FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

From HyperLTL modelcheckers to Alloy: Enabling Hyperproperty Verification

Miguel Lourenço Pregal de Mesquita Montes

Mestrado em Engenharia Informática e Computação

Supervisor: Prof. Nuno Macedo

Supervisor: Prof. Hugo Pacheco

Supervisor: Prof. Jácome Cunha

July 31, 2026

From HyperLTL modelcheckers to Alloy: Enabling Hyperproperty Verification

Miguel Lourenço Pregal de Mesquita Montes

Mestrado em Engenharia Informática e Computação

Approved in oral examination by the committee:

President: Prof. João Fernando Ferreira
Examiner: Prof. José Proença
Member: Prof. Nuno Macedo

July 31, 2026

Resumo

As hiperpropriedades são propriedades que relacionam múltiplos traços de execução de um sistema e são úteis para especificar requisitos de segurança, privacidade e de fluxo de informação. Embora os verificadores de modelos para HyperLTL consigam verificar este tipo de propriedades, operam normalmente sobre representações de baixo nível e produzem resultados difíceis de interpretar. Isto cria uma lacuna entre linguagens de especificação de alto nível, como Alloy, e as ferramentas de verificação existentes.

Esta dissertação explora uma arquitetura que permite a verificação de hiperpropriedades a partir de modelos de alto nível e de propriedades HyperLTL. Em particular, o trabalho foca-se no ecossistema Alloy, utilizando especificações em Alloy para modelar sistemas e expressar propriedades HyperLTL e traduzindo-as para os formatos exigidos pelos verificadores de modelos de HyperLTL existentes, nomeadamente o AutoHyper e o HyperQube. A pipeline proposta integra o Alloy, o Electrod, o HyperSMV, verificadores de backend, o Inst2SMV e uma extensão do visualizador de Alloy capaz de exibir múltiplos traços. O objetivo não é apenas integrar verificadores externos, mas também reconstruir as suas testemunhas e contraexemplos ao nível de abstração do Alloy.

A abordagem é avaliada por meio de casos de estudo representativos. Os resultados mostram que a pipeline consegue fornecer feedback visual significativo quando o resultado produzido pelo backend contém informação suficiente para a reconstrução. No entanto, a avaliação também revela limitações relacionadas à escalabilidade, aos formatos específicos de cada backend e à robustez da reconstrução de contraexemplos. No geral, este trabalho demonstra que a integração de Alloy com verificadores de modelos para HyperLTL é viável e útil, ao mesmo tempo que identifica os desafios que devem ser ultrapassados para alcançar uma pipeline de verificação mais geral e fiável.

Palavras-chave: Verificação Formal • Verificação de Modelos • Hiperpropriedades • HyperLTL • Alloy • Visualização de Contraexemplos

Abstract

Hyperproperties are properties that relate multiple execution traces of a system and are useful for specifying security, privacy, and information-flow requirements. Although HyperLTL model checkers can verify such properties, they typically operate on low-level representations and produce outputs that are difficult to interpret. This creates a gap between high-level specification languages, such as Alloy, and existing backend verification tools.

This dissertation explores an architecture for enabling hyperproperty verification of high-level models and HyperLTL properties. In particular, the work focuses on the Alloy ecosystem, using Alloy specifications to model systems and express HyperLTL properties, and translating them into the formats required by existing HyperLTL model checkers, namely AutoHyper and HyperQube. The proposed pipeline integrates Alloy, Electrod, HyperSMV, backend solvers, Inst2SMV, and an extended Alloy visualizer capable of displaying multiple traces. The goal is not only to integrate external solvers but also to reconstruct their witnesses and counterexamples at the Alloy level.

The approach is evaluated using representative case studies. The results show that the pipeline can provide meaningful visual feedback when the backend output contains sufficient information for reconstruction. However, the evaluation also reveals limitations related to scalability, backend-specific formats, and the robustness of counterexample reconstruction. Overall, this work demonstrates that integrating Alloy with HyperLTL model checkers is feasible and useful, while also identifying the challenges that must be addressed to achieve a more general and reliable verification pipeline.

Keywords: Formal Verification • Model Checking • Hyperproperties • HyperLTL • Alloy • Counterexample Visualization

The United Nations Sustainable Development Goals

The United Nations Sustainable Development Goalss (SDGs) provide a global framework for achieving a better and more sustainable future for all. It includes 17 goals (https://sdgs.un.org/goals) to address the world's most pressing challenges, including poverty, inequality, climate change, environmental degradation, peace, and justice.

This dissertation contributes to the field of formal methods and software verification. Its main goal is not to address a Sustainable Development Goal directly, but to improve the methods and tools available for specifying, verifying, and interpreting the behavior of software systems. In particular, the work focuses on hyperproperties, which are relevant for reasoning about security, privacy, information flow, and other relational properties of systems. By supporting the use of high-level specifications and improving the interpretation of verification results, the proposed approach can contribute indirectly to the development of more reliable, transparent, and trustworthy computational systems.

The specific Sustainable Development Goals mentioned have the following names:

SDG 4 Ensure inclusive and equitable quality education and promote lifelong learning opportunities for all

SDG 9 Build resilient infrastructure, promote inclusive and sustainable industrialization and foster innovation

SDG 16 Promote peaceful and inclusive societies for sustainable development, provide access to justice for all and build effective, accountable and inclusive institutions at all levels

SDG	Target	Contribution	Performance Indicators and Metrics
4	4.4	The dissertation contributes to advanced technical knowledge in formal methods, particularly in Alloy, HyperLTL, and hyperproperty verification.	Explanation of the pipeline; examples written in Alloy; discussion of intermediate verification formats.
9	9.5	The work contributes to research and innovation in software verification by integrating high-level Alloy specifications with existing HyperLTL model checkers.	Implemented pipeline components; integration with AutoHyper and HyperQube; evaluation using representative case studies.
16	16.6	The work is indirectly related to trustworthy software systems, since hyperproperties can express security, privacy, and information-flow requirements.	Discussion of hyperproperty verification; interpretation of counterexamples; visual feedback for relational behaviors.

Acknowledgements

I would like to start by thanking my coordinators, without whom this thesis wouldn't have been possible. Especially Professors Nuno and Hugo, not only by introducing me to the theme, but also by being responsible for parts of the developed solution and helping and guiding me to the end line. But also Professor Jácome, whom I may not have gotten in touch with, but decided to help me with the bureaucratic side of a dissertation. I would also like to extend my gratitude to all my previous teachers and professors, whose teachings shaped what I know and how I think. I also owe thanks to all my course colleagues, mainly from CJ, with whom I shared many good memories and who helped me throughout our academic journey. Among them, a special thanks to Lora and Mariana, who were there from day 1 at FEUP and stayed until the end. They helped me a lot, either working or escaping the work. Mentioning Lora, I have to thank him and Diego because at some points I spent more time with them than at home, and I am extremely grateful for all they've done. In the wave of thanking my friends, from Os Seis, Bea, Maria, Martim, Mário, and Paixão, or the ones I know since a kid, João and Mauro, or from the other group that lacks a constant name, Onha, João, Lora, Pereira, Pia, Manel, Monteiro, Carol, Bid, Sima, Tiago, Fel, and Toni, or, from OMI, Fernando and Sargo, and for all other friends I had throughout my life, a massive thanks to you all because I value a lot my time with you, and it's always what I look for the most. I know if I work, I'll be able to enjoy my time with you. To finalize, I would like to thank my family. To my cousins, Filipe, Francisco, and Margarida, thanks for all the fun times growing up. To my aunts, uncles, grandparents, and family friends, you taught me a lot. To my siblings, Nuno and Sofia, because I spend all my time with you, and you are my day to day entertainment. And, finally, to my parents because without them I wouldn't be here. You supported my life, my education, my leisure. You encouraged me to be where I am and always try to keep an eye on me to check if I'm going in the right direction. To everyone I mentioned and everyone I forgot, thank you!

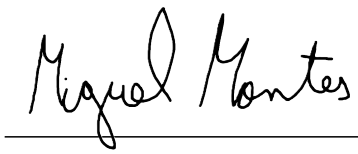
Miguel Montes

Declaration of honour

I declare that this dissertation is my own work and has not previously been submitted or assessed at this or any other institution. References to other authors (statements, ideas, thoughts), as well as the use of published works, strictly comply with the rules of attribution and are duly identified in the text and in the bibliographic references, in accordance with the applicable referencing standards. I am aware that the practice of plagiarism or self-plagiarism constitutes an academic offence, as established in the U.Porto Code of Ethics.

I also declare that I have used Artificial Intelligence tools to complete part(s) of this dissertation, and that this use and its results are duly noted and have been carefully checked for errors, inaccuracies, unfounded attributions or other forms of bias in content and arguments, as well as determining authorship.

Miguel Montes



"And these other earths, [...] they all exist at the same time, giving us endless alternatives to what we have here."

Martin Stein

Contents

List of Figures

List of Tables

List of Acronyms

GUI Graphical User Interface
SDG Sustainable Development Goals
LTL Linear Temporal Logic
PRISMA Preferred Reporting Items for Systematic Reviews and Meta-Analyses
RV Runtime Verification
HHL Hyper Hoare Logic
BDD Binary Decision Diagram
CTL Computation Tree Logic
BMC Bounded Model Checking
IC3 Incremental Construction of Inductive Clauses for Indubitable Correctness
MTL Metric Temporal Logic
SoC System-on-Chip
PTCTL Parametric Timed Computation Tree Logic
ATL Alternating-Time Temporal Logic
LDLf Linear Dynamic Logic over Finite Traces
STL Signal Temporal Logic
TSL Temporal Stream Logic
CPS Cyber-Physical System
QBF Quantified Boolean Formula
CHC Constrained Horn Clause
CFG control-flow graph
TLA Temporal Logic of Actions
MDP Markov Decision Process
MIT Massachusetts Institute of Technology
SMV symbolic model verifier
XML eXtensible Markup Language
LLM Large Language Model

Chapter 1

Introduction

Good software development increasingly relies on meeting security and privacy requirements. Modern software systems manage large amounts of sensitive information and are often distributed, concurrent, and highly interconnected, making them more susceptible to vulnerabilities, unintended information flow, and privacy violations. With the emergence of more complex systems and growing security concerns, relying on testing alone to satisfy these requirements is insufficient. Therefore, formal verification techniques, such as model checking, play an important role in the early validation of system designs and specifications by facilitating the identification of security and privacy issues at the conceptual modeling stage. Although originally developed to verify safety properties, these techniques are also widely applied to security and privacy policies.

