refinement in Figure 3.11 below. In this example, the interface $\mathbb{F}$ refines $\mathbb{F}'$ by specifying an alternative transition from the initial state that allows more environments while restricting the implementation and preserving the closed-guarantee.

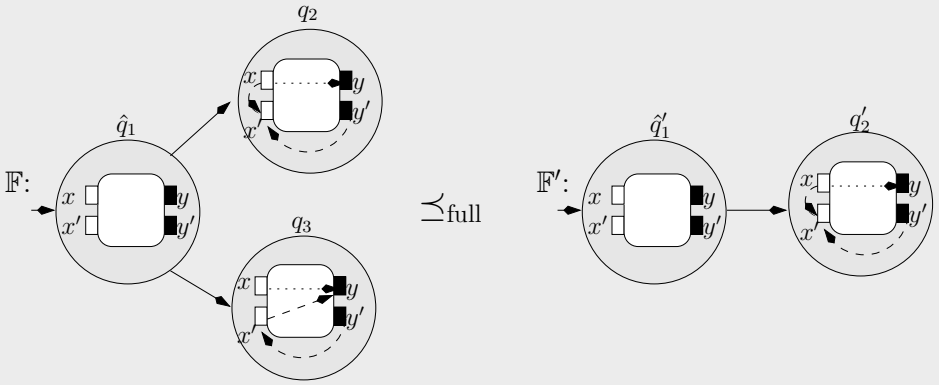


Figure 3.11: Stateful interface $\mathbb{F}$ refines the stateful interface $\mathbb{F}'$, witnessed by the relation $H_2 = \{(\hat{q}_1, \hat{q}'_1), (q_2, q'_2), (q_3, q'_2)\}$.

As in the previous example, the initial states in the interfaces above define the same stateless information-flow interface, thus, clearly $(\hat{q}_1, \hat{q}'_1)$ are in the relation

witnessing the refinement. We present below the input and output steps derived from the initial states, where δ_R and δ_A are the transition functions from $\mathbb{F}$ and $\mathbb{F}'$, respectively.

$$\begin{aligned}
\delta_R^Y(\hat{q}_1) &= \{\delta_R^Y(\hat{q}_1, \{(x,y)\}, \{(x,y)\}),\ \delta_R^Y(\hat{q}_1, \{(x,y),(x',y)\}, \{(x,y)\})\} = \\
&= \{\{q_2\}, \{q_3\}\} \\
\delta_A^Y(\hat{q}_1') &= \{\delta_A^Y(\hat{q}_1', \{(x,y)\}, \{(x,y)\})\} = \{\{q_2'\}\} \\
\delta_A^X(\hat{q}_1') &= \{\delta_A^X(\hat{q}_1', \{(x,x'),(y',x')\})\} = \{\{q_2'\}\} \\
\delta_R^X(\hat{q}_1) &= \{\delta_R^X(\hat{q}_1, \{(x,x'),(y',x')\}, \{(y',x')\})\} = \{\{q_2\}, \{q_3\}\}
\end{aligned}$$

The refined interface has two possible *output transitions* from the initial state $\hat{q}_1$: it can either move to state q_2 that specifies a stateless interface with both open- and closed-guarantee requiring only a no-flow form x to y; or it can move to state q_3 that specifies the set of no-flows $\{(x,y),(x',y)\}$ as its stateless open-guarantee and $\{(x,y)\}$ as the closed-guarantee. The abstract interface, $\mathbb{F}'$, can simulate any of these possibilities by its transition from its initial state $\hat{q}_1'$ to its only successor, q_2' because the stateless interface in q_2' does have new no-flow requirements in its open- and closes-guarantees when compared to the interfaces in q_2 and q_3.

As for the input steps, the abstract interface only has one possible *input transition* from its initial state, to the state q_2'. In this state, the stateless interface specifies the assumption that x does not flow to x' and y' does not flow to x. Both input steps from $\hat{q}_1$ simulate the input step from $\hat{q}_1'$ because,for both states accessible from the initial state in the refined interface, $\mathbb{A}(q_2) \subseteq \mathbb{A}'(q_2')$ and $\mathbb{A}(q_3) \subseteq \mathbb{A}'(q_2')$. Hence $H_2 = \{(\hat{q}_1, \hat{q}_1'), (q_2, q_2'), (q_3, q_2')\}$ witnesses the refinement between $\mathbb{F}$ and $\mathbb{F}'$.

In the theorem below, we prove that a refined stateful information-flow interface may accepted less implementations while allowing more environments.

Theorem 3.2.4. *Let* $\mathbb{F}_1{=}(X, Y, Q_1, \hat{q}_1, \delta_1, \mathbb{A}_1, \mathbb{G}_1, \mathbb{P}_1)$ *and* $\mathbb{F}_2{=}(X, Y, Q_2, \hat{q}_2, \delta_2, \mathbb{A}_2, \mathbb{G}_2, \mathbb{P}_2)$ *be stateful information-flow interfaces s.t.* $\mathbb{F}_1 \preceq_{\text{full}} \mathbb{F}_2$. *For all components* $\mathbb{f}$ *and* $\mathbb{f}_{\mathcal{E}}$*:*

(a) If $\mathbb{f} \models_{\mathbb{G}_1} \mathbb{F}_1$, *then* $\mathbb{f} \models_{\mathbb{G}_2} \mathbb{F}_2$.

(b) If $\mathbb{f}_{\mathcal{E}} \models_{\mathbb{A}_2} \mathbb{F}_2$, *then* $\mathbb{f}_{\mathcal{E}} \models_{\mathbb{A}_1} \mathbb{F}_1$.

Proof. Assume that $\mathbb{F}_1 \preceq_{\text{full}} \mathbb{F}_2$. Then, there exists a simulation relation $H_{\preceq} \subseteq Q_1 \times Q_2$ that witnesses it.

For the item (a), assume that $\mathbb{f} \models_{\mathbb{G}_1} \mathbb{F}_1$. Then, there exists a simulation relation $H_{\models} \subseteq Q_{\mathbb{f}} \times Q_1$ that witnesses it. Consider the relation $H = H_{\models} \circ H_{\preceq}$. By definitions of refinement and implementation, $(\hat{q}_{\mathbb{f}}, \hat{q}_1) \in H_{\models}$ and $(\hat{q}_1, \hat{q}_2) \in H_{\preceq}$. So, $(\hat{q}_{\mathbb{f}}, \hat{q}_2) \in H$. Additionally, $\mathbb{F}_1(\hat{q}_1) \preceq \mathbb{F}_2(\hat{q}_2)$ and $\mathbb{f}(\hat{q}_{\mathbb{f}}) \models_{\mathbb{G}_1(\hat{q}_1)} \mathbb{F}_1(\hat{q}_1)$. Then, $\mathbb{f}(\hat{q}_{\mathbb{f}}) \models_{\mathbb{G}_2(\hat{q}_2)} \mathbb{F}_2(\hat{q}_2)$. Consider arbitrary $(q_{\mathbb{f}}, q_2) \in H$. By construction of H there exists $(q_{\mathbb{f}}, q_1) \in H_{\models}$

and $(q_1, q_2) \in H_{\preceq}$. We want to prove that: if $q'_{\mathbb{f}} \in \delta_{\mathbb{f}}(q_{\mathbb{f}})$, then there exists $q'_2 \in \delta_2(q_2)$ s.t. $(q'_{\mathbb{f}}, q'_2) \in H$ and $\mathbb{f}(q'_{\mathbb{f}}) \models_{\mathbb{G}_2(q'_2)} \mathbb{F}_2(q'_2)$.

Assume that $q'_{\mathbb{f}} \in \delta_{\mathbb{f}}(q_{\mathbb{f}})$. By $(q_{\mathbb{f}}, q_1) \in H_{\models}$, then there exists a state $q'_1 \in \delta_1(q_1)$ s.t. $(q'_{\mathbb{f}}, q'_1) \in H_{\models}$. So, $\mathbb{M}(q'_{\mathbb{f}}) \subseteq \overline{\mathbb{G}}_{\mathbb{F}_1}(q'_1)$. Additionally, by $(q_1, q_2) \in H_{\preceq}$ and $q'_1 \in \delta_1(q_1)$, there exists $q'_2 \in \delta_2(q_2)$ s.t. $\mathbb{G}_{\mathbb{F}_2}(q'_2) \subseteq \mathbb{G}_{\mathbb{F}_1}(q'_1)$. Thus, $(q'_{\mathbb{f}}, q'_2) \in H$ and $\mathbb{M}(q'_{\mathbb{f}}) \subseteq \overline{\mathbb{G}_1}(q'_1) \subseteq \overline{\mathbb{G}_2}(q'_2)$. So, by definition of implements for stateless interfaces, $\mathbb{f}(q'_{\mathbb{f}}) \models_{\mathbb{G}_2(q'_2)} \mathbb{F}_2(q'_2)$. Hence H witnesses $\mathbb{f} \models_{\mathbb{G}} \mathbb{F}$.

For the second item, assume that $\mathbb{f}_{\mathcal{E}} \models_{\mathcal{A}_2} \mathbb{F}_2$. Then, there exists a simulation relation $H_{\models} \subseteq Q_2 \times Q_{\mathcal{E}}$ that witnesses it. Consider the relation $H = H_{\preceq} \circ H_{\models}$. We can prove analogously to the previous case that H witnesses $\mathbb{f}_{\mathcal{E}} \models_{\mathbb{A}_1} \mathbb{F}_1$. □

We can now prove that stateful information-flow interface satisfies the independent implementation of systems.

Theorem 3.2.5. *For all well-formed interfaces $\mathbb{F}'_1$, $\mathbb{F}_1$ and $\mathbb{F}_2$, if $\mathbb{F}'_1 \preceq_{\text{full}} \mathbb{F}_1$ and $\mathbb{F}_1 \sim \mathbb{F}_2$, then $\mathbb{F}'_1 \sim \mathbb{F}_2$ and $\mathbb{F}'_1 \otimes \mathbb{F}_2 \preceq_{\text{full}} \mathbb{F}_1 \otimes \mathbb{F}_2$.*

Proof. Assume that: (i) $\mathbb{F}'_1 \preceq_{\text{full}} \mathbb{F}_1$; and (ii) $\mathbb{F}_1 \sim \mathbb{F}_2$. $\mathbb{F}'_1 \sim \mathbb{F}_2$ follows from (i) and Theorem 3.1.9 for stateless interfaces. We prove now that $\mathbb{F}'_1 \otimes \mathbb{F}_2 \preceq_{\text{full}} \mathbb{F}_1 \otimes \mathbb{F}_2$. From (i), there exists a relation $H_{\preceq} \subseteq Q'_1 \times Q_1$ that witnesses the refinement. Consider the relation: $H = \{((q_{\mathbb{F}'_1}, q_{\mathbb{F}_2}), (q_{\mathbb{F}_1}, q_{\mathbb{F}_2})) \mid (q_{\mathbb{F}'_1}, q_{\mathbb{F}_1}) \in H_{\preceq} \text{ and } \mathbb{F}_1(q_{\mathbb{F}_1}) \sim \mathbb{F}_2(q_{\mathbb{F}_2})\}$.

By (i) and (ii), $((\hat{q}_{\mathbb{F}'_1}, \hat{q}_{\mathbb{F}_2}), (\hat{q}_{\mathbb{F}_1}, \hat{q}_{\mathbb{F}_2})) \in H$. Additionally, $\mathbb{F}'_1(\hat{q}_{\mathbb{F}'_1}) \preceq \mathbb{F}_1(\hat{q}_{\mathbb{F}_1})$. Then, by Theorem 3.1.9 for stateless interface, $\mathbb{F}'_1(\hat{q}_{\mathbb{F}'_1}) \otimes \mathbb{F}_2(\hat{q}_{\mathbb{F}_2}) \preceq \mathbb{F}_1(\hat{q}_{\mathbb{F}_1}) \otimes \mathbb{F}_2(\hat{q}_{\mathbb{F}_2})$.

Consider arbitrary $((q_{\mathbb{F}'_1}, q_{\mathbb{F}_2}), (q_{\mathbb{F}_1}, q_{\mathbb{F}_2})) \in H$ and arbitrary $O \in \delta^Y_{\mathbb{F}'_1 \otimes \mathbb{F}_2}((q_{\mathbb{F}'_1}, q_{\mathbb{F}_2}))$. Then, there exists $\mathcal{G}'$ and $\mathcal{P}'$ s.t. $O = \delta^Y_{\mathbb{F}'_1 \otimes \mathbb{F}_2}((q_{\mathbb{F}'_1}, q_{\mathbb{F}_2}), \mathcal{G}', \mathcal{P}')$. By $H_{\preceq}$ witnessing (i) and Definition 3.2.4, there exists $\mathcal{G} \subseteq \mathcal{G}'$ and $\mathcal{P} \subseteq \mathcal{P}'$ s.t. $O' = \delta^Y_{\mathbb{F}_1 \otimes \mathbb{F}_2}((q_{\mathbb{F}_1}, q_{\mathbb{F}_2}), \mathcal{G}, \mathcal{P})$.

Consider arbitrary $I \in \delta^X_{\mathbb{F}_1 \otimes \mathbb{F}_2}((q_{\mathbb{F}_1}, q_{\mathbb{F}_2}))$. Then, by $H_{\preceq}$ witnessing (i) and Definition 3.2.4, there exists $\mathcal{A}$ s.t. $I' = \delta^X_{\mathbb{F}_1 \otimes \mathbb{F}_2}((q_{\mathbb{F}_1}, q_{\mathbb{F}_2}), \mathcal{A})$. By $H_{\preceq}$ witnessing (i), there exists $\mathcal{A}' \subseteq \mathcal{A}$ s.t. $I = \delta^X_{\mathbb{F}'_1 \otimes \mathbb{F}_2}((q_{\mathbb{F}'_1}, q_{\mathbb{F}_2}), \mathcal{A}')$.

Consider arbitrary $((q'_{\mathbb{F}'_1}, q'_{\mathbb{F}_2}), (q'_{\mathbb{F}_1}, q'_{\mathbb{F}_2})) \in (O \cap I) \times (O' \cap I')$. Then, by (i) and H definition, $\mathbb{F}'_1(q'_{\mathbb{F}'_1}) \preceq \mathbb{F}_1(q'_{\mathbb{F}_1})$ and $\mathbb{F}_1(q'_{\mathbb{F}_1}) \sim \mathbb{F}_2(q'_{\mathbb{F}_2})$ So, by Theorem 3.1.9 for stateless interfaces, $\mathbb{F}'_1(q'_1) \otimes \mathbb{F}_2(q_2) \preceq \mathbb{F}_1(q_1) \otimes \mathbb{F}_2(q_2)$.

Hence H is a witness relation for $\mathbb{F}'_1 \otimes \mathbb{F}_2 \preceq \mathbb{F}_1 \otimes \mathbb{F}_2$. □

3.3 Related Work

The interface theory introduced in this chapter adopts the contract-based design approach (explained in Chapter 2) and, as the name suggests, follows the interface theory philosophy (c.f., Section2.2) In particular, our theory assumes an optimistic view on composition (i.e., all parts involved are willing to collaborate to meet other parts' requirements) and an alternating view on refinement. The novelty of this work is that it presents the first interface language that specifies information-flow requirements while supporting the incremental design of systems and their independent implementability. Our main contribution was to introduce the notion of closed-guarantees, allowing the designer to express its intended requirements for the closed system without interfering with the compositional flavor of the interface theory. Closed-guarantees act as a consistency check between the interaction of assumptions and open-guarantees, which can be used to steer the refinement process to satisfy the intended global behavior of the system.

Language-based techniques, like type systems [FM11] to program analysis using program-dependency graphs (PDGs) [HS09, GHM13], are a successful approach to the verification and enforcement information flow policies [SM03]. These techniques are tailored for specific implementation languages, while our goal is to introduce composition and refinement definitions that are independent of the language adopted for the implementations. Therefore, language oriented techniques are orthogonal to our work.

The theory introduced here focuses on the structural aspect of information flow. Thus, it abstracts away from the specifics of the model, like their interaction or semantics. This is distinct from Interface Automata (IA), whose primary focus is specifying the iteration between the system's parts. Another prominent approach to defining interface languages are Assume/Guarantee interfaces. However, this approach requires a complete separation between input and output requirements, which is not possible when reasoning about the flow of information.

More recently, Lee and D'Argenio propose *interface for structure and security* [LD10b], extending IA with actions labelled as having either low or high confidentiality. They enforce a bisimulation-based variant of non-interference that checks if the system's behaviour does not change when actions labelled with high confidentiality occur. Later, in [LD10a], the same authors extend their approach to enforcing an alternative refinement-based notion of non-interference. Our approach is orthogonal to both works. Instead of labeling variables, our theory specifies no-flow requirements directly with the variables. Additionally, to aid the compositional design, we introduce closed-guarantees as a meta-specification over all system's parts assumptions and open-guarantees.

Closer to our approach are *relational interfaces* (RIs) [TLHL11], specifying for each legal component's input the output it generates. As for information-flow interfaces, RIs do not require assumptions and guarantees to be defined solely over inputs and output variables. A RI specified the desired input-output behavior over traces. Hence they have limited expressive power to capture information-flow policies, which often require comparing multiple traces at a time.

In a recent work, Incer et al. [IBSS22] introduced *Hypercontracts* as a meta-theory for assumption-guarantee contracts supporting hyperproperties. In a nutshell, a hypercontract is defined by a pair of assumptions and guarantees of the closed system. Both assumptions and guarantees are interpreted as set of components. To compute the guarantees of the open system, the authors introduce a notion of quotient between component sets, defining open guarantees as the quotient between the guarantee of the closed system and the assumption. Our theory can also be interpreted as the set of components derived by the no-flow relations specifying the interface. The main difference between these two approaches, is that we include both the open and closed system requirements at the interface level. The explicit specification of guarantees for both the open and the closed system allows the designer to specify further assumptions and open-guarantees, even if they are not necessary to support the requirements of the closed-system.

3.4 Final Remarks

This chapter introduced a framework for stateless and stateful interfaces to specify information-flow requirements. The main contribution lies in the stateless information-flow interfaces. Stateful interfaces naturally extend the previous, allowing for dynamic specifications by labeling states with stateless interfaces and defining appropriate transitions between different specification states. Both kinds of interfaces are defined by specifying assumptions, open-guarantees, and closed-guarantees. We introduce a refinement and compatibility predicate, together with a composition operator for the stateless and stateful versions. We then prove that our calculus of information-flow interfaces adheres to the principles of incremental design and independent implementability.

In later work that followed the results described in this chapter, we focused on studying semantics view of this theory. In particular, in the journal version [BFH+24], we include a section where we present information-flow contracts where assumptions and guarantees are defined with sets of flow relations. We introduce the composition and refinement of such contracts and show that they define an appropriate semantic interpretation for stateless information-flow interfaces. This work was followed by a short paper [BHNOdC24] where we define how to go from information-flow contracts to contracts over sets of security lattices. With these semantic investigations and future work that may follow, we hope to close the gap between the modeling phase of security requirements and their concrete implementation into software or hardware components.

Part II

Information-flow Specification

CHAPTER 4

Trace Properties and Hyperproperties

In this work, we abstract systems by their observable behavior. In particular, we represent each system execution by a trace, i.e., a sequence of assignments of the system's variables (also referred to as program variables) describing how they evolve during a system execution. A system is then characterized by a set of traces.

This chapter introduces definitions and notations related to traces, which are used throughout this document. We start the chapter by introducing *trace properties*, which are sets of traces. As trace properties are not expressive enough to specify information-flow requirements, we then present *hyperproperties*, which generalize trace properties to sets of trace sets. This chapter focuses on trace properties and hyperproperties as specification formalisms, highlighting their relation to linear-temporal logic. We assume the reader to be familiar with classical logic (propositional and first-order). Otherwise, a good introduction to classical logic can be found in the book by Elliott Mendelson [Men09].

4.1 Trace Properties

We model the system's executions as traces, which are sequences of valuations. A valuation assigns a value of a finite domain to a system (or program) variable.

Traces. Let X be a finite set of variables, ranging over a finite domain Σ. A *valuation (or variable assignment)* for variables in X over a domain Σ is a partial mapping $v : X \to \Sigma$ assigning a value in Σ to a variable in X. For a given assignment v, its *domain* is the set of all variables that have an assignment in v, denoted by $X(v) \subseteq X$, and its *size* is the size of its domain, that is, $|v| = |X(v)|$. We denote by $v[x \mapsto b]$ the valuation

resulting from updating the value assigned to the variable x in v to the value b, i.e., all the assignments in v to other variables remain unchanged and only the assignment to x is changed to $b \in \Sigma$. To simplify notation, we may represent a finite set of variables $\{x_0, \ldots, x_n\}$ as a string $[x_0 \ldots x_n]$ and their valuations v by the string $[v(x_0) \ldots v(x_n)]$. *The set of all valuations from X to Σ* is denoted by Σ^X.

A *trace τ over the set of variables X with domain Σ* is a sequence of valuations in Σ^X, i.e., $\tau \in (\Sigma^X)^*$. The *alphabet of a trace over X and Σ* is Σ^X. For traces defined over boolean domains (i.e., $\Sigma = \{0, 1\}$) and a set of variables X, we may abuse of notation and refer to X as their domain. The *length* of a finite trace $\tau = v_0 v_1 \ldots v_n$ is defined as $|\tau| = n + 1$, while for an infinite trace τ we define $|\tau| = \omega$. For a trace $\tau = v_0 v_1 \ldots$ and an index i within its length (i.e., $i < |\tau|$), we adopt the following indexing notations: $\tau[i] = v_i$, $\tau[i \ldots] = v_i v_{i+1} \ldots$, and $\tau[\ldots i] = v_0 v_1 \ldots v_{i-1}$. When an index falls outside a trace length (i.e., $j \geq |\tau|$), we adopt the following convention: $\tau[j \ldots]$ is the empty trace, and $\tau[\ldots j] = \tau$. The set of all *finite* traces over X and Σ is denoted $(\Sigma^X)^*$, while the set of all *infinite* traces over X is denoted by $(\Sigma^X)^\omega$.

Example: Traces

The program $\mathcal{P}$, shown in Algorithm 4.1, is the running example for concepts related to traces. In a later chapter, we use the same program to highlight the impact of asynchronous state changes in the expressivity of information-flow requirements. In a nutshell, our goal later will be to prove that the program satisfies the stateful information-flow policy that the value of y must be independent of the secret variable x until the state changes and the value of z must now be independent of x.

Algorithm 4.1: Program $\mathcal{P}$

```
1  state := 0;
2  do
3  |  if (state = 0) then
4  |  |  read(c1, state in {0,1});
5  |  end
6  |  read(c0, x in {0,1});
7  |  if (state = 0) then
8  |  |  z := x; y = default;
9  |  else
10 |  |  y := x; z = default;
11 |  end
12 |  output(c2, y);
13 |  output(c3, z);
14 while True;
```

The program $\mathcal{P}$ starts with the state variable set to 0 ($state = 0$). In every while loop iteration, the *state* value is non-deterministically assigned via the input channel with address c_1. When the value of *state* changes from 0 to 1, the program will stop updating *state*, which will remain as 1 for the subsequent iterations.

Regarding the variable x, its value is non-deterministically assigned via the input channel c_0. If the value of *state* is 0, then the program $\mathcal{P}$ assigns x to z while it sets y a default value. Otherwise, when the value of *state* is 1, the inverse occurs: the program $\mathcal{P}$ assigns x to y while it sets z a default value. The default value is a boolean value set at the start of the program execution. At the end of each loop iteration program exposes y and z via the output channels c_2 and c_3, respectively.

The set of (program) variables of $\mathcal{P}$ is $X = \{x, y, z, state\}$. We assume that all variables are boolean, so the domain of the traces derived by $\mathcal{P}$ is $\{0,1\}^X$. We do not include c_1, c_2 and c_3 as program variables because they are constants that encode the address of three communication channels.

	0			1			2			3		
	x	y	z	x	y	z	x	y	z	x	y	z
τ_1	0	0	0	1	1	0	1	1	0	1	1	0
τ_2	1	0	1	1	1	0	1	1	0	1	1	0
τ_3	1	0	1	1	0	1	0	0	0	0	0	0
τ_4	0	0	0	1	0	1	0	0	0	1	1	0

Table 4.1: A set of traces over the variables x, y, and z of $\mathcal{P}$, with $default = 0$, transparent cells indicating that $state = 0$, and gray cells, that $state = 1$.

In Table 4.1, we depict four traces derived from program $\mathcal{P}$. Each step in a trace represents the variable values at the end of the respective while loop iteration. We represent the traces in Table 4.1 as, for the string of program variables [x y z $state$]:

$$\tau_1 = [0000]\ [1101]\ [1101]\ [1101]$$
$$\tau_2 = [1010]\ [1101]\ [1101]\ [1101]$$
$$\tau_3 = [1010]\ [1010]\ [0001]\ [0001]$$
$$\tau_4 = [0000]\ [1010]\ [0000]\ [1101].$$

Trace Properties. A *trace property* is a set of traces containing all traces that satisfy the behavior it specifies. Formally, a trace property T over a set X of variables and domain Σ is a set of infinite traces over Σ^X, that is, $T \subseteq (\Sigma^X)^\omega$. A trace τ satisfies a trace property T iff it is one of its elements ($\tau \in T$).

We refer to the set of all trace properties as $\mathbb{T} = 2^{(\Sigma^X)^\omega}$. Note that both systems and trace properties define sets of traces. The distinction between these two types of sets

is semantics: the first (trace properties) specifies intended behavior, while the second (systems) specifies observed behavior. Then, *a system S satisfies a trace property T* iff all behaviors of S are included in the set of intended behaviors in T, i.e., $S \subseteq T$.

LTL. A successful formalism to specify trace properties is Linear Temporal Logic (LTL), introduced by Pnueli in [Pnu77]. LTL formulas are defined by the following grammar:

$$\varphi ::= \; a \mid \neg\varphi \mid \varphi \vee \varphi \mid \mathbf{X}\,\varphi \mid \varphi\,\mathbf{U}\,\varphi$$

where $a \in X$ is a propositional variable, and $\mathbf{X}$ ("next") and $\mathbf{U}$ ("until") are temporal modalities.

LTL formulas are interpreted over infinite traces and boolean domains. Given a LTL formula φ and an infinite trace $\tau \in (\{0,1\}^X)^\omega$, we define that the trace τ satisfies φ, denoted $\tau \models_{\text{LTL}} \varphi$, inductively over the structure of φ, as follows:

$\tau \models_{\text{LTL}} a$ iff $\tau[0](a) = 1$;
$\tau \models_{\text{LTL}} \neg\psi$ iff $\tau \not\models_{\text{LTL}} \psi$;
$\tau \models_{\text{LTL}} \psi_1 \vee \psi_2$ iff $\tau \models_{\text{LTL}} \psi_1$ or $\tau \models_{\text{LTL}} \psi_2$;
$\tau \models_{\text{LTL}} \mathbf{X}\,\psi$ iff $\tau[1\ldots] \models_{\text{LTL}} \psi$;
$\tau \models_{\text{LTL}} \psi_1\,\mathbf{U}\,\psi_2$ iff exists $j \geq 0$ s.t. $\tau[j\ldots] \models_{\text{LTL}} \psi_2$ and for all $0 \leq j' < j, \tau[j'\ldots] \models_{\text{LTL}} \psi_1$.

When it is clear from the context, we may omit the LTL subscript from $\models_{\text{LTL}}$. We adopt the usual abbreviations for common boolean and temporal operators: conjunction is defined as $\psi_1 \wedge \psi_2 \stackrel{\text{def}}{=} \neg\psi_1 \vee \neg\psi_2$; and the temporal operators $\mathbf{G}$ ("globally", also called "always") and $\mathbf{F}$ ("eventually") are defined as $\mathbf{G}\,\psi \stackrel{\text{def}}{=} \psi\,\mathbf{U}$ false and $\mathbf{F}\,\psi \stackrel{\text{def}}{=}$ true $\mathbf{U}\,\psi$.

Given a set of traces T and an LTL formula φ, the *model-checking problem* asks whether all traces in T satisfy the formula φ. Formally, a set of traces T is a *model* of an LTL formula φ iff, for all traces $\tau \in T$, $\tau \models_{\text{LTL}} \varphi$.

Example: LTL Semantics

The LTL formula below specifies how the value of y and z changes depending on *state*:

$$\varphi \stackrel{\text{def}}{=} (\neg \mathit{state} \wedge \neg y)\;\mathbf{U}\;(\mathbf{G}\,(\mathit{state} \wedge \neg z)).$$

In particular, φ states that the variable *state* has value 0 until its its value changes to 1 and, for then on, it will always be 1. Additionally, the formula specifies that depending on the state of the program (identified by the value of the variable *state*), either y or z is assigned to 0. We observe that all executions of the program $\mathcal{P}$ in the previous example satisfy the property specified by φ, i.e., the set of traces derived from $\mathcal{P}$ are a model of φ.

First-order Logic of Order. In his seminal work [Kam68], Kamp initiated the study of relative expressiveness between the classical first-order approach to specify linear-time (the first-order logic of order, which we denote by FO[<]) and LTL. The first-order logic FO[<] is interpreted over labeled linear orders with all uninterpreted predicates being unary. Formally, FO[<] formulas φ are defined by the grammar:

$$\psi ::= P(i) \mid i < i \mid i = i$$
$$\varphi ::= \psi \mid \neg\varphi \mid \varphi \vee \varphi \mid \exists i\, \varphi$$

where i is a first-order variable, $=$ is equality, $<$ is the order, and P is predicate from a given set of monadic predicates. We interpret this logic over labeled linear-orders, which are defined by a tuple $(\Lambda, <, \mathcal{I})$ where $<$ defines a linear order over the domain Λ, and $\mathcal{I}$ is a function from predicates to sets of elements of Λ interpreting the predicates over the given domain.