1.1 Context

Model checking is a formal verification technique that checks whether a given property holds for a given trace of the system. A trace is a sequence of states that a system passes through during its execution. Alloy [1] is an example of a successful approach to formal software design. It is an abstract, flexible, open-source specification language that provides a modeling tool based on first-order logic, in which these properties can be expressed. It is accompanied by automated analysis procedures from Alloy Analyzer, which are traditionally based on bounded analysis rather than full model checking. The most recent version of Alloy (Alloy 6) already supports the verification of temporal properties using model-checking techniques and has been used in real-world systems [2]. However, it does not support verifying a more complex class of properties, called *hyperproperties* [3], that can cover aspects such as security and concurrency and extend traditional temporal properties by involving reasoning over multiple system traces. A typical example is noninterference, which requires that changes in secret input do not affect publicly observable outputs (see Section 2.2.1.4 for a detailed description).

1.2 Motivation

Trace properties are commonly expressed using temporal logics such as Linear Temporal Logic (LTL) and Computation Tree Logic (CTL). While LTL is interpreted over individual execution traces and CTL extends reasoning to branching-time systems through path quantification, both are limited in their ability to express hyperproperties, which require relating multiple execution traces simultaneously. Although general-purpose logics for specifying hyperproperties and associated model-checking algorithms have been proposed, such as HyperLTL [4], extending LTL with explicit trace quantification, allowing the specification of a wide range of security and privacy properties, the area remains complex and actively evolving. Existing implementations, such as AutoHyper and HyperQube, are mostly at the prototype level, operate on low-level system descriptions, and produce outputs that are not human-friendly to interpret. Since there is no high-level language, such as Alloy, that allows users to analyze the output of low-level algorithms, there is a need to parse this backend information and bring it to a higher level. Previous work [5] has begun to address this issue, but it remains in the early stages of development. It attempts to connect high-level specifications (through Alloy) to the lower-level verification of hyperproperties via model-checking backends for HyperLTL, serving as an intermediate step towards the solvers. This way, users can define hyperproperties more abstractly over more expressive models while still using existing low-level HyperLTL model-checking techniques. It is still necessary to interpret the low-level outcomes of the HyperLTL model checkers and return them to the high-level tool for presentation as readable, usable information. As an alternative, if existing model checkers for HyperLTL are too opaque, this may motivate reimplementing parts of the verification pipeline more transparently, enabling developers to better understand and interpret the results and reducing reliance on existing low-level HyperLTL model checkers. There is also a lack of examples of hyperproperties suitable for testing these solutions. To illustrate the challenge of interpreting low-level verification results, consider a user who specifies a noninterference property over an Alloy model. Existing work can translate the model towards HyperLTL model checkers and determine whether the property holds. However, the verification process ends with the low-level output produced by the solver, leaving the user without a way to reconstruct witnesses or counterexamples as Alloy instances. As a result, although verification is possible, interpreting and understanding the results remains difficult.

1.3 Problem

One of the main challenges is the gap that separates high-level specification and low-level verification of hyperproperties. At a higher level of specification, Gonçalves [5] aims to address the lack of a user-friendly tool for verifying hyperproperties, though it is still in its early stages of development. Converting high-level models to low-level solvers raises the question of how to interpret the resulting outputs and present them in a way the user can easily understand. Current solvers return solver-level assignments that even experts struggle to understand seamlessly, limiting their

usability for non-experts. Besides, this process typically involves a pipeline of multiple tools operating at different levels of abstraction and relying on various input and output formats, which increases overall complexity. This fragmentation calls for a modular approach to facilitate the integration of various tools and to account for upcoming backends as the state of the art evolves. These challenges collectively represent the principal problem regarding high-level hyperproperty model checking.

1.4 Research Questions

To address this problem, the following research questions were established to guide its solution.

- RQ1: How can high-level hyperproperty specifications be effectively translated into low-level formal representations for the back-end model checker for HyperLTL?
- RQ2: How can low-level feedback from HyperLTL verification tools be abstracted back into high-level human-readable outputs?

1.5 Hypothesis

Considering the problem at hand, the following hypothesis can be formulated:

> An architecture can be designed to describe a pipeline to formally express hyperproperties at a high level, translate them into low-level formal representations to be verified with existing model checkers for HyperLTL, and present their feedback in a human-friendly way.

To evaluate this hypothesis, this dissertation proposes an extension of Alloy as a high-level tool for hyperproperty specification and verification. The proposed approach consists of designing and implementing a modular pipeline that translates high-level Alloy models and hyperproperties into low-level representations accepted by existing HyperLTL verification solvers, while preserving information needed to reconstruct and visualize the verification outputs at the Alloy level. The evaluation will focus on verifying whether the architecture can successfully integrate diverse tools and representations, support the verification of hyperproperties via existing backends, and provide understandable feedback to users by visualizing witnesses and counterexamples.

1.6 Thesis Overview

This dissertation addresses the challenge of verifying hyperproperties specified over high-level Alloy models. Rather than requiring users to manually encode specifications for existing hyperproperty model checkers, it proposes a verification pipeline that automatically translates Alloy models into the formats expected by state-of-the-art verification tools and reconstructs the verification results back at the Alloy level. Figure 1.1 illustrates the verification workflow before this

dissertation. While existing tools supported the translation of Alloy models towards HyperLTL verification, the workflow terminated at the solver, with no mechanism to reconstruct verification results back into Alloy.

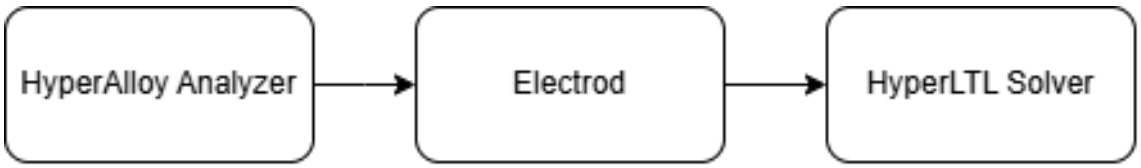


Figure 1.1: Architecture before the solution

The proposed solution builds upon several existing tools. A HyperLTL extension of Alloy provides the modeling language and analyzer, a modified Electrod translates Alloy specifications into declarative SMV, HyperSMV optimizes the declarative SMV into solver-ready simplified SMV, AutoHyper and HyperQube perform hyperproperty verification, and the Alloy Visualizer serves as the basis for presenting verification results. Although these tools collectively support different stages of the verification process, the workflow remained incomplete, ending at the HyperLTL solver without reconstructing verification results at the Alloy level. The main contribution of this work is the design and implementation of an end-to-end verification pipeline connecting these tools. The resulting architecture is shown in Figure 1.2.

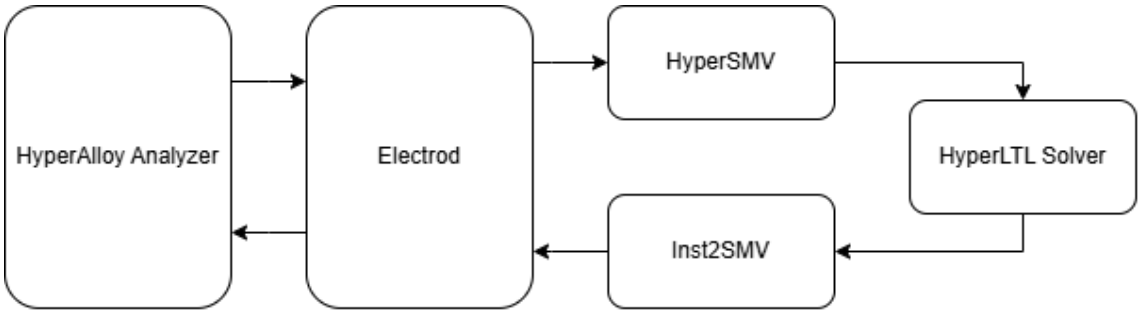


Figure 1.2: Architecture of the solution

This includes the development of an intermediate translation layer from the low-level model checkers to the SMV level, the extension of the Alloy Visualizer to reconstruct multiple witnesses and counterexamples from model-checking outputs, and the integration of all components into a unified framework that enables hyperproperty verification directly from Alloy models. The proposed approach demonstrates that Alloy can be effectively used as a front-end for hyperproperty verification without requiring users to interact directly with lower-level verification languages. Experimental evaluation shows that the generated translations are correct for some of the considered benchmarks, making the verification process more accessible, although still with limitations.

1.7 Dissertation Structure

This dissertation is structured into 6 chapters, each detailing a different aspect of the work done. Chapter 1 introduced the context and motivation for this work, presented the problem being addressed, and defined the research questions and hypothesis that guide the dissertation. Chapter 2 presents a systematic literature review and the state of the art, exploring formal verification, HyperLTL, and techniques for hyperproperty verification. Chapter 3 dives into the existing tools relevant to this work. Chapter 4 presents the design of the solution and its architecture, as well as how the evaluation process will be conducted. Chapter 5 presents the solution implementation, highlighting the components developed in this dissertation and how they were integrated. Chapter 6 presents the evaluation results, discusses them, and revisits the research questions and the earlier proposed hypothesis. Finally, Chapter 7 discusses the main conclusions drawn from the evaluation, reflects on the limitations of the proposed approach, and presents possible directions for future work.

Chapter 2

Literature Review and State of the Art

In this chapter, the literature review process will be described, and its key takeaways will be presented in a state-of-the-art review.

2.1 Literature Review

The process followed for a systematic literature review was the Preferred Reporting Items for Systematic Reviews and Meta-Analyses (PRISMA) methodology [6]. This methodology consists of three phases: identification (Section 2.1.1), screening (Section 2.1.2), and eligibility (Section 2.1.3).

2.1.1 Identification phase

The first phase of the PRISMA methodology is the identification phase. This phase involves identifying data sources and defining search keywords, ultimately leading to the formulation of queries to retrieve relevant references for this dissertation.

2.1.1.1 Data Sources

First, data sources were selected from which to retrieve the references. The selected sources were Scopus [7], IEEE Xplore [8], ScienceDirect [9], and ACM Digital Library [10].

2.1.1.2 Search Strings

The next step was to define the search keywords. The first keyword was "`HyperLTL`", and, after it, came "`hyperproperties`". After realizing that "`hyperproperty`" also came up frequently, "`hyperproperties`" was changed to "`hyperpropert*`" to encompass both. To address any potentially missing relevant documents, "`model checking`" and "`verification`" were also considered.

2.1.1.3 Search Queries

The search queries were defined and redefined while using Scopus. After some meddling with the keywords and logical operators, three search queries were established:

- **QS1:** `"HyperLTL" AND "hyperpropert*"`
- **QS2:** `"model checking" AND ("HyperLTL" OR "hyperpropert*")`
- **QS3:** `"verification" AND ("HyperLTL" OR "hyperpropert*")`

This way, references to both `"HyperLTL"` and `"hyperpropert*"` were considered. If not both, they had to mention `"model checking"` or `"verification"`.

To reduce duplicates, the queries were merged into a single query using an `OR` operation. The number of records per data source and per query is described in Table 2.1.

Data Source	QS1	QS2	QS3	Total	Total (without duplication)
Scopus	67	110	136	313	205
IEEE Xplore	11	26	23	60	31
ScienceDirect	2	5	6	13	8
ACM Digital Library	0	15	13	28	22
All	80	156	178	414	266

Table 2.1: Search string results per data source and per query

2.1.2 Screening Phase

The second phase of the PRISMA methodology is the screening phase. This phase consists of defining inclusion and exclusion criteria and analyzing each unique record. The result of this section is an initial list of potentially relevant references.

2.1.2.1 Inclusion and Exclusion Criteria

To shorten the list of references, a set of criteria was defined (see Table 2.2). All the records passed through these steps.

Phases	Criteria
1	Search engine results after executing a search query
2	Full-text access
3	Published in the English language
4	Article, Conference Paper or Book
5	Not duplicated

Table 2.2: Inclusion and Exclusion Criteria

Phases 1-4 were conducted on the search engines. A total of 137 entries were left after those steps and were put into EndNote [11]. Duplicate detection was initially performed by EndNote's

automatic feature, which excluded 19 instances. After that, I manually checked for more and found another 15.

2.1.2.2 Screened Records

The remaining 103 records were screened by reviewing their titles and abstracts to identify the most relevant to the dissertation's theme. A total of 6 records were excluded because they either met the fourth criterion by mistake or were not relevant. Most of the records were maintained because the search queries were specific.

2.1.3 Eligibility Phase

The third and final phase of the PRISMA methodology is the eligibility phase. This phase involves selecting the most relevant records from those that met the criteria defined in Section 2.1.2.1. This process was conducted in parallel with the screening by reading relevant sections of the papers.

2.1.3.1 Selected Records

The selected records are listed in Table 2.3. They represent the final set of studies identified through the review process and explore different perspectives around the domain of this dissertation. They serve as the basis for the State of the Art presented in Section 2.2.

2.2 State of the Art

In [3], hyperproperties are presented as sets of trace properties that express security policies that trace properties cannot. Hyperproperties include both safety and liveness hyperproperties (hypersafety and hyperliveness). Hypersafety ensures that nothing bad ever happens across multiple system executions; a finite set of bad behaviors across traces is sufficient to refute a hypersafety property. It also defines k-safety hyperproperties as a subset of hypersafety properties for which a violation can be demonstrated by observing at most k traces. Hyperliveness ensures that for any finite observation of the traces, it is always possible to construct a set of traces that satisfies the hyperproperty; refuting a hyperliveness property requires showing that no such trace exists among all possible traces. A significant amount of research focuses on verifying specific classes of hyperproperties (e.g., k-safety). While these approaches are tailored to particular subclasses, this work addresses user-specified hyperproperties expressed in unrestricted logics, thereby also supporting k-safety hyperproperties.

2.2.1 Logics for Hyperproperties

Standard temporal logics such as LTL, CTL, and CTL* are used to formally express trace properties. In [15], hyper temporal logics (HyperLTL and HyperCTL*) are proposed to address the

Reference	Venue	Research Purpose
[3]	JCS '10	Hyperproperties
[12]	CSF '16	Monitorability in HyperLTL and monitoring over k-safety and co-k-safety hyperproperties
[13]	TACAS '18	RVHyper, a runtime verification tool for hyperproperties
[14]	OOPSLA '24	Hypra, a deductive program verifier for arbitrary hyperproperties
[15]	POST '14	HyperLTL
[16]	TECS '23	Automatically infer hyperproperties specifications from system executions
[17]	VIS '21	Visual analysis of hyperproperties model checking results
[18]	CAV '22	HyperLTL counterexamples
[4]	CAV '15	Automata-based algorithm for checking finite state systems for hyperproperties specified in HyperLTL
[19]	TACAS '23	AutoHyper, an existing model checker that supports HyperLTL
[20]	TACAS '21	Bounded model checking for Hyperproperties
[21]	TACAS '23	BMC techniques for HyperLTL with loop conditions
[22]	ESOP '24	Verification of hyperproperties as CHC formulas, instead of first-order
[23]	OOPSLA '24	Automated detection of hyperproperties violations with bug-finding techniques
[24]	TACAS '24	Automated verification of hyperproperties that combine universal and existential quantification
[25]	CAV '19	Hyperliveness properties expressed as HyperLTL formulas with quantifier alternation
[26]	CSF '21	Verification of hyperproperties with TLA
[27]	ATVA '23	Combining the scalability of statistical approaches with the verification of hyperproperties
[28]	POPL '17	Derive static analyses for hyperproperties

Table 2.3: Selected Records

expression of hyperproperties, since they cannot be expressed in the classic temporal logics. HyperLTL and HyperCTL* extend LTL and CTL*, respectively, by introducing quantification over traces ($\forall\pi$, $\exists\pi$) and variables that can refer to specific traces. As a simple example, LTL can specify that a request is eventually followed by a response in a single execution. However, it cannot specify that two executions differing only in confidential inputs are indistinguishable from the perspective of a public observer. This requires quantification over multiple traces, as provided by HyperLTL.

2.2.1.1 LTL

LTL [29] is a formal language used to specify properties of individual traces over time. It is used to verify properties such as safety (nothing bad will ever happen) and liveness (something good will eventually happen).

Syntax The syntax of LTL is represented by the following grammar:

$$\varphi ::= a \mid \neg\varphi \mid \varphi \vee \varphi \mid X\varphi \mid \varphi U \varphi$$

where a is an atomic proposition, i.e., a Boolean variable that represents a property of the system at a given state.

Operators

- **Propositional Operators:** $\neg$ (negation), $\wedge$ (and), $\vee$ (or), $\rightarrow$ (implies).
- **Temporal Operators:**
  - $X\varphi$: φ holds in the ne**X**t state.
  - $F\varphi$: φ eventually holds at some **F**uture state.
  - $G\varphi$: φ always (**G**lobally) holds in all future states.
  - $\varphi U \psi$: φ holds **U**ntil ψ becomes true, and ψ eventually becomes true.
  - $\varphi R \psi$: ψ must hold until and including the point where φ holds, or ψ holds forever if φ never holds (**R**elease).
  - $\varphi W \psi$: φ holds until ψ becomes true, or φ continues to hold if ψ never becomes true (**W**eak until).

The propositional operators *negation* ($\neg$) and *disjunction* ($\vee$) are the fundamental propositional logical operators of LTL. The remaining propositional operators and Boolean constants can be defined using $\neg$ and $\vee$ (assuming at least one propositional variable a exists).

- $\varphi \wedge \psi \equiv \neg(\neg\varphi \vee \neg\psi)$

- $\varphi \rightarrow \psi \equiv \neg\varphi \vee \psi$
- $\varphi \leftrightarrow \psi \equiv (\varphi \rightarrow \psi) \wedge (\psi \rightarrow \varphi)$
- true $\equiv a \vee \neg a$
- false $\equiv \neg$true

The operators *next* (X) and *until* (U) are the fundamental temporal operators of LTL. The other temporal operators can be defined using X and U in combination with propositional operators.

- $F\varphi \equiv$ true $U\ \varphi$
- $G\varphi \equiv$ false $R\ \varphi \equiv \neg F \neg\varphi$
- $\varphi R \psi \equiv \neg(\neg\varphi\ U\ \neg\psi)$
- $\varphi W \psi \equiv (\varphi U \psi) \vee G\varphi \equiv \varphi U(\psi \vee G\varphi) \equiv \psi R(\psi \vee \varphi)$

2.2.1.2 Examples

LTL is used to specify temporal properties over a single execution trace. The following examples illustrate common classes of properties that can be expressed in LTL.

Safety: Safety properties ensure that "nothing bad ever happens." They specify that undesirable states are never reached during a system execution.

$$G\neg error$$

This formula states that the system never enters an error state.

Liveness: Liveness properties ensure that "something good eventually happens." They guarantee that progress is eventually made.

$$G(request \rightarrow F\ grant)$$

This formula states that every request is eventually granted.

Response: Response properties specify that the occurrence of one event must eventually be followed by another.

$$G(send \rightarrow F\ receive)$$

This formula states that every message sent is eventually received.

Mutual Exclusion: Mutual exclusion ensures that no two processes are simultaneously in their critical sections.

$$G\left(\bigwedge_{1\leq i<j\leq n} \neg(cs_i \wedge cs_j)\right)$$

This formula states that, at every point in time, every pair of processes is prevented from being in the critical section simultaneously.

2.2.1.3 HyperLTL

While LTL can only express properties of individual traces, HyperLTL extends LTL to express hyperproperties that involve relationships among multiple traces. This is achieved by introducing:

- **Trace Variables:** Variables like π_1, π_2 represent specific traces.
- **Trace Quantifiers:** $\forall\pi$ (for all traces) and $\exists\pi$ (there exists a trace).
- **Indexed Propositions:** Propositions such as a_π refer to the value of a in trace π.

Syntax The syntax of HyperLTL is a superset of the syntax of LTL. A subset of its fundamental operators is represented by the following grammar, as the others can be derived from these, as explained in Section 2.2.1.1.

$$\psi ::= \exists\pi.\psi \mid \forall\pi.\psi \mid \varphi$$
$$\varphi ::= a_\pi \mid \neg\varphi \mid \varphi \vee \varphi \mid X\varphi \mid \varphi U \varphi$$

2.2.1.4 Examples

HyperLTL allows the expression of hyperproperties, which capture relationships across multiple system executions. A common application of hyperproperties is in the specification of security policies. Below are some examples of hyperproperties in the context of security and how they are represented in HyperLTL, with variables partitioned into input and output variables, and into two security levels, *high* and *low*, represented as I_L (set of low-level input variables), O_L (set of low-level output variables), I_H (set of high-level input variables), and O_H (set of high-level output variables):

Noninterference: Noninterference [30] ensures that the outputs observable by low-security users are independent of the inputs from high-security users. That means the absence of the latter should yield the same low outputs. In HyperLTL, this is expressed by comparing two execution traces, π and π'. The universal quantifiers require the property to hold for every pair of traces, while the temporal operator G specifies that the comparison holds at every point in time.

$$\forall\pi.\forall\pi'.G\Big(\bigwedge_{i\in I_L} i_\pi = i_{\pi'}\Big) \rightarrow G\Big(\bigwedge_{o\in O_L} o_\pi = o_{\pi'}\Big)$$

The formula should be read from left to right: "for every pair of traces π and π', if they always agree on the low-security inputs, then they must always agree on the low-security outputs."