From a trace τ, defined over the set of propositional variables X, we can derive the labeled linear-order $(\mathbb{N}, <, \mathcal{I}_\tau)$ over the set of unary predicates $\{P_a \mid a \in X\}$, where $<$ is interpreted as usual for natural numbers and $\mathcal{I}(P_a) = \{j \mid \tau[j](a) = 1\}$. Given such a translation from a trace to a labelled linear order, it becomes clear how to effectively translate LTL formulas (with the semantics introduced above) to a FO[<] interpreted over $(\mathbb{N}, <)$ (where $<$ is interpreted as usual over the natural numbers). Hence, FO[<] subsumes LTL. The only question remaining is whether LTL subsumes FO[<] interpreted over $(\mathbb{N}, <)$, i.e., whether LTL is expressively complete for this fragment of FO[<].

Kamp started by proving in [Kam68] that LTL with both past and future temporal operators is complete for FO[<] over the integers with order (i.e., $(\mathbb{Z}, <)$). Later, in [GPSS80], Gabbay et al. proved that when considering only future operators (as in the LTL definition introduced here), LTL is complete for FO[<] over the natural numbers. For the remaining of the manuscript, we are only interested in the linear order over natural numbers[1].

4.2 Hyperproperties

A general formalism to specify security policies must allow to specify relations between multiple system executions. Consider, for example, the program below, which has a variable with high-confidentiality (`secret`) and with low-confidentiality (`public`):

```
if secret>0 then public:=public+1
```

In our policy, we want to prevent an attacker from learning about the value of `secret` from observing the public behavior of the program. The program above violates this requirement because an attacker can compare the values of `public` before and after the

[1] For an overview of results comparing other versions of LTL with FO[<] interpreted over different linear orders, a good source is the manuscript by Rabinovich [Rab14].

program execution and observe that the observed behavior differs for the same public input (which the attacker controls). This is an example of an *implicit flow of information*. In this example, the information flows through program control.

Trace properties cannot express requirements across multiple executions [McL96, CS10]. For this reason, Clarkson and Schneider introduced, in [CS10], *hyperproperties* that generalize trace properties to sets of trace sets, setting the foundations for a general theory to specify security-related properties.

Formally, a hyperproperty $\mathbf{T} \subseteq \mathbb{T}$ is a set of trace properties, i.e., a set of trace sets. In trace semantics, a system S is characterized by the set of its execution traces; hence systems and trace properties have the same type: $S \in \mathbb{T}$. A hyperproperty characterizes a set of systems.

HyperLTL. HyperLTL, introduced in [CFK$^+$14], extends LTL with trace quantifiers. It has emerged as a popular formalism for both the specification and verification of hyperproperties with tools supporting important reasoning tasks, from model checking [FRS15, BF23, HSB21], to satisfiability checking [FHS17] and runtime verification [Bon16].

Let $\mathcal{V} = \{\pi, \pi', \ldots, \pi_0, \pi_1, \ldots\}$ be a set of trace variables and $X = \{a, b, \ldots, x, y, \ldots\}$ be a set of propositional variables. HyperLTL formulas φ are defined by the grammar:

$$\psi ::= a_\pi \mid \neg\psi \mid \psi \vee \psi \mid \mathbf{X}\,\psi \mid \psi\,\mathbf{U}\,\psi$$
$$\varphi ::= \exists \pi\, \varphi \mid \forall \pi\, \varphi \mid \psi$$

where $\pi \in \mathcal{V}$ is a trace variable and $a \in X$ is a propositional variable.

We evaluate HyperLTL formulas over sets of traces. To keep track of the current assignment to trace variables, we define trace assignments, $\Pi_T : \mathcal{V} \to T$, which are partial functions from traces in a given set of traces T to trace variables in $\mathcal{V}$. We denote by $\Pi_T[\pi \mapsto \tau]$ updating the assignment to π by a trace τ, i.e., only the value assigned to π changes while all the others remain the same. We define the models relation for HyperLTL formulas over pairs with a trace assignment Π_T and a time $i \in \mathbb{N}$ inductively over the structure of HyperLTL formulas as follows:

$$\begin{aligned}
&(\Pi_T, i) \models_{\mathrm{H}} \exists \pi\ \psi \text{ iff there exists } \tau \in T : (\Pi_T[\pi \mapsto \tau], i) \models_{\mathrm{H}} \psi;\\
&(\Pi_T, i) \models_{\mathrm{H}} \forall \pi\ \psi \text{ iff for all } \tau \in T : (\Pi_T[\pi \mapsto \tau], i) \models_{\mathrm{H}} \psi;\\
&(\Pi_T, i) \models_{\mathrm{H}} a_\pi \text{ iff } \Pi_T(\pi)[i](a) = 1;\\
&(\Pi_T, i) \models_{\mathrm{H}} \neg\psi \text{ iff } (\Pi_T, i) \not\models_{\mathrm{H}} \psi;\\
&(\Pi_T, i) \models_{\mathrm{H}} \psi_1 \vee \psi_2 \text{ iff } (\Pi_T, i) \models_{\mathrm{H}} \psi_1 \text{ or } (\Pi_T, i) \models_{\mathrm{H}} \psi_2;\\
&(\Pi_T, i) \models_{\mathrm{H}} \mathbf{X}\,\psi \text{ iff } (\Pi_T, i+1) \models_{\mathrm{H}} \psi;\\
&(\Pi_T, i) \models_{\mathrm{H}} \psi_1\,\mathbf{U}\,\psi_2 \text{ iff there exists } i \le j : (\Pi_T, j) \models_{\mathrm{H}} \psi_2\\
&\qquad\qquad \text{and for all } i \le j' < j : (\Pi_T, j') \models_{\mathrm{H}} \psi_1.
\end{aligned}$$

A set T of traces is a model of a HyperLTL formula φ, denoted $T \models_{\text{H}} \varphi$, iff there exists an assignment Π_T such that $(\Pi_T, 0) \models_{\text{H}} \varphi$. A formula is closed (a sentence) when all occurrences of trace variables are in the scope of a quantifier. For all closed formulas φ, $T \models_{\text{H}} \varphi$ iff $(\Pi_T^{\emptyset}, 0) \models_{\text{H}} \varphi$, where $\Pi_T^{\emptyset}$ is the empty assignment. We may omit the subscript H in $\models_H$ when it is clear from context.

Example: HyperLTL

We specify with the HyperLTL formula below that for all two executions (π and π') it always hold that if they agree on the *state* variable value then either they agree also in the value of y or in the value of z:

$$\forall\pi\forall\pi' \, \mathbf{G}(state_\pi = state_{\pi'} \rightarrow (y_\pi = y_{\pi'} \vee z_\pi = z_{\pi'})).$$

The Program $\mathcal{P}$ in the first example satisfies this HyperLTL formula.

Other Liner-time Hyperlogics. HyperLTL [CFK+14] was not the first extension to LTL proposed to reason about security properties. Earlier instances include epistemic temporal logic (ETL) [FMHV95], which extends LTL with *modal operators for knowledge*; and SecLTL [DFK+12], which introduces the *hide* operator. We now delve into literature results comparing these formalisms with HyperLTL.

The first attempt to compare HyperLTL and ETL was presented in the seminal HyperLTL paper [CFK+14]. In this paper, the authors proved that HyperLTL subsumes ETL when HyperLTL is extended with the possibility to quantify over propositional variables that are not part of the system. This extension defines, indeed, a formalism that is strictly more expressive than HyperLTL[CFHH19a], called HyperQPTL. It was proved later by Bozzelli et al. in [BMP15] that HyperLTL and ETL have incomparable expressive power. In [BMP15], they proved the stronger result that HyperCTL* (extending CTL* with trace quantifiers) and KCTL* (extending CTL* with the knowledge operator) have incomparable expressive power. Note that LTL is subsumed by CTL*. Thus, the results transfer seamlessly to HyperLTL and ETL. To prove that HyperLTL does not subsume ETL, they observe that all trace quantification in ETL is done implicitly through the knowledge operator. Then, no ETL formula can specify the requirement that two traces in a set of traces differ only at a time point, while we can do it with HyperLTL formulas. As for the other direction, in HyperLTL formulas, trace quantifiers always precede time quantification (done implicitly with the time operators). Then, existentially quantified time variables always depend on universally quantified trace variables. Following this observation, they show that HyperLTL cannot express synchronized behavior across all traces in a trace set, while ETL can. Synchronized termination requires that there exists a unique time point for which all traces agree on the value of a propositional variable.

The hide modality in SecLTL expresses that the system's observable behavior must be independent of secret values. The semantic interpenetration of hiding compares all

possible outcomes from execution paths starting with the same secret value. As SecLTL is defined in terms of execution paths, in [CFK+14], the authors point out that there are SecLTL formulas that distinguish systems defining the same set of traces but with different computation paths. Hence, SecLTL is not subsumed by HyperLTL (as HyperLTL with only one existential quantifier is equivalent to LTL).

A different approach to change LTL to support security policies is to change the semantics interpretation of LTL from the classical first-order view to the alternative *team semantics* approach [Vää07]. In the team semantics perspective, formulas are evaluated over sets of assignments (referred to as *teams*) instead of single assignments. In [KMVZ18], Krebs et al. propose to reinterpret LTL under *team semantics*, proposing both a synchronous and an asynchronous interpretation for the temporal operators. The two semantics differ only on how they slice the teams while interpreting time: in synchronous semantics, the slicing is done simultaneously for all traces, while in the asynchronous case, the slicing time is local to each trace. In the same work, they prove that HyperLTL and LTL interpreted with team semantics and synchronous entailment have incomparable expressive power.

CHAPTER 5

Specifying Information-flow

This chapter presents an overview on the specification of information-flow requirements. We first introduce *information-flow policies*, which are high-level descriptions of required and forbidden flows within a system. In particular, we introduce policies defined for sets of security labels, which, under reasonable assumptions, define *security lattices*. We then present different trace-based semantics interpretations for information-flow policies with a focus on non-interference related properties.

5.1 Policies

Information security is a field of computer science concerned with protecting information from being compromised or misused. Its main goals can be summarized by the CIA (*Confidentiality Integrity Availability*) triad explained below [San93]:

Confidentiality concerns protecting private information from unauthorized agents, i.e., it enforces that *secret* information (high-confidentiality) can not be learned through a *public* channel (low-confidentiality);

Integrity concerns preserving the quality of information by preventing *untrusted* agents (low-integrity) from interfering with *trusted* channels (high-integrity);

Availability assures that information with high-availability requirements is available when needed.

These goals cut across different computational layers, from hardware and software to communication between them, with information being the common denominator between these concepts, as depicted in Figure 5.1. Information security policies are often specified

[1]Figure derived from file *CIAJMK1209-en.svg* by Michel Bakni under license CC BY-SA 4.0 `https://creativecommons.org/licenses/by-sa/4.0` via Wikimedia Commons.

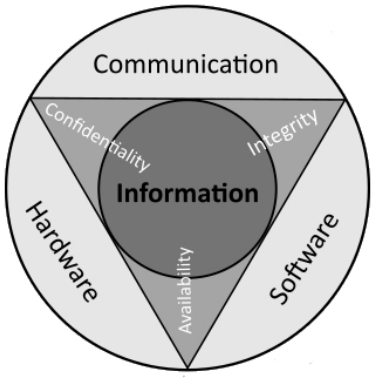


Figure 5.1: CIA triad of information security. [1]

as information-flow requirements on a system. In this work, we abstract from the specifics of each layer (for example, hardware vs software), referring to the target entities of an information flow policy as *systems*.

Instead of defining security policies over concrete objects of a system, it is common practice to define them over a set of *security labels* or, equivalently, *security classes*. The system's objects are later assigned a label (for example, at design time or dynamically while the system runs) reflecting their role within the policy.

The seminal work by Bell and LaPadula [BL75, Bel05] introduced a state-based model to enforce access control policies defined over security labels. Their model focused primarily on the confidentiality dimension of information security. Later, Biba [Bib77] introduced a similar approach to capture integrity, which behaves inversely from confidentiality (i.e., confidentiality prevents information flow from a high to a low level of confidentiality, while integrity prevents flows from a low to a high level of integrity). Denning, in [Den76], showed that under reasonable assumptions, these models define (finite) *security lattices* that are now a fundamental notion across multiple information security mechanisms. Notable examples of such techniques are type systems for security [FM11], static analysis to identify problematic information-flows [HYH+04] or the runtime enforcement of security policies with secure multi-execution of systems [DGDNP12, AF12].

Security Lattices. In this work, we focus on security policies defined over a set of security labels, SC, with the allowed flows specified as a *label can-flow* relation,[2] denoted $\sqsubseteq$, over SC. Then, $L \sqsubseteq L'$ specifies the requirement that information from entities labeled with label $L \in \text{SC}$ can flow to entities labeled with $L' \in \text{SC}$. For an entity x, its labeling with L is denoted as x^L or, equivalently, we may say that x is in the label L and denote it as $x \in L$. To support dynamic assignment of labels, a policy must provide a joint

[2] In the literature, label can-flow relations are referred to as flow or can-flow relations. We, however, define flow relations in terms of objects and not labels later in this manuscript in Section 3.1.

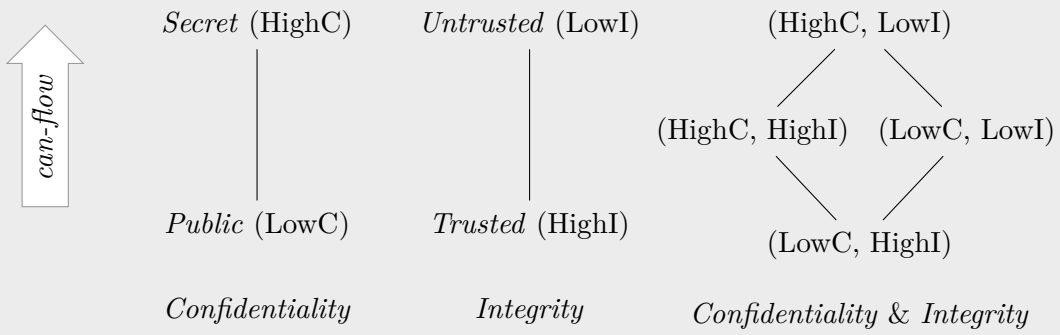
operator $\oplus \subseteq SC \times SC$ specifying how to combine entities assigned with different labels. In summary, a policy is specified by a tuple $(SC, \sqsubseteq, \oplus)$. In [Den76], Denning proposed the following axioms for security policies $(SC, \sqsubseteq, \oplus)$:

- *Finiteness*: the set of security classes SC is finite;
- *Order*: $(SC, \sqsubseteq)$ defines a partial order;
- *Public information*: there exists an unique lower bound in SC with respect to $\sqsubseteq$;
- *All information can be combined*: the joint operator $\oplus$ is a least upper bound operator defined for every pair of security classes.

When a security policy satisfies these assumptions, it defines a bounded lattice [Den76]. We denote by high its unique upper bound and by low its unique lower bound.

Example: Security Lattices

We depict security lattices as Hasse diagrams, with reflexive and transitive arrows omitted. Below, we depict three security lattices encoding the basic requirements for confidentiality, integrity and their constraints combined. As illustrated by the lattices, confidentiality and integrity adopt inverse views: the first lattice does not allow flows from high to low confidentiality, while the second does not allow flows from low to high integrity.



Dynamic Policies. It is often necessary to specify how security policies change during a system execution. For example, a user password should not be leaked to the system before a successful login. However, this requirement must be relaxed once the login is successful. Otherwise, the policy would prevent the user from being informed that the login was successful, as this leaks the information that the input password is correct. This is an example of *declassification* of information [SS05], where information previously secret to a user becomes available to the same user. Other use cases may be related, for example, to a change in the system's overall status from a normal operation state to a repair/debug state.

5.2 Properties

An information-flow property instantiates a security policy by providing a semantic interpretation for the can-flow relation over the system's observations. In this work, we refer to policies as the *structural view on information-flow* (defining *where* information can flow), and properties as the *semantic view on information-flow* (defining *what* is to flow).

A property specification depends on many factors, from how the system is observed and represented to the system's computational and thread model. For simplicity of presentation, we do not elaborate further on these considerations (for a more in-depth discussion, see the monograph by Kozyri et.al. [KCM+22]). For what follows, we assume the system is conveniently abstracted as a set of traces T and follow the approach of Clarkson and Schneider in [CS10] of using set comprehension and first-order logic to introduce our properties of interest.

Non-interference. Given a security policy defining a can-flow relation between a set of security labels, *non-interference* properties require that behavior visible to a security label L should not depend on the behavior of a higher security label (i.e., a label that is not in the can-flow relation with L). For example, non-interference, as defined originally by Goguen and Meseguer [GM82], requires that removing inputs labeled with high-confidentiality should not change what users holding a low-confidentiality security clearance observe.

In the seminal work by Zdancewic and Myers [ZM03], the authors propose to define non-interference for concurrent programs as *observational determinism*. In particular, they define a program to be observational deterministic iff, for all program executions that start with equivalent observations for a low-level security user, they remain equivalent to the same user along their execution.

While this definition is simple and intuitive, its formalization can be subtle, and it is particularly challenging for multi-threaded programs. Note that the observable behavior from different executions of a multi-thread program is not easily comparable because (i) they depend on the scheduling policy (which may differ significantly between different executions of the system) and (ii) program state changes may not occur at the same time in all executions. To address this misalignment between different system observations, Zdancewic and Myers [ZM03] propose to interpret equivalence between traces up to their *stuttering* and *prefixing* (i.e., one execution may run for longer). Other authors later refined this definition [HWS06, Ter08], always keeping the requirement to compare traces modulo their stuttering.

Side Note: Specifying Observational Determinism

In this example, we show how to specify different versions of observational determinism depending on our assumptions about the system and how we observe it. We consider the case of confidentiality with only two security labels, low and high, where, as expected, $\mathsf{low} \sqsubseteq \mathsf{high}$.

We start by considering a *synchronous* interpretation of observational determinism. Here we assume that the time of the observations between different system's executions is synchronized, i.e., each time point corresponds to that same execution time of the system. We specify observational determinism over the program's public input and output variable, denoted in^{low} and $\text{out}^{\mathsf{low}}$, respectively. In our first specification, we assume that the behavior of each iteration of the program corresponds to one time point in the trace (i.e., in the same time point we see the input and output for that program iteration). Then, we define observational determinism as a globally property as follows:

$$\begin{aligned}\text{OD}_{\text{sync},\mathbf{G}} = \{T \mid \forall \tau \in T\, \forall \tau' \in T\, \forall i \in \mathbb{N} :\\ \tau[i](\text{in}^{\mathsf{low}}) = \tau'[i](\text{in}^{\mathsf{low}}) \rightarrow \tau[i](\text{out}^{\mathsf{low}}) = \tau'[i](\text{out}^{\mathsf{low}})\}.\end{aligned}$$

This specification is equivalent to the following HyperLTL formula:

$$\varphi_{\text{sync},\mathbf{G}} \stackrel{\text{def}}{=} \forall \pi \forall \pi'\, \mathbf{G}\, ((\text{in}^{\mathsf{low}}_{\pi} \leftrightarrow \text{in}^{\mathsf{low}}_{\pi'}) \rightarrow (\text{out}^{\mathsf{low}}_{\pi} \leftrightarrow \text{out}^{\mathsf{low}}_{\pi'})).$$

Alternatively, it may be the case that the input value is only relevant at the beggining of the traces, and from then on we are only interested in the output behavior. We specify this variant below:

$$\begin{aligned}\text{OD}_{\text{sync},\mathbf{X}} = \{T \mid \forall \tau \in T\, \forall \tau' \in T' :\ \tau[0](\text{in}^{\mathsf{low}}) = \tau'[0](\text{in}^{\mathsf{low}}) \rightarrow\\ (\forall i \in \mathbb{N} : i > 0 \wedge \tau[i](\text{out}^{\mathsf{low}}) = \tau'[i](\text{out}^{\mathsf{low}})\},\end{aligned}$$

which is equivalent to the following HyperLTL formula:

$$\varphi_{\text{sync},\mathbf{X}} \stackrel{\text{def}}{=} \forall \pi \forall \pi' (\text{in}^{\mathsf{low}}_{\pi} \leftrightarrow \text{in}^{\mathsf{low}}_{\pi'}) \rightarrow \mathbf{X}\, \mathbf{G} (\text{out}^{\mathsf{low}}_{\pi} \leftrightarrow \text{out}^{\mathsf{low}}_{\pi'}).$$

If we do not assume the program executions to be synchronous (i.e., they are *asynchronous*), then we can not compare each time point directly. Instead, we can follow the approach by Zdancewic and Myers and require that every publicly visible variable (i.e., $x \in \mathsf{low}$) must be stutter-equivalent up-to prefixing:

$$\text{OD}_{\text{async}} = \{T \mid \forall \tau \in T\, \forall \tau' \in T : \bigwedge_{x \in \mathsf{low}} (\tau(x) \precsim \tau'(x) \vee \tau'(x) \precsim \tau(x))\}$$

where $\precsim$ represents the stutter-equivalence up-to prefixing relation and $\tau(x)$ is the projection of all values of x in a given trace τ.

The original definition of non-interference by Goguen and Meseguer is for systems represented as deterministic state machines. McCullough in [McC87] introduce *generalized non-interference (GNI)* extending the original definition of non-interference to handle non-deterministic systems. In this work, we adopt the definition by McLean [McL96] of GNI, requiring that for any two executions there exists a third execution that has the same high-level inputs (with respect to given can-flow relation) as the first execution and the same low-level outputs of the second execution. Formally, generalized non-interference can be specified as follows:

$$\begin{aligned}\mathrm{GNI} = \{T \mid \forall \tau \in T\, \forall \tau' \in T\, \exists \tau_\exists \in T\, \forall i \in \mathbb{N} :\\ \tau[i](\mathrm{in}^{\mathsf{high}}) = \tau_\exists[i](\mathrm{in}^{\mathsf{high}}) \wedge \tau'[i](\mathrm{out}^{\mathsf{low}}) = \tau_\exists[i](\mathrm{out}^{\mathsf{low}})\}.\end{aligned}$$

We remark that this definition of generalized non-interference is akin to the notion of *independence*, which is ubiquitous across multiple scientific disciplines [GV13].

CHAPTER 6

Expressing Information-flow with Linear Hyperlogics

Motivated by the observation that specifying hyperproperties requires both quantification over time and traces (as opposed to trace properties with quantification only over time), we study in this chapter how constraints over these two types of quantifiers affect the expressiveness of logics for linear-time hyperproperties.

We first introduce a first-order logic, called *Hypertrace Logic*, with an explicit distinction between trace and time quantification as different sorts. We then use this logic to specify variants of a two-state information-flow policy. In particular, we look into *sequential generalized non-interference* (with non-interference specified as independence), allowing us to explore different quantification alternation patterns in depth. Guided by this example, we propose fragments of Hypertrace Logic by restricting the order of the trace and time quantifiers. A fragment of particular importance is the one that forces all trace quantifiers to occur before time quantifiers, which is equivalent to HyperLTL.

For the fragments of Hypertrace logic identified, we introduce relations characterizing indistinguishability between sets of traces for the said fragments. Using these relations, we prove that HyperLTL cannot specify most of the studied two-state information flow variants, including variants with an asynchronous transition between the two states of the specification.

This chapter extends on the invited paper below, which results from a collaboration with Ezio Bartocci, Thomas Ferrère, Thomas Henzinger and Dejan Nickovic:

> [BFH+22a] 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, *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 1–19. Springer International Publishing, 2022.

6.1 Hypertrace Logic

Hypertrace Logic, denoted FO[$<, \mathbb{T}$], extends the first-order logic of linear order with equality, denoted FO[$<$], with a time sort $\mathbb{N}_<$ and a trace sort $\mathbb{T}$. The logic FO[$<$], introduced previously in Section 4.1, is a first-order logic interpreted over labeled linear orders allowing only unary uninterpreted predicates. We are interested in discrete linear time (i.e., systems' executions abstracted as traces), so we adopt the interpretation of FO[$<$] over the theory of natural numbers (i.e., the linear order $(\mathbb{N}, <)$ where $<$ is defined as usual over the set of natural numbers). Under this theory, FO[$<$] is expressively equivalent to LTL as introduced in this manuscript (i.e., only future operators) [GPSS80].

While FO[$<$] allows only unary uninterpreted predicates (i.e., all predicates that are not equality, $=$, or the linear order, $<$); in hypertrace logic, we allow binary uninterpreted predicates defined over pairs of a trace and a time variable. Formally, all hypertrace formulas φ are defined by the grammar:

$$\psi ::= P(\pi, i) \mid i < i \mid i = i$$
$$\varphi ::= \psi \mid \neg\varphi \mid \varphi \vee \varphi \mid \exists \pi\, \varphi \mid \exists i\, \varphi$$

where π is a trace variable, i is a time variable and Q is a binary predicate over pairs of trace and time variables. From now on, $\mathcal{V}_\mathbb{T} = \{\pi, \pi', \ldots, \pi_1, \ldots\}$ is a set of trace variables, and $\mathcal{V}_\mathbb{N} = \{i, i', \ldots, j, \ldots\}$ is a set of time variables.

Let T be a set of traces and Π_T be a trace assignment over it. We define $\mathcal{V}(\Pi_T) = \{\pi \mid \Pi_T(\pi) \text{ is defined}\}$ as the set of trace variables with an assignment in Π_T; and $|\Pi_T| = |\mathcal{V}(\Pi_T)|$ as the trace assignment size. Another useful definitions in what follows are of *sequential update* of two valuations v and v', where $X(v') = \{x_1, \ldots, x_n\}$, which is defined as $v \cdot v' = v[x_1 \mapsto v'(x_1)] \ldots [x_n \mapsto v'(x_n)]$; and of *sequential update* of two two traces $\tau = v_0 v_1 \ldots$ and $\tau' = v'_0 v'_1 \ldots$ defined as $\tau \cdot \tau' = (v_0 \cdot v'_0)(v_1 \cdot v'_1) \ldots$.

Model-checking. In the model-checking problem, we check whether a set of traces is a model of a hypertrace formula. As Hypertrace Logic is a first-order formalism, we start by defining how to go from a set of traces to a first-order structure.

In the translation from LTL to FO[$<$], each propositional variable $a \in X$ is translated to a monadic predicate $P_a(k)$, asserting that "variable a is true at time $k \in \mathbb{N}_<$". Hypertrace formulas are defined with binary predicates over pairs of trace and time variables, i.e., with type $\mathbb{T} \times \mathbb{N}_<$. Then, we translate each propositional variable $a \in X$ to the binary predicate $P_a(\tau, k)$, asserting that "variable a is true in trace $\tau \in \mathbb{T}$ at time $k \in \mathbb{N}_<$". Additionally, to accommodate for finite traces, we include the binary predicate *def* including all time positions within the length of a given trace. Formally, given a set T of traces, we translate T to a structure $\overline{T}$ with signature:

$$(\mathbb{N}, T;\ < : \mathbb{N} \times \mathbb{N}, (P_a : T \times \mathbb{N})_{a \in X}, \mathit{def} : T \times \mathbb{N})$$

where $\mathbb{N}$ and T are the time and trace sort domains, respectively. The predicate $<$ is

interpreted as the usual partial order over the theory of natural numbers, while for all variables $a \in X$, we have:

$$P_a = \{(\tau, k) \mid \tau \in T, k \in \mathbb{N}, \text{ and } \tau[k](a) = 1\}$$
$$def = \{(\tau, k) \mid \tau \in T, k \in \mathbb{N}, \text{ and } 0 \leq k < |\tau|\}.$$

As we have two sorts of variables in hypertrace formulas, we evaluate whether a set of traces T satisfies a hypertrace formula over pair of assignments:

$$(\Pi_T^{\mathbb{T}}, \Pi^{\mathbb{N}}) : (\mathcal{V}_{\mathbb{T}} \to T) \times (\mathcal{V}_{\mathbb{N}} \to \mathbb{N}).$$

We denote an assignment update as $(\Pi_T^{\mathbb{T}}, \Pi^{\mathbb{N}})[x \mapsto v]$ where x is mapped to v, and other variables remain unchanged. Then, a set T of traces is a model of a hypertrace formula $\varphi \in \text{FO}[<, \mathbb{T}]$, denoted $T \models_{\mathbb{T}} \varphi$, iff $\overline{T}$ models φ under the standard first-order semantics. Formally, $T \models_{\mathbb{T}} \varphi$, iff there exists a pair of assignments $(\Pi_T^{\mathbb{T}}, \Pi^{\mathbb{N}})$ such that $(\overline{T}, (\Pi_T^{\mathbb{T}}, \Pi^{\mathbb{N}})) \models \varphi$, where $\models$ is the standard first-order logic models relation. For the hypertrace formula φ we define the hyperproperty containing all sets of traces that are models of φ, denoted $[\![\varphi]\!]$, as $[\![\varphi]\!] = \{T \mid T \models_{\mathbb{T}} \varphi\}$. From now on, we refer to P_a as a, and omit the subscript $\mathbb{T}$ in $\models_{\mathbb{T}}$ when clear from context.

Example: Hypertrace Logic Semantics

The hypertrace formula φ below specifies that *"there exists a time point i such that for all traces, the proposition a holds at that time i"*:

$$\varphi \overset{\text{def}}{=} \exists i \forall \pi \; a(\pi, i).$$

An example of a formula with a different order for the trace and time quantifiers is the formula φ' below requiring that *"there exist two traces that are each other complement with respect to the value of a"*:

$$\varphi' \overset{\text{def}}{=} \exists \pi \exists \pi' \forall i \; a(\pi, i) \leftrightarrow \neg a(\pi', i).$$

Now consider, for example, the two trace sets below defined with traces over a represented as strings of length 1 (i.e., $[v(a)]$):

$$T = \{[0][1][0][1][0]^\omega, \; [0][0][0][1]^\omega\} \qquad T' = \{([0][1])^\omega, \; ([1][0])^\omega, \; [0]^\omega\}$$

The set T satisfies the formula φ ($T \models_{\mathbb{T}} \varphi$) because for both traces in T at time 3 the value of a is 1. While the set T' satisfies the formula φ' ($T' \models_{\mathbb{T}} \varphi'$) with the two top most traces witnessing the satisfaction of the existential requirement of φ'.