Noninference: Noninference [31] is a liveness variant of noninterference that ensures that, for all traces, the low-observable behavior must be the same when high inputs are replaced by dummy inputs, that is, when they are removed.

$$\forall\pi.\exists\pi'.G\Big(\bigwedge_{i\in I_H} i_{\pi'} = \lambda\Big) \wedge G\Big(\bigwedge_{l\in(I_L\cup O_L)} l_\pi = l_{\pi'}\Big)$$

Here, $i_{\pi'} = \lambda$ represents that all high inputs in π' are assigned a distinguished dummy value λ, representing the absence of high inputs, and the rest ensures low-equivalent traces.

As noninterference was envisioned for deterministic programs, variants have emerged to address nondeterministic programs, including observational determinism and generalized noninterference.

Observational determinism: Observational determinism [32] is a safety variant that states that for every pair of traces with the same initial low observation, they remain indistinguishable for low users. This means that the program appears to be deterministic for low users.

$$\forall\pi.\forall\pi'.\Big(\bigwedge_{i\in I_L} i_\pi = i_{\pi'}\Big) \rightarrow G\Big(\bigwedge_{o\in O_L} o_\pi = o_{\pi'}\Big)$$

Generalized Noninterference: Generalized Noninterference (GNI) [33] is a liveness variant that allows nondeterminism in low-observation contexts, but stipulates that high-security inputs must not alter low-security outputs.

$$\forall\pi.\forall\pi'.\exists\pi''.G\Big(\bigwedge_{i\in I_H} i_\pi = i_{\pi''}\Big) \wedge G\Big(\bigwedge_{l\in I_L\cup O_L} l_{\pi'} = l_{\pi''}\Big)$$

Here, π'' is a trace whose high-security inputs coincide with those of π, while its low-security inputs and outputs coincide with those of π'.

Declassification: Noninterference policies would not allow a program to reveal secret information, but in some cases, this is necessary to fulfill functional requirements, such as revealing whether a password is correct or not. To facilitate the declassification of information [34], more flexible policies have been developed. For a system that, in its initial state, inputs a password and, in the next state, determines whether that password is correct, here is an example of a safety hyperproperty stating that observational determinism must hold unless the system is leaking the password's correctness.

$$\forall\pi.\forall\pi'.\Big(\Big(\bigwedge_{i\in I_L} i_\pi = i_{\pi'}\Big) \wedge X(\mathrm{pw}_\pi \leftrightarrow \mathrm{pw}_{\pi'})\Big) \rightarrow G\Big(\bigwedge_{o\in O_L} o_\pi = o_{\pi'}\Big)$$

Here, pw is a Boolean proposition that evaluates to true if the password provided by the user is correct, and false otherwise.

Quantitative Noninterference: Quantitative information-flow policies [35–38] permit leakage of information at restricted rates. One can quantify the amount of information an attacker can gain from their response to a single guess about the secret as min-entropy [39], a measure of leakage. Using min-entropy as a metric, the bounding problem [40] determines whether the leakage is bounded by a constant n. Quantitative noninterference extends noninterference by bounding the amount of information leakage to low-security outputs. Assuming that the program whose leakage is being quantified is deterministic and its secret input is uniformly distributed, the bounding problem is reduced to determining that there is no tuple of $2^n + 1$ low-distinguishable traces and can be expressed as the following formula.

$$\neg\exists\pi_0,\ldots,\exists\pi_{2^n}.\bigwedge_i G\Big(\bigwedge_{a\in I_L} a_{\pi_i} = a_{\pi_0}\Big) \wedge \bigwedge_{i\neq j} G\Big(\bigwedge_{a\in O_L} a_{\pi_i} = a_{\pi_j}\Big)$$

This restricts low-security outputs to a bounded number of distinguishable values.

2.2.1.5 Computational Tree Logics

As mentioned above, there are other temporal logics beyond LTL and HyperLTL. CTL [41] is another classical temporal logic.

CTL CTL is a branching-time model, i.e., it represents the possible executions of a system as a computation tree. Rather than quantifying over complete execution traces, CTL uses path quantifiers, which range over the branches (paths) originating from the current state:

- A: "For **A**ll paths."
- E: "There **E**xists a path."

A CTL formula is built by combining these quantifiers with LTL operators. Some properties can be expressed in CTL but not in LTL, whereas others cannot be expressed in CTL, even though they are expressible in LTL. This is due to CTL's syntax, which requires every temporal operator to be immediately preceded by a path quantifier.

Examples of CTL formulas include:

- $AF\ \varphi$: on all execution paths, φ eventually holds;
- $EG\ \varphi$: there exists an execution path in which φ always holds.

CTL* CTL* [42] is a superset of LTL and CTL. It removes the restriction that path quantifiers must be immediately followed by a temporal operator, allowing arbitrary nesting.

Examples of CTL* formulas include:

- $A(FG\,\varphi)$: on all execution paths, the system eventually reaches a point after which φ always holds;
- $E(GF\,\varphi)$: there exists an execution path in which φ will always eventually hold.

HyperCTL* HyperCTL* [15] is an extension of CTL* similar to HyperLTL's extension on LTL, introducing trace quantifiers and indexed propositions. It combines the branching-time reasoning of CTL* with the ability to compare relationships across multiple traces. This additional expressiveness allows HyperCTL to specify branching-time hyperproperties that cannot be expressed in HyperLTL, although it comes at the cost of increased reasoning and verification complexity.

2.2.1.6 Other Temporal Logics

While conducting the literature review, other temporal logics beyond HyperLTL were identified. Rather than presenting them individually, they can be grouped broadly by the types of systems and properties they target.

Some approaches extend branching-time temporal logics with hyperproperty reasoning. HyperATL* [43] combines hyperlogic with multi-agent strategy reasoning, extending Alternating-Time Temporal Logic (ATL)* to include path and strategy quantifiers, enabling reasoning about asynchronous hyperproperties in multi-agent systems. Similarly, TeamCTL* [44] extends CTL* with team semantics and asynchronous trace progression via time-evaluation functions. Other works also explore branching-time semantics together with timing aspects, such as HCMTL* [45], which extends Metric Temporal Logic (MTL), a temporal logic that explicitly incorporates quantitative timing constraints between events, with quantification over multiple traces and real-time constraints; verification is undecidable in general, but becomes decidable for bounded horizons.

Another group of works focuses on reasoning about quantitative time and timed systems. HyperMITL [46] extends HyperLTL for reasoning about hyperproperties in systems with timing constraints, while HyperPTCTL [47] extends Parametric Timed Computation Tree Logic (PTCTL) with hyperlogic capabilities, supporting reasoning over multiple traces and timing parameters. These approaches are often motivated by the verification of real-time systems, where the timing between events is as important as the events themselves. In general, unrestricted verification is often undecidable, motivating the study of restricted fragments or bounded horizons.

Several hyperlogics have also been proposed for domain-specific applications. HyperLDLf [48] extends Linear Dynamic Logic over Finite Traces (LDLf), a temporal logic inspired by dynamic logic, which models program behavior through actions and their possible executions, and reasons about finite executions using regular-expression-like path expressions, with hyperlogic features, making it useful for checking multi-trace properties in finite process logs. HyperPLTL [49] introduces past-time reasoning into HyperLTL and targets hardware/software co-validation scenarios, including adversarial reasoning and fuzzing in System-on-Chip (SoC) designs. HyperTSL [50] extends Temporal Stream Logic (TSL) for specifying and synthesizing hyperproperties in smart contracts; it detects trace-equivalent hyperproperties and refines contracts to satisfy them.

Other works focus on probabilistic or signal-based systems. HyperPCTL* [51] extends PCTL* to support probabilistic hyperproperties, such as anonymity and resistance to timing side-channel attacks. HyperPSTL [52] combines probabilistic reasoning with Signal Temporal Logic (STL), which is commonly used to specify temporal properties over real-valued signals in Cyber-Physical Systems (CPSs), targeting CPSs with continuous-time dynamics. HyperSTL [16] extends STL to infer hyperproperties from system executions, allowing quantifier alternation and inference of hyperproperties without relying on predefined templates.

There are also proposals aimed at increasing HyperLTL's expressive power. HyperQPTL [53] combines HyperLTL with propositional quantification, enabling the specification of regular hyperproperties and bounded liveness properties (promptness). OHyperLTL [54] introduces observation-based semantics and asynchronous reasoning, allowing traces to progress at different speeds. Additionally, asynchronous variants of HyperLTL, such as A-HLTL [55], further explore asynchronous trace semantics and synchronization mechanisms between traces.

Beyond the theoretical definition of temporal logics, practical applications of these logics for identifying and verifying hyperproperties are emerging. One such approach leverages HyperSTL to mine hyperproperties from execution data. The paper [16] explores how temporal logics, particularly HyperSTL, can be applied to extract hyperproperties from system executions. By analyzing execution data, this approach identifies patterns that satisfy predefined hyperproperty templates or infers hyperproperties without relying on predefined patterns.

2.2.2 Formal Verification

Verification of (hyper)properties is essential to ensure system security. Different strategies can be followed to achieve this verification, including runtime verification, deductive verification, and model checking. Since this work aims to leverage Alloy for hyperproperty verification, it focuses on model checking.

2.2.2.1 Runtime Verification of Hyperproperties

Runtime Verification (RV) [56] is a technique that monitors system executions at runtime to ensure compliance with specified properties. It complements exhaustive verification methods by analyzing execution traces directly on the running system, avoiding the complexity and potential errors of formal modeling. This approach is particularly effective for detecting policy violations caused by unanticipated threats or vulnerabilities. Unlike traditional methods such as model checking, RV operates on the actual system, enabling applications such as security and safety policy monitoring, debugging, testing, verification, validation, profiling, fault protection, and behavior modification.