On the other hand, the set T does not satisfy φ' ($T \not\models_{\mathbb{T}} \varphi'$) and the set T' does not satisfy φ ($T' \not\models_{\mathbb{T}} \varphi$).

Linear Logic with Equal-level Predicate. Finkbeiner and Zimmermann introduce in [FZ17] an alternative extension of FO[<] to express linear-time hyperproperties. They add to FO[<] the equal-level predicate E [Tho92] and denoted it by FO[$<, E$]. Their domain of interpretation are pairs of type $T \times \mathbb{N}$, where T is a set of traces. We call the domain elements *time positions*. The equal-level predicate is a binary predicate over time positions specifying which positions correspond to the same time point. Besides the binary equal-level predicate, FO[$<, E$] allows only unary predicates. To solve the model-checking problem for FO[$<, E$] we translate sets of traces T to the first order structure $\overline{T}^E$ with signature:

$$(T \times \mathbb{N}; <^E : (T \times \mathbb{N}) \times (T \times \mathbb{N}), E : (T \times \mathbb{N}) \times (T \times \mathbb{N}), (P_a : T \times \mathbb{N})_{a \in X})$$

where:

$$\begin{aligned} <^E &= \{((\tau, n), (\tau, n')) \mid \tau \in T \text{ and } n < n'\}, \\ E &= \{((\tau, n), (\tau', n)) \mid \tau, \tau' \in T \text{ and } n \in \mathbb{N}\} \text{ and} \\ P_a &= \{(\tau, n) \mid \tau[n](a) = 1\}. \end{aligned}$$

As usual, the direct successor predicate is defined as $\text{Succ}(x, y) \stackrel{\text{def}}{=} x < y \wedge \neg\exists z(x < z < y)$. To allow to distinguish between trace and time quantification, we define minimal pairs as $\min(x) \stackrel{\text{def}}{=} \neg\exists y\ \text{Succ}(y, x)$ as the time positions that mark the beginning of each trace.

Given a set T of traces, formulas φ of FO[$<, E$] are interpreted over assignments $\Pi_T^E : \mathcal{V} \to (T \times \mathbb{N})$. Then, $T \models_E \varphi$ iff there exists an assignment Π_T^E that witnesses the model relation, i.e., $(\overline{T}^E, \Pi_T^E) \models \varphi$ under the standard first-order semantics. While there is no explicit distinction between trace and time quantifiers, it is possible to encode this distinction implicitly. In [FZ17], the authors introduce *minimal-time quantifiers* to define implicit quantification over traces. Minimal-time quantifiers are denoted by $Q^M x\, \varphi$, with $Q \in \{\forall, \exists\}$, and defined by the following shorthands:

$$\forall^M x\, \varphi \stackrel{\text{def}}{=} \forall x\, (\min(x) \to \varphi) \quad \text{and} \quad \exists^M x\, \varphi \stackrel{\text{def}}{=} \exists x\, (\min(x) \wedge \varphi).$$

Example: From Hypertrace Formulas to Equal-level Formulas

The hypertrace formula $\varphi \stackrel{\text{def}}{=} \exists i \forall \pi\, a(\pi, i)$ from our previous example is equivalent to the equal-level formula below, which we will explain next:

$$\exists x_i\, \forall^M x_\pi\, \exists x_{(\pi,i)}\, E(x_{(\pi,i)}, x_i) \wedge x_\pi \leq x_{(\pi,i)} \wedge P_a(x_{(\pi,i)}).$$

To help distinguish between time quantifiers and minimal-time quantifiers, we annotate the time-position variables (denoted by x) with either a time reference (denoted by i), a trace reference (denoted by π or π') or a pair of them. We use the predicate $E(x_{(\pi,i)}, x_i)$ to guarantee that $x_{(\pi,i)}$ has the same time index as x_i, i.e., we make sure our annotation is correct and both time-position variables refer to the time i. Moreover, since x_π is in the scope of a minimal time quantifier, the predicate $x_\pi \leq x_{(\pi,i)}$ guarantees that $x_{(\pi,i)}$ has the same trace identifier as x_π.

The hypertrace formula $\exists\pi\exists\pi'\forall i\,(a(\pi,i) \leftrightarrow \neg a(\pi',i))$ is equivalent to the equal-level formula:

$$\exists^M x_\pi\, \exists^M x_{\pi'}\, \forall x_i\, \exists x_{(\pi,i)}\ E(x_{(\pi,i)}, x_i) \wedge x_\pi \leq x_{(\pi,i)} \wedge \\ \exists x_{(\pi',i)}\, E(x_{(\pi',i)}, x_i) \wedge x_{\pi'} \leq x_{(\pi',i)} \wedge (P_a(x_{(\pi,i)}) \leftrightarrow \neg P_a(x_{(\pi',i)})).$$

We prove next that Hypertrace Logic and FO[$<, E$] are equally expressive for sets of infinite traces. Translating from equal-level formulas to hypertrace formulas is straightforward due to the explicit trace and time quantification in FO[$<, \mathbb{T}$]. The other direction is more challenging. We define in the proof below a translation that first encodes binary predicates of FO[$<, \mathbb{T}$] into unary predicate over variables named with the intended trace and time variables (i.e., a predicate $x(\pi, i)$ will define the variable $x_{(\pi,i)}$). We then introduce minimal quantifiers appropriately to guarantee that $x_{(\pi,i)}$ has the same trace identifier as a minimal variable x_π with $x_\pi \leq x_{(\pi,i)}$, and has the same time variable as x_i using the equal-level predicate $E(x_i, x_{(\pi,i)})$.

Theorem 6.1.1. *For all equal-level sentences $\varphi_E \in FO[<, E]$ there exists a hypertrace sentence $\varphi \in FO[<, \mathbb{T}]$ such that for all sets $T \subseteq \mathbb{V}_X^\omega$ of infinite traces, we have $T \models_E \varphi_E$ iff $T \models_{\mathbb{T}} \varphi$. For all hypertrace sentences $\varphi \in FO[<, \mathbb{T}]$ there exists an equal-level sentence $\varphi_E \in FO[<, E]$ such that for all sets $T \subseteq \mathbb{V}_X^\omega$ of infinite traces, we have $T \models_{\mathbb{T}} \varphi$ iff $T \models_E \varphi_E$.*

Proof. For what follows, $\mathcal{V}$ is a set of variables (as used by equal-level formulas), T is a set of traces, $\mathcal{V}_{\mathbb{T}} = \{v_\pi \mid v \in \mathcal{V}\}$ is a set of trace variables and $\mathcal{V}_{\mathbb{N}_<} = \{v_i \mid v \in \mathcal{V}\}$ is a set of time variables. We start with the $\Rightarrow$-direction and start by defining a translation from equal-level formulas to hypertrace formulas, $\mathrm{tr}_{\mathbb{T}}$, recursively as follows:

$$\mathrm{tr}_{\mathbb{T}}(\forall x\, \varphi) = \forall x_i \forall x_\pi \mathrm{tr}_{\mathbb{T}}(\varphi) \qquad \mathrm{tr}_{\mathbb{T}}(\exists x\, \varphi) = \exists x_i \exists x_\pi \mathrm{tr}_{\mathbb{T}}(\varphi)$$
$$\mathrm{tr}_{\mathbb{T}}(E(x,y)) = (x_i = y_i) \qquad \mathrm{tr}_{\mathbb{T}}(x < y) = (x_\pi = y_\pi \wedge x_i < y_i).$$

Additionally, we define below a translation from equal-level assignments to hypertrace assignments with type: $\mathrm{tr} : (\mathcal{V} \to (T \times \mathbb{N})) \to ((\mathcal{V}_{\mathbb{T}} \to T) \times (\mathcal{V}_{\mathbb{N}_<} \to \mathbb{N}))$. Then, given an equal-level assignment $\Pi_T^E : \mathcal{V} \to (T \times \mathbb{N})$ its translation for all variables $x \in \mathcal{V}$ where the assignment is defined, i.e. $\Pi_T^E(x) = (\tau, i)$, is just the assignment of each element of the pair to the respective sort, formally $\mathrm{tr}(\Pi)(x_\pi) = \tau$ and $\mathrm{tr}(\Pi)(x_i) = i$. Then, it follows by

structural induction on equal-level formulas φ_E that for all set of traces T and equal-level assignments Π_T^E, $(\overline{T}^E, \Pi_T^E) \models \varphi_E$ iff $(\overline{T}, \mathrm{tr}(\Pi_T^E)) \models \mathrm{tr}_{\mathbb{T}}(\varphi_E)$.

We proceed now to the $\Leftarrow$-direction. We translate hypertrace sentences $\varphi \in \mathrm{FO}[<, \mathbb{T}]$ to equal-level predicate sentence $\varphi_E \in FO[<, E]$ recursively with tr_E as follows:

$$\begin{aligned}
&\mathrm{tr}_E(\forall \pi\ \varphi) = \forall^M x_\pi\, \mathrm{tr}_E(\varphi) && \mathrm{tr}_E(\exists \pi\ \varphi) = \exists^M x_\pi\, \mathrm{tr}_E(\varphi)\\
&\mathrm{tr}_E(\forall i\ \varphi) = \forall x_i\, \mathrm{tr}_E(\varphi) && \mathrm{tr}_E(\exists i\ \varphi) = \exists x_i\, \mathrm{tr}_E(\varphi)\\
&\mathrm{tr}_E(\mathit{def}(\pi, i)) = \mathrm{true} && \mathrm{tr}_E(a(\pi, i)) = \exists x_{(\pi,i)}\, E(x_{(\pi,i)}, x_i) \wedge x_\pi \leq x_{(\pi,i)} \wedge P_a(x_{(\pi,i)}).
\end{aligned}$$

We define below a translation from hypertrace assignments to equal-level assignments with type $\mathrm{tr} : ((\mathcal{V}_{\mathbb{T}} \to T) \times (\mathcal{V}_{\mathbb{N}_<} \to \mathbb{N})) \to (\mathcal{V}_{(\mathbb{T},\mathbb{N}_<)} \to (T \times \mathbb{N}))$. We use the same name as in the other direction of the proof to avoid introducing more notation. Then, we define the translation between assignments recursively as follows, for all pair of variables $\pi \in \mathcal{V}_{\mathbb{T}}$ and $i \in \mathcal{V}_{\mathbb{N}_<}$ that are defined in a given $(\Pi_T^{\mathbb{T}}, \Pi^{\mathbb{N}_<})$, i.e., $\Pi_T^{\mathbb{T}}(\pi) = \tau$ and $\Pi^{\mathbb{N}_<}(i) = n$:

$$\begin{aligned}
&\mathrm{tr}((\Pi_T^{\mathbb{T}}, \Pi^{\mathbb{N}_<}))(x_\pi) = (\tau, 0)\\
&\mathrm{tr}((\Pi_T^{\mathbb{T}}, \Pi^{\mathbb{N}_<}))(x_i) = (\tau', n), \text{ for some } \tau' \in T\\
&\mathrm{tr}((\Pi_T^{\mathbb{T}}, \Pi^{\mathbb{N}_<}))(x_{(\pi,i)}) = (\tau, n).
\end{aligned}$$

It follows by structural induction on hypertrace formulas φ that for all set of traces T and pair assignments $(\Pi_T^{\mathbb{T}}, \Pi^{\mathbb{N}_<})$, $(\overline{T}, (\Pi_T^{\mathbb{T}}, \Pi^{\mathbb{N}_<}) \models \varphi$ iff $(\overline{T}^E, \mathrm{tr}((\Pi_T^{\mathbb{T}}, \Pi^{\mathbb{N}_<}))) \models \mathrm{tr}_E(\varphi)$. □

Fragments of Hypertrace Logic. In this work, we focus on two fragments of hypertrace logic with restrictions on the type of quantifiers' relative ordering. For example, in the specification of the two variants of stateless independence in Definition 6.2.1, the only difference between them is the relative ordering between the trace and time quantifiers: segment semantics has trace quantifiers first, followed by a time quantifier, while for point semantics, it is the other way around. For each fragment, we will present an equivalence relation over sets of traces, establishing which sets of traces are indistinguishable for the respective hypertrace logic fragment. Our goal is to provide a framework to study the expressive power of linear-time hyperlogics, like HyperLTL.

6.1.1 Trace-prefixed

We start by studying the fragment where trace quantifiers occur first, followed by time quantifiers. We call this the *trace-prefixed hypertrace logic*, denoted $\mathbf{T}$-FO$[<, \mathbb{T}]$. Formally, trace-prefixed formulas $\varphi \in \mathbf{T}$-FO$[<, \mathbb{T}]$ are defined by the grammar:

$$\begin{aligned}
&\psi ::= \forall i\ \psi \mid \psi \vee \psi \mid \neg\psi \mid i < i \mid i = i \mid P(\pi, i)\\
&\varphi ::= \forall \pi\, \varphi \mid \exists \pi\, \varphi \mid \psi
\end{aligned}$$

where π is a trace variable, i is a time variable, and P is a binary predicate.

Without surprise, $\mathbf{T}$-FO$[<, \mathbb{T}]$ is expressively equivalent to HyperLTL [CFK+14] when interpreted over sets of infinite traces. We prove this result in Theorem 6.1.3 below, lifting the equivalence between FO$[<]$ and LTL by Gabbay et.al. [GPSS80].

Our first observation is that a quantifier-free HyperLTL formula φ over trace variables $\mathcal{V}$ and propositions X is a LTL formula over the alphabet $\{a_\pi \mid a \in X, \pi \in \mathcal{V}\}$. So, to prove our results we first need to define a translation between assignments over sets of traces and propositional variables, and assignments only over propositional variables. Formally, for a set of traces T and an assignment $\Pi_T : \mathcal{V} \to T$ over it and variables in $\mathcal{V}$, we define its *flattening* as:

$$\langle \Pi_T \rangle[i](a_\pi) = \Pi_T(\pi)[i](a)$$

for all time variables $i \in \mathbb{N}$, trace variables $\pi \in \mathcal{V}$ and propositional variables $a \in X$. We prove below that a trace assignment over a set of traces satisfies a quantifier-free HyperLTL formula iff its flattening satisfies the same formula under the LTL semantics.

Proposition 6.1.2. *Let φ be a quantifier-free HyperLTL formula. For all $i \in \mathbb{N}$, all trace sets T, and all corresponding trace assignments Π_T, $(\Pi_T, i) \models_{\mathrm{H}} \varphi$ iff $\langle \Pi_T \rangle[i \ldots] \models \varphi$.*

Proof. We prove the result by structural induction on quantifier-free HyperLTL formulas. The inductive cases follow directly from induction hypothesis. As for the base case, it follows directly from definition of flattening. □

Using the results by Gabbay et.al. [GPSS80] and the proposition above, we prove below that hypertrace logic and HyperLTL are equivalent.

Theorem 6.1.3. *For all HyperLTL sentences φ_H there exists a trace-prefixed hypertrace sentence φ such that for all sets of infinite traces $T \subseteq \mathbb{V}_X^\omega$, we have $T \models_{\mathrm{H}} \varphi_H$ iff $T \models_{\mathbb{T}} \varphi$. For all trace-prefixed hypertrace sentences φ there exists a HyperLTL sentence φ_H such that for all sets of infinite traces $T \subseteq \mathbb{V}_X^\omega$, we have $T \models_{\mathrm{H}} \varphi_H$ iff $T \models_{\mathbb{T}} \varphi$.*

Proof. The translation from HyperLTL formulas to an equivalent trace-prefixed hypertrace formula works as follows. We keep the trace quantifiers as they are and we use the translation from LTL to FO$[<]$ introduced in [GPSS80] to translate the quantifier-free part. Then, we apply the following change in the quantifier-free part: $P_a(\pi, i) = P_{a_\pi}(i)$. Let us call this translation tr_H. It follows from structural induction on HyperLTL formulas that for all sets of traces and their assignments they satisfy an HyperLTL formula iff they satisfy its translation to the trace-prefixed hypertrace formula. This follows from the result by Gabbay et al. in [GPSS80] and Proposition 6.1.2 for the base case of this induction. Hence for all HyperLTL formulas φ_H there exists the trace-prefixed hypertrace formula $\mathrm{tr}_H(\varphi_H)$ s.t. $T \models_{\mathrm{H}} \varphi_H$ iff $T \models_{\mathbb{T}} \mathrm{tr}_H(\varphi_H)$.

The translation from trace-prefixed hypertrace formulas to HyperLTL is similar. We use instead the translation from FO$[<]$ to LTL from [GPSS80]. □

From now on, when dealing with infinite traces, we may refer to trace-prefixed hyperlogic as HyperLTL. This name convention emphasises its direct connection to LTL and, by doing so, allows us to seamlessly lift LTL results to this fragment of hypertrace logic.

From LTL to HyperLTL Indistinguishable Models

We recall that quantifier-free HyperLTL formulas are LTL formulas. Moreover, we know that the number of quantifiers determines an upper bound on the number of traces that can be simultaneously compared while evaluating HyperLTL formulas. From these two remarks, it is a natural step to define equivalences on trace sets for HyperLTL formulas by lifting results from the LTL literature. We formalize this intuition below.

Let $\mathbb{C}$ be a class of LTL formulas and $\approx$ be an equivalence relation on traces. We say that $\approx$ is $\mathbb{C}$-*preserving* if for all LTL formulas in that class, $\varphi \in \mathbb{C}$, and all traces τ and τ' in the equivalence relation, $\tau \approx \tau'$, we have $\tau \models \varphi$ iff $\tau' \models \varphi$. For example, if $\mathbb{C}$ is the set of LTL formulas without next ($\mathbf{X}$) operator, and the equivalence classes of $\approx$ are closed under stuttering, then $\approx$ is $\mathbb{C}$-preserving, i.e., stutter-invariant [PW97].

Let let $\varphi = Q_0\pi_0 \dots Q_k\pi_k\psi$ be a HyperLTL formula with ψ being quantifier-free and $Q_i \in \{\forall, \exists\}$ for all $0 \leq i \leq k$. For a class of LTL formulas $\mathbb{C}$, φ is in the *k-extension* of $\mathbb{C}$, denoted $\varphi \in 2^{\mathbb{C}}_k$, if $\psi \in \mathbb{C}$. We can now lift a $\mathbb{C}$-preserving equivalence $\approx$ on traces to a $2^{\mathbb{C}}_k$-preserving equivalence on trace sets, by requiring a bijective translation between trace sets which preserves $\approx$ for all assignments of size k.

Definition 6.1.1. *Let $k \in \mathbb{N}$, let $\mathbb{C}$ be a class of LTL formulas, and let $\approx$ be a $\mathbb{C}$-preserving equivalence on traces. Two sets T and U of traces are $(k, \mathbb{C})$-equivalent, denoted $T \approx_{(k,\mathbb{C})} U$, iff there exists a bijective and total function $f : T \to U$, such that for all sets $\mathcal{V}$ of k trace variables and all trace assignments $\Pi : \mathcal{V} \to T$ and $\Pi' : \mathcal{V} \to U$, we have $\langle \Pi \rangle \approx \langle f(\Pi) \rangle$ and $\langle \Pi' \rangle \approx \langle f^{-1}(\Pi') \rangle$, where $f(\Pi)(\pi) = f(\Pi(\pi))$ for all $\pi \in \mathcal{V}$.*

We prove below that the lift from indistinguishability relations for LTL formulas defined above specifies a proper indistinguishability relation between sets of traces concerning HyperLTL formulas.

Theorem 6.1.4. *Let $\mathbb{C}$ be a class of LTL formulas and $\approx$ a $\mathbb{C}$-preserving equivalence on traces. Let $\varphi \in 2^{\mathbb{C}}_k$ be a HyperLTL sentence in the k-extension of $\mathbb{C}$, for some $k \in \mathbb{N}$. For all sets T and U of traces with $T \approx_{(k,\mathbb{C})} U$, we have $T \models \varphi$ iff $U \models \varphi$.*

The theorem follows from Lemma 6.1.5 below, which is shown by induction on the number k of trace quantifiers.

Lemma 6.1.5. *Let $\mathbb{C}$ be a class of LTL formulas and $k \in \mathbb{N}$. For all HyperLTL formulas $\varphi \in 2^{\mathbb{C}}_k$, all trace sets T and U with $T \approx_{(k,\mathbb{C})} U$, all functions $f : T \to U$ that witness the $(k, \mathbb{C})$-equivalence of T and U, and all trace assignments $\Pi : \mathit{free}(\varphi) \to T$ and $\Pi' : \mathit{free}(\varphi) \to U$ to the free variables in φ, we have*

$$(\Pi, 0) \models \varphi \text{ iff } (f(\Pi), 0) \models \varphi, \text{ and } (\Pi', 0) \models \varphi \text{ iff } (f^{-1}(\Pi'), 0) \models \varphi.$$

Proof. We prove this statement by structural induction on HyperLTL formulas on a class $2^{\mathbb{C}}$. The class $\mathbb{C}$ affects only the quantifier-free part of the formula.

We start with the base case where $\varphi \in 2^{\mathbb{C}}$ is quantifier-free, i.e., $\text{free}(\varphi) = \mathcal{V}(\varphi)$. Note that φ can be interpreted as an LTL formula over the set of propositional variables $X_{\mathcal{V}(\varphi)}$ with $\varphi \in \mathbb{C}$. Consider arbitrary set of traces s.t. $T \approx_{(|\mathcal{V}(\varphi)|,\mathbb{C})} U$ with f being a function that witnesses it. Now, consider an arbitrary Π_T and Π_U over U and T, respectively, s.t. $\mathcal{V}(\Pi_T) = \mathcal{V}(\Pi_U) = \text{free}(\varphi)$. By definition of $\approx_{(|\mathcal{V}(\varphi)|,\mathbb{C})}$, $\langle \Pi_T \rangle \approx_{\mathbb{C}} \langle f(\Pi_T) \rangle$ and, by definition of $\approx_{\mathbb{C}}$: $(\star)$ $\langle \Pi_T \rangle \models \varphi$ iff $\langle f(\Pi_T) \rangle \models \varphi$. By Proposition 6.1.2, $(\Pi_T, 0) \models \varphi$ iff $\langle \Pi_T \rangle[0 \ldots] \models \varphi$; and $(f(\Pi_T), 0) \models \varphi$ iff $\langle f(\Pi_T) \rangle[0 \ldots] \models \varphi$. Thus, $(\Pi_T, 0) \models \varphi$ iff $(f(\Pi_T), 0) \models \varphi$. Analogously, $(\Pi_U, 0) \models \varphi$ iff $(\langle f^{-1}(\Pi_U) \rangle, 0) \models \varphi$.

We proceed now to the induction steps and assume by induction hypothesis (IH) that the statement holds for arbitrary $\varphi \in 2^{\mathbb{C}}$. We start with the case for the universal quantifier – $\forall \pi\ \varphi$. Assume that *(i)* $T \approx_{(|\mathcal{V}(\forall \pi\ \varphi)|,\mathbb{C})} U$. Note that, wlog we can assume that quantifiers bind a variable already occurring in φ, i.e. $|\mathcal{V}(\forall \pi\ \varphi)| = |\mathcal{V}(\varphi)|$. Then, *(i')* $T \approx_{(|\mathcal{V}(\varphi)|,\mathbb{C})} U$, and it has the same witnesses as assumption *(i)*. Let $f : T \to U$ be a function that witnesses *(i)*. Now, consider arbitrary Π_T and Π_U, over T and U, s.t. $\mathcal{V}(\Pi_T) = \mathcal{V}(\Pi_U) = \text{free}(\forall \pi\ \varphi) = \text{free}(\varphi) \setminus \{\pi\}$. We prove next that:

$$(\Pi_T, 0) \models \forall \pi\ \varphi \text{ iff } (f(\Pi_T), 0) \models \forall \pi\ \varphi.$$

We start with the $\Rightarrow$-direction of the statement. Assume that $(\Pi_T, 0) \models \forall \pi\ \varphi$, then by HyperLTL satisfaction: $(\star)$ for all $\tau \in T : (\Pi_T[\pi \mapsto \tau], 0) \models \varphi$. By Definition 6.1.1, $\mathcal{V}(f(\Pi_T)) = \mathcal{V}(\Pi_T)$. Thus, $\mathcal{V}(f(\Pi_T)[\pi \mapsto \tau']) = \mathcal{V}(\Pi_T) \cup \{\pi\} = \text{free}(\varphi)$. We can apply the (IH), because $T \approx_{(|\mathcal{V}(\varphi)|,\mathbb{C})} U$, f witnesses it, and for all $\tau \in T$ then $\Pi_T[\pi \mapsto \tau]$ is an assignment over T. So, it follows: for all $\tau \in T : (f(\Pi_T[\pi \mapsto \tau]), 0) \models \varphi$.

Assume towards a contradiction that $(f(\Pi_T), 0) \not\models \forall \pi\ \varphi$. Then, by definition of HyperLTL satisfaction: there exists $\tau' \in U : (f(\Pi_T)[\pi \mapsto \tau'], 0) \not\models \varphi$. We can apply the (IH), because $\mathcal{V}(f(\Pi_T)[\pi \mapsto \tau']) = \text{free}(\varphi)$, *(i')* with f being one of its witnesses, and for all $\tau' \in U$ the $f(\Pi_T)[\pi \mapsto \tau']$ is an assignment over U. So, it follows:

$$\text{there exists } \tau' \in U : (f^{-1}(f(\Pi_T)[\pi \mapsto \tau']), 0) \not\models \varphi.$$

And by Definition 6.1.1: there exists $\tau' \in U : (f^{-1}(f(\Pi_T))[\pi \mapsto f^{-1}(\tau')], 0) \not\models \varphi$. As f is a bijective function, $(f^{-1}(f(\Pi_T)) = \Pi_T$, and so: there exists $\tau' \in U : (\Pi_T[\pi \mapsto f^{-1}(\tau')], 0) \not\models \varphi$. And this is equivalent to:

$$\text{there exists } \tau' \in U : \tau = f^{-1}(\tau') \text{ and } (\Pi_T[\pi \mapsto \tau], 0) \not\models \varphi.$$

Given that f is a surjective function, then: there exists $\tau \in T : (\Pi_T[\pi \mapsto \tau], i) \not\models \varphi$. This contradicts $(\star)$. So, the $\Rightarrow$-direction holds.

We now prove the $\Leftarrow$-direction by contra-position. Assume that $(\Pi_T, 0) \not\models \forall\pi\ \varphi$, then by HyperLTL satisfaction: there exists $\tau \in T : (\Pi_T[\pi \mapsto \tau], 0) \not\models \varphi$. We can apply now the (IH), because $\mathcal{V}(f(\Pi_T)[\pi \mapsto \tau']) = \text{free}(\varphi)$, *(i')* with f being one of its witnesses, and for all $\tau \in T$ the $\Pi_T[\pi \mapsto \tau]$ is an assignment over T. Then, it follows: there exists $\tau \in T : (f(\Pi_T[\pi \mapsto \tau]), 0) \not\models \varphi$. By Definition 6.1.1: there exists $\tau \in T : \tau' = f(\tau)$ and $(f(\Pi_T)[\pi \mapsto \tau']), 0) \not\models \varphi$. By f being surjective, it follows: there exists $\tau' \in U : (f(\Pi_T)[\pi \mapsto \tau']), 0) \not\models \varphi$. And by Definition of HyperLTL satisfaction: $(f(\Pi_T), 0) \not\models \forall\pi\ \varphi$. Thus, the $\Leftarrow$-direction of the statement holds, as well. Hence $(\Pi_T, 0) \models \forall\pi\ \varphi$ iff $(f(\Pi_T), 0) \models \forall\pi\ \varphi$.

Now, we prove $(\Pi_U, 0) \models \forall\pi\ \varphi$ iff $(f^{-1}(\Pi_U), 0) \models \forall\pi\ \varphi$. We start with the $\Rightarrow$-direction of the statement. Like in the previous case, we assume that $(\Pi_U, 0) \models \forall\pi\ \varphi$, and then we assume towards a contradiction that $(f^{-1}(\Pi_U), 0) \not\models \forall\pi\ \varphi$. The proof is analogous to the previous case up to the point that we infer:

$$\text{there exists } \tau \in T : \tau' = f(\tau) \text{ and } (\Pi_U[\pi \mapsto \tau'], i) \not\models \varphi.$$

Then, by f being total, there exists $\tau' \in U : (\Pi_U[\pi \mapsto \tau'], i) \not\models \varphi$. And this contradicts the assumption that $(\Pi_U, i) \models \forall\pi\ \varphi$.

The $\Leftarrow$- direction is analogous to the previous case, as well, up to the point that we infer:

$$\text{there exists } \tau' \in U : \tau = f(\tau') \text{ and } (f^{-1}(\Pi_T)[\pi \mapsto \tau]), 0) \not\models \varphi.$$

Then, by f being total we infer: there exists $\tau \in T : (f^{-1}(\Pi_T)[\pi \mapsto \tau]), 0) \not\models \varphi$. So, $f^{-1}(\Pi_T), 0) \not\models \forall\pi\ \varphi$. Hence the $\Leftarrow$- direction holds.

We consider now the induction case $\exists\pi\ \varphi$. We assume that *(i)* $T \approx_{(|\mathcal{V}(\exists\pi\ \varphi)|, \mathbb{C})} U$. Note that, wlog we can assume that quantifiers bind a variable already occurring in φ, i.e. $|\mathcal{V}(\exists\pi\ \varphi)| = |\mathcal{V}(\varphi)|$. So, *(i')* $T \approx_{(|\mathcal{V}(\varphi)|, \mathbb{C})} U$, and it has the same witnesses as assumption *(i)*. Let $f : T \to U$ be a function that witnesses *(i)*. Now, consider arbitrary Π_T and Π_U, over T and U, s.t. $\mathcal{V}(\Pi_T) = \mathcal{V}(\Pi_U) = \text{free}(\exists\pi\ \varphi) = \text{free}(\varphi) \setminus \{\pi\}$. We prove next that $(\Pi_T, 0) \models \exists\pi\ \varphi$ iff $(f(\Pi_T), 0) \models \exists\pi\ \varphi$.

We start with the $\Rightarrow$-direction of the statement. Assume that $(\Pi_T, 0) \models \exists\pi\ \varphi$, then by HyperLTL satisfaction: there exists $\tau \in T : (\Pi_T[\pi \mapsto \tau], 0) \models \varphi$.

By Definition 6.1.1, $\mathcal{V}(f(\Pi_T)) = \mathcal{V}(\Pi_T)$, and thus $\mathcal{V}(f(\Pi_T)[\pi \mapsto \tau']) = \mathcal{V}(\Pi_T) \cup \{\pi\} = \text{free}(\varphi)$. Then, we can apply the (IH), because $T \approx_{(|\mathcal{V}(\varphi)|, \mathbb{C})} U$ with f being one of its witnesses, and for all $\tau \in T$ the $\Pi_T[\pi \mapsto \tau]$ is an assignment over T. So, we get:

$$\text{there exists } \tau \in T : (f(\Pi_T[\pi \mapsto \tau]), 0) \models \varphi.$$

By Definition 6.1.1, there exists $\tau \in T, \tau' = f(\tau)$ and $(f(\Pi_T)[\pi \mapsto \tau'], 0) \models \varphi$. Then, by f being a total function: there exists $\tau' \in U : (f(\Pi_T)[\pi \mapsto \tau'], 0) \models \varphi$. Hence by HyperLTL satisfaction definition: $(f(\Pi_T), 0) \models \exists\pi\ \varphi$.

We now prove the $\Leftarrow$-direction of the statement.

Assume that $(f(\Pi_T), 0) \models \exists \pi\ \varphi$, then by HyperLTL satisfaction:

$$\text{there exists } \tau \in T : (f(\Pi_T[\pi \mapsto \tau]), 0) \models \varphi.$$

By Definition 6.1.1, $\mathcal{V}(\Pi_T) = \mathcal{V}(f(\Pi_T))$, and thus $\mathcal{V}(\Pi_T[\pi \mapsto \tau']) = \mathcal{V}(f(\Pi_T)) \cup \{\pi\} = \text{free}(\varphi)$. Then, we can apply the (IH), because $T \approx_{(|\mathcal{V}(\varphi)|, \mathbb{C})} U$ with f being one of its witnesses, and for all $\tau \in T$ the $f(\Pi_T[\pi \mapsto \tau])$ is an assignment over U. So, there exists $\tau \in T : (\Pi_T[\pi \mapsto \tau], 0) \models \varphi$. And by HyperLTL satisfaction definition, $(\Pi_T, 0) \models \exists \pi\ \varphi$.

Finally, we prove: $(\Pi_U, 0) \models \exists \pi\ \varphi$ iff $(f^{-1}(\Pi_U), 0) \models \exists \pi\ \varphi$.

We start with the $\Rightarrow$-direction of the statement. It is analogous to the previous case, up to the point that we infer: there exists $\tau' \in U : \tau = f^{-1}(\tau')$ and $(f^{-1}(\Pi_U)[\pi \mapsto \tau], 0) \models \varphi$. Then, by f being surjective, there exists $\tau \in T : (f^{-1}(\Pi_U)[\pi \mapsto \tau], 0) \models \varphi$. Hence by HyperLTL satisfaction definition: $(f^{-1}(\Pi_U), 0) \models \exists \pi\ \varphi$.

The $\Leftarrow$-direction is analogous to the previous case. □

Side note: Incompleteness of the lift

The other direction of the implication in Theorem 6.1.4 does not hold. We support this claim with an example next. Consider the following trace sets with valuations over a single variable x:

$$T = \{[1][0][1][0]^\omega\}$$
$$U = \{[1][0][0][1][0]^\omega, [1][0][0][0][1][0]^\omega\}$$

We first observe that as the two trace sets have different number of elements, there is no natural number k and class of LTL formulas $\mathbb{C}$ for which they are $(k, \mathbb{C})$-equivalent. According to Definition 6.1.1, the function witnessing the equivalence must be bijective and total. Thus, the sets must have the same number of elements. However, they are indeed indistinguishable for all HyperLTL formulas with only one trace quantifier and no next operator because the two traces in U are stutter-equivalent to the trace in T.

6.1.2 Time-prefixed

In this section, we look into the fragment of Hypertrace Logic with time constraints defined before trace quantifiers, denoted by $<$-FO$[<, \mathbb{T}]$. The formulas $\varphi \in <$-FO$[<, \mathbb{T}]$ are defined by the following grammar:

$$\psi ::= \forall \pi\ \psi \mid i < i \mid i = i \mid \psi \vee \psi \mid \neg\psi \mid P(\pi, i)$$
$$\varphi ::= \forall i\, \varphi \mid \exists i\, \varphi \mid \psi$$

where π is a trace variable, i is a time variable, and P is a binary predicate.

Example: Time-prefixed hypertrace formulas

We can use time-prefixed hypertrace logic to specify *synchronized behavior across all traces*, i.e., require that all executions of a system agree on a variable (or multiple variables) assignment at the same time. For example, if we want all traces to agree on p at the same time, we specify it with the formula: $\exists i \, \forall \pi \; p(\pi, i)$.

Bozelli et al. in [BMP15], proved that HyperLTL cannot express a similar property.

Indistinguishable Time Points

Intuitively, time-prefixed hypertrace formulas cannot distinguish trace sets with the same projection at each time point because time-prefixed formulas cannot fix a trace assignment before quantifying over time. Formally, for a time-prefixed formula that quantifies over k time points, two sets of traces are k-point equivalent if there exists a bijection relating all k-tuple of time points in the two sets.

Definition 6.1.2. *Let T and U be sets of traces. We say that T and U are* k-point equivalent*, denoted $T \approx_k^{point} U$, iff for all k-tuples $(i_1, \cdots i_k) \in \mathbb{N}^k$ of time positions, there exists a bijective and total function $f : T \to U$ such that for all traces $\tau \in T$ and $\tau' \in U$, and all $1 \leq j \leq k$, we have $\tau[i_j] = f(\tau)[i_j]$ and $\tau'[i_j] = f^{-1}(\tau)[i_j]$.*

We prove below that this definition captures trace sets indistinguishability for time-prefixed hypertrace formulas.

Theorem 6.1.6. *For all time-prefixed hypertrace sentences $\varphi \in <\text{-}FO[<, \mathbb{T}]$, and all sets T and U of traces with $T \approx_k^{point} U$, where k is the number of time variables in φ, we have $T \models \varphi$ iff $U \models \varphi$.*

Proof. We evaluate time-prefixed formulas as in FOL with sorts. We denote by $(\Pi_{\mathbb{N}}, \Pi_T)$ a pair with the assignments for variables over the sort time and the sort of traces, respectively. Wlog, we can assume that the variables can be identified by the position they are quantified. Then, given a k-point equivalence and an assignment over time, $\Pi_{\mathbb{N}}$ that has assignments for the variables $\mathcal{I}(\Pi_{\mathbb{N}}) = \{\pi_1, \ldots \pi_n\}$ with $n \leq k$, then there exists a witness function for the tuple $(\pi_1, \ldots, \pi_n)$ which we denote by $f_{\Pi_{\mathbb{N}}}$. We prove the theorem by proving the following lemma first:

For all time-prefixed Hypertrace formulas $\varphi \in <\text{-}FO[<, \mathbb{T}]$ and all sets of traces, T and U, that are $|\mathcal{I}(\varphi)|$-point equivalent, $T \approx_{|\mathcal{I}(\varphi)|}^{point} U$, where $|\mathcal{I}(\varphi)|$ is the number of time variables in φ, then for all time assignments $\Pi_{\mathbb{N}}$, and for all trace assignments Π_T and Π_U, which are over free(φ), we have: $(\Pi_{\mathbb{N}}, \Pi_T) \models \varphi$ iff $(\Pi_{\mathbb{N}}, f_{\Pi_{\mathbb{N}}}(\Pi_T)) \models \varphi$; and $(\Pi_{\mathbb{N}}, \Pi_U) \models \varphi$ iff $(\Pi_{\mathbb{N}}, f_{\Pi_{\mathbb{N}}}^{-1}(\Pi_T)) \models \varphi$;

We prove this lemma by structural induction on time-prefixed formulas. The induction step for the time prefix part is trivial, as the assignment over time variables in both sides of the implication is the same. For the trace quantifier part, the proof in analogous to Lemma 6.1.5. The only difference is the base case, that follows from the definition of k-point equivalence. □

6.2 Flavors of Two-state Independence

In this section, we present different specification variants of the following sequential information-flow policy:

The value of observable variable y is independent of the value of observable variable x
until state *changes, and from then on*
the value of observable variable z is independent of the value of x.

Stateless. We start by defining variants for independence without state change with hypertrace logic. Independence between two variables x and y requires that for any two traces τ and τ' there exists a third trace $\tau_\exists$ that has the same values for x as the first trace τ and the same values for y as the second trace τ'. The variants we will discuss differ in how we define the notion of *'having the same values'*.

In our first variant, we require the comparison to consider the complete sequence of values altogether. It adopts a trace-based view for the comparison, which we call the *segment semantics* of independence. An alternative semantics is of time-invariant combinational independence. Under this interpretation, we compare values at each time independently. This variant adopts a time-point view, which we refer to as the *point semantics* of independence.

We specify point and segment independence with hypertrace formulas below. We use the predicate *def* to accommodate for the bounds of finite traces.

Definition 6.2.1. *Two variables x and y are* point independent*, $ind_{point}(x, y)$, iff*

$$\forall i \forall \pi \forall \pi' \exists \pi_\exists \ \big(def(\pi, i) \wedge def(\pi', i)\big) \rightarrow \\ \big(def(\pi_\exists, i) \wedge (x(\pi, i) \leftrightarrow x(\pi_\exists, i)) \wedge (y(\pi', i) \leftrightarrow y(\pi_\exists, i))\big).$$

Two variables x and y are segment independent*, $ind_{seg}(x, y)$, iff:*

$$\forall \pi \forall \pi' \exists \pi_\exists \forall i \ \big(def(\pi, i) \wedge def(\pi', i)\big) \rightarrow \\ \big(def(\pi_\exists, i) \wedge (x(\pi, i) \leftrightarrow x(\pi_\exists, i)) \wedge (y(\pi', i) \leftrightarrow y(\pi_\exists, i))\big).$$

Stateful. We look now at the state change. First, we distinguish between the state change being either *observable* or *hidden* in the set of traces abstracting the system's behavior. Besides the visibility of the state change, we distinguish between systems

in which the state change occurs at the same time in all of its traces (i.e., the state change is *synchronous* across all traces) and systems without any guarantees on the state changes (i.e., the change is *asynchronous* across all traces). In the specification below, the propositional variable a indicates the state change (it becomes true after the change).

We start by defining a slicing operator over sets of traces with two variants: the prefix and the suffix slicer. We use these slicers to get prefixes (or suffixes) of all traces in a given set of traces, characterizing all that happens before (or after) a given propositional variable becomes true for the first time. We formalize the statement that a *'propositional variable a becomes true for the first time at time i in a trace τ'* with the following formula:

$$\min(\tau, a, i) \stackrel{\text{def}}{=} a(\tau, i) \wedge \forall j \; a(\tau, j) \rightarrow j \geq i.$$

The *slicing* of a set of traces T for a propositional variable a, is defined as follows:

$$T[a \ldots] = \{\tau[k \ldots] \mid \tau \in T, \; k < |\tau|, \text{ and } \min(\tau, a, k)\};$$
$$T[\ldots a] = \{\tau[\ldots k] \mid \tau \in T, \text{ if exists } l \text{ s.t. } \min(\tau, a, l) \text{ then } k = l \text{ else } k = \omega\}.$$

We now have all the definitions we need to define the different variants of sequential independence we introduced so far.

Definition 6.2.2. Two-state local independence *is defined with regard to a propositional variable a and to an independence interpretation $ind \in \{ind_{point}, ind_{seg}\}$.*

Observable asynchronous state change:

$$\boldsymbol{T}_{ind}^{async} = \{T \mid T[\ldots a] \models ind(x, y) \; \textit{and} \; T[a \ldots] \models ind(x, z)\}.$$

Observable synchronous state change:

$$\boldsymbol{T}_{ind}^{sync} = \{T \mid T \in \boldsymbol{T}_{ind}^{async} \; \textit{and} \; T \models \exists i \forall \pi \; \min(\pi, a, i)\}.$$

Hidden asynchronous state change:

$$\boldsymbol{T}_{ind}^{async,hidden} = \{T|_a \mid \exists a \; T[\ldots a] \models ind(x, y) \; \textit{and} \; T[a \ldots] \models ind(x, z)\}.$$

Hidden synchronous state change:

$$\boldsymbol{T}_{ind}^{sync,hidden} = \{T \mid \exists k \; T[\ldots k] \models ind(x, y) \; \textit{and} \; T[k \ldots] \models ind(x, z)\}.$$

Here $T|_a$ is the same set of traces as T except for the assignments of a being removed.

Example: Two-state independence

We use the program $\mathcal{P}$, described in Algorithm 4.1, to illustrate the differences between the two-state independence definitions. Recall that, we abstract the

behavior of the program $\mathcal{P}$ by the updates of four variables: *state*, x, y and z. In each program's loop iteration, *state*'s value determines the values of y and z, while the variable x is non-deterministically assigned via an input channel.

The program $\mathcal{P}$ starts with the state variable set to 0 ($state = 0$) and, at each iteration, the *state* value is non-deterministically assigned via an input channel. When the value of *state* changes from 0 to 1, the program stops updating *state*, which remains as 1 for the subsequent iterations. If the value of *state* is 0, then the program $\mathcal{P}$ assigns x to z while it sets y with a default value. Otherwise, when the value of *state* is 1, the inverse occurs: the program $\mathcal{P}$ assigns x to y while it sets z with a default value. At the end of each loop iteration, the program outputs y and z.

We first observe that the program does not satisfy the two-state independence property for observable synchronous state change because the state change is non-deterministically determined in each execution. As for the observable asynchronous state change variant, the program intuitively satisfies it for both the segment and point-based interpretation.

Recall the four traces depicted in Table 4.1, defined over the sequence of variables $[x\ y\ z\ state]$. We depict below the result of applying the prefix and the suffix slicers for the action *state* over the set with the four traces:

	$\{\tau_1,\tau_2,\tau_3,\tau_4\}[\ldots state]$	$\{\tau_1,\tau_2,\tau_3,\tau_4\}[state\ldots]$
τ_1 :	[000**0**]	[110**1**] [110**1**] [110**1**]
τ_2 :	[101**0**]	[110**1**] [110**1**] [110**1**]
τ_3 :	[101**0**] [101**0**]	[000**1**] [000**1**]
τ_4 :	[000**0**] [101**0**] [000**0**]	[110**1**]

In these traces, we can observe that $ind_{seg}(x, y)$ holds in the first slice, while $ind_{seg}(x, z)$ holds in the second slice.

6.3 Expressing Two-State Independence

After introducing different natural variants of the two-state independence property using hypertrace formulas combined with a trace set slicing operator, the next natural step is to investigate which variants can be expressed only using hypertrace logic formulas. In particular, in this section, we focus on the most successful fragment of hypertrace logic: the trace-prefix, known as HyperLTL.

In our negative expressiveness results for HyperLTL, we use the fact that HyperLTL formulas build from LTL formulas belonging to a class $\mathbb{C}$ with up to k trace quantifiers cannot distinguish trace sets that are $(k, \mathbb{C})$-equivalent (see Definition 6.1.1), as proved in Theorem 6.1.4. The proof process follows the following steps:

1. we identify the class of LTL formulas (i.e., the class of formulas in the quantifier-free part of an HyperLTL formula) we are interested in;
2. we define two families of trace sets parameterized by a natural number k;
3. we prove that, for one of the families, all its trace sets satisfy the property we are interested in, while for the other family, all its trace sets do not satisfy it;
4. we prove that for each k, the trace sets from each family are $(k, \mathbb{C})$-equivalent.

Finally, once we have all those results, it follows from Theorem 6.1.4 that HyperLTL (or the fragment of it we are working with) cannot express the property of interest.

Can HyperLTL express the variants of two-state independence?

In the table below, we summarize the findings presented in this section, where we explore which variants of two-state independence can be expressed using HyperLTL.

Independence semantics	**State change**			
	Sync	*Async*	*Hidden async*	*Hidden sync*
Point	No? [Thm 6.3.4]	No [Thm 6.3.11]	No [Thm 6.3.11]	No? [Thm 6.3.4]
Segment	Yes [Thm 6.3.6]	No [Thm 6.3.11]	No [Thm 6.3.11]	?

Table 6.1: Summary of results on expressing variants of two-state independence with HyperLTL.

In a nutshell, for point semantics and synchronous state change, we prove in Theorem 6.3.4 that HyperLTL formulas with a single globally (**G**) operator cannot express both the visible and the hidden variants of two-state independence. We conjecture this result extends to the full HyperLTL. In Theorem 6.3.11, we prove that HyperLTL cannot express asynchronous state change. This is unsurprising because, in HyperLTL formulas, time evaluation is done synchronously over the traces assigned to the trace variables. Finally, we show that HyperLTL can express the segment semantics variant with visible synchronous state change, while for the hidden and synchronous state change the problem remains open.

6.3.1 Two-state Point Semantics

We start by considering the point semantics of independence, i.e., independence predicates of the form $ind_{point}(x, y)$ as in Definition 6.2.1. We observe that $ind_{point}(x, y)$ is a time-prefixed hypertrace formula, and the only time quantifier is a universal quantifier.

A natural HyperLTL fragment to capture point semantics is the fragment where time is also only universally quantified. So, we are interested in the HyperLTL fragment where the inner LTL formula belongs to the class LTL formulas that start with the globally

($\mathbf{G}$) operator and contain no other modal operators, which we refer to as *Global LTL* and denote by $\mathbb{G}$. Formally, $\mathbb{G} = \{\mathbf{G}\,\psi \mid \psi \text{ is a propositional formula}\}$. *Global HyperLTL* extends global LTL with trace quantifiers.

We prove next that Global HyperLTL cannot express *one-state independence with point semantics*. We denote the hyperproperty specifying one-state independence with point semantics as $\mathbf{T}^1_{point}$, i.e., $\mathbf{T}^1_{point} = [\![ind_{point}(x,y)]\!]$.

As we are interested in Global LTL ($\mathbb{G}$), we need to first define an relation between all pair of traces that cannot be distinguished by formulas in the class $\mathbb{G}$. We say that traces τ and τ' are $\approx_{\mathbb{G}}$-*equivalent* if for all formulas $\varphi \in \mathbb{G}$: $\tau \models \varphi$ iff $\tau' \models \varphi$.

Proposition 6.3.1. *For all traces τ and τ', we have $\tau \approx_{\mathbb{G}} \tau'$ iff $\{\tau[i] \mid i \in \mathbb{N}\} = \{\tau'[j] \mid j \in \mathbb{N}\}$.*

Proof. Consider arbitrary traces τ and τ'. We start with the $\Rightarrow$-direction. Assume that $\tau \approx_{\mathbb{G}} \tau'$. Consider an arbitrary $\varphi \in \mathbb{G}$, so $\varphi = \mathbf{G}\,\psi$ where ψ is a propositional formula. Then, $\tau \not\models \varphi$ iff there exists $i \in \mathbb{N}$ s.t. $\tau[i] \not\models \psi$. And this is equivalent to, there exists $v \in \{\tau[i] \mid i \in \mathbb{N}\}$ s.t. $v \not\models \psi$. By $\tau \approx_{\mathbb{G}} \tau'$, $v \in \{\tau[i] \mid i \in \mathbb{N}\}$ iff $v \in \{\tau'[j] \mid j \in \mathbb{N}\}$. Then from an analogous reasoning, the former is equivalent to $\tau' \not\models \varphi$.

For the $\Leftarrow$-direction, we assume that for all $\varphi \in \mathbb{G}$ we have $(\star)$ $\tau \models \varphi$ iff $\tau' \models \varphi$. Now, assume towards a contradiction that $\tau \not\approx_{\mathbb{G}} \tau'$. Consider first the case that there exists $i \in \mathbb{N}$ s.t. $\tau[i] \notin \{\tau'[j] \mid j \in \mathbb{N}\}$. This contradicts $(\star)$, because it entails that there exists $\varphi \in \mathbb{G}$ with $\varphi = \mathbf{G}\,\psi$ s.t. $\tau[i] \models \psi$ and for all $j \in \mathbb{N}$ $\tau[j] \not\models \psi$. The case that there exists $j \in \mathbb{N}$ s.t. $\tau'[j] \notin \{\tau'[i] \mid i \in \mathbb{N}\}$ is analogous. □

Our next step to prove that Global HyperLTL cannot express $\mathbf{T}^1_{point}$ is to define two families of trace sets that are equivalent for Global HyperLTL formulas with one of the families being in $\mathbf{T}^1_{point}$ while the other is not.

Example: Trace Sets Witnessing Inexpressibility of $\mathbf{T}^1_{point}$ with One Trace Quantifier Global HyperLTL

Consider the set of traces:

$$T_1^{point} = \{[11]\ [11]\ [11]\ [00]^\omega,\ [10]\ [00]\ [00]\ [00]^\omega,\ [01]\ [00]\ [00]\ [00]^\omega,\ [00]\ [10]\ [10]\ [00]^\omega,\ [00]\ [01]\ [01]\ [00]^\omega\}$$

$$U_1^{point} = \{[11]\ [11]\ [11]\ [00]^\omega,\ [10]\ [00]\ [00]\ [00]^\omega,\ [01]\ [00]\ [00]\ [00]^\omega,\ [00]\ [10]\ [00]\ [00]^\omega,\ [00]\ [01]\ [00]\ [00]^\omega\}$$

We first observe that T_1^{point} satisfies the condition that x is independent of y: at

all times, we have all possible combinations of observations for x and y. While the set T_1^{point} does not fulfil the independence condition because at time 2 we are missing traces with valuations [10] and [01] for [x y].

When there is only one trace quantifier in a Global HyperLTL formula, then two trace sets are $(1, \mathbb{G})$-equivalent iff there is a bijective mapping between their traces s.t. the traces in the relation are $\mathbb{G}$-equivalent. The mapping here is evident: each trace in T_1^{point} is mapped to the trace to their right in U_1^{point}. We observe that for both trace sets, we get the following set of valuations from their traces (which we present below in the same order as the traces above):

$$\{[11], [00]\}$$
$$\{[10], [00]\}$$
$$\{[01], [00]\}$$
$$\{[00], [10]\}$$
$$\{[00], [01]\}$$

Then, $T_1^{point} \approx_{(1,\mathbb{G})} U_1^{point}$.

Given enough trace quantifiers, we can distinguish the two trace sets with a Global HyperLTL formula. One example of such a formula is:

$$\forall \pi\, \exists \pi_0\, \exists \pi_1\, \exists \pi_2\ \mathbf{G}\, ((x_\pi \wedge y_\pi) \rightarrow (y_{\pi_0} \vee y_{\pi_1} \vee y_{\pi_2})).$$

We generalize the construction presented in the example to accommodate any number of quantifiers. Our goal is to ensure that there are enough traces in the sets such that no n-tuple of them can be distinguished by a Global HyperLTL formula with up to n quantifiers.

Definition 6.3.1. *For all $n \in \mathbb{N}$, the sets T_n^{point} and U_n^{point} of traces with valuations over [x y] are defined as:*

$$E_n = \{[11]^{n+2}[00]^\omega\} \cup \bigcup_{0 \leq j < n} \{[00]^j\ [10]\ [00]^\omega, [00]^j\ [01]\ [00]^\omega\};$$
$$T_n^{point} = E_n \cup \{[00]^n\ [10]\ [10]\ [00]^\omega, [00]^n\ [01]\ [01]\ [00]^\omega\};$$
$$U_n^{point} = E_n \cup \{[00]^n\ [10]\ [00]\ [00]^\omega, [00]^n\ [01]\ [00]\ [00]^\omega\}.$$

As expected, the family of sets T_n^{point} satisfies one-state point-wise independence, while U_n^{point} does not.

Lemma 6.3.2. *For all $n \in \mathbb{N}$, we have $T_n^{point} \in \boldsymbol{T}^1_{point}$ and $U_n^{point} \notin \boldsymbol{T}^1_{point}$.*

Proof. $T \in \mathbf{T}^1_{point}$ iff for all $i \in \mathbb{N}$:

$$T[i] \in \{M \mid M \models (x(\pi, i) \leftrightarrow x(\pi_\exists, i)) \wedge (y(\pi', i) \leftrightarrow y(\pi_\exists, i)).\}$$

Consider arbitrary $n \in \mathbb{N}$. We represent the valuations in (x, y).

For all $n+1 < i$, $T^{point}_n[i] = T'^{point}_n[i] = \{00\}$.

For all $i < n+1$, $T^{point}_n[i] = T'^{point}_n[i] = \{00, 01, 10, 11\}$.

As $T^{point}_n[n+1] = \{00, 01, 10, 11\}$, then $T^{point}_n \in \mathbf{T}^1_{point}$. And, as $T'^{point}_n[n+1] = \{10, 00\}$, then $T^{point}_n \notin \mathbf{T}^1_{point}$. □

We prove next that T^{point}_n and U^{point}_n are $(n, \mathbb{G})$-equivalent, for all natural numbers n.

Lemma 6.3.3. *For all $n \in \mathbb{N}$, we have $T^{point}_n \approx_{(n,\mathbb{G})} U^{point}_n$.*

Proof. Consider arbitrary $n \in \mathbb{N}$. We define the witness function $f_n : T^{point}_n \to T'^{point}_n$ below:

$$f_n(\tau) = \begin{cases} (00)^n\ 10\ 00^\omega & \text{if } \tau = (00)^n\ 10\ 10\ 00^\omega \\ (00)^n\ 01\ 00^\omega & \text{if } \tau = (00)^n\ 01\ 01\ 00^\omega \\ \tau & \text{otherwise.} \end{cases}$$

Clearly, this function is both bijective and total.

By definition of f_n and T^{point}_n, then for all assignments of size n over it, $\Pi_{T^{point}_n}$:

(a) for all $i \neq n+1$, $\langle \Pi_{T^{point}_n} \rangle[i] = \langle f_n(\Pi_{T^{point}_n}) \rangle[i]$; and

(b) for all $\pi \in \mathcal{V}$, if $\Pi_{T^{point}_n}(\pi) \notin \{(00)^n\ 10\ 10\ 00^\omega, (00)^n\ 01\ 01\ 00^\omega\}$, then $\Pi_{T^{point}_n}(\pi) = f_n(\Pi_{T^{point}_n}(\pi))$.

Analogously for all assignments over T'^{point}_n of size n and f^{-1}.

It follows from the definition of T^{point}_n, that for all assignments of size $m < n$ over T^{point}_n, $\Pi^m_{T^{point}_n}$, there exists $0 \leq k < n$ s.t. for all $\pi \in \mathcal{V}$, $\Pi^m_{T^{point}_n}(\pi) \notin \{(00)^k\ 10\ 00^\omega, (00)^k\ 01\ 00^\omega\}$. Then, (†) $\Pi^m_{T^{point}_n}[k] \in \{\{11\}, \{00\}, \{11, 00\}\}$, because the only way to get valuations 10 and 10 at time k is with the missing traces.

Consider arbitrary $n \in \mathbb{N}$ and assignment over T^{point}_n of size n, $\Pi_{T^{point}_n}$. If for all $\pi \in \mathcal{V}$, $\Pi_{T^{point}_n}(\pi) \notin \{(00)^n\ 10\ 10\ 00^\omega, (00)^n\ 01\ 01\ 00^\omega\}$, then by (b), for all $i \in \mathbb{N}$, $\langle f_n(\Pi_{T^{point}_n}) \rangle[i] = \langle \Pi_{T^{point}_n} \rangle[i]$. Now we assume that there exists $Y = \{\pi_0, \ldots, \pi_l\}$, with $0 \leq l < n$ s.t. $\Pi_{T^{point}_n}(\pi) \in \{(00)^n\ 10\ 10\ 00^\omega, (00)^n\ 01\ 01\ 00^\omega\}$, with $\pi \in Y$, and for all $\pi \notin Y$, $\Pi_{T^{point}_n}(\pi) \notin \{(00)^n\ 10\ 10\ 00^\omega, (00)^n\ 01\ 01\ 00^\omega\}$. We can prove from (†) that there exists k s.t. $\langle \Pi_{T^{point}_n|_Y} \rangle[n+1] = \langle \Pi_{T^{point}_n|_Y} \rangle[k]$, where $\Pi_{T^{point}_n|_Y}$ is $\Pi_{T^{point}_n}$ without the assignments to the variables in Y. Moreover, it follows as well, that for all $\pi \in Y$, $\langle \Pi_{T^{point}_n} \rangle[k] = 00$, and so there exists k s.t. $\langle f_n(\Pi_{T^{point}_n}) \rangle[n+1] = \langle \Pi_{T^{point}_n} \rangle[k]$. Thus, ($\star$)

for all assignments over T_n^{point} of size n, $\Pi_{T_n^{point}}$ and all $i \in \mathbb{N}$ there exists $j \in \mathbb{N}$ s.t. $\langle f_n(\Pi_{T_n^{point}})\rangle[i] = \langle \Pi_{T_n^{point}}\rangle[j]$.