For hyperproperties, RV involves reasoning about relationships across multiple traces to ensure compliance with properties such as k-safety. In [12], monitorability is defined for HyperLTL, and algorithms are proposed for verifying if a subset of k-safety hyperproperties is held by comparing system executions against their respective specification in HyperLTL.

In [13], RVHyper is presented as a tool to verify HyperLTL properties at run-time by monitoring. It optimizes RV by sequentially analyzing system traces and leveraging properties such as symmetry, transitivity, and reflexivity in HyperLTL formulas. If a violation is detected, RVHyper provides a counterexample to highlight the failure, demonstrating its utility for RV of temporal hyperproperties. However, this approach is incomplete because it considers only the finite set of traces observed during system execution. In particular, for existential quantification, traces are searched only within this observed set, rather than over the (infinite) universe of all possible system executions.

2.2.2.2 Deductive Verification

Deductive verification [57] is a formal method that uses logical proofs to verify that a system satisfies its specification. Unlike model checking, which typically operates on system models, deductive verification is applied directly to programs. This approach involves generating proof obligations from the program, its specification, and user-provided annotations, such as invariants, preconditions, and postconditions. These proof obligations, which ensure the program conforms to its specification, are fulfilled using tools such as SMT solvers and proof assistants. While deductive verification is typically semi-automated and requires user annotations, it is particularly effective for verifying hyperproperties because it can handle complex relationships across multiple traces.

Hypra [14] is a deductive program verifier that uses Hyper Hoare Logic (HHL). It automates the deductive verification process by encoding HHL into an intermediate verification language, which is then processed by SMT-based solvers. Hypra extends traditional methods by supporting the verification of more complex hyperproperties, including those with arbitrary quantifier alternations. It achieves this by combining overapproximation and underapproximation reasoning, which minimizes user-provided annotations while efficiently handling complex proof patterns.

Despite its automation capabilities, deductive verification often requires a deep understanding of the program being verified. Users must provide detailed insights into why the program works correctly, conveyed through annotations or sequences of theorems. Hypra alleviates some of this burden by automating much of the deductive reasoning, making it a powerful tool for hyperproperty verification.

2.2.2.3 Model Checking

Model checking [58] is a formal verification technique that automates the exploration of all possible states of a system to ensure that it satisfies a certain property. It utilizes a formal model of the system and a specification (in temporal logic) to detect violations. This process systematically examines the state space of a system, typically represented as a state-transition system, and determines whether all possible executions satisfy the specified property. If the property is violated, model checking provides a counterexample that demonstrates how the failure occurs.

Modern model-checking techniques employ various strategies to handle the computational complexity of exploring large or infinite state spaces. For finite models, explicit state enumeration is often used, where every possible state is analyzed. However, for infinite models or large-scale systems, abstraction techniques, such as symmetry reduction (grouping symmetric states to reduce redundancy), symbolic representations (like Binary Decision Diagrams (BDDs), which compactly represent Boolean functions, or SAT solvers, which encode problems into Boolean satisfiability), and abstraction-refinement methods (iteratively refining abstract models to eliminate inaccuracies), enable efficient verification by compactly representing sets of states. Temporal logics, such as LTL, CTL, and CTL*, provide the expressive power needed to specify a wide range of properties, including safety and liveness.

Bounded Model Checking (BMC) is a popular approach that limits verification to a finite depth, enabling the efficient detection of counterexamples within bounded executions. Symbolic methods, such as those that use BDDs or SAT solvers, further reduce the memory and time complexity of verification tasks. For unbounded verification, techniques such as Incremental Construction of Inductive Clauses for Indubitable Correctness (IC3) (which incrementally constructs inductive invariants to prove properties) and interpolation (which generates intermediate assertions to break complex proofs into simpler steps) improve the scalability of symbolic model checking.

Although model checking is highly automated and precise, it faces scalability limitations. Explicit state enumeration is constrained by the size of the state space, while symbolic approaches are typically limited to representing systems with a few hundred bits of state.

Model checking has also been extended to support hyperproperties. Since classical temporal logics such as LTL reason about individual traces, they are insufficient to express relations across multiple executions. To address this, hyperlogics such as HyperLTL introduce explicit trace quantification, enabling the specification of such properties. As in standard model checking, both explicit-state and symbolic techniques have been adapted to this context. Since model checking is the main focus of this dissertation, these techniques will be discussed in more detail in Section 2.2.3.

2.2.3 Automatic Verification for HyperLTL

Traditional verification techniques focus on properties of individual execution traces using temporal logics such as LTL. However, hyperproperties, which express relationships across multiple traces, require different verification techniques. While trace properties can be checked by exploring the state space of a single execution, hyperproperties involve reasoning about multiple executions simultaneously. This shift introduces new challenges for hyperproperty verification, including the need to reason about sets of traces and to handle the computational complexity introduced by quantifier alternations. Existing approaches differ in how they reason about the temporal dimension of executions. Some techniques are complete and can reason about executions of arbitrary length, while bounded approaches restrict the analysis to a finite number of states. Within bounded techniques, loops may still be used to represent infinite behaviors through lasso-shaped

traces, although reasoning may also rely solely on finite trace prefixes when no loop is found within the explored bound.

Some approaches provide automated verification tools for hyperproperties. Among these, AutoHyper and HyperQube are particularly relevant to this dissertation, as they illustrate different verification strategies and motivate the need for a higher-level specification approach, which is the focus of the proposed solution.

2.2.3.1 MCHyper

The paper [4] presents an automata-theoretic algorithm specifically designed to verify finite-state systems against hyperproperties defined in HyperLTL and HyperCTL*. The method leverages alternating Büchi automata (finite-state automata used to model systems with infinite behaviors) in its construction, extending existing model-checking technologies to handle alternation-free formulas within these hyperlogics. To verify their algorithms, the authors developed a prototype tool, MCHyper [59].

MCHyper focuses on alternation-free formulas in HyperLTL but supports quantifier alternation up to one alternation. In some cases, a witness may be required to assist in the verification process. In the context of formal verification, a witness is a supplementary input that guides the verification process, such as a trace demonstrating compliance with a given property. Alternating Büchi automata allow the verification of infinite behaviors. To use MCHyper as a model checker for a given system and a property, the system must be specified as a Verilog module or as an Aiger circuit, and the property as a HyperLTL formula, either in the EAHyper [60] or in the MCHyper input format. It is recommended to use the EAHyper format, as the MCHyper format is more complex and difficult to write and read.

The EAHyper input format allows the specification of HyperLTL formulas using a structured grammar. It follows the general structure of HyperLTL but imposes stricter rules on quantifier ordering and syntax. It does not allow arbitrary quantifier alternation; it limits alternation to formulas with at most one alternation.

For satisfied properties, MCHyper outputs a confirmation that the system satisfies the given HyperLTL formula. For violated properties, the tool generates counterexamples as specific trace combinations that demonstrate the violation.

2.2.3.2 AutoHyper

AutoHyper [19] is introduced as a tool for model checking HyperLTL properties using an explicit-state automata-based approach. It can handle the full range of HyperLTL formulas, including those with arbitrary quantifier alternations, enabling comprehensive verification without manual intervention. The tool operates by iteratively eliminating trace quantification via automata complementation, thereby reducing the problem to checking the emptiness of the constructed automaton. To optimize performance, it integrates language inclusion checks and leverages mature external tools such as Spot, RABIT, BAIT, and FORKLIFT for automata operations. Although AutoHyper

is not temporally bounded (it supports infinite traces using Büchi automata), it is bounded by the size of the model it can handle. Since it uses an explicit-state automata-based approach, the tool's performance is inherently tied to the size of the system's state space. Larger models with complex state spaces can pose scalability challenges.

The system specifications can be expressed in either explicit-state systems (given in a HANOI-like [61] input format) or symbolic systems, which are internally converted to an explicit-state representation. The support for symbolic systems includes Aiger circuits, symbolic models written in a fragment of the `NuSMV` input language [62], and a simple boolean programming language [63]. The HyperLTL formulas in AutoHyper adhere to the conventional HyperLTL syntax, with no major differences.

AutoHyper checks whether a property holds or is violated, and its output is either `SAT` or `UNSAT`. They mention that the inclusion checkers return usable counterexamples. There previously existed an extension of AutoHyper for HyperQPTL (see Section 2.2.1.6), called AutoHyperQ [64]. While AutoHyperQ is now discontinued, its functionality has been integrated into AutoHyper, which currently supports HyperQPTL and produces mappings that assign values to variables.

2.2.3.3 HyperQube

BMC for LTL reduces the verification problem to SAT solving, but the trace quantifications inherent in HyperLTL need Quantified Boolean Formula (QBF) solving. HyperQube encodes both the system and the HyperLTL specification as a QBF problem and explores the state space incrementally, with a state-space bound of `k`. HyperQube [20] is the first BMC tool designed to verify hyperproperties specified in HyperLTL. It supports HyperLTL properties with quantifier alternations. It is bounded by the sizes of the model and the traces (which are finite) and does not handle loops, making it incomplete for some properties without exhaustive exploration.

HyperQube accepts models in the `NuSMV` format, a widely used model-checking language. The `process` keyword, used to model asynchronous processes, is not supported. To represent asynchronicity, users can use nondeterministic updates with the `next()` function in `TRANS`. It uses its own custom grammar to specify HyperLTL formulas, since the `NuSMV SPEC` keyword does not support HyperLTL. Their grammar is not that different from AutoHyper's. The tool generates examples/counterexamples, but they are not understandable.

The paper [21] introduces a simulation-based framework to efficiently handle BMC for HyperLTL formulas. Traditional BMC techniques for LTL employ loop conditions to reason about infinite behaviors within finite exploration. However, HyperLTL's trace-quantifying nature complicates this process, as traces from different models may loop in inconsistent ways. The authors propose efficient algorithms to address this challenge, enabling reasoning about infinite traces through finite exploration via simulation-based techniques. The study introduces two simulation variants:

- SIM_{EA}: Handles formulas of the type $\exists\pi.\forall\pi'.\square \mathit{Pred}$, focusing on verifying that a single trace in one model matches all traces in another.

- SIM_{AE}: Addresses formulas of the type $\forall\pi.\exists\pi'.\square \mathit{Pred}$, ensuring that every trace in one model is matched by a trace in another.