By $\langle \Pi_{T_n^{point}}\rangle[n+1] = \langle f_n(\Pi_{T_n^{point}})\rangle[n]$ and (a), then $(\star\star)$ for all assignments of size n over T_n^{point}, $\Pi_{T_n^{point}}$ and for all $i \in \mathbb{N}$ there exists $j \in \mathbb{N}$ s.t. $\langle \Pi_{T_n^{point}}\rangle[i] = \langle f_n(\Pi_{T_n^{point}})\rangle[j]$.

By $(\star)$ and $(\star\star)$, for all assignments of size n over T_n^{point}, $\Pi_{T_n^{point}}$, we have $\langle \Pi_{T_n^{point}}\rangle[i] \approx_{\mathbb{G}} \langle f_n(\Pi_{T_n^{point}})\rangle[j]$.

We prove analogously that for all assignments of size n over T'^{point}_n, $\Pi_{T'^{point}_n}$, we have $\langle \Pi_{T'^{point}_n}\rangle[i] \approx_{\mathbb{G}} \langle f_n^{-1}(\Pi_{T'^{point}_n})\rangle[j]$. □

Finally, we combine all the results above to conclude that Global HyperLTL cannot express the point semantics variant of two-state independence.

Theorem 6.3.4. *Global HyperLTL cannot express neither one-state independence, nor synchronous two-state local independence under point semantics for both observable and hidden action: for all global HyperLTL formulas φ, we have $[\![\varphi]\!] \neq \boldsymbol{T}^{1}_{point}$, $[\![\varphi]\!] \neq \boldsymbol{T}^{sync}_{point}$ and $[\![\varphi]\!] \neq \boldsymbol{T}^{sync,hidden}_{point}$.*

Proof. From Lemma 6.3.2, Lemma 6.3.3, and Theorem 6.1.4, it follows that for all global HyperLTL formulas φ, we have $[\![\varphi]\!] \neq \mathbf{T}^{1}_{point}$. Assume towards a contradiction that there exists a global HyperLTL formula φ with $[\![\varphi]\!] = \mathbf{T}^{sync}_{point}$. Define $\varphi_y = \varphi[z \mapsto y]$, where $[z \mapsto y]$ replaces all occurrence of z by y. Then $[\![\varphi_y]\!] = \mathbf{T}^{1}_{point}$, which is a contradiction. Analogously, the assumption that there exists a global HyperLTL formula φ that specifies hidden synchronous change $[\![\varphi]\!] = \mathbf{T}^{sync,hidden}_{point}$ lead us to a contradiction. □

We conjecture that this result extends to full HyperLTL due to the requirement of reasoning over unbound possibilities over time, which, for HyperLTL, must be done by a bounded trace assignment. It is not surprising, however, that we can express two-state independence under point semantics with a synchronous state change with a time-prefixed hypertrace formula, which we prove below.

Theorem 6.3.5. *Consider the following time-prefixed hypertrace formula:*

$$\begin{aligned}\varphi^{sync}_{time} \equiv \exists j \forall i < j \forall k \leq j \forall \pi \forall \pi' \exists \pi_\exists \\ (\neg a(\pi,i) \wedge \neg a(\pi',i) \wedge (x(\pi,i) \leftrightarrow x(\pi_\exists,i)) \wedge (y(\pi',i) \leftrightarrow y(\pi_\exists,i))) \wedge \\ (a(\pi,j) \wedge a(\pi',j) \wedge (x(\pi,k) \leftrightarrow x(\pi_\exists,k)) \wedge (z(\pi',k) \leftrightarrow z(\pi_\exists,k)))\end{aligned}$$

Then $[\![\varphi^{sync}_{time}]\!] = \boldsymbol{T}^{sync}_{point}$.

Proof. Note that $T \models \varphi^{sync}_{\text{time}}$ iff $T \models \exists j \forall i < j \forall \pi\, \neg a(\pi,i) \wedge a(\pi,j)$. Thus, it only includes sets of traces with synchronous action.

Additionally, for all T that have synchronous action:

$$
\begin{aligned}
T \models \varphi^{sync}_{\text{time}} \text{ iff } T \models\ & \exists j \forall i < j \forall \pi \forall \pi' \exists \pi_\exists \\
& (\neg a(\pi, i) \wedge \neg a(\pi', i) \wedge (x(\pi, i) \leftrightarrow x(\pi_\exists, i)) \wedge (y(\pi', i) \leftrightarrow y(\pi_\exists, i))) \wedge a(\pi, j) \\
& \text{and} \\
& T \models \exists j \forall k \leq j \forall \pi \forall \pi' \exists \pi_\exists \\
& \quad a(\pi, j) \wedge a(\pi', j) \wedge (x(\pi, k) \leftrightarrow x(\pi'_\exists, k)) \wedge (z(\pi', k) \leftrightarrow z(\pi'_\exists, k))) \\
\text{iff } & T[\ldots a] \models \forall i \forall \pi \forall \pi' \exists \pi_\exists (\mathit{def}(\pi, i) \wedge \mathit{def}(\pi', i)) \rightarrow \\
& \quad (\mathit{def}(\pi_\exists, i) \wedge (x(\pi, i) \leftrightarrow x(\pi_\exists, i)) \wedge (y(\pi', i) \leftrightarrow y(\pi_\exists, i)) \\
& \text{and} \\
& T[a\ldots] \models \forall k \forall \pi \forall \pi' \exists \pi_\exists (\mathit{def}(\pi, k) \wedge \mathit{def}(\pi', k)) \rightarrow \\
& \quad (\mathit{def}(\pi_\exists, k) \wedge (x(\pi, k) \leftrightarrow x(\pi_\exists, k)) \wedge (z(\pi', k) \leftrightarrow z(\pi_\exists, k)). \qquad \square
\end{aligned}
$$

6.3.2 Segment semantics

In this section, we prove that HyperLTL can express two-state local independence under segment semantics (i.e., independence predicates of the type $\mathit{ind}_{seg}(x, y)$) with a synchronous state change; while the asynchronous state change variants (both the observable and the hidden) are not expressible in HyperLTL.

First, as the segment semantics of independence compares complete traces with trace quantifiers occurring before time quantifiers, the HyperLTL formula for independence is straightforward. Second, the synchronous state change can be easily captured by the until operator, as we illustrate in the theorem below.

Theorem 6.3.6. *Consider the following HyperLTL formula:*

$$
\begin{aligned}
\varphi^{sync}_{seg} \equiv \forall \pi \forall \pi' \exists \pi_\exists \exists \pi'_\exists\ & (\neg a_\pi \wedge \neg a_{\pi'} \wedge x_\pi = x_{\pi_\exists} \wedge y_{\pi'} = y_{\pi_\exists}) \\
& \mathbf{U}\ (a_\pi \wedge a_{\pi'} \wedge \mathbf{G}(x_\pi = x_{\pi_\exists} \wedge z_{\pi'} = z_{\pi'_\exists})).
\end{aligned}
$$

Then $[\![\varphi^{sync}_{seg}]\!] = \boldsymbol{T}^{sync}_{seg}$.

Proof. Note that $x(\pi, i) \leftrightarrow x(\pi_\exists, i)$ in Hypertrace Logic corresponds to $x_\pi = x_{\pi_\exists}$ in HyperLTL. By Definition 6.2.2, $T \in \mathbf{T}^{sync}_{seg}$ iff: (i) $T \models \exists i\ \forall \pi\ \min(\tau, a, i)$; (ii) $T[\ldots a] \models \mathit{ind}_{seg}(x, y)$; and (iii) $T[a\ldots] \models \mathit{ind}_{seg}(x, z)$. Then, by HyperLTL satisfaction, for all set of traces T:

$$
\begin{aligned}
& T \models \varphi^{sync}_{seg} \text{ iff} \\
& T \models \forall \pi \forall \pi' (\neg a_\pi \wedge \neg a_{\pi'})\ \mathbf{U}\ (a_\pi \wedge a_{\pi'}), \\
& T \models \forall \pi \forall \pi' \exists \pi_\exists\ (\neg a_\pi \wedge \neg a_{\pi'} \wedge x_\pi = x_{\pi_\exists} \wedge y_{\pi'} = y_{\pi_\exists})\ \mathbf{U}\ (a_\pi \wedge a_{\pi'}), \text{ and} \\
& T \models \forall \pi \forall \pi' \exists \pi'_\exists\ (\neg a_\pi \wedge \neg a_{\pi'})\ \mathbf{U}\ \ (a_\pi \wedge a_{\pi'} \wedge \Box(x_\pi = x_{\pi_\exists} \wedge z_{\pi'} = z_{\pi'_\exists})).
\end{aligned}
$$

We can prove, by satisfaction for HyperLTL and Hypertrace Logic formulas, that:

$$T \models_H \forall\pi\forall\pi'(\neg a_\pi \wedge \neg a_{\pi'})\ \mathbf{U}\ (a_\pi \wedge a_{\pi'}) \text{ iff } T \models \exists i\, \forall\pi\ a(\pi, i) \wedge \forall 0 \leq j < i\ \neg a(\pi, j)$$

Hence $[\![\varphi^{sync}_{seg}]\!] = \mathbf{T}^{sync}_{seg}$. □

For the asynchronous state change, the challenge is in expressing the unpredictable time difference between the state change in two different traces. In fact, in general HyperLTL cannot compare arbitrarily distant time points from different traces. To prove this shortcoming we use the trace equivalence introduced by Kučera and Strejček [KS05] for the class of LTL formulas with up to n nested next ($\mathbf{X}$) operators, denoted $\mathbb{X}^n$.

In [KS05], the authors define that a valuation $\tau[i]$ at time i is *n-redundant* in a trace τ if it is repeated consecutively for at least $n+1$ times, that is, if $\tau[i] = \tau[i+j]$ for all $1 \leq j \leq n$. We then say that traces τ and τ' are *n-stutter equivalent*, denoted $\tau \approx^n \tau'$, if they are equal up to the removal of n-redundant valuations. We can now define the relation $\approx^n$ has the least equivalence over the set of all finite or infinite traces containing $\prec^n$, where $\tau \prec^n \tau'$ iff there is a time i such that the valuation $\tau'[i]$ is n-redundant in τ', and τ is obtained from τ' by removing $\tau'[i]$. The following proposition is a direct consequence of the results in [KS05].

Proposition 6.3.7 ([KS05])**.** *For all formulas $\varphi \in \mathbb{X}^n$ and all traces τ and τ' with $\tau \approx^n \tau'$, we have $\tau \models \varphi$ iff $\tau' \models \varphi$.*

For the families we want to use for the proof, our goal is to parameterize them by an upper bound on the number of next operators ($\mathbf{X}$) in the HyperLTL formula that is insufficient to encode the distance between the state change in the current assigned traces. In a nutshell, in the construction defined below, the traces in the sets from the family that does not satisfy the property are equal to those in the family that does satisfy it, except for the position $2n+1$, which is deleted. The deleted position defines, by construction, a n-stuttering step over all traces in the family that satisfies the property.

Definition 6.3.2. *For each $n \in \mathbb{N}$, we define two sets $T^{async}_n = \{t_1, t_2, t_3, t_4\}$ and $U^{async}_n = \{u_1, u_2, u_3, u_4\}$ of trace sets with valuations over $[a\ x\ y\ z]$:*

$$\begin{aligned}
\tau_0 &= [1110]\,[1000]^{n+4}\,[1001]^{n+4}\,[1111]\,[1001]^{n+4}\,[1000]^{n+4},\\
\tau_1 &= [1111]\,[1001]^{n+4}\,[1000]^{n+4}\,[1110]\,[1000]^{n+4}\,[1001]^{n+4},\\
t_1 &= [0000]\ \tau_1\ [1001]^{\omega},\ t_2 = [0010]\ \tau_1\ [1001]^{n+4}\ [1111]^{\omega},\\
t_3 &= [0000]^{n+4}\ \tau_0\ [1001]^{\omega},\ t_4 = [0010]^{n+4}\ \tau_0\ [1111]^{\omega},\\
u_i &= t_i[0]t_i[1]\ldots t_i[2n+10]t_i[2n+12]\ldots \text{ for } 1 \leq i \leq 4.
\end{aligned}$$

We start by proving that the position that differs in the family of traces introduced above indeed defines a n-redundancy in the family T^{async}_n.

Lemma 6.3.8. *For all $n \in \mathbb{N}$ and all valuations Π over T_n^{async}, the valuation at time $2n+11$ is n-redundant in the trace $\langle \Pi \rangle$.*

Proof. We prove this by induction on the size of trace assignments Π over T_n^{async}.

For the case case $|\Pi| = 1$, wlog, let $\mathcal{V}(\Pi) = \{\pi\}$ for some $\pi \in \mathcal{V}$. If $\Pi(\pi) \in \{t_1, t_2\}$, then at $2n+11$ we have the block $(1001)^{n+4}$. Hence $t_1[2n+11] = t_1[2n+11+j]$ for all $1 \leq j \leq n+1$. If $\Pi(\pi) \in \{t_3, t_4\}$, then at $2n+10$ we have the block $(1000)^{n+4}$. Hence $t_2[2n+11] = t_1[2n+11+j]$ for all $1 \leq j \leq n+1$.

For the inductive case, assume as induction hypothesis (IH) that the statement holds for all assignments of size k.

Consider an arbitrary assignment Π_{k+1} with size $k+1$ Then, there exists an assignment Π_k with size k s.t. $\Pi_{k+1} = \Pi_k[\pi \mapsto \tau]$ and $\Pi_k(\pi)$ is undefined, for some $\pi \in \mathcal{V}$ and $\tau \in T_n^{async}$. By (IH), the valuation at position $2n+11$ in $\langle \Pi_k \rangle$ is n-redundant. As argued in the base case, the letter at position $2n+11$ for all $\tau \in T_n^{async}$ is n-redundant, as well.

As $\Pi_k(\pi)$ is undefined, then $\langle \Pi_{n+1}[\pi \mapsto \tau] \rangle = \langle \Pi_n \rangle \otimes \langle \Pi^{\emptyset}[\pi \mapsto \tau] \rangle$ where $\otimes$ is the composition of traces. Then, by the $2n+1$ letter being n-redundant in both $\langle \Pi_n \rangle$ and τ, it follows that the letter at $2n+11$ in $\langle \Pi_{n+1}[\pi \mapsto \tau] \rangle$ is n-redundant,as well. □

With the result above, it follows directly that both families and $(k, \mathbb{X}^n)$-equivalent, where k specifies an upper-bound on the number of trace quantifiers and n is an upper-bound on the number of nested next operators on HyperLTL formulas.

Lemma 6.3.9. *For all $k, n \in \mathbb{N}$, we have $T_n^{async} \approx_{(k,\mathbb{X}^n)} U_n^{async}$ and $T_n^{async}|_a \approx_{(k,\mathbb{X}^n)} U_n^{async}|_a$.*

Proof. Consider arbitrary $k, n \in \mathbb{N}$. We define the witness function $f : T_n^{async} \to U_n^{async}$ as $f(t_i) = t'_i$ for $1 \leq i \leq 4$. Clearly, the function is both bijective and total. Let Π be an arbitrary valuation over T_n^{async} such that $|\Pi| = k$. We proved in Lemma 6.3.8 that the valuation at time $2n+11$ in trace $\langle \Pi \rangle$ is n-redundant. By the definition of U_n^{async}, the trace $\langle f(\Pi) \rangle$ is the same as $\langle \Pi \rangle$ except that the valuation at time $2n+11$ is deleted. Therefore $\langle \Pi \rangle \approx^n \langle f(\Pi) \rangle$. We prove analogously that for all valuations Π' of size k over U_n^{async}, we have $\langle \Pi' \rangle \approx^n \langle f^{-1}(\Pi') \rangle$. Hence $T_n^{async} \approx_{(k,\mathbb{X}^n)} U_n^{async}$. We use the same witness function to prove that $T_n^{async}|_a \approx_{(k,\mathbb{X}^n)} U_n^{async}|_a$. Note that for all $n \in \mathbb{N}$, since $T_n^{async}|_a$ is the same as T_n^{async} except for the values of a that are removed, Lemma 6.3.8 holds for $T_n^{async}|_a$ as well. □

Finally, we are only missing to prove that, for all natural numbers n, T_n^{async} satisfies two-state independence under segment semantics and with asynchronous state change, while U_n^{async} does not.

Lemma 6.3.10. *For all $n \in \mathbb{N}$, $T_n^{async} \in \boldsymbol{T}_{seg}^{async}$, $U_n^{async} \notin \boldsymbol{T}_{point}^{async}$, and $U_n^{async}|_a \notin \boldsymbol{T}_{point}^{hidden}$.*

Proof. We start by proving $T_n^{async} \in \mathbf{T}_{seg}^{async}$.

First, we prove that $T_n^{async}[\dots a] \models ind_{seg}(x, y)$. By definition of slicing of sets of traces:

$$T_n^{async}[\dots a] = \{0000, 0010, (0000)^{n+4}, (0010)^{n+4}\}.$$

Then, by Definition 6.2.1, $T_n^{async}[\dots a] \models ind_{seg}(x, y)$ holds because we can choose $\pi_\exists = \pi'$.

Now, we prove that $T_n^{async}[a\dots] \models ind_{seg}(x, z)$. By definition of slicing of sets of traces, $T_n^{async}[a\dots] = \{\tau_0(1000)^\omega, \tau_1(1000)^\omega, \tau_0(1110)^\omega, \tau_1\ (1000)^{n+4}\ (1110)^\omega\}$, where τ_0 and τ_1 are as in Definition 6.3.2. Then, as in the previous case, we can choose $\pi_\exists = \pi'$ to show that $T_n^{async}[\dots a] \models ind_{seg}(x, z)$ holds.

We prove now that ${T'}_n^{async} \notin \mathbf{T}_{point}^{async}$.

We show that ${T'}_n^{async}[\dots a] \not\models ind_{point}(x, z)$. By Definition 6.3.2 and definition of slicing:

$$\begin{aligned}
{t'}_1[a\dots] &= t_1[1]t_1[2]\dots t_1[2n+10]t_1[2n+12]\dots \\
&= \tau_1[0]\tau_1[1]\dots\tau_1[2n+9]\tau_1[2n+11]\dots \\
&= {t'}_2[a\dots] \\
{t'}_3[a\dots] &= t_3[n+4]t_3[n+5]\dots t_3[2n+10]t_3[2n+12]\dots \\
&= \tau_0[0]\tau_1[1]\dots\tau_0[n+6]\tau_0[n+8]\dots \\
&= {t'}_4[a\dots]
\end{aligned}$$

Note that, $2n + 10 - (n + 4) = n + 6$.

Note that $(\star)$ $({t'}_1[a\dots])[2n+9] = ({t'}_2[a\dots])[2n+9] = 1000$ and $({t'}_3[a\dots])[2n+9] = ({t'}_4[a\dots])[2n+9] = 1111$. If we chose $i = 2n + 9$, $\pi = {t'}_3$ and $\pi' = {t'}_1$, then there should exist a trace $t_\exists \in {T'}_n^{async}$ s.t. $(t_\exists[a\dots])[2n+9](x) = ({t'}_3[a\dots])[2n+9](x) = 1$ and $(t_\exists[a\dots])[2n+9](z) = ({t'}_1[a\dots])[2n+9](z) = 0$. However, by $(\star)$ we know that there is not such trace in ${T'}_n^{async}$. Hence ${T'}_n^{async} \notin \mathbf{T}_{point}^{async}$.

The set of set of traces ${T'}_n^{async}|_a$ is the set ${T'}_n^{async}$ where all valuations of a are removed. We need to prove that there is no extension of ${T'}_n^{async}|_a$ with (possibly new) valuations in a that makes it an element of $\mathbf{T}_{point}^{hidden}$. We will abstract the extension of ${T'}_n^{async}|_a$ by defining a function $g : {T'}_n^{async}|_a \to \mathbb{N}$ that given a set of traces in ${T'}_n^{async}|_a$ returns the index where a first holds. We then redefine the slicing operator to slice w.r.t. this function, as follows: $T[\dots g] = \{\tau[\dots g(\tau)] \mid \tau \in T\}$.

We refer to the elements of ${T'}_n^{async}|_a$ by the same names as in the definition of ${T'}_n^{async}$. By construction of ${T'}_n^{async}|_a$, the function g needs to guarantee the following conditions for $T[\dots g] \models ind_{point}(x, y)$ to hold: $g(t'_1) \leq 5n + 23$ and $g(t'_3) \leq 5n + 23$, because $t'_1[5n + 23] = 000 = t'_3[5n + 23]$ and $t'_2[4n + 23] = 111 = t'_4[5n + 23]$. So, we are missing valuations 10 and 01 in (x, y), to prove the independence of y w.r.t. x.

If $g(t'_1) = g(t'_2) = g(t'_3) = g(t'_4) = 1$, then $T[g\dots] \not\models ind_{point}(x, z)$, because ${T'}_n^{async}|_a[1] = \{000, 010, 111\}$ and so we are missing the valuation 01 in (x, z).

We proceed by case analysis.

Case $g(t'_3) = g(t'_4) = 1$**:** We show below the first $n+4$ steps of the slice of t'_3 and t'_4:

$$\tau'_3[1 \dots n+4] = (000)^{n+3}\ 110$$
$$\tau'_4[1 \dots n+4] = (010)^{n+3}\ 110$$

To find a compatible slicing of t'_1 and t'_2 we need it to satisfy the following:

- for the first $n+3$ we can only have the valuation 00 in (x, z), as there is no time point where we can get at the same time 10 and 11;
- at the $n+4$ we cannot have 01 as it is not possible with only one trace left cover all the valuations missing (00 and 11).

Then, the time $n+7$ is the only slicing of t'_1 and t'_2 that satisfies this conditions and guarantees that x is independent of z for the first $n+4$ elements of the slicing suffix, as we show below:

$$\tau'_1[n+7 \dots 2n+10] = (000)^{n+3}\ 110$$
$$\tau'_2[n+7 \dots 2n+10] = (000)^{n+3}\ 110$$
$$\tau'_3[1 \dots n+4] = (000)^{n+3}\ 110$$
$$\tau'_4[1 \dots n+4] = (010)^{n+3}\ 110$$

However, if $g(t'_1) = g(t'_2) = n+7$, then $\tau'_1[4n+20] = \tau'_2[4n+20] = 000$ while $\tau'_3[3n+13] = \tau'_4[3n+13] = 111$. So, we are missing valuations 01 and 10 in (x, z). Hence, for $g(t'_3) = g(t'_4) = n+7$, $T[g \dots] \not\models ind_{point}(x, z)$. So, $g(t'_3) = g(t'_4) > 1$.

Case $g(t'_1) = g(t'_2) > 1$**:** As $g(t'_3) = g(t'_4) > 1$, then the prefix of a slicing with $g(t'_1) = g(t'_2) > 1$ does not satisfy $ind_{point}(x, y)$. Note that $t'_1[1] = t'_2[1] = 111$ while $t'_3[1] = 000$ and $t'_4[1] = 010$, so we are missing the valuation 10 in (x, y). Hence $g(t'_1) = g(t'_2) = 1$.

Case $1 < g(t'_3) \leq 5n+23$ **and** $1 < g(t'_4) \leq 5n+23$**:** If $g(t'_3) < n+4$ and $g(t'_4) < n+4$, then we will be missing the assignment 01 in (x, z). If $g(t'_3) = n+4 = g(t'_4)$, then we know from the case with visible action that the property does not hold. If $n+4 < g(t'_3) < 2n+9$ and $n+4 < g(t'_4) < 2n+9$, then we will be missing the assignment 01 in (x, z). If either $g(t'_4) = 2n+9$, then $g(t'_4)[2n+9...(2n+9)+3n+12] = 111$ while $g(t'_1)[1...3n+13] = 000$ and we will be missing the assignment 10 on (x, z). Then, $g(t'_3) \neq 2n+9$ because the suffix of the trace starts with 000, so there will be not enough traces to cover for observation 111. The same reasoning holds for the next 3 positions. The next $2n+9$ positions cover the deleted letter from t'_1 and t'_2, while the deleted letter from t'_3 and t'_4 happens in a earlier part of the trace. So, the position $2n+11$ of t'_1 and t'_2, with assignment 110, will miss the assignment 11 on (x, z). Note that at that point in the slice of t'_3 and t'_4 z is constantly 1. □

It is clear that all trace sets that are models under the segments semantics are models under the point semantics, as well. Therefore $\mathbf{T}^{async}_{seg} \subseteq \mathbf{T}^{async}_{point}$.

Theorem 6.3.11. *For all HyperLTL sentences φ, we have $[\![\varphi]\!] \neq \boldsymbol{T}^{async}_{point}$, $[\![\varphi]\!] \neq \boldsymbol{T}^{async}_{seg}$, $[\![\varphi]\!] \neq \boldsymbol{T}^{hidden}_{point}$, and $[\![\varphi]\!] \neq \boldsymbol{T}^{hidden}_{seg}$.*

Proof. From $\mathbf{T}^{async}_{seg} \subseteq \mathbf{T}^{async}_{point}$ and Lemma 6.3.10, it follows that for all $n \in \mathbb{N}$, we have $T^{async}_n \in \mathbf{T}^{async}_{seg}$ and $U^{async}_n \notin \mathbf{T}^{async}_{seg}$, as well as $T^{async}_n \in \mathbf{T}^{async}_{point}$ and $U^{async}_n \notin \mathbf{T}^{async}_{point}$. Let φ be a closed HyperLTL formula, let n be the nesting depth of its next operators, and let $k \in \mathbb{N}$ be the number of trace quantifiers in φ. It follows from Lemma 6.3.9 and Theorem 6.1.4 that $T^{async}_n \in [\![\varphi]\!]$ iff $U^{async}_n \in [\![\varphi]\!]$. Hence for all HyperLTL sentences φ, we have $[\![\varphi]\!] \neq \mathbf{T}^{async}_{point}$ and $[\![\varphi]\!] \neq \mathbf{T}^{async}_{seg}$. For all $n \in \mathbb{N}$, since $T^{async}_n|_a$ is the same as T^{async}_n except for the values of a that are removed, we have $T^{async}_n|_a \in \mathbf{T}^{hidden}_{seg}$ and $T^{async}_n|_a \in \mathbf{T}^{hidden}_{point}$. Lemma 6.3.10 implies that $U^{async}_n|_a \notin \mathbf{T}^{hidden}_{point}$, and thus $U^{async}_n|_a \notin \mathbf{T}^{hidden}_{seg}$, for all $n \in \mathbb{N}$. As in the previous case, from Lemma 6.3.9 and Theorem 6.1.4, it follows that for all HyperLTL sentences φ, we have $[\![\varphi]\!] \neq \mathbf{T}^{hidden}_{point}$ and $[\![\varphi]\!] \neq \mathbf{T}^{hidden}_{seg}$. □

6.4 Related Work on HyperLTL Expressive Power

In Section 4.2, we introduced several LTL extensions to reason about hyperproperties. The first HyperLTL inexpressibility result is by Bozzelli et al. in [BMP15], where they prove that HyperLTL and ETL have incomparable expressive power. In this result, the authors prove that HyperLTL cannot express the requirement that there exists a global time-point such that a property holds for all traces at that time, which we refer to as the *globally synchronized behavior property*. In [FZ17], the authors use the same property to prove that $FO[<, E]$ is strictly more expressive than HyperLTL. As for LTL interpreted with team semantics [KMVZ18], Krebs et al. prove that HyperLTL and LTL under team semantics with synchronous entailment have incomparable expressive power. As for the hyperlogics before, they use the globally synchronized behavior property to prove their result. In [CFHH19b], the authors present an overview of the relative expressiveness results for linear-time hyperlogics.

We observe that besides the natural result that HyperLTL cannot distinguish systems that generate the same sets of traces [FR14], up to our work, all expressive comparisons rely on the result by Bozelli et al. [BMP15]. Unfortunately, their result is difficult to generalize to properties beyond the globally synchronized behavior property. In the proof by Bozelli et al. [BMP15], they define an equivalence relation for a specific family of models to show that no HyperCTL* can distinguish them. In the work we presented in this section, we showcase a proof strategy that easily generalizes to sets of indistinguishable traces for classes of HyperLTL formulas parameterized by their quantifier-free part.

Part III

Information-flow Verification

CHAPTER 7

Hypernode Automata

This chapter introduces *hypernode automata*, a formalism to specify *asynchronous hyperproperties* that has a decidable model-checking problem. *Asynchronous hyperproperties* are hyperproperties where the portions we need to compare in different traces may be arbitrarily far. Such asynchronicity may arise for various reasons, like differences in scheduling decisions in a concurrent system or how the system is observed or abstracted. Hypernode automata explicitly support synchronicity and asynchronicity between different traces by combining automata and logic.

A hypernode automaton is a finite automaton with nodes labeled with hypernode logic formulas and transitions labeled with actions. *Hypernode logic* is a fully asynchronous logic that allows specifying properties comparing how variables' values evolve in different traces independently of their concrete timing. We use actions in the transitions in a hypernode automaton to re-synchronize traces.

We start the chapter presenting hypernode logic and automata, followed by a section in which we introduce and solve their respective model-checking problems. To solve the model checking problem for hypernode logic, we define stutter-free automata, a formalism that encodes stutter-free and independent value progressing of program variables. We finish this chapter by comparing hypernode automata with related formalisms and presenting our conjecture that they are expressively incomparable.

This chapter is based on a collaboration with Ezio Bartocci, Thomas Henzinger and Dejan Nickovic, which resulted in the conference paper published in CONCUR 2023 proceedings:

> [BHNOdC23] Ezio Bartocci, Thomas A. Henzinger, Dejan Nickovic, and Ana Oliveira da Costa. Hypernode Automata. In Guillermo A. Pérez and Jean-François Raskin, editors, *International Conference on Concurrency Theory (CONCUR)*, volume 279 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 21:1–21:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023.

We extended the work mentioned above by adding more examples, presenting the complete proofs, and expanding on the sections about stutter-free automata and how to translate from Kripke structures to stutter-free automata.

Example: Mutually Exclusive Declassification

In this example, we specify a *mutually exclusive declassification policy* using hypernode automata. The policy we are interested in requires that y and z only expose the secret value of x when it is necessary to debug the system and that they do not do it at the same time.

We depict below, in Figure 7.1, a possible specification of this policy as a automaton with labeled states. In the automaton below, transitions between states are labeled with the actions that signal when the system starts debugging using variable y (action Deb_y) or with variable z (action Deb_z), or when it is not in debug mode (action *Clear*). As for the states, they are labeled with a formula specifying what is expected from the system at that point. In this example, at different states we require the system to be observationally deterministic for a subset of $\{y, z\}$. When we instantiate the observational determinism requirements with hypernode formulas, the automaton in Figure 7.1 can be interpreted as a hypernode automaton.

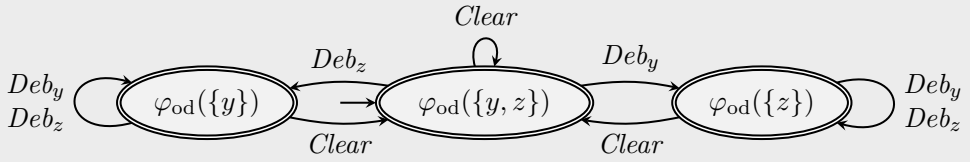


Figure 7.1: Hypernode automaton $\mathcal{H}$ specifying the mutually exclusive declassification of secure information in x by y and z.

When interpreting the declassification policy presented above, we assume that the time of each system execution may not be aligned (i.e., the executions may be asynchronous) and use actions to synchronize the dynamic transitions to different specification states across the system's executions.

We can interpret the observational determinism requirement as introduced by Zdancewic-Myers[ZM03], which can be specified by the following hypernode formula for a given set of (output public) variables L:

$$\varphi_{\text{od}}(L) \stackrel{\text{def}}{=} \forall\pi\forall\pi' \bigwedge_{l\in L} (l(\pi) \precsim l(\pi') \vee l(\pi') \precsim l(\pi)).$$

We observe that we have only output variables; thus, we only need to guarantee that they are deterministic. In this definition of observational determinism, we require that for any two system executions (specified by $\forall\pi\forall\pi'$), their projections of each publicly visible program variable (all variables l in the set L) are equivalent up to

stuttering and prefixing (specified by $l(\pi) \precsim l(\pi') \vee l(\pi') \precsim l(\pi)$). This specification accommodates the asynchronicity between executions by removing repeating steps (i.e., stuttering) and comparing executions of different durations (i.e., prefixing).

7.1 Hypernode Automata

Hypernode automata are transition systems with states labeled with hypernode formulas (which we introduce in Section 7.1.1) and transitions labeled with (synchronizing) actions. In what follows, we work with program executions abstracted as finite or infinite sequences of trace segments with synchronization actions. Unless mentioned otherwise, X is a finite set of *program variables* over a domain Σ, and A is a finite set of *(synchronizing) actions* A, with $A_\varepsilon = A \cup \{\varepsilon\}$ where ε is the empty action.

A hypernode automaton reads simultaneously all traces in an input set of *action-labeled traces*, with actions dictating when each trace transitions to a new state in the automaton. An *action-labeled trace* ρ is a finite or infinite sequence of pairs, each consisting of a valuation and an action label (which can be the empty action). Formally, an action-labeled trace is either $\rho \in (\Sigma^X \times A_\varepsilon)^*$ or $\rho \in (\Sigma^X \times A_\varepsilon)^\omega$. For technical simplicity, we require that infinite action-labeled traces have infinitely many non-empty action labels.

7.1.1 Hypernode Logic

Hypernode logic adopts a maximally asynchronous view on finite traces (which we may refer to as *trace segments*). In particular, hypernode formulas specify hyperproperties where each program variable's progress is fully independent from each other.

Formally, hypernode logic is a first-order logic with 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. The predicate $x(\pi) \precsim y(\pi')$, with $\precsim$ referred to as *stuttering-prefix* relation, holds when the ordered value changes of x in the trace assigned to π is equivalent to the changes of the values y in the trace in π'. As the name of $\precsim$ suggests, while comparing $x(\pi)$ with $y(\pi')$ we allow their value changes to happen at different times (i.e., they must be equal up to stuttering), and values of y in π' to include additional changes (i.e., $x(\pi)$ only needs to define a prefix of $y(\pi)$ changes). The formulas of hypernode logic are defined by the following grammar:

$$\varphi \;::=\; \exists \pi\, \varphi \mid \neg\varphi \mid \varphi \wedge \varphi \mid x(\pi) \precsim x(\pi)$$

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.

We interpret hypernode formulas over sets of *unzipped trace segments*. A *trace segment* is just a finite trace, i.e., a finite sequence of valuations in Σ^X, where each *valuation* $v : X \to \Sigma$ maps program variables to domain values. While an *unzipped trace segment* τ maps program variables to finite strings of values (i.e., $\tau : X \to \Sigma^*$), with each variable

progressing independently of the others. We refer to the trace of each variable mapped in an unzipped trace as a *variable-trace*. The *unzipping of a trace segment* $\tau = v_0 v_1 \cdots v_n$ over $X = \{x_0, \ldots, x_m\}$ is defined as: $\mathit{unzip}(\tau) = \{x_0 : v_0(x_0) \cdots v_n(x_0),\ \ldots,\ x_m : v_0(x_m) \cdots v_n(x_m)\}$. We extend this notion to unzipping of a set of traces T as $\mathit{Unzip}(T) = \{\mathit{unzip}(\tau) \mid \tau\ \in T\}$.

Hypernode logic formulas 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 trace variables in $\mathcal{V}$ to an unzipped trace segment in T. As usual, we denote by $\Pi_T[\pi \mapsto \tau]$ the update of Π_T, where π is assigned to τ and all other trace assignments remain the same. The satisfaction relation for a hypernode formula φ over an assignment Π_T is defined inductively as follows:

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

A set T is a *model* of a hypernode formula φ, $T \models \varphi$, iff there exists an assignment Π_T such that $\Pi_T \models \varphi$. We adopt the usual abbreviations $\forall \pi \varphi \stackrel{\text{def}}{=} \neg \exists \pi \neg \varphi$ and $\varphi \wedge \varphi' \stackrel{\text{def}}{=} \neg(\neg \varphi \vee \neg \varphi')$.

Example: Specifying Non-interference with Hypernode Formulas

In this example, we show how to use hypernode formulas to specify different variants of non-interference proposed in the literature.

Our first variant is from the foundational work by Zdancewic and Myers [ZM03], where they explore how to use observational determinism to capture non-interference over concurrent programs. In this work, they observe that for the public behavior of a program to be deterministic, the program's execution does not necessarily need to be deterministic. Instead, they point out that some degree of non-determinism may be unavoidable (for example, it may not be possible to control how the runtime environment affects the program's scheduling) and advocate that the challenge is in identifying permissible non-determinism that does not open the possibility for a security exploit. They propose to compare different programs' executions up to their stuttering and the duration of the shortest one. Formally, for every pair of program executions, all their publicly visible variables (specified by a set L) must be stutter-equivalent up to prefixing, which we specify as follows:

$$\forall \pi \forall \pi' \bigwedge_{l \in L} (l(\pi) \precsim l(\pi') \vee l(\pi') \precsim l(\pi)).$$

The use of prefixing in the definition above allows leakage of information related to the termination of the program. To address this limitation. Huisman, Worah,

and Sunesen [HWS06] proposed to strengthen the previous definition by requiring every publicly visible variable to be stutter-equivalent, which we express below:

$$\forall\pi\forall\pi' \bigwedge_{l\in L} l(\pi) \precsim l(\pi').$$

The universal quantification forces the relation $l(\pi) \precsim l(\pi')$ to be symmetric, practically enforcing $l(\pi)$ and $l(\pi')$ to have the same size (i.e., to be stuttering equivalent).

A third variant of observational determinism in the literature was presented by Terauchi[Ter08], requiring that combined behavior of all publicly visible variables to be stutter-equivalent up to prefixing. To compare all public variables combined, we exploit the fact that hypernode logic allows for arbitrary finite domains, and combine all values of public variables in a variable L. We can then express their property with hypernode logic, as follows:

$$\forall\pi\forall\pi' \ (L(\pi) \precsim L(\pi') \vee L(\pi') \precsim L(\pi)).$$

We can also specify independence [BFH+22b], also known as generalized noninterference. We say that the values in variables x and y are *independent* if for all sequence of value changes of x witnessed by a trace π, and sequence of value changes for y witnessed by a trace π', there exists a third trace that witnesses their combination $(x(\pi), y(\pi'))$. In this encoding, we required that the trace witnessing the value combinations is equivalent to them up to stuttering and prefixing:

$$\forall\pi\forall\pi'\exists\pi_\exists \ (x(\pi) \precsim x(\pi_\exists) \wedge y(\pi') \precsim y(\pi_\exists)).$$

Stutter-reduced traces

In the semantic interpretation of hypernode formulas presented above, the stuttering-prefix predicate cares only about how variables' values progress, disregarding the number of consecutive steps the variables stay in the same value. In fact, as we will prove below, hypernode formulas cannot distinguish sets of unzipped traces where the only difference is in the number of stuttering valuations in the variable-traces.

Given variable $x \in X$ and an unzipped trace τ s.t. $\tau(x) \in \sigma_0^+ \dots \sigma_n^+$, where $\sigma_i \neq \sigma_{i+1}$ for $0 \leq i < n$, its *x-stutter-reduction* is $\lfloor \tau(x) \rfloor = \sigma_0 \dots \sigma_n$. We extend this notion naturally to the stutter-reduction of an unzipped trace τ as $\lfloor \tau \rfloor(x) = \lfloor \tau(x) \rfloor$, for all variables $x \in X$; and to the stutter reduction of a set of unzipped traces T as $\lfloor T \rfloor = \{\lfloor \tau \rfloor \mid \tau \in T\}$. We prove below that hypernode formulas cannot distinguish between a set of unzipped trace segments T and its stutter-reduction $\lfloor T \rfloor$.

Proposition 7.1.1. *Let $T \subseteq (\Sigma^*)^X$ be a set of unzipped trace segments and φ a formula of hypernode logic. Then, $T \models \varphi$ iff $\lfloor T \rfloor \models \varphi$.*

Proof. We prove this statement by proving the analogous for open formulas and trace assignments. The only interesting case is the base case for atomic formulas $x(\pi) \precsim y(\pi')$, which follows from both definitions of satisfaction of hypernode formulas and stutter-reduction of unzipped traces. □

7.1.2 Hypernode Automata

A hypernode automaton is a finite automaton with transition labeled with actions and states (which we refer to as *hypernodes*) labeled with hypernode formulas.

Definition 7.1.1. *A deterministic and finite* hypernode automaton *(HNA) defined over a set of actions A and program variables X is a tuple $\mathcal{H} = (Q, \hat{q}, \gamma, \delta)$, where:*

- *Q is a finite set of states (also called* hypernodes*) 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 relation $\delta : Q \times A \to Q$ is a total function that assigns to each state and action a unique successor state.*

We define HNA as complete and deterministic automata to ease the presentation of the results in this chapter. For the same reason, in this work we assume that all states in a HNA are accepting (such automata are often called *safety automata*).

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, starting in the initial hypernode $q_0 = \hat{q}$ and satisfying the transition function, $\delta(q_i, a_i) = q_{i+1}$ with $i \geq 0$. We refer to the sequence of actions $p = a_0 a_1 a_2 \ldots$ derived from a run r as the *action sequence* of r. Given that HNA are deterministic and complete, then each action sequence p identifies a unique run in each HNA $\mathcal{H}$, denoted by $\mathcal{H}[p]$.

Hypernode automata read sets of action-labeled traces. Let $\rho = (v_0, a_0)(v_1, a_1)(v_2, a_2) \ldots$ be an action-labeled trace with $v_i \in \Sigma^X$ and $a_i \in A_\varepsilon$ for all $i \geq 0$. Its *action sequence* is the projection of its actions with all empty labels ε removed. Formally, $\rho|_A = a'_0 a'_1 \ldots$ with $a_0 a_1 \ldots \in a'_0 \varepsilon^* a'_1 \varepsilon^* \ldots$ and $a'_i \in A$ for all $i \geq 0$. We extend this definition to sets R of action-labeled traces, and define the *projection of a set of traces R with respect to a finite action sequence* $p \in A^*$ as $R[p] = \{\rho \in R \mid \rho|_A = p\, p'$ for some suffix $p' \in A^* \cup A^\omega\}$.

We assume that actions in action-labeled traces synchronize observable system behavior with specification states, with each new non-empty action defining a segment (or slice) in its trace. Additionally, we assume our actions to be enabled, i.e., while reading a set of action-labeled traces, as soon as a hypernode automaton reads a non-empty action, it must transition accordingly. Then, each step in a hypernode automaton run defines a *slicing* on the set of action-labeled traces it reads. Let $p = a_0 a_1 \ldots a_n$ be a finite action sequence, and $\rho = (v_0, a'_0)(v_1, a'_1) \ldots$ be an action-labeled trace that

has prefix p. We write $\rho(\varnothing, a_0)$ for the initial trace segment of ρ which ends with the action label a_0. Formally, $\rho(\varnothing, a_0) = v_0 \dots v_k$ such that $a'_k = a_0$, and $a'_i = \varepsilon$ for all $0 \leq a_i < k$. For the trace segments of ρ that follow the initial one, we write $\rho(a_0 a_1 \dots a_{i-1}, a_i)$, specifying the segment ending with the action label a_i after having seen the action sequence $a_0 a_1 \dots a_{i-1}$. Formally, we define $\rho(a_0 a_1 \dots a_i, a_{i+1}) = v_{k+1} \dots v_m$ inductively, where $\rho(a_0 a_1 \dots a_{i-1}, a_i) = v_0 \dots v_k$, and $a'_j = \varepsilon$ for all $k < j < m$, and $a'_m = a_{i+1}$. We extend the slicing sets of action-labeled traces naturally, where, for example, $R(\varnothing, a) = \{\rho(\varnothing, a) \mid \rho \in R\}$.

Definition 7.1.2. *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, $Unzip(R[p](\varnothing, a_0)) \models \gamma(q_0)$, and $Unzip(R[p](a_0 \dots a_{i-1}, a_i)) \models \gamma(q_i)$ 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 $\mathcal{L}(\mathcal{H})$.

Example: Hypernode Automata Runs

In this example, we illustrate how hypernode automata read sets of traces. In particular, we consider the hypernode automaton $\mathcal{H}$, in Figure 7.1, specifying a mutually exclusive declassification property and the set of traces generated by the program $\mathcal{Q}_v$ in Algorithm 7.1.

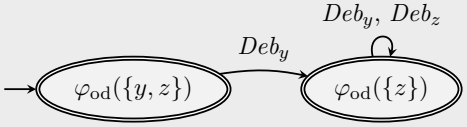
Algorithm 7.1: Program $\mathcal{Q}_v$

```
1 do
2 |   v := 0;
3 |   read(x);
4 |   read(status);
5 |   if (status = Deb_v) then
6 |   |   v := x;
7 |   end
8 |   output(v);
9 while true;
```

Table 7.1: Executions of $\mathcal{Q}_y \parallel \mathcal{Q}_z$.

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

Figure 7.2: Path induced by pattern Deb_y Deb_z on $\mathcal{H}$ depicted in Figure 7.1.



The program $\mathcal{Q}_v$ is a reactive program parameterized by a variable v. The functionality of $\mathcal{Q}_v$ is to expose the value of x directly through the value of v to

its output channel when it is notified that variable v is responsible to debug x, i.e., $status = Deb_v$. Note that v is a parameter and can be instantiated with any variable name. Each iteration of the program loop reads the input variable x and action *status* from an input channel, and when the program reads the status Deb_v, it behaves as described above.

Consider the set of traces shown in Figure 7.1. We first observe that the set of traces T exhibit the same sequence of actions: Deb_y followed by Deb_z. This action pattern partitions each trace into three trace segments, called *slices*, which we distinguish in Table 7.1 using different cell colors. Moreover, the action sequence defines a unique sequence of hypernodes, which we depict in Figure 7.2.

The next step in our verification process is to map each of the slices to the corresponding hypernodes. The mapping is straightforward: the slice colored with white in Table 7.1 is mapped to the initial node; the light-gray portion is mapped to the hypernode labeled with $\varphi_{\text{od}}(\{z\})$ with the dark-gray fragment also mapped to the same hypernode.

7.2 Model-Checking Hypernode Logic and Automata

This section presents a novel approach to model-check asynchronous hyperproperties specified by hypernode automata. In particular, we focus on the problem of checking if a given hypernode automaton accepts a set of infinite traces generated by an action-labeled Kripke structure. Our algorithm works at two levels. First, it uses the action labels to slice the Kripke structure (the *model*) and synchronize each slice with the matching step in the hypernode automata (the *specification*). Then, in each step, we solve the model-checking problem for the hypernode formula in the hypernode assigned to the step over the matching slice of the original Kripke structure. This approach's novelty lies in the combination of the model slicing and the introduction of new automata-theoretic constructions over a new type of automata, called *stutter-free* automata. Stutter-free automata is a simple formalism that allows seamless reasoning about variables' independent value progression.

In what follows, we assume that all program variables are propositional; that is, $\Sigma = \{0, 1\}$ is boolean. Note that all finite domains can be encoded by propositional variables.

7.2.1 Problem Statement

A *Kripke structure* is defined by a tuple $K = (W, X, \Delta, V)$ where W is a set of worlds, X is a set of atomic propositions, $\Delta \subseteq W \times W$ is a transition relation, and $V : W \times X \to \{0, 1\}$ is a function assigning a truth value to each proposition in each world. We denote by (K, w_0) the *pointed Kripke structure* where the world $w_0 \in W$ is the structure initial world. Our Kripke structures are labeled with actions from a set A. In particular, for a Kripke structure with a transition relation Δ, 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 denote by $\mathrm{Paths}(K, \mathbb{A})$ the set of all such paths. The path $r = w_0 a_0\, w_1 a_1 \ldots$ defines the action-labeled trace $\mathit{sync}(r) = V(w_0) a_0\, V(w_1) a_1 \ldots$. We denote by $\mathit{Zip}(K, \mathbb{A})$ the set of action-labeled traces defined by paths in $\mathrm{Paths}(K, \mathbb{A})$. Naturally, $\mathrm{Paths}(K, \mathbb{A}, w_0)$ denotes the set of all paths in $\mathrm{Paths}(K, \mathbb{A})$ that start at the world w_0, and $\mathit{Zip}(K, \mathbb{A}, w_0)$ refers to the set of all action-labeled traces defined by paths in $\mathrm{Paths}(K, \mathbb{A}, w_0)$.

We now have all the necessary definitions to formalize the central verification question solved in this section: the model-checking problem for specifications given as hypernode automata over pointed Kripke structures with an action labeling.

Model-checking problem for hypernode automata

Let (K, w_0) be a pointed Kripke structure with the set X of propositions, and let $\mathbb{A}$ an action labeling for K over a set A of actions. Let $\mathcal{H}$ be a hypernode automaton over the set X of boolean program variables and the set A of actions. Is the set of initialized action-labeled traces generated by $(K, \mathbb{A}, w_0)$ accepted by $\mathcal{H}$; that is, $\mathit{Zip}(K, \mathbb{A}, w_0) \in \mathcal{L}(\mathcal{H})$?

Example: Model-checking Hypernode Automata

In this example, we show how to model-check the set of traces generated by a program against the mutually exclusive declassification property specified by the hypernode automaton in Figure 7.1. As in the original example, we interpret observational determinism as proposed by a Zdancewic and Myers [ZM03], i.e.:

$$\varphi_{\mathrm{od}}(L) \stackrel{\mathrm{def}}{=} \forall\pi \forall\pi' \bigwedge_{l \in L} (l(\pi) \precsim l(\pi') \vee l(\pi') \precsim l(\pi)).$$

The set of traces generated by the parallel composition $\mathcal{Q}_y \,||\, \mathcal{Q}_z$, where $\mathcal{Q}_v$ is defined in is not a model of the hyperproperty specified by the hypernode automaton $\mathcal{H}$, in Figure 7.1. The set of traces $T = \{\tau_1, \tau_2\}$ shown in the previous example in Table 7.1, witness the specification violation.

Starting from the sequence of hypernodes derived from the action pattern $\mathit{Deb}_y \mathit{Deb}_z$ (illustrated in Figure 7.2), the next step is to show that the slicing of T induced by $\mathit{Deb}_y \mathit{Deb}_z$ satisfies the matching hypernode formulas in $\mathcal{H}$. The violation occurs

because the dark-gray slice of T (the third slice) does not satisfy $\varphi_{\text{od}}(\{z\})$. More specifically, the 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.

To solve the model-checking problem stated above, we need to solve the model-checking problem for hypernode logic over Kripke structures, as this constitutes the key subroutine for model-checking hypernode automata. Due to the need to slice the Kripke structure before we model check it against the formula in the respective hypernode, we are interested in *open Kripke structures*. Open Kripke structures are Kripke structures extended with two sets of worlds: the *entry worlds*, signalling where the segment (or slice) begins, and the *exit worlds*, marking where the segment ends. Formally, an *open Kripke structure* consists of a Kripke structure $K = (W, X, \Delta, V)$, and a pair $\mathbb{W} = (W_{\text{in}}, W_{\text{out}})$ consisting of a set $W_{\text{in}} \subseteq W$ of entry worlds, and a set $W_{\text{out}} \subseteq W$ of exit worlds.

A *path* of the open Kripke structure $(K, \mathbb{W})$ is finite sequence of worlds $w_0 \dots w_n$ that starts in a entry world, $w_0 \in W_{\text{in}}$, ends in an exit world, $w_n \in W_{\text{out}}$, and is consistent with the transition relation, $(w_i, w_{i+1}) \in \Delta$ for all $0 \leq i < n$. Similar to the definition over Kripke structures, we denote by $\text{Paths}(K, \mathbb{W})$ the set with all paths defined by the open Kripke structure $(K, \mathbb{W})$. Let $X = \{x_0, \dots, x_m\}$. The set of unzipped trace segments generated by the open Kripke structure $(K, \mathbb{W})$ is defined as $\mathit{Unzip}(K) = \{\{x_0{:}V(w_0, x_0)..V(w_n, x_0), \dots, x_m{:}V(w_0, x_m)..V(w_n, x_m)\} \mid w_0 \dots w_n \in \text{Paths}(K, \mathbb{W})\}$.

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 propositional variables. Is the set of unzipped trace segments generated by $(K, \mathbb{W})$ a model for φ; that is, $\mathit{Unzip}(K, \mathbb{W}) \models \varphi$?

7.2.2 Stutter-free Automata

Before we dive into the model-checking procedure, we introduce its main building block: *stutter-free automata*. We proved in Proposition 7.1.1 that when we want to check if a set of traces is a model of a given hypernode formula, it suffices to consider the stutter reduction of the given set. Likewise, to model-check hypernode formulas over the set of traces a given Kripke structure generates, we can instead reason about all variable-traces generated by paths with no stuttering steps. Stutter-free automata is a formalism to represent and reason about such (independent) stutter-free paths. In particular, stutter-free automata are a restricted form of nondeterministic finite automata (NFA) that read unzipped trace segments and guarantee that, for each state, there are no repeated variable assignments on their incoming and outgoing transitions.

In what follows, Σ^X denotes all assignments of variables in X to values in Σ or the termination symbol $\#$, i.e., for $X = \{x_0, \ldots, x_m\}$:

$$\Sigma^X = \{x_0 : \sigma_0, \ldots, x_m : \sigma_m \mid \forall 0 \leq i \leq m \text{ with } \sigma_i \in \Sigma \cup \{\#\}\} \setminus \{x_0 : \#, \ldots, x_m : \#\}.$$

Definition 7.2.1. *Let X be a finite set of variables over Σ. A nondeterministic* stutter-free automaton *(NSFA) over the alphabet Σ^X 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 $x \in X$:*

- stutter-freedom *requiring $In(q, x) \cap Out(q, x) \subseteq \{\#\}$, and*
- termination *requiring that if $\# \in In(q, x)$, 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:

$$In(q, x) = \{v(x) \mid q \in \delta(q', v) \text{ for some } q' \in Q\} \text{ and } Out(q, x) = \{v(x) \mid \delta(q, v) \neq \emptyset\}.$$

A *run* of the stutter-free automaton $\mathcal{A}$ is a finite sequence $q_0 v_0 q_1 v_1 \ldots v_{n-1} q_n$ alternating between states and variable assignments starting with an initial state, $q_0 \in \hat{Q}$, and following $\mathcal{A}$ transition function δ (i.e., $q_{i+1} \in \delta(q_i, v_i)$ for all $i < n$); with a run being *accepting* if it ends in a final state.

A stutter-free automaton reads unzipped finite traces. Let τ be an unzipped trace segment over a set of variables X with domain Σ. The trace τ is *accepted* by the stutter-free automaton $\mathcal{A}$ iff there exists an accepting run $q_0 v_0 \ldots v_{n-1} q_n$ where the sequence of assignments for x in that sequence defines the x-trace in τ; formally, for all variables $x \in X$, $\tau(x) = v_0(x) \ldots v_{n-1}(x)$. Then, naturally, the *language* of $\mathcal{A}$, denoted $\mathcal{L}(\mathcal{A})$, is the set with all unzipped trace segments accepted by $\mathcal{A}$.

Without loss of generality, we will often refer to the language of a stutter-free automaton as the accepted language with the termination symbol removed,i.e., $\mathcal{L}(\mathcal{A})|_\# = \{\tau|_\# : X \to \Sigma^* \mid \tau \in \mathcal{L}(\mathcal{A})\}$, where $\tau|_\#$ removes all occurrences of $\#$ in a trace segment τ. When $\mathcal{A}$ is a stutter-free automaton, then the accepted language is already stutter-reduced, i.e., $\mathcal{L}(\mathcal{A})|_\# = \lfloor \mathcal{L}(\mathcal{A})|_\# \rfloor$.

Example: Stutter-free Automata

In Figure 7.3 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 Figure 7.3 is:

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

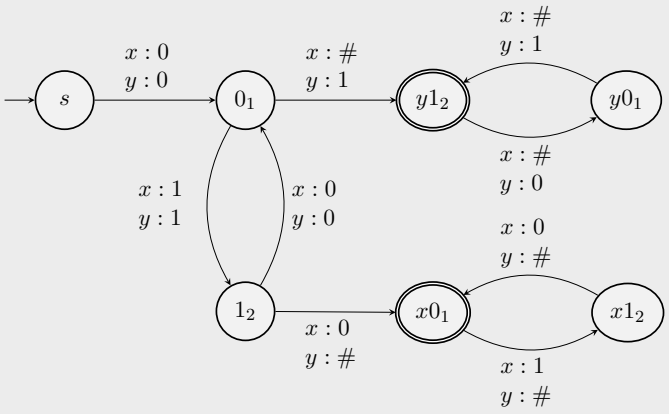


Figure 7.3: Stutter-free automaton $\mathcal{A}$ where x-traces are of odd size while y-traces are of even size, and the first valuation for both x and y is 0.

The union, intersection, and determinization for NSFA 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 stutter-free automata over the same alphabet Σ^X. We define:

- their *union* as $\mathcal{A}_1 \cup \mathcal{A}_2 = (Q_1 \dot\cup Q_2, \hat{Q}_1 \dot\cup \hat{Q}_2, F_1 \dot\cup F_2, \delta_\cup)$ over the same alphabet Σ^X where $\dot\cup$ is 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 Σ^X 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 $v \in \Sigma^X$.

We prove below that stutter-free automata are closed under the operators defined above.

Proposition 7.2.1. *Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be two stutter-free automata over the propositional variables X. Then, both $\mathcal{A}_1 \cup \mathcal{A}_2$ and $\mathcal{A}_1 \cap \mathcal{A}_2$ are stutter-free automata over X 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 automaton $\mathcal{A}$, the determinization $det(\mathcal{A})$ is a deterministic stutter-free automaton over X with the same language, that is, $\mathcal{L}(det(\mathcal{A})) = \mathcal{L}(\mathcal{A})$.*

Proof. We observe that union and disjunction do not affect the stutter-free condition. Then, the result follows from a direct translation from stutter-free automata to NFA.

Let $\mathcal{A}_1$ be an arbitrary stutter-free automata. We start by proving that $\det(\mathcal{A}_1)$ satisfies stutter-freedom, i.e. $In(S,x) \cap Out(S,x) = \emptyset$, for all states S of $\det(\mathcal{A}_1)$ and variables $x \in X$. By definition:

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

Thus, $(\star)$ for all $q \in S$, $In(q,x) = In(S,x)$. From $\mathcal{A}_1$ being a stutter-free automaton we know that, for all $q \in S$, $In(q,x) \cap Out(q,x) = \emptyset$. Assume towards a contradiction that there exists a value that is in both the incoming and outgoing transitions of S for a variable x, i.e. $l \in In(S,x) \cap Out(S,x)$. 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 $(\star)$.

We prove now that $\det(\mathcal{A}_1)$ satisfies the #-ending requirement. By $(\star)$, it follows that if $\# \in In(x)$ then $\# \in In(x)$ for all $q \in S$. So, by $\mathcal{A}$ being a stutter-free automaton, for all $q \in S$ we have $Out(x) = \{\#\}$. Hence $Out(x) = \{\#\}$.

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

To complement a stutter-free automaton, 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 nonfinal states. The only challenging step is completing the automaton, as we must ensure that all added transitions satisfy the stutter-freedom requirement. In particular, when completing a stutter-free automaton, we must add a sink state representing all the missing transitions. By definition, all missing transitions lead to non-accepting runs, independently of what is read next by the automaton. The stutter-freedom 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 stutter-free automata*, to represent such sink states.