Pred is a relational predicate (a predicate over a pair of states). Both algorithms leverage simulation relations to avoid exhaustive exploration of the entire state space. For cases where nondeterminism complicates direct simulation, the authors enhance SIM_{AE} with prophecy variables, providing the necessary foresight to resolve ambiguities. Prophecy variables are auxiliary variables introduced in formal verification, used when the satisfaction of a property depends on future behaviors or states that cannot be determined in the current execution step. The algorithms generate SAT queries to determine whether a given formula is satisfiable. The algorithms were implemented using Z3Py (Python API for the Z3 SAT solver). For SIM_{AE}, the algorithm successfully reduced the explored state space by focusing on relevant portions of the larger model, and for SIM_{EA}, the method efficiently identified lasso traces satisfying the required properties. Lasso traces are finite paths in a system that represent infinite behaviors: a sequence of states leading to a point where the behavior starts repeating (a prefix), and a sequence of states forming a cycle that repeats indefinitely (a lasso). Their methods outperformed HyperQube by incorporating loop conditions, enabling reasoning about infinite traces in finite steps.

2.2.3.4 HyHorn

The HyHorn tool [22] introduces a novel approach to verifying hyperproperties using Constrained Horn Clauses (CHCs). A CHC is a first-order formula of the form:

$$\forall \mathscr{X} \cdot \bigwedge_i P_i(\mathscr{X}_i) \wedge \varphi(\mathscr{X}) \rightarrow H(\mathscr{X}_H),$$

where:

- $\mathscr{X}$ is a vector of logical variables.
- $P_i \in \mathscr{P}$ are predicates over subsets $\mathscr{X}_i \subseteq \mathscr{X}$.
- Predicates may repeat, i.e., $P_{i_1} = P_{i_2}$ is allowed for $i_1 \neq i_2$.
- H is the *head*, which is either a predicate from $\mathscr{P}$ or $\bot$ (false).
- $\varphi(\mathscr{X})$ is a constraint, typically expressed in a decidable logic.
- $\mathscr{X}_H \subseteq \mathscr{X}$ are the variables relevant to the head predicate.

The universal quantification $\forall \mathscr{X}$ is often omitted for simplicity.

HyHorn focuses on hyperproperties that involve $\forall * \exists *$ quantifications, a structured quantifier sequence in which all universal quantifiers precede all existential quantifiers. The challenge in verifying such properties lies in aligning multiple traces, discovering existential witnesses, and simultaneously constructing relational inductive invariants. Unlike traditional approaches that fix alignments or require specialized solvers, HyHorn reduces hyperproperty verification problems

to CHC-SAT problems, leveraging the strengths of existing CHC solvers. The process involves encoding the verification problem as first-order logic formulas using uninterpreted predicates representing alignment, invariants, and witnesses. These formulas are transformed into CHCs, which are resolved by solvers like Spacer [65]. It employs relational abstractions to reason about infinite-state systems, with the caveat that the branching degree must be finite (bounded by a constant) if the hyperproperty includes $\exists*$ quantification, to ensure soundness and completeness. For infinite branching systems, the approach is sound but incomplete.

HyHorn takes as input a control-flow graphs (CFGs) whose transitions are annotated with two-vocabulary first-order formulas to represent the system. Hyperproperties are represented with $\forall*\exists*$-quantified HyperLTL formulas. If an initial state is identified as doomed (i.e., the CHCs are unsatisfiable), then the property is violated, and a counterexample can be retrieved. Otherwise, if the set of initial states does not intersect the set of doomed states, then the hyperproperty is proved. A state is doomed if it reaches a state that violates the hyperproperty along every valid alignment (as opposed to some in the direct encoding). This tool is not yet public.

2.2.3.5 Hy∃n∀ (zero)

The paper [23] targets hyperproperties expressed in a fragment of OHyperLTL (see Section 2.2.1.6), focusing on $\forall\exists$ hyperproperties where each universally quantified trace requires the existence of a corresponding trace that satisfies a specific relationship. The approach utilizes symbolic execution engines (substituting normal inputs, such as numbers, for symbolic values and formulae) to generate symbolic representations of traces, thereby significantly reducing the need for exhaustive exploration. It introduces a bounded semantics for OHyperLTL, making reasoning over both terminating and non-terminating executions feasible by focusing on finite-length prefixes. Two algorithms were proposed:

- A naive algorithm that checks satisfaction for increasing bounds.
- An optimized "lazy" algorithm that processes traces incrementally, identifying counterexamples by searching for universal traces lacking existential matches.

Hy∃n∀ (zero) is temporally bounded, focusing on finite execution prefixes to detect violations, but not specifically bounded on model size, although limited by available computational resources. Systems are described with CFGs, annotated with first-order formulas to represent transition relations. Hyperproperties are specified in a fragment of OHyperLTL, referred to as $OHyperLTL_{safe}$, and are expressed using $\forall*\exists*$-quantified HyperLTL formulas. The tool outputs whether the property is satisfied or violated. If violated, a counterexample will be generated. Hy∃n∀ (zero) reasons at the program level, instead of the model level, which is not exactly the intended approach for this work, but it is still relevant and can be compared with HyHorn.

2.2.4 Other Verification Techniques for HyperLTL

Other publications present alternative techniques without necessarily presenting new tools.

The paper [24] uses ForEx [66], a tool for the automated verification of $\forall k \exists l$ hyperliveness properties, which also employs symbolic execution, to generate symbolic trace representations, which are then checked for property violations using constraint solving. It supports reasoning over finite execution prefixes to detect violations of hyperliveness properties.

In [25], the authors address the challenge of verifying hyperliveness properties expressed in HyperLTL, focusing on formulas with quantifier alternations. They propose a game-theoretic approach for model checking and synthesis, emphasizing strategies for existential quantifiers and leveraging prophecy variables and bounded synthesis to improve efficiency. MCHyper is extended to handle quantifier alternations and demonstrates practical applications in security and symmetry within reactive systems.

The paper [26] uses the Temporal Logic of Actions (TLA) for system-level verification. It utilizes self-composition to model relationships between traces, with TLA+ providing tools for formal proofs and verification through logical formulas. TLA+ tools include a model checker and a proof checker. The paper focuses on verifying a large class of hyperproperties, termed "finitary hyperproperties," which includes a subset of $\forall\exists$-hyperproperties.

In [27], the authors present a novel approach to the verification of hyperproperties by combining statistical model checking with scalable methods to handle nondeterminism in Markov Decision Processs (MDPs). Their lightweight approach extends the PLASMA [67] tool to verify bounded, unquantified HyperLTL properties using statistical techniques, allowing black-box and gray-box system analysis. The method enables verification in systems with large state spaces by focusing on trace sampling and hypothesis testing.

The paper [28] introduces hypercollecting semantics, a fixpoint-based framework that elevates abstract interpretation to directly reason about hyperproperties, such as noninterference and quantitative information flow. It uses Galois connections to systematically approximate hypersafety properties and derive static analyses, including cardinality-based information-flow bounds that surpass traditional 2-safety analyses.

2.2.5 Hyperproperty Visualization Techniques

Beyond approaches focused solely on verification, there are also tools dedicated to improving the interpretation and visualization of hyperproperty violations and counterexamples.

In [17], the authors present HYPERVIS, an interactive tool for visualizing and analyzing hyperproperty counterexamples. Using MCHyper [59] for model checking, HYPERVIS emphasizes intuitive, linked visualizations and causal analyses to better understand violations, thereby enabling iterative refinement of system models and specifications. This foundational work is extended in [18], where the authors propose methods to explain hyperproperty violations by generating minimal causative subformulas and providing concise textual and visual representations, further enhancing interpretability and aiding in debugging during model checking.

2.2.6 Discussion

Table 2.4 summarizes the characteristics of the aforementioned tools, mentioning aspects like:

- **Expressiveness:** The hyperproperties or classes of hyperproperties supported by this tool. By default, regarding HyperLTL
- **Model Checking:** If the model checking is bounded, i.e., restricted to a finite number of paths or states
- **Infinite:** If the model reasons about infinite traces, i.e., if it is temporally bounded (length of the traces)
- **Inputs:** What are the inputs for the tool
- **Outputs:** What are the outputs from the tool
- **Loops:** If the tool supports loops
- **Witness:** If the tool needs user input for witness generation

Tool	**MCHyper**	**AutoHyper**	**HyperQube**	**HyHorn**	**Hy∃n∀ (zero)**
Expressiveness	Up to one quantifier alternation	All	Quantifier alternations	$\forall*\exists*$	OHyperLTL $\forall\exists$ hyper-properties
Model Checking	Bounded	Bounded	Bounded	Unbounded* (sound but incomplete)	Unbounded* (theoretically)
Infinite	Yes	Yes	No	Yes	No
System Inputs	Verilog/Aiger	Explicit-state or symbolic systems	`NuSMV`	Annotated CFG	Annotated CFG
Formula Inputs	EAHyper or MCHyper input format	Own HyperLTL-like	Own HyperLTL-like	$\forall*\exists*$-quantified HyperLTL formulas	$\forall*\exists*$-quantified HyperLTL formulas
Outputs	Confirmation or counterexample	SAT or UNSAT and Witness	Solver-level assignment	Confirmation or counterexample	Confirmation or counterexample
Loops	Yes	Yes	No	Yes	Yes
Witness	Yes	No	No	No	No

Table 2.4: Comparison of Tools for Automated Hyperproperty Verification

The outputs as solver-level assignments are difficult to comprehend. Besides, all the tools presented lack in some aspect, therefore, highlighting the need for a tool that combines all these features while offering higher-level, user-friendly inputs and outputs.

Chapter 3

Existing Tools

This chapter will detail the behavior of existing model-checking tools, specifically Alloy and existing extensions for HyperLTL as a high-level specification language, and state-of-the-art model-checking solvers for HyperLTL identified in the previous chapter, specifically AutoHyper and HyperQube. Alloy allows users to input a model and its properties in their preferred format and verifies whether the properties are valid. A previously proposed Alloy extension for HyperLTL [5] extends the syntax to represent hyperproperties and verify them using HyperQube. This dissertation uses HyperQube, along with AutoHyper (already introduced in Section 2.2.3), as the low-level model checkers for HyperLTL. These two tools were selected because they support HyperLTL verification over SMV transition systems.

3.1 Alloy

The Alloy language [1] is a declarative modeling language based on first-order relational logic. It enables designers to specify the structure and constraints of a system in terms of sets and relations, using a concise and expressive syntax. Developed by Daniel Jackson at Massachusetts Institute of Technology (MIT), Alloy supports the creation of small, analyzable models that help reveal design flaws early in the development process.

Complementing the language, the Alloy Analyzer is the tool that performs BMC: it automatically translates Alloy specifications into relational logical formulas via Kodkod [68], which in turn translates them into propositional formulas solvable by SAT solvers and searches for instances that satisfy the constraints (or counterexamples that do not) within a user-defined scope. Despite its bounded nature, this approach is useful in practice due to the so-called small-scope hypothesis, which holds that most specification errors occur in small instances. Alternatively, if the user has `NuSMV` [69] or `nuXmv` [70] installed, Alloy 6 supports unbounded model checking. The Analyzer also includes an interactive visualizer that displays generated instances as intuitive graphs, helping users explore and understand the system behavior. With the introduction of Alloy 6, the tool expanded to support mutable state and temporal properties, enabling the modeling and analysis of dynamic systems.

Alloy provides a Graphical User Interface (GUI) that enables users to specify the model and its intended properties in a specific format. The GUI provides various settings that the user can change, with the most relevant ones being the selection of the solver and the choice of which commands to execute. The default solver is `SAT4J`, but for the HyperLTL extension of Alloy (see Section 3.1.1), a different solver is required. Alloy allows executing any individual command or all of them.

This dissertation does not aim to provide a complete presentation of the Alloy language. Instead, Alloy is considered primarily from a tooling perspective: it is the high-level modeling environment in which specifications are written and analyzed, and the results of that analysis are visualized. To illustrate the parts of Alloy that are most relevant for this work, Listing 3.1 presents a small example of an alloy model and a property. The example models a simple machine that can receive requests, be refilled, and produce coffee or tea. It also defines a mutated version of the same machine and checks whether the mutant can eventually be distinguished from the original under the same inputs.

Listing 3.1: Example of an Alloy model and property

```
sig Machine {
  var water : one Int,
  var input : lone Input,
  var output : lone Output
}
enum Output { Coffee, Tea }
enum Input { Req, Fill }
pred machine[m:Machine] {
  no m.output
  no m.input
  m.water = 2
  always {
    m.input = Req and m.water > 0 implies (some m.output' and m.water' = m
    .water.minus[1])
    not (m.input = Req and m.water > 0) implies no m.output'
    m.input = Fill and m.water = 0 implies m.water' = 2
    not (m.input = Fill and m.water = 0) and
    not (m.input = Req and m.water > 0) implies
      m.water' = m.water
  }
}
pred mutant[m:Machine] {
  no m.output
  no m.input
  m.water = 2
  always {
```

```
26      m.input = Req and m.water > 0 implies (some m.output' and m.water' = m
        .water.minus[1])
27      not (m.input = Req and m.water > 0) implies no m.output'
28      m.input = Fill and m.water = 0 implies m.water' in 1 + 2 // Difference
29      not (m.input = Fill and m.water = 0) and
30      not (m.input = Req and m.water > 0) implies
31        m.water' = m.water
32    }
33  }
34  run PotentiallyKillableNonHyper {
35    some disj mut, original: Machine |
36      mutant[mut] and
37      machine[original] and
38      always (mut.input = original.input) and
39      eventually not (mut.output = original.output)
40  } for 10 steps, exactly 2 Machine, 3 Int
```

The declaration `sig Machine` introduces a signature, which can be defined as a set of atoms. Its fields introduce relations between signatures. In this case, `water`, `input`, and `output` are mutable fields, indicated by the keyword `var`, meaning their values may change from one state to the next. The multiplicities `one` and `lone` restrict the number of values associated with each field: `one Int` means that each machine has exactly one integer value for `water`, while `lone Input` and `lone Output` mean that the machine may have either zero or one input/output at each state.

The declarations `enum Output` and `enum Input` define finite sets of atoms. In this example, the possible outputs are `Coffee` and `Tea`, while the possible inputs are `Req` and `Fill`.

The declaration `pred machine[m: Machine]` introduces a predicate, which can be defined as a reusable constraint that describes a condition or behavior. The first set of constraints describes the initial state: there is no input or output, and the water level is `2`. The temporal operator `always` then specifies constraints that must hold in every state of the trace. Inside this block, the prime operator `'` is used to refer to the value of a mutable field in the next state.

The predicate `mutant` is similar to `machine`, but introduces a deliberate behavioral difference. When the mutant is refilled from zero, its next water level may be either `1` or `2`, expressed by `m.water' in 1 + 2`. Here, + denotes set union, and `in` checks membership or inclusion.

The `run` command asks Alloy to search for an instance satisfying the predicate `PotentiallyKillableNonHyper`. This command introduces two distinct machines, using `some disj mut, original: Machine`. The keyword `some` is an existential quantifier, requiring such machines to exist, while `disj` requires them to be distinct. The command then constrains one machine to follow the mutant behavior and the other to follow the original behavior. It also requires that they always receive the same input and that their outputs eventually differ. The scope at the end of the command bounds the analysis. The command is executed for `10 steps`, with exactly two machines and an integer bitwidth of `3`. Alloy searches only within the finite space determined by these limits.

Table 3.1 describes the code blocks available within Alloy succinctly. For more in-depth details on Alloy, refer to Daniel Jackson's book [71], and for more details on Alloy 6/temporal operators, refer to the practical Alloy guide [72].

Alloy Block	**Description**
`sig`	Signature, a set of atoms
`fact`	Fact, a global constraint that must always hold
`pred`	Predicate, a reusable constraint that describes a condition or behavior
`fun`	Function, a reusable expression that returns a relation
`assert`	Assertion, a property that should hold in all valid instances
`check`	Check, a command to verify whether an `assert` can be falsified within a given scope
`run`	Run, a command to find an instance that satisfies a `pred`

Table 3.1: Alloy blocks

3.1.1 HyperLTL Extension of Alloy

Since Alloy lacks support for reasoning about hyperproperties, a prototype HyperLTL extension of Alloy [5] was developed to tackle some of its limitations. It extends Alloy to support the specification and verification of hyperproperties, utilizing HyperQube as its backend model checker for HyperLTL. It has some additional keywords to allow for verification of hyperproperties, like:

- `hyperltl`: defines hyperproperties.
- `forall`: universal quantifier.
- `exists`: existential quantifier
- `A`, `B`, `C`, `D`,...: used to identify the traces associated to each quantifier. The user needs to create a signature `sig Traces {}` and, when specifying each trace, must extend `Traces`.
- `verify`: command to verify an hyperproperty.

Despite verifying the hyperproperties with HyperQube, the only output is the HyperQube output (see section 3.3.2.3), which is not easily readable. Furthermore, it would be ideal if the output were returned to Alloy so it could use its visualization tool to display the examples and counterexamples (see Figures 6.1 and 6.2 and the rest of Chapter 6, for examples).

3.2 Electrod

Electrod [73] is the backend used in the Electrum Analyzer [74], an expansion of the Alloy Analyzer that supports temporal logic operators. This expansion serves as the basis for the latest version of Alloy, Alloy 6, which integrates support for temporal logic operators. Electrod is particularly relevant to this work because it provides the translation from Alloy models to the low-level

symbolic model verifier (SMV) representation that can be processed by model checkers, a capability leveraged by the proposed approach. Electrod is a model finder inspired by Kodkod that takes in models with bounded domains and an unbounded time horizon, being expressed via a mixture of relational first-order logic and LTL. The model is then translated into an SMV file for evaluation by low-level solvers, such as NuSMV and nuXmv.

3.2.1 Usage

Electrod was intended to be launched from within Electrum or similar tools, but it can still be run as a standalone command-line tool. To run Electrod, you must install either NuSMV or nuXmv; at least one must be installed on the system. Its usage follows the following format:

```
electrod <options> <eloFile>
```

where <eloFile> is the path to the file that will be passed to Electrod. Elo files are representations of the model supported by Electrod, produced by the Analyzer; the options available for Electrod are the following:

- -t: the analysis tool to rely upon, either nuXmv (default) or NuSMV.
- -bmc: stands for "bounded model checking" and, when followed by a positive integer, sets the bound on the number of trace steps.
- -s: path to the script file that will be passed to the analysis tool.
- -kg: keep the generated model and script files.
- -na: generate files but perform no analysis.
- -pg: print the generated files to the standard output.
- -ln: use long names for relations and atoms in the generated file.
- -of: outcome format of the analysis, either chrono (default), plain or xml.
- -ts: generate the full temporal symmetry-breaking predicate (otherwise, a single-state symmetry predicate is used).
- -so: followed by a positive integer, sets the offset of the single-state symmetry-breaking predicate. Only used if -ts is not selected. The default is 0.
- -sf: generate a single formula as the specification for the underlying solver.

3.2.2 Format

Electrod takes in `Elo` files that follow the structure in Listing 3.2, where `univ` is the universe of atoms and each `<atom>` is represented as `<sig>#<index>`, with `<sig>` corresponding to the originating Alloy/Electrum signature and `<index>` is a natural number uniquely identifying the atom within that signature. The universe is followed by a list of constants and variables. Constants are time-invariant relations, and variables are time-varying relations. Constants do not need to precede variables. The `<relName>` is the name of that relation and is a string, `<arity>` is a positive integer, and `<tuple>` is a tuple of atoms with length equal to `arity`, written as `( <atom> ... <atom> )`.

Listing 3.2: Format of an `Elo` file

```
1  univ : { <atom> ... <atom> };
2
3  const <relName> :<arity> { <tuple> ... <tuple> };
4  ...
5  const <relName> :<arity> { <tuple> ... <tuple> };
6
7  var <relName> :<arity> { <tuple> ... <tuple> } ... { <tuple> ... <tuple> };
8  ...
9  var <relName> :<arity> { <tuple> ... <tuple> } { <tuple> ... <tuple> };
10
11 run
12 <constraint>;
13 ...
14 <constraint>;
```

Each `<constraint>` can be either a structural or a temporal constraint and follow the format used in Alloy. The constraint `true` is used to denote the end of that listing.

Electrod outputs `SMV` files, following the structure described in Section 3.3.1.2, specifically for `NuSMV` systems.

3.3 HyperLTL Model Checkers

HyperLTL model checkers verify whether models satisfy the specified HyperLTL properties. This section provides an in-depth explanation of how AutoHyper and HyperQube are utilized. They have already been briefly introduced in Sections 2.2.3.2 and 2.2.3.3, respectively. Sections 3.3.1 and 3.3.2 will present a more technical overview, in particular their user interface and input and output formats. Understanding these aspects is important because these HyperLTL model checkers will serve as components of the proposed solution, making it necessary to understand their interfaces and formats.