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 7.2.1. Note that we generalize the set of all incoming and outgoing variable valuations as follows:

$$In(q) = \{v(x) \mid x \in X, v(x) \in In(q,x)\} \text{ and } Out(q) = \{v(x) \mid x \in X, v(x) \in Out(q,x)\}.$$

The *universal* stutter-free automaton $\mathcal{U}_{\Sigma^X}$ over Σ^X is a deterministic and complete automaton, accepting stutter-free unzipped traces over Σ^X. This means that the $\mathcal{U}_{\Sigma^X}$ accepts the language $\mathcal{L}(\mathcal{U}_{\Sigma^X})|_{\#} = \lfloor (\Sigma^*)^X \rfloor$.

Definition 7.2.2. *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

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

Example: Universal Stutter-free Automaton

In the Figure 7.4 below, we depict the universal stutter-free automaton $\mathcal{U}_X$ for the set of boolean variables $X = \{x, y\}$.

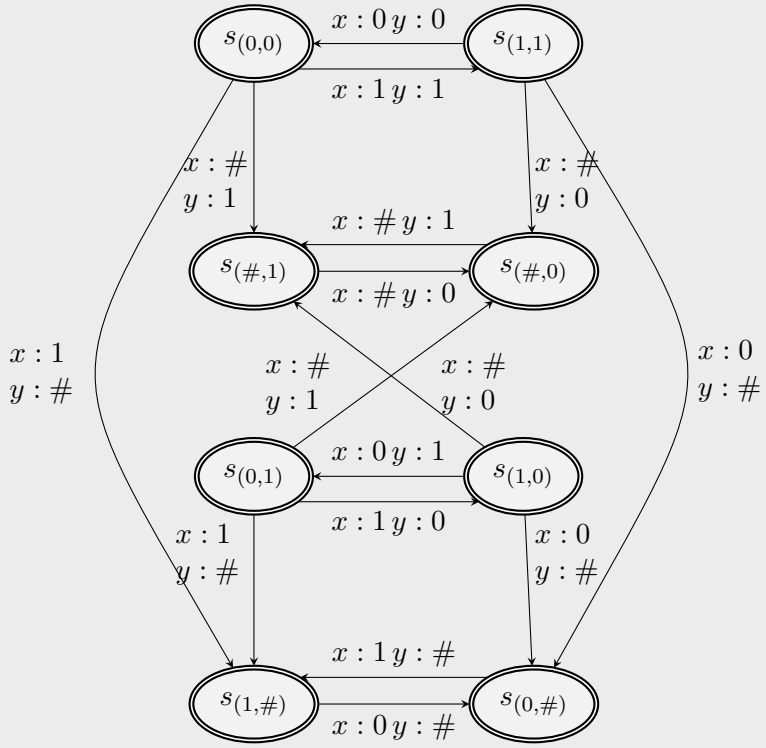


Figure 7.4: The universal stutter-free automaton $\mathcal{U}_{\{x,y\}}$ over the boolean variables $\{x, y\}$. It accepts all stutter-free unzipped trace segments over $\{x, y\}$. All states are both initial and final.

Definition 7.2.3. *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 $\mathcal{U}_{\Sigma^X}$*, and for all states* $q \in Q \cup Q_{\mathcal{U}}$ *and all valuations* $v \in \Sigma^X$*:*

$$\delta'(q,v) = \begin{cases} \delta_{\mathcal{U}}(q,v) & \text{if } q \in Q_{\mathcal{U}}, \\ \delta(q,v) \cup \{v' \mid \text{for all } x \in X \text{ if } v(x) = \# \text{ then } v'(x) = \# & \\ \qquad \text{otherwise } v'(x) \in (\Sigma \setminus \mathit{In}(q,x) \cup \mathit{Out}(q,x))\} & \text{if } q \in Q. \end{cases}$$

where In *and* Out *are defined with respect to the transition relation* δ *of* $\mathcal{A}$*, and* $\delta_{\mathcal{U}}$ *is the transition relation of the universal stutter-free automaton* $\mathcal{U}_{\Sigma^X}$*.*

We can now prove that our completion procedure returns an automaton accepting the same language and the input automaton.

Proposition 7.2.2. *Let* $\mathcal{A}$ *be a stutter-free automaton. Then,* 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 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 complete($\mathcal{A}$) is a stutter-free automaton is only missing to prove that extension of the $\mathcal{A}$ transition relation (pointing to states in the universal automaton) preserves both requirements. By definition, if a variable trace is terminated, it will remain terminated (if $v(x) = \#$ then $v'(x) = \#$), 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 edges of a state (for all $x \in X$, $v'(x) \notin \mathit{In}(q) \cup \mathit{Out}(q)$), hence it satisfies the stutter-free requirement. We now prove that complete($\mathcal{A}$) is complete. By 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 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 7.2.3. *Let* $\mathcal{A}$ *be a deterministic and complete stutter-free automaton over* Σ^X*. Then,* $\overline{\mathcal{A}}$ *is a stutter-free automaton and* $\mathcal{L}(\overline{\mathcal{A}}) = \lfloor (\Sigma^*)^X \rfloor \setminus \mathcal{L}(\mathcal{A})$*.*

Proof. Let $\mathcal{A}$ be an arbitrary stutter-free automaton. It follows directly from $\mathcal{A}$ being a stutter-free automaton that $\overline{\mathcal{A}}$ is also stutter-free automaton (complementing an

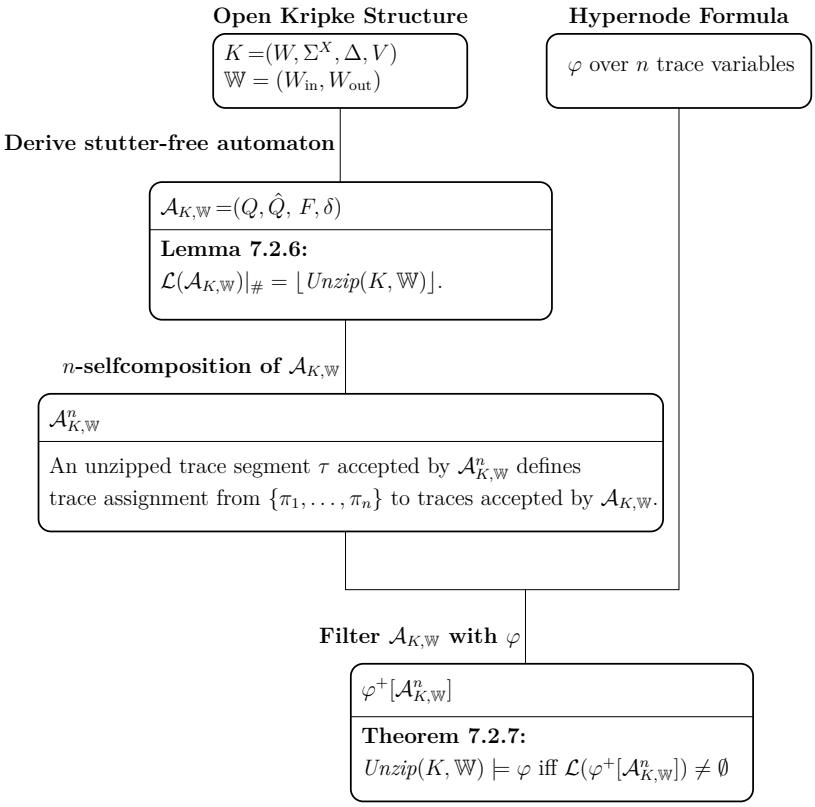


Figure 7.5: Model-checking algorithm for hypernode formulas with relevant results.

automaton does not change its transition function). For the same reason, $\overline{\mathcal{A}}$ is a deterministic and complete stutter-free automaton, as well. Then, for all unzipped traces $\tau\in\lfloor(\Sigma^*)^X\rfloor$ there exists a run r in $\overline{\mathcal{A}}$ for τ (from completeness); and it this the only run for τ (from determinism). As $\overline{\mathcal{A}}$ and $\mathcal{A}$ share the same transition function, then r is also 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 $\mathcal{L}(\overline{\mathcal{A}})=\lfloor(\Sigma^*)^X\rfloor\setminus\mathcal{L}(\mathcal{A})$. □

7.2.3 Model-checking Hypernode Formulas

We start by showing how to model-check hypernode formulas over stutter-free automata. Our algorithm is based on a filtration of hypernode automata by a hypernode formula. The filtration guarantees that if its input automaton is a model for the given formula, then the language of the resulting automaton is non-empty. With the model-checking problem of stutter-free automata solved, the next step is to show how to translate Kripke structures to stutter-free automata. In particular, we define a translation from an open Kripke structure $(K,\mathbb{W})$ to a stutter-free automaton such that the automaton accepts the same unzipped trace segments as the stutter reduction of the set of trace segments generated by $(K,\mathbb{W})$. We give an overview of the model-checking algorithm in Fig. 7.5.

When defining the filtration of stutter-free automata, all boolean entities (both operators and quantifiers) are defined using the stutter-free automata operators introduced in the previous section. The only ingredient missing is the filtration by atomic formulas, i.e., formulas of the form $x(\pi) \precsim y(\pi')$. Below, we introduce a sub-automaton of the universal automaton that captures the requirement specified by an atomic hypernode formula, which we use later in the filtration definition.

Definition 7.2.4. *Let $\mathcal{U}_X = (Q_\mathcal{U}, Q_\mathcal{U}, Q_\mathcal{U}, \delta_\mathcal{U})$ be the universal stutter-free automaton over the set X of propositional variables, and let $x, y \in X$. The* stutter-free automaton for the atomic formula $x \precsim y$ of hypernode logic *is defined as the stutter-free automaton $\mathcal{A}_{x \precsim y} = (Q, Q, Q, \delta)$ over X, 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 $v \in \Sigma^X$.*

We extend the set X of propositional variables with reference to trace variables in $\mathcal{V}$ as $X(\mathcal{V}) = \{x(\pi) \mid x \in X \text{ and } \pi \in \mathcal{V}\}$. Given an accepting run $\rho = q_0\sigma_0q_1\sigma_1 \dots \sigma_{n-1}q_n$ of the stutter-free automaton $\mathcal{A}_{x(\pi) \precsim y(\pi')}$ over $\Sigma^{X(\mathcal{V})}$, the trace assignment generated by ρ is defined as $\Pi_\rho(\pi, x) = \sigma_0(x(\pi)) \dots \sigma_{n-1}(x(\pi))$, for all $\pi \in \mathcal{V}$ and $x \in X$. We prove that all accepting run for the automaton derived for$x \precsim y$ define assignments that satisfy $x \precsim y$.

Lemma 7.2.4. *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 $\Pi_\tau \models x(\pi) \precsim y(\pi')$.*

Proof. Consider arbitrary unzipped trace segment τ over $(\Sigma^{X_\mathcal{V}})^*$. We start with the $\Rightarrow$-direction and assume that $\tau \in \mathcal{L}(\mathcal{A}_{x_\pi \precsim y_{\pi'}})|_\#$. Then, there exists an accepting run $q_0\sigma_0q_1\sigma_1 \dots \sigma_nq_{n+1}$ where, for all $x_\pi \in X_\mathcal{V}$, if $0 \leq i < |\tau(x_\pi)|$, then $\sigma_i(x_\pi) = \tau(x_\pi, i)$, otherwise $\sigma_i(x_\pi) = \#$. By definition of universal stutter-free automaton and $\mathcal{A}_{x_\pi \precsim y_{\pi'}}$, it follows that for all $0 \leq j \leq n$, $\sigma_j(x_\pi) = \sigma_j(y_{\pi'})$ or $\sigma_j(x_\pi) = \#$. Moreover, by the termination requirement of stutter-free automaton, if there exists j s.t. $\sigma_j(x) = \#$, then the value of x will not change anymore (i.e, for all $k \geq j$, $\sigma_k(x) = \#$). We can then prove by induction of the trace τ size that $\Pi_\tau \models x(\pi) \precsim y(\pi')$. For the $\Leftarrow$-direction, we assume that τ is not accepted by $\mathcal{A}_{x_\pi \precsim y_{\pi'}}$. As all states in $\mathcal{A}_{x_\pi \precsim y_{\pi'}}$ are final, then there exists a step $0 \leq i \leq n$ where $\sigma_i(x_\pi) \neq \sigma_i(y_{\pi'})$ and $\sigma_i(x_\pi) \neq \#$. Given that τ is stutter-free, then $\Pi_\tau \not\models x(\pi) \precsim y(\pi')$. □

We now define the main component of the model-checking algorithm: the filtration of stutter-free automata over hypernode formulas.

Definition 7.2.5. *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:*

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

W 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 *n-self-composition of* $\mathcal{A}$, denoted by $\mathcal{A}^n$, is the standard synchronous product construction of $\mathcal{A}$ with itself for n times.

Theorem 7.2.5. *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 $\mathcal{L}(\mathcal{A}) \not\models \varphi$.*

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

Let τ be an unzipped trace segment over $(\Sigma^{X_\mathcal{V}})^*$.
Then, $\tau \in \mathcal{L}(\varphi^+[\mathcal{A}^n])|_\#$ iff $\Pi_\tau \models \varphi$; and $\tau \in \mathcal{L}(\varphi^-[\mathcal{A}^n])|_\#$ iff $\Pi_\tau \not\models \varphi$.

We prove the lemma by induction on the structure of the hypernode formula. To prove the base case, $x_\pi \precsim y_{\pi'}$ we use Lemma 7.2.4. In particular, $\tau \in \mathcal{L}(x_\pi \precsim y^+_{\pi'}[\mathcal{A}^n])|_\#$ iff $\tau \in \mathcal{L}(\mathcal{A}^n)|_\#$ and $\tau \in \mathcal{L}(\mathcal{A}_{x_\pi \precsim y_{\pi'}})|_\# \Leftrightarrow$. As $\tau \in \mathcal{L}(\mathcal{A}^n)|_\#$, then Π_τ defines a trace assignment from $\{\pi_1, \dots \pi_n\}$ to traces accepted by $\mathcal{A}$ and, by Lemma 7.2.4, $\Pi_\tau \models x_\pi \precsim y_{\pi'}$. We prove analogously for $\tau \in \mathcal{L}(x_\pi \precsim y^-_{\pi'}[\mathcal{A}^n])|_\#$.

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 7.2.1. While the inductive case $\neg\varphi$ follows from IH and Proposition 7.2.3.

The case for the existential quantifier – $\exists\pi\varphi$ – is more challenging. We first prove the positive filtration and start with the $\Rightarrow$-direction. Consider arbitrary $\tau \in \mathcal{L}((\exists\pi\varphi)^+[\mathcal{A}^n])|_\#$. By definition of filtration, this is equivalent to $\tau \in \mathcal{L}(\varphi^+[\mathcal{A}^n])|_\#$. By IH, $\Pi_\tau \models \varphi$, and so $\Pi_\tau \models \exists\pi\varphi$. 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 Σ^X 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, $\tau_\pi \in \exists\pi\varphi^+[\mathcal{A}^n]$.

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 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 $\tau_\pi \in \mathcal{L}((\exists\pi\varphi)^-[\mathcal{A}^n])$. □

From open Kripke structures to stutter-free automata

Our final step is to translate open Kripke structures to stutter-free automata. In a nutshell, the generated stutter-free automaton keeps an independent record of how each variable value changes along the paths defined by the Kripke structure. To build the record of the independent variable progression, we use the function defined below, returning the next stutter-free transition for each variable x from state w of a given Kripke structure. Formally, for a Kripke structure $K = (W, X, \Delta, V)$ over the set of variables X with domain Σ, the set with all worlds reachable from $w \in W$ with a transition to a world where $x \in X$ is not labeled with $\sigma \in \Sigma \cup \{\#\}$ is:

$$N(w, \sigma, x) = \{w' \mid \mathit{Unzip}(K, w, w')[x] \in \sigma^+, (w', w'') \in \Delta \text{ and } V(w'', x) \neq \sigma\},$$

where $\mathit{Unzip}(K, w, w')$ is the set of unzipped traces defined by paths in K from w to w'. We define a set of terminated states, $W^\# = \{w^\# \mid w \in W\}$, to encode early termination of some of the variable value progression. We define below the transition relation $\text{next}(w \in W \cup W^\#, \sigma \in \Sigma \cup \{\#\}, x \in X)$ that puts all of this together:

$$\text{next}(w, \sigma, x) = \begin{cases} \{w^\#\} & \text{if } V(w, x) = \sigma \text{ and } N(w, \sigma, x) = \emptyset, \\ \{w\} & \text{if } \sigma = \# \text{ and } w \in W^\#, \\ N(w, \sigma, x) & \text{otherwise.} \end{cases}$$

We introduce below the reduction of open Kripke structures to stutter-free automata and prove in Lemma 7.2.6 that the language accepted by the stutter-free automaton induced by a Kripke structure accepts the same set of traces as the stutter-reduction of the set of unzipped trace segments defined by the Kripke structure.

Definition 7.2.6. *Let $(K, \mathbb{W})$ with $\mathbb{W} = (W_{\text{in}}, W_{\text{out}})$ be an open Kripke structure defined over the set of variables $X = \{x_1, \ldots, x_m\}$. The* stutter-free automaton induced by $(K, \mathbb{W})$ *is $\mathcal{A}_{K,\mathbb{W}} = (Q, \hat{Q}, F, \delta)$, where:*

- $Q = (W \cup W^\#)^m$;
- $\hat{Q} = \{(w_1, \ldots, w_m) \mid w_i \in W_{\text{in}} \text{ for all } 1 \leq i \leq m\}$;
- $F = \{(w_1, \ldots, w_m) \mid w_i \in W_{\text{out}} \cup W^\#_{\text{out}} \text{ for all } 1 \leq i \leq m\}$;
- $\delta((w_1, \ldots, w_m), v) = \{(w'_1, \ldots, w'_m) \mid w'_i \in \text{next}(w_i, v[i], x_i) \text{ for all } i\}$.

Lemma 7.2.6. *For all open Kripke structures* $(K, \mathbb{W})$, $\mathcal{L}(\mathcal{A}_{K,\mathbb{W}})|_{\#} = \lfloor \mathit{Unzip}(K, \mathbb{W}) \rfloor$.

Proof. $\mathcal{L}(\mathcal{A}_{K,\mathbb{W}})|_{\#} \subseteq \lfloor \mathit{Unzip}(K, \mathbb{W}) \rfloor$: Let τ be an unzipped trace segment over $(\Sigma^X)^*$ accepted by $\mathcal{A}_{K,\mathbb{W}}$. Then, there exists an accepting $\mathcal{A}_{K,\mathbb{W}}$ run $q_0\sigma_0 q_1 \sigma_1 \dots \sigma_n q_{n+1}$ where $\sigma_i = \tau[i]$, for $i \geq n$. We prove that the property holds by induction on the size of run independently for each variable x_i with $i \leq m$. By definition of next, we prove that for the current step $j \leq n$ – $q_j[i]$ $\sigma_j[i]$ $q_{j+1}[i]$ – either (i) $N(q_j[i], \sigma_j[i], x_i) \neq \emptyset$ and then there exists a path ϱ from $q_j[i]$ to $q_{j+1}[i]$ where the value of x_i remains $\sigma_j[i]$; (ii) $N(q_j[i], \sigma_j[i], x_i) = \emptyset$ and the next state is terminated ($q_{j+1}[i] = q_j^{\#}[i]$); or (iii) $q_j[i]$ is a terminated stated and so $\sigma_j[i] = \#$ and $q_{j+1}[i] = q_{j+1}^{\#}[i]$. Using this we build a path in the Kripke structure by extending the partial path given by the stutter-free automata with ϱ when (i) holds and not doing anything when (ii) and (iii) holds. Note that we made sure to include the terminated exited paths in the final states of the stutter-free automaton, guaranteeing the the accepting condition matches the definition of a complete path over an open Kripke structure. When we stutter reduce the new path we get $\tau[x_i]$.

$\mathcal{L}(\mathcal{A}_{K,\mathbb{W}}) \supseteq \lfloor \mathit{Unzip}(K, \mathbb{W}) \rfloor$: Let τ be an unzipped stutter-free trace segment in $\lfloor \mathit{Unzip}(K, \mathbb{W}) \rfloor$. Then, there exists a path in $(K, \mathbb{W})$ that generates that trace segment. From that path, for each variable, we build a sequence of wolds that removes all stutter transitions. We prove then, by induction, that that sequence defines an accepting run of $\mathcal{A}_{K,\mathbb{W}}$. □

Using the translation of open Kripke structures to stutter-free automata (Definition 7.2.6) and the filtration from Definition 7.2.5, we have an effective way to solve the model-checking problem for hypernode logic over Kripke structures.

Theorem 7.2.7. *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,* $\mathit{Unzip}(K, \mathbb{W}) \models \varphi$ *iff* $\mathcal{L}(\varphi^+[\mathcal{A}^n_{K,\mathbb{W}}]) \neq \emptyset$.

The proof follows from Theorem 1 and 2, and Lemma 3. This gives us our main result.

Theorem 7.2.8. *Model checking of hypernode logic over open Kripke structures is decidable.*

With the presented algorithm, the running time of model checking a hypernode formula 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 7.2.9. *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* $\mathcal{O}(2^{n \cdot k^m})$.

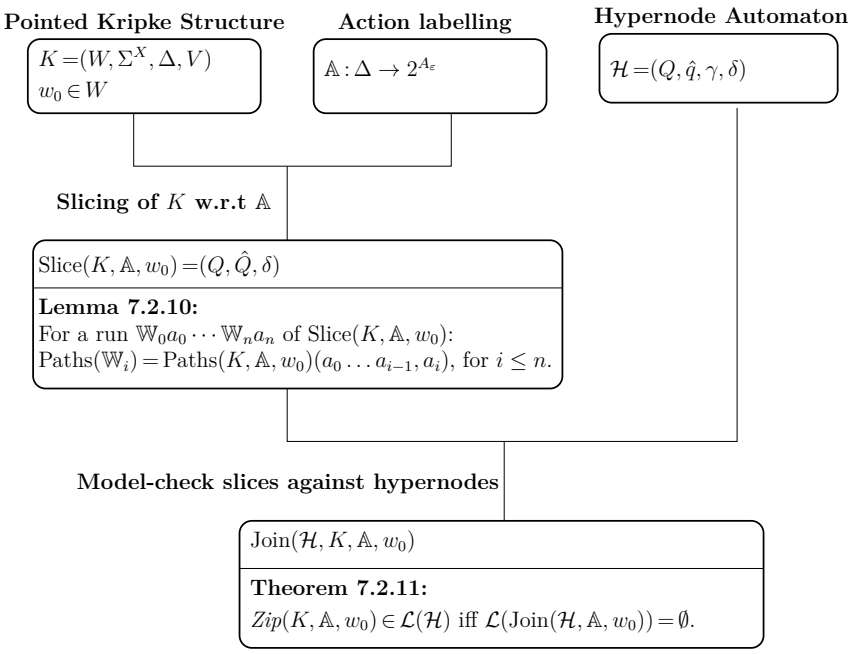


Figure 7.6: Model-checking algorithm for hypernode automata with relevant results.

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. Completing the deterministic stutter-free automaton 2^m states. The n-self-composition of the resulting automaton has $\mathcal{O}(2^{n\cdot k^m})$ states. □

7.2.4 Model Checking Hypernode Automata

Model-checking hypernode automata require slicing the model according to its action-labelled steps and matching the different 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 $\mathrm{Slice}(K,\mathbb{A},w_0)$. 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, $\mathrm{Join}(\mathcal{H},K,\mathbb{A},w_0)$, is non-empty. In a nutshell, the composite automaton accepts all sequence 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. 7.6.

The building blocks of the 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 $W'=\{w\mid(w,w')\in\Delta \text{ or } (w',w)\in\Delta \text{ for some } w'\in W\}$.

We are interested in open sub-structures that contain all transitions needed to define

the paths from a given entry set of worlds to the next transition labeled with action a, using only transitions labeled with an 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:

$$(K, \mathbb{A}, W_{\text{in}}) \downarrow a = \{(w_j, w_{j+1}) \mid w_0 \varepsilon \ldots w_{n-1} \varepsilon\, w_n a \in \text{Paths}(K, \mathbb{A})), w_0 \in W_{\text{in}} \text{ and } 0 \leq j < n\}.$$

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* $(K, \mathbb{A}, W_{\text{in}}) \downarrow a$, which we denote by $\mathbb{K}[(K, \mathbb{A}, W_{\text{in}}) \downarrow a]$. Formally, $\mathbb{K}[(K, \mathbb{A}, W_{\text{in}}) \downarrow a] = (K_s, (W_{\text{in}}, W_{\text{out}}))$ where $K_s = K[(K, \mathbb{A}, W_{\text{in}}) \downarrow a]$ and the set $W_{\text{out}} = \{w \mid w \in W \text{ and } \mathbb{A}(w, a) \neq \emptyset\}$ contains all possible exit points for 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 the source structure.

Definition 7.2.7. *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:*

- *$Q = \{\mathbb{K}[(K, \mathbb{A}, W_{\text{in}}) \downarrow a] \mid a \in A$ and $W_{\text{in}} \subseteq W\}$ is a set of states with initial states $\hat{Q} = \{(K, (\{w_0\}, W_{\text{out}})) \in Q\}$;*
- *$\delta : Q \times A \to Q$ 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 $(K', (W'_{\text{in}}, W'_{\text{out}}))$:*
    if $W''_{\text{in}} \subseteq \{w \mid a \in \mathbb{A}(w', w)$ for some $w' \in W_{\text{out}}\}$, then $W'_{\text{in}} \not\subseteq W''_{\text{in}}$.

We remark that, for all open Kripke structures $\mathbb{K}$ and action a, there is an 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 substructures 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 7.2.10 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 of traces derived from the action-labeled Kripke structure with action pattern p.

Lemma 7.2.10. *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 $Zip(K, \mathbb{A}, w_0)[p] \neq \emptyset$, then p defines a unique run $\mathbb{K}_0 a_0 \cdots \mathbb{K}_n a_n$ of $\mathrm{Slice}(K, \mathbb{A}, w_0)$ such that for all $0 \leq i \leq n$, $\mathrm{Paths}(\mathbb{K}_i) = \mathrm{Paths}(K, \mathbb{A}, w_0)(a_0 \dots a_{i-1}, a_i)$.*

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 $\mathrm{Slice}(\mathbb{A}(K, w_0))$ being deterministic, it follows that all action sequences $p \in A^*$ in (K, w_0) with labeling $\mathbb{A}$, i.e. $Zip(\mathbb{A}(K, w_0))[p] \neq \emptyset$, define an unique path in $\mathrm{Slice}(\mathbb{A}(K, w_0))$.

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 $\mathrm{Slice}(\mathbb{A}(K, w_0))$ is $\mathbb{K}_0$, then we need to prove that $\mathrm{Paths}(\mathbb{K}_0) = \mathrm{Paths}(\mathbb{A}(K))(\varnothing, a_0)$. By Definition 7.2.7, $\mathrm{Paths}(\mathbb{K}_0) = \mathrm{Paths}((\mathbb{K}[(K, \{w_0\}) \downarrow_{\mathbb{A}} a_0]))$. And, by definition open substructure induced by a, $\mathrm{Paths}(\mathbb{K}[(K, \{w_0\}) \downarrow_{\mathbb{A}} a_0]) = \{w_0 \dots w_n \mid (w_0, \varepsilon) \dots (w_{n-1}, \varepsilon)(w_n, a_0) \in \mathrm{Paths}(K)\}$. Thus, $\mathrm{Paths}(\mathbb{K}[(K, \{w_0\}) \downarrow_{\mathbb{A}} a_0]) = \mathrm{Paths}(\mathbb{A}(K))(\varnothing, a_0)$.

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 $\mathrm{Paths}(\mathbb{K}_i) = \mathrm{Paths}(\mathbb{A}(K))(a_0 \dots a_{i-1}, a_i)$ for all $0 \leq i < n$. We are only missing to prove that $\mathrm{Paths}(\mathbb{K}_n) = \mathrm{Paths}(\mathbb{A}(K))(a_0 \dots a_{n-1}, a_n)$. By IH, we know that $\mathrm{Paths}(\mathbb{K}_{n-1})$ and $\mathrm{Paths}(\mathbb{A}(\mathbb{K}))(a_0 \dots a_{n-2}, a_{n-1})$ have the same terminal states. Then, it follows that $\mathrm{Paths}(\mathbb{K}_n)$ and $\mathrm{Paths}(\mathbb{A}(\mathbb{K}))(a_0 \dots a_{n-1}, a_n)$ have the same initial states W_{in}. And, from an analogous reasoning from the base case, $\mathrm{Paths}(\mathbb{K}_n) = \mathrm{Paths}(\mathbb{K}[(K, W_{\mathrm{in}}) \downarrow_{\mathbb{A}} a_n]) = \mathrm{Paths}(\mathbb{A}(K))(a_0 \dots a_{n-1}, a_n)$ □

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 7.2.8. *Let $\mathcal{H} = (Q_h, \hat{q}, \gamma, \delta_h)$ be an hypernode automaton. The* intersection of $\mathcal{H}$ with the slicing of a pointed, action-labeled Kripke structure $(K, \mathbb{A}, w_0)$, $\mathrm{Slice}(K, \mathbb{A}, w_0) = (Q_s, \hat{Q}_s, F_s, \delta_s)$, *is the finite automaton* $\mathrm{Join}(\mathcal{H}, K, \mathbb{A}, w_0) = (Q, \hat{Q}, F, A, \delta)$ *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; with initial state* $\hat{Q} = \{(\mathbb{K}, \hat{q}) \in Q \mid \mathbb{K} \in \hat{Q}_s\} \cup \{(\mathbb{K}, \hat{\overline{q}}) \in Q \mid \mathbb{K} \in \hat{Q}_s\}$ *and final state* $F = \{(\mathbb{K}, \overline{q}) \mid (\mathbb{K}, \overline{q}) \in Q\}$*; and*
- *transition function* $\delta : Q \times A \to Q$*, 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 $\text{Join}(\mathcal{H}, K, \mathbb{A}, w_0)$ reads sequences of actions with run and accepting run defined as usual. The language of the automaton is empty, if there are no accepting runs.

Theorem 7.2.11. *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,* $\mathit{Zip}(K, \mathbb{A}, w_0) \in \mathcal{L}(\mathcal{H})$ *iff the language of the finite automaton* $\text{Join}(\mathcal{H}, K, \mathbb{A}, w_0)$ *is empty.*

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

$\mathit{Zip}(\mathbb{A}(K, w_0)) \notin \mathcal{L}(\mathcal{H})$ iff there exists a sequence of actions $p = a_0 \dots a_n$ that is in $\mathit{Zip}(\mathbb{A}(K, w_0))$, i.e. $\mathit{Zip}(\mathbb{A}(K, w_0))[p] \neq \emptyset$, and $\mathit{Zip}(\mathbb{A}(K, w_0))[p] \not\models \mathcal{H}[p]$. Wlog, we can assume that only the last slice does not satisfy the corresponding node in $\mathcal{H}$. Let $\mathcal{H}[p] = q_0 a_0 \dots q_n a_n$. Then, $\mathit{Zip}(\mathbb{A}(K, w_0))[p] \not\models \mathcal{H}[p]$ iff $(\star)$ for all $0 \leq j < n$, $\mathit{Zip}(\mathbb{A}(K, w_0))(a_0 \dots a_{j-1}, a_j) \models q_j$ while $\mathit{Zip}(\mathbb{A}(K, w_0))(a_0 \dots a_{n-1}, a_n) \not\models q_n$. By Lemma 7.2.10, $\mathit{Zip}(\mathbb{A}(K, w_0))[p] \neq \emptyset$ defines an unique path in $\text{Slice}(\mathbb{A}(K, w_0))$, $\mathbb{K}_0 a_0 \dots \mathbb{K}_n a_n$ that preserves the path slicing defined by $\mathbb{A}(K, w_0)$. Thus, from $(\star)$, definition of $\text{Join}(\mathcal{H}, \mathbb{A}(K, w_0))$ and Lemma 7.2.10, $(\mathbb{K}_0, q_0) a_0 \dots (\mathbb{K}_n, \overline{q_n}) a_n$ defines an accepting run in $\text{Join}(\mathcal{H}, \mathbb{A}(K, w_0))$. Note that $(\mathbb{K}_n, \overline{q_n})$ is in $\text{Join}(\mathcal{H}, \mathbb{A}(K, w_0))$ (and it is final) because $\mathit{Zip}(\mathbb{A}(K, w_0))(a_0 \dots a_{n-1}, a_n) \not\models q_n$. □

The following theorem puts all results from this section together.

Theorem 7.2.12. *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 7.2.8). Evaluating $\text{Join}(\mathcal{H}, K, \mathbb{A}, w_0)$ 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. □

As the most expensive computation of model checking a hypernode automaton is model-checking the hypernodes, then the running time for model checking hypernode automata is also dominated by a doubly exponential dependency on the number of propositional

variables. Additionally, the model-checking algorithm depends singly exponentially on both the size of the Kripke structure and the size of the hypernode automaton.

Corollary 7.2.13. *Let A be a set of actions and X a set of m propositional 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 $Zip(K, \mathbb{A}, w_0)) \in \mathcal{L}(\mathcal{H})$ 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 $\mathcal{H}$.*

7.3 Related Work

The first general approaches to reason about hyperproperties [CFHH19b] (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 [BFH+22a], and presented in Section 6.3, 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 [GMO21].

The first logic studied to express asynchronous hyperproperties was hyper μ-calculus [GMO21], called Hμ. It extends the linear-time μ-calculus [Var88] 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 [GMO21]. The undecidability of asynchronous hyperproperties verification techniques 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 [GMO21], 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.

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 [BFH$^+$22b], 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 Figure 7.7. 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.

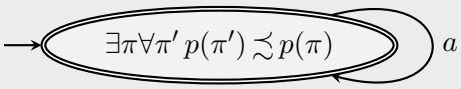


Figure 7.7: Hypernode automaton specifying that within each slice of a trace set induced by observing action a, there exists a trace in each slice that describes the progress of the propositional variable p in the slice.

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 (HyperLTL$_S$) and *context* HyperLTL (HyperLTL$_C$), both introduced in [BPS21], extend HyperLTL with new asynchronous operators; while, Asynchronous HyperLTL (A-HyperLTL) [BCB$^+$21] 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 [BPS22] the relative expressiveness of all the mentioned formalisms and proved that all of them are subsumed by Hμ [BPS22]. 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 [BPS22], 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^{2}LTL [BFFM23]. First-order quantifiers in Hyper^{2}LTL assign traces to trace variables (as in HyperLTL), while second-order quantifiers range over sets of traces. The full Hyper^{2}LTL (i.e., second-order quantifiers ranging over all possible trace sets) has a highly-undecidable model-checking problem [BFFM23]. To remedy this problem, they introduce Hyper^{2}LTL$_{\text{fp}}$ where sets are constrained to satisfy given minimality or maximality requirements. For this fragment, the authors present an algorithm to approximate solutions to the model-checking problem. 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^{2}LTL 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.

Finally, in the team semantics reinterpretations of LTL presented by Krebs et al. [KMVZ18], 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.

CHAPTER 8

Conclusion and Future Work

The work presented in this thesis was prompted by the following general research question:

Are there classes of hyperproperties suitable for the compositional design and verification of information-flow requirements?

From a high-level point of view, the work in this thesis falls into the study of the foundational aspects of languages to express information-flow constraints, with a focus on general languages (i.e., allowing user-defined security properties) that are independent of implementation details of the system to be verified. Orthogonal to this line of research is the rich landscape of formalisms and techniques addressing different facets of security verification and enforcement (e.g., language-based approaches like type systems and taint-analysis [SM03]) and tailored for specific systems or security properties.

Our first step in this research journey was to question how we could lift the interface theory approach to support hyperproperties in a way that is also useful to help design secure systems. At this point, we observed that an interface theory for the trace-based view on information-flow (i.e., having assumptions and guarantees in our interfaces defined in terms of the set of trace sets it accepts) would require the designer to fix early on assumptions on how the system is observed, which may not be possible at early stages of the design process or, at all, for heterogeneous systems. Following this insight, we decided to focus on the design of the structural aspect of information-flow. We devised the information-flow interfaces, presented in Chapter 3, which support a step-wise design on information flow requirements across the different components of the system to guarantee that the whole system satisfies the intended security policy. Information-flow interfaces also support the specification of dynamic information-flow policies defined as transition systems with states annotated with stateless information-flow interfaces. In Figure 8.1, we illustrate an example of a stateful specification.

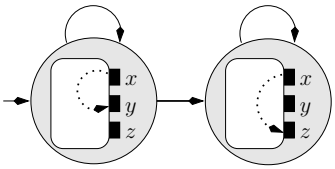


Figure 8.1: Example of a stateful information-flow specification: information in x do not flow to y until information in x do not flow to z.

The natural next research step was to go back to the semantic considerations and investigate whether we could use formalisms for hyperproperties in the current state-of-the-art to instantiate the no-flow requirements in our interface theory. We started by considering simple stateful policies like the one illustrated in Figure 8.1, where the specification changes during the system execution. As we are working with hyperproperties, we need to take into account that such a specification state change may not coincide in all system executions, i.e., the change is asynchronous, and it may happen at arbitrary time points in each execution. The dominant formalism to express hyperproperties, HyperLTL, adopts a synchronous analysis of traces, which is incompatible with such asynchronous state changes. This limitation stems from HyperLTL formulas starting with trace quantifiers, with the time quantification being implicit in its inner LTL formula. Having identified this limitation, our goal was to formalize it and highlight the importance of studying other possible combinations between time and trace quantification to support asynchronous hyperproperties.

We introduced hypertrace logic (c.f., Chapter 6), a two-sorted first-order logic, and an operator to slice trace sets to specify the two-state information-flow policy in Figure 8.1. Hypertrace formulas have explicit quantification over both traces and time, allowing us to identify and study fragments with different relative ordering between the two types of quantification. Using hypertrace logic, we lifted ideas and techniques from the model theory of first-order logic to prove expressivity results for semantic variants of the policy in Figure 8.1. In particular, we proved that, in general, we cannot express asynchronous specification state changes with HyperLTL formulas.

When we look at stateless information-flow interfaces, we face various sources of asynchronicity between observations of system executions, like different scheduling decisions, the granularity of observations or active waiting for external input. Therefore, for our stateless specifications, we are interested in a language that supports as much asynchronicity between executions as possible without compromising the quality of its security analysis outcome. Inspired by our stateful information-flow interfaces, we proposed hypernode automata (c.f., Chapter 7), which mixes automata with logic, where the automata structure synchronizes transitions between states labeled with the fully asynchronous logic. This novel approach in the literature opens new possible research directions to deal with the highly undecidable problem of model-checking asynchronous hyperproperties.

In the end, to answer the question we introduced at the beginning of this chapter, we collected the following insights addressed in this thesis:

i) before we specify an information-flow requirement as a hyperproperty, it may be useful to first reason about it from the structural point of view, enabling a divide-and-conquer approach by first organizing the verification process into smaller tractable tasks;

ii) we should focus our study on asynchronous hyperproperties because, otherwise, we may compromise on the ability to specify dynamic security policies and security properties that are independent of how we observe the system;

iii) hybrid specification languages for hyperproperties, for example, combining automata with logic, may turn intractable problems into tractable ones.

Future Work. The work we present here opened new research directions with many interesting problems left for future work. For the information-flow interfaces, we would like to investigate further how to interpret its requirements with trace-based semantics. Our first step in this direction was the introduction of hypernode automata, which seems a good candidate for interpreting stateful information-flow interfaces. In future work, we want to study and formalize the connection between these two formalisms. A different related line of research is to study if the results on compositionality between trace-based security policies [Man00, Man02, MSS11] can be lifted to the be a trace-based semantics for our interface theory. Related to asynchronous hyperproperties, we would like to further investigate expressiveness-related questions in this area. In particular, we would like to work on finding a classification within asynchronous hyperproperties capturing the nature of their asynchronicity (for example, whether the misalignment between traces is bounded or can be solved by slicing the trace appropriately). With such classification, we could design and identify adequate specification languages for different systems and observation settings. Finally, for the hypernode automata, we would like to extend its expressiveness, for example, by looking into different acceptance conditions or making the hypernode logic more expressive.

List of Figures

List of Tables

List of Algorithms

Bibliography

[AF12] Thomas H Austin and Cormac Flanagan. Multiple facets for dynamic information flow. In *Proceedings of ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 165–178, 2012.

[AFRP15] Mohammad Al Faruque, Francesco Regazzoni, and Miroslav Pajic. Design methodologies for securing cyber-physical systems. In *Proceedings of the 10th International Conference on Hardware/Software Codesign and System Synthesis*, pages 30–36. IEEE Press, 2015.

[AHKV98] Rajeev Alur, Thomas A Henzinger, Orna Kupferman, and Moshe Y Vardi. Alternating refinement relations. In *CONCUR'98 Concurrency Theory*, volume 1466 of *LNCS*, pages 163–178. Springer, 1998.

[AL93] Martín Abadi and Leslie Lamport. Composing specifications. *ACM Trans. Program. Lang. Syst.*, 15(1):73—-132, 1993.

[ALW89] Martín Abadi, Leslie Lamport, and Pierre Wolper. Realizable and unrealizable specifications of reactive systems. In Giorgio Ausiello, Mariangiola Dezani-Ciancaglini, and Simonetta Ronchi Della Rocca, editors, *Automata, Languages and Programming*, pages 1–17. Springer, 1989.

[BCB+21] 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.

[BCF+08] Albert Benveniste, Benoît Caillaud, Alberto Ferrari, Leonardo Mangeruca, Roberto Passerone, and Christos Sofronis. Multiple viewpoint contract-based specification and design. In *Proc. of FMCO 2007: the 6th International Symposium on Formal Methods for Components and Objects*, volume 5382 of *Lecture Notes in Computer Science*, pages 200–225. Springer, 2008.

[BCN+18] Albert Benveniste, Benoît Caillaud, Dejan Nickovic, Roberto Passerone, Jean-Baptiste Raclet, Philipp Reinkemeier, Alberto L. Sangiovanni-Vincentelli, Werner Damm, Thomas A. Henzinger, and Kim G. Larsen.

Contracts for system design. *Foundations and Trends in Electronic Design Automation*, 12(2-3):124–400, 2018.

[BDH+12] Sebastian S. Bauer, Alexandre David, Rolf Hennicker, Kim Guldstrand Larsen, Axel Legay, Ulrik Nyman, and Andrzej Wasowski. Moving from specifications to contracts in component-based design. In *Proc. of FASE 2012*, volume 7212 of *Lecture Notes in Computer Science*, pages 43–58. Springer, 2012.

[Bel05] David Elliott Bell. Looking back at the bell-la padula model. In *21st Annual Computer Security Applications Conference (ACSAC'05)*, pages 15–pp. IEEE, 2005.

[BF23] Raven Beutner and Bernd Finkbeiner. AutoHyper: Explicit-State Model Checking for HyperLTL. In Sriram Sankaranarayanan and Natasha Sharygina, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 145–163. Springer Nature Switzerland, 2023.

[BFFM23] Raven Beutner, Bernd Finkbeiner, Hadar Frenkel, and Niklas Metzger. Second-order hyperproperties. In Constantin Enea and Akash Lal, editors, *Computer Aided Verification (CAV)*, pages 309–332. Springer Nature Switzerland, 2023.

[BFH+22a] 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, *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 1–19. Springer International Publishing, 2022.

[BFH+22b] Ezio Bartocci, Thomas Ferrère, Thomas A. Henzinger, Dejan Nickovic, and Ana Oliveira da Costa. Information-flow interfaces. In *International Conference on Fundamental Approaches to Software Engineering (FASE)*, volume 13241 of *LNCS*, pages 3–22, 2022.

[BFH+24] Ezio Bartocci, Thomas Ferrère, Thomas A. Henzinger, Dejan Nickovic, and Ana Oliveira da Costa. Information-flow interfaces. *Formal Methods in System Design*, 2024.

[BHNOdC23] Ezio Bartocci, Thomas A. Henzinger, Dejan Nickovic, and Ana Oliveira da Costa. Hypernode Automata. In Guillermo A. Pérez and Jean-François Raskin, editors, *International Conference on Concurrency Theory (CONCUR)*, volume 279 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 21:1–21:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023.

[BHNOdC24] Ezio Bartocci, Thomas A. Henzinger, Dejan Nickovic, and Ana Oliveira da Costa. Information-flow interfaces and security lattices. Festschrift' Engineering Safe and Trustworthy Cyber Physical Systems – Essays Dedicated to Werner Damm on the Occasion of His 71st Birthday, 2024. Accepted for publication.

[Bib77] Kenneth J Biba. Integrity considerations for secure computer systems. Technical report, MITRE CORP BEDFORD MA, 1977.

[BL75] David E Bell and Leonard J LaPadula. Computer security model: Unified exposition and multics interpretation. *MITRE Corp., Bedford, MA, Tech. Rep. ESD-TR-75-306, June*, 1975.

[BMP15] Laura Bozzelli, Bastien Maubert, and Sophie Pinchinat. Unifying hyper and epistemic temporal logics. In *Foundations of Software Science and Computation Structures (FoSSaCS)*, volume 9034 of *LNCS*, pages 167–182, 2015.

[Bon16] Bonakdarpour, Borzoo and Finkbeiner, Bernd. Runtime Verification for HyperLTL. In Falcone, Yliès and Sánchez, César, editor, *Runtime Verification*, pages 41–45. Springer International Publishing, 2016.

[BPS21] Laura Bozzelli, Adriano Peron, and César Sánchez. Asynchronous extensions of hyperltl. In *2021 36th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–13, 2021.

[BPS22] Laura Bozzelli, Adriano Peron, and Cesar Sanchez. Expressiveness and decidability of temporal logics for asynchronous hyperproperties. In *Proc. of 33nd International Conference on Concurrency Theory (CONCUR 2022) (To appear)*, 2022.

[BRLEK17] Ryad Benadjila, Mathieu Renard, José Lopes-Esteves, and Chaouki Kasmi. One car, two frames: attacks on hitag-2 remote keyless entry systems revisited. In *11th USENIX Workshop on Offensive Technologies*, 2017.

[CDAHM02] Arindam Chakrabarti, Luca De Alfaro, Thomas A. Henzinger, and Freddy YC Mang. Synchronous and bidirectional component interfaces. In *International Conference on Computer Aided Verification*, pages 414–427. Springer, 2002.

[CdAHS03] Arindam Chakrabarti, Luca de Alfaro, Thomas A. Henzinger, and Mariëlle Stoelinga. Resource interfaces. In *Embedded Software*, volume 2855 of *LNCS*, pages 117–133, 2003.

[CFHH19a] Norine Coenen, Bernd Finkbeiner, Christopher Hahn, and Jana Hofmann. The hierarchy of hyperlogics. In *2019 34th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–13. IEEE, 2019.

[CFHH19b] Norine Coenen, Bernd Finkbeiner, Christopher Hahn, and Jana Hofmann. The hierarchy of hyperlogics. In *Proc. of LICS: the 34th Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 1–13. IEEE, 2019.

[CFK+14] Michael R. Clarkson, Bernd Finkbeiner, Masoud Koleini, Kristopher K. Micinski, Markus N. Rabe, and César Sánchez. Temporal logics for hyperproperties. In *Principles of Security and Trust (POST)*, volume 8414 of *LNCS*, pages 265–284, 2014.

[CS10] Michael R. Clarkson and Fred B. Schneider. Hyperproperties. *Journal of Computer Security*, 18(6):1157–1210, 2010.

[dA03] Luca de Alfaro. *Game Models for Open Systems*, pages 269–289. Springer Berlin Heidelberg, 2003.

[dAH01a] Luca de Alfaro and Thomas A. Henzinger. Interface automata. In *European Software Engineering Conference/Foundations on Software Engineering (ESEC/FSE)*, page 109–120, 2001.

[dAH01b] Luca de Alfaro and Thomas A. Henzinger. Interface theories for component-based design. In *Embedded Software*, volume 2211 of *LNCS*, pages 148–165, 2001.

[dAH05] Luca de Alfaro and Thomas A. Henzinger. Interface-based design. In *Engineering Theories of Software Intensive Systems*, volume 195 of *NATO Science Series (Series II: Mathematics, Physics and Chemistry)*, pages 83–104, 2005.

[dAHS02] Luca de Alfaro, Thomas A. Henzinger, and Mariëlle Stoelinga. Timed interfaces. In *Embedded Software*, volume 2491 of *LNCS*, pages 108–122, 2002.

[Den76] Dorothy E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, may 1976.

[DFK+12] Rayna Dimitrova, Bernd Finkbeiner, Máté Kovács, Markus N Rabe, and Helmut Seidl. Model checking information flow in reactive systems. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 169–185. Springer, 2012.

[DGDNP12] Willem De Groef, Dominique Devriese, Nick Nikiforakis, and Frank Piessens. Flowfox: A web browser with flexible and precise information flow control. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS)*, page 748–759, 2012.

[Dil89] David L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits*. The MIT Press, 09 1989.

[DLL+10] Alexandre David, Kim G. Larsen, Axel Legay, Ulrik Nyman, and Andrzej Wasowski. Timed I/O automata: a complete specification theory for real-time systems. In *Proceedings of the 13th ACM International Conference on Hybrid Systems: Computation and Control (HSCC)*, pages 91–100, 2010.

[FHS17] Bernd Finkbeiner, Christopher Hahn, and Marvin Stenger. EAHyper: Satisfiability, Implication, and Equivalence Checking of Hyperproperties", booktitle="Computer Aided Verification. pages 564–570. Springer International Publishing, 2017.

[Flo67] Robert W. Floyd. Assigning meanings to programs. *Proceedings of Symposium on Applied Mathematics*, 19:19–32, 1967.

[FM11] Riccardo Focardi and Matteo Maffei. Types for security protocols. *Formal Models and Techniques for Analyzing Security Protocols*, 5:143–181, 2011.

[FMHV95] Ronald Fagin, Yoram Moses, Joseph Y Halpern, and Moshe Y Vardi. *Reasoning about knowledge*. MIT Press, 1995.

[FR14] Bernd Finkbeiner and Markus N Rabe. The linear-hyper-branching spectrum of temporal logics. *it Inf. Technol.*, 56(6):273–279, 2014.

[FRS15] Bernd Finkbeiner, Markus N. Rabe, and César Sánchez. Algorithms for Model Checking HyperLTL and HyperCTL*. In Daniel Kroening and Corina S. Păsăreanu, editors, *Computer Aided Verification*, pages 30–48. Springer International Publishing, 2015.

[FZ17] Bernd Finkbeiner and Martin Zimmermann. The first-order logic of hyperproperties. In *34th Symposium on Theoretical Aspects of Computer Science*, 2017.

[GHM13] Jürgen Graf, Martin Hecker, and Martin Mohr. Using JOANA for information flow control in Java programs - a practical guide. In *Software Engineering 2013 - Workshopband*, volume P-215 of *LNI*, pages 123–138, 2013.

[GM82] J. A. Goguen and J. Meseguer. Security policies and security models. In *1982 IEEE Symposium on Security and Privacy*, pages 11–11, April 1982.

[GMO21] Jens Oliver Gutsfeld, Markus Müller-Olm, and Christoph Ohrem. Automata and fixpoints for asynchronous hyperproperties. *Proc. ACM Program. Lang.*, 5(POPL):1–29, 2021.

[GPSS80] Dov Gabbay, Amir Pnueli, Saharon Shelah, and Jonathan Stavi. On the temporal analysis of fairness. In *Proceedings of the 7th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 163–173, 1980.

[GV13] Erich Grädel and Jouko Väänänen. Dependence and independence. *Studia Logica*, 101(2):399–410, 2013.

[Hel18] Edward Helmore. Uber shuts down self-driving operation in arizona after fatal crash. `https://www.theguardian.com/technology/2018/may/23/uber-shuts-down-self-driving-operation-in-arizona-two-months-aft` 2018. Accessed: 2019-10-10.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[Hoa80] Charles Antony Richard Hoare. A model for communicating sequential process. 1980.

[HS09] Christian Hammer and Gregor Snelting. Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. *International Journal of Information Security*, 8(6):399–422, 2009.

[HSB21] Tzu-Han Hsu, César Sánchez, and Borzoo Bonakdarpour. Bounded Model Checking for Hyperproperties. In Jan Friso Groote and Kim Guldstrand Larsen, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 94–112. Springer International Publishing, 2021.

[HWS06] M. Huisman, P. Worah, and K. Sunesen. A temporal logic characterisation of observational determinism. In *19th IEEE Computer Security Foundations Workshop (CSFW'06)*, pages 13 pp.–3, 2006.

[HYH+04] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, and Sy-Yen Kuo. Securing web application code by static analysis and runtime protection. In *Proceedings of the 13th International Conference on World Wide Web*, WWW '04, page 40–52, 2004.

[IBSS22] Inigo Incer, Albert Benveniste, Alberto L. Sangiovanni-Vincentelli, and Sanjit A. Seshia. Hypercontracts. In *Proc. of NFM 2022: the 14th International Symposium*, volume 13260 of *LNCS*, pages 674–692, 2022.

[JHC15] Cliff B Jones, Ian J Hayes, and Robert J Colvin. Balancing expressiveness in formal approaches to concurrency. *Formal Aspects of Computing*, 27:475–497, 2015.

[Jon81] Cliff B Jones. *Development methods for computer programs including a notion of interference*. Oxford University Computing Laboratory, 1981.

[Kam68] Hans Kamp. *Tense Logic and the Theory of Linear Order*. PhD thesis, UCLA, 1968.

[KCM+22] Elisavet Kozyri, Stephen Chong, Andrew C Myers, et al. Expressing information flow properties. *Foundations and Trends® in Privacy and Security*, 3(1):1–102, 2022.

[KMVZ18] A Krebs, A Meier, J Virtema, and M Zimmermann. Team semantics for the specification and verification of hyperproperties. *Leibniz International Proceedings in Informatics, LIPIcs*, 117, 2018.

[KS05] Antonín Kučera and Jan Strejček. The stuttering principle revisited. *Acta Informatica*, 41(7-8):415–434, 2005.

[LD10a] M. Lee and P. R. D'Argenio. A refinement based notion of non-interference for interface automata: Compositionality, decidability and synthesis. In *International Conference of the Chilean Computer Science Society*, pages 280–289, Nov 2010.

[LD10b] Matias Lee and Pedro R. D'Argenio. Describing secure interfaces with interface automata. *Electronic Notes in Theoretical Computer Science*, 264(1):107–123, 2010.

[LD10c] Matias Lee and Pedro R. D'Argenio. A refinement based notion of non-interference for interface automata: Compositionality, decidability and synthesis. In *2010 XXIX International Conference of the Chilean Computer Science Society*, pages 280–289, 2010.

[Lee08] Edward A. Lee. Cyber physical systems: Design challenges. In *2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*, pages 363–369, 2008.

[LNW07] Kim G. Larsen, Ulrik Nyman, and Andrzej Wasowski. Modal I/O automata for interface and product line theories. In *European Symposium on Programming*, pages 64–79. Springer, 2007.

[LSS05] Kerstin Lemke, Ahmad-Reza Sadeghi, and Christian Stüble. An open approach for designing secure electronic immobilizers. In *Proc. of ISPEC 2005*, volume 3439 of *LNCS*, pages 230–242, 2005.

[LT87] Nancy A. Lynch and Mark R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '87, page 137–151, New York, NY, USA, 1987. Association for Computing Machinery.

[LV12] Gerald Lüttgen and Walter Vogler. Modal interface automata. In *IFIP International Conference on Theoretical Computer Science*, pages 265–279. Springer, 2012.

[Man00] Heiko Mantel. Possibilistic definitions of security-an assembly kit. In *Proceedings 13th IEEE Computer Security Foundations Workshop. CSFW-13*, pages 185–199, 2000.

[Man02] Heiko Mantel. On the composition of secure systems. In *IEEE Symposium on Security and Privacy*, pages 88–101, 2002.

[McC87] Daryl McCullough. Specifications for multi-level security and a hook-up. In *1987 IEEE Symposium on Security and Privacy*, pages 161–161, 1987.

[McL96] John McLean. A general theory of composition for a class of "possibilistic" properties. *IEEE Transactions on Software Engineering*, 22(1):53–67, 1996.

[Men09] Elliott Mendelson. *Introduction to mathematical logic*. CRC press, 2009.

[Mey92] Bertrand Meyer. Applying 'design by contract'. *Computer*, 25(10):40–51, 1992.

[Mey09] Bertrand Meyer. *Touch of class: Learning to program well with Object Technology and Design by Contract*. Springer, 2009.

[MHGG19] Marcus Mikulcak, Paula Herber, Thomas Göthel, and Sabine Glesner. Information flow analysis of combined simulink/stateflow models. *Information Technology And Control*, 48(2):299–315, 2019.

[MSS11] Heiko Mantel, David Sands, and Henning Sudbrock. Assumptions and guarantees for compositional noninterference. In *IEEE Computer Security Foundations Symposium (CSF)*, pages 218–232, 2011.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *Annual Symposium on Foundations of Computer Science (FOCS)*, pages 46–57, 1977.

[PW97] Doron Peled and Thomas Wilke. Stutter-invariant temporal properties are expressible without the next-time operator. *Information Processing Letters*, 63(5):243–246, 1997.

[Rab14] Alexander Rabinovich. A Proof of Kamp's theorem. *Logical Methods in Computer Science*, Volume 10, Issue 1, February 2014.

[RKG+19] Denise Ratasich, Faiq Khalid, Florian Geissler, Radu Grosu, Muhammad Shafique, and Ezio Bartocci. A roadmap toward the resilient internet of things for cyber-physical systems. *IEEE Access*, 7:13260–13283, 2019.

[San93] R.S. Sandhu. Lattice-based access control models. *Computer*, 26(11):9–19, 1993.

[SDK19] Florian Sommer, Jürgen Dürrwang, and Reiner Kriesten. Survey and classification of automotive security attacks. *Information*, 10(4), 2019.

[SM03] Andrei Sabelfeld and Andrew C Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.

[SS05] A. Sabelfeld and D. Sands. Dimensions and principles of declassification. In *18th IEEE Computer Security Foundations Workshop (CSFW'05)*, pages 255–269, 2005.

[SVDP] Alberto Sangiovanni-Vincentelli, Werner Damm, and Roberto Passerone. Taming dr. frankenstein: Contract-based design for cyber-physical systems.

[Ter08] Tachio Terauchi. A type system for observational determinism. In *2008 21st IEEE Computer Security Foundations Symposium*, pages 287–300, 2008.

[Tho92] Wolfgang Thomas. Infinite trees and automaton- definable relations over ω-words. *Theoretical Computer Science*, 103(1):143–159, 1992.

[TLHL11] Stavros Tripakis, Ben Lickly, Thomas A. Henzinger, and Edward A. Lee. A theory of synchronous relational interfaces. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 33(4):14, 2011.

[Tri16] Stavros Tripakis. Compositionality in the science of system design. *Proceedings of the IEEE*, 104(5):960–972, 2016.

[Vää07] Jouko Väänänen. *Dependence Logic: A New Approach to Independence Friendly Logic*. London Mathematical Society Student Texts. Cambridge University Press, 2007.

[Var88] M. Y. Vardi. A temporal fixpoint calculus. In *Symposium on Principles of Programming Languages (POPL)*, page 250–259. Association for Computing Machinery (ACM), 1988.

[Wol96] Elizabeth Susan Wolf. *Hierarchical models of synchronous circuits for formal verification and substitution*. Stanford University, 1996.

[ZM03] 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.