3.3.1 AutoHyper

AutoHyper supports the input systems represented as a subset of `NuSMV` models, explicit-state systems, and Boolean programs.

3.3.1.1 Usage

The general usage follows the following format:

```
AutoHyper <options> <systemPaths> <formulaPath>
```

where `<options>` are the command line options, `<systemPaths>` are the file paths to the systems about which the properties will be verified, and `<formulaPath>` is the file path to the HyperLTL formula representing the hyperproperty. The number of `<systemPaths>` should match the number of quantifier prefixes in the HyperLTL property.

Depending on the system type, different flags must be passed. The default option, `-nusmv`, is used for `NuSMV` models represented in a restricted subset of the SMV language. These models define the system through variables and transition expressions, and AutoHyper only supports single-module systems. Other systems can be converted into this form using the `NuSMV` toolchain [62]. The `-explicit` flag is used when states, transitions, and variable valuations are given directly, while `-bp` is used for Boolean programs, where the system is described through Boolean variables and Boolean operations.

Another set of command-line options defines the model-checking mode. AutoHyper supports different strategies, but only one of these flags can be used at a time. The `-comp` flag uses automaton complementation, while the remaining options use language-inclusion checks. The default option, `-incl-spot`, uses Spot [75], a C++17 library for LTL, ω-automata manipulation, and model checking. AutoHyper also supports inclusion checks through external tools, namely RABIT with `-incl-rabit` [76], BAIT with `-incl-bait` [77], and FORKLIFT with `-incl-forklift` [78], each requiring a working copy of the corresponding tool and a valid path to its `.jar` file in `paths.json`. Finally, `-incl-forq` uses the FORQ-based inclusion check implemented in Spot version 2.12 or later, and should be preferred over `-incl-forklift` when applicable.

The remaining options are not mutually exclusive. By default, AutoHyper computes a bisimulation quotient for each system, reducing its size by merging states with equivalent behavior; this optimization can be disabled with `-no-bisim`. The `-witness` option instructs AutoHyper to compute witness paths for the outermost quantifier block. If the formula starts with an existential prefix ($\exists^*$) and is satisfied, AutoHyper prints lasso-shaped witness traces for the existential quantifiers; if it starts with a universal prefix ($\forall^*$) and is not satisfied, it prints counterexample traces for the universal quantifiers. This option also enables `-comp` and disables `-no-bisim`. The `-simplification` option simplifies the automaton at each step using Spot, while `-log` enables verbose logging and more detailed error messages. Finally, `-version`, `-license`, and `-help` display the version, license information, and help message, respectively.

3.3.1.2 Input Format

Depending on the chosen type of system representation, its structure varies accordingly.

HyperLTL Formulas The abstract syntax of HyperLTL formulas was described in Section 2.2.1. The concrete syntax in AutoHyper is the following:

`<qfPrefix> <body>`

where the quantifier prefix `<qfPrefix>` can be the empty string, universal quantification (`forall <tvar>. <qfPrefix>`), or existential quantification (`exists <tvar>. <qfPrefix>`). Here, `<tvar>` is a trace variable. The `<body>` of a formula can be the Boolean constants `1` and `0`, representing true and false, respectively; an atomic expression written as `{<atomicExpression>}`, which may be a Boolean constant, a Boolean variable, or an arithmetic comparison between variables or integer constants; a parenthesized formula `(<body>)`; a binary operation `<body> <bopp> <body>`; or a unary operation `<uopp> <body>`. Variables inside atomic expressions are written in the format `"<varName>"_<tvar>`. The supported binary operators are conjunction `&`, disjunction `|`, implication `->`, equivalence `<->`, until `U`, weak until `W`, and release `R`. The supported unary operators are negation `!`, next `X`, globally `G`, and eventually `F`.

Explicit-State Systems The syntax of explicit-state-systems follows the structure presented in Listing 3.3.

Listing 3.3: Format of an explicit-state system

```
1 Variables: ("<var>" <type>) ... ("<var>" <type>)
2 Init: <stateID> ... <stateID>
3 --BODY--
4 State: <stateID> {("<var>" <val>) ... ("<var>" <val>)}
5 <stateID> ... <stateID>
6 ...
7 State: <stateID> {("<var>" <val>) ... ("<var>" <val>)}
8 <stateID> ... <stateID>
9 --END--
```

where `<var>` is a variable name, which can be any non-reserved string consisting of letters, digits, and _; `<type>` is the variable type, either `Int` or `Bool`; `<stateID>` is a natural number used to identify a state; and `<val>` is the value assigned to a variable in a state, either `true`, `false`, or an integer `<n>`.

The initial declaration defines all variables in the system and assigns a type to each. Then it declares the initial states. Each state declares the evaluation of all the variables and their successor states. Listing 3.4 is an example of an explicit state system:

Listing 3.4: Example of an explicit-state system

```
Variables: ("x" Int) ("y" Bool)
Init: 0 1
--BODY--
State: 0 {("x" 3) ("y" true)}
0 2 3
State: 1 {("x" 2) ("y" true)}
0 1 2
State: 2 {("x" 15) ("y" false)}
0 2 3
State: 3 {("x" 1) ("y" true)}
2 3
--END--
```

The HyperLTL formulas used alongside explicit-state systems use the variables declared in the `Variables:` line. Listing 3.5 is an example of a property that holds for the previous system:

Listing 3.5: Example of a property for an explicit-state system

```
forall A. exists B. G {"x"_A = "x"_B} & F {"y"_B}
```

NuSMV Transition Systems The syntax of NuSMV transition systems follows the structure presented in Listing 3.6.

Listing 3.6: Format of a NuSMV transition system

```
MODULE main
VAR
    <varName> : <type>;
    ...
    <varName> : <type>;

ASSIGN
    init(<varName>) := <expression>;
    ...
    next(<varName>) := <expression>;
    ...

DEFINE
    <varName> := <expression>;
    ...
    <varName> := <expression>;
```

where the model may contain several `ASSIGN` and `DEFINE` blocks, and

`<varName>` is a variable name, which can be any non-reserved string starting with a letter or _, followed by letters, digits, or the special characters _, `$`, `#`, `-`, `[`, `]`, and `..`

`<type>` is the type of the variable. The supported types are `boolean`, whose values are `TRUE` and `FALSE`; set types of the form `{n_1, ..., n_M}`, whose values are natural numbers contained in the set; ranges of the form `l..h`, whose values are integers between `l` and `h`, with `l <= h`; and arrays of the form `array l..h of <type>`, whose indices range from `l` to `h`, with `l <= h`, and whose elements have type `<type>`.

`<expression>` is any supported `NuSMV` expression. This includes the Boolean constants `TRUE` and `FALSE`; integers `<n>`; references to declared or defined variables `<varName>` in the current state; conversions of the form `toInt(<expression>)`, which returns `1` if the expression is `TRUE` and `0` otherwise, and `toBool(<expression>)`, which returns `FALSE` if the expression is `0` and `TRUE` otherwise; case expressions of the form `case <expression>: <expression>; ... <expression>: <expression>;`; set expressions of the form `{<expression>, ..., <expression>}`; binary operations `<expression> <opp> <expression>`, where `<opp>` can be `&`, `|`, `->`, `=`, `!=`, `<=`, `>=`, `<`, `>`, `+`, or `-`; and unary negation `! <expression>`.

Listing 3.7 is an example of a `NuSMV` transition system:

Listing 3.7: Example of a `NuSMV` transistion system

```
MODULE main
VAR
    mutation: boolean;
    action: 0..2;
    beverage: 0..2;
    water: 0..3;

ASSIGN
    init(mutation) := {TRUE, FALSE};
    next(mutation) := {TRUE, FALSE};

    init(action) := 0;
    next(action) := {0,1,2};

    init(beverage) := 0;
    next(beverage) :=
        case
            (action=1 & !(water=0)): 1;
            TRUE: beverage;
```

```
        esac;

    init(water) := 2;
    next(water) :=
        case
            (action=1 & !(water=0)): water - 1; -- make beverage
            (mutation & water=0): 1;
            (mutation & water=1): 2;
            (mutation & water=2): 3;
            (mutation & water=3): 3;
            (!mutation & water=0): 2;
            (!mutation & water=1): 3;
            (!mutation & water=2): 3;
            (!mutation & water=3): 3;
            TRUE: water;
        esac;

DEFINE
    NO_water := (action=1) & (water=0);
    NO_output := (action!=1)  | ((action=1)&(beverage=0));
```

The HyperLTL formulas used alongside NuSMV transition systems use variables declared in variable declaration blocks or defined in definition blocks. Listing 3.8 is an example of a property for the system from Listing 3.7.

Listing 3.8: Example of a property for a NuSMV transistion system

```
1 forall A. exists B.
2 (
3     ({"action"_A = "action"_B}) U ({"NO_water"_B})
4 )
```

AutoHyper Boolean Programs The syntax of boolean programs for AutoHyper follows the structure presented in Listing 3.9.

Listing 3.9: Format of a Boolean program

```
1 <varName> : <bitwidth>;
2 ...
3 <varName> : <bitwidth>;
4 <statement>
```

In this syntax, `<varName>` is any non-empty (non-reserved) sequence of letters starting with a letter, and `<bitwidth>` is a natural number that defines the bitwidth of each variable. A `<statement>` can be an assignment `<varName> = <expression>;`, which assigns the value of `<expression>` to `<varName>`; a nondeterministic assignment `<varName> = *;`; a conditional statement `if <expression> {<statement>} else {<statement>}`, which branches depending on whether `<expression>` evaluates to `[t]`; a sequence `<statement> <statement> ... <statement>`, which executes all statements sequentially; a loop `while <expression> {<statement>}`, which executes `<statement>` as long as `<expression>` evaluates to `[t]`; or a nondeterministic branch `if * {<statement>} else {<statement>}`.

An `<expression>` can be a variable reference `<varName>`, representing the value currently bound to the variable with that name; the Boolean constants `t` and `f`; a pointwise conjunction `<expression> & <expression>`; a pointwise disjunction `<expression> | <expression>`; a pointwise negation `! <expression>`; a bit selection `<expression>[l, u]`, representing the bits ranging from position `l` to position `u` of `<expression>`, where `l` and `u` are natural numbers and `<expression>[i]` is the same as `<expression>[i, i]`; or a repeated concatenation `(i * <expression>)`, which concatenates the evaluated `<expression>` `i` times.

Listing 3.10 is an example of a Boolean program.

Listing 3.10: Example of a Boolean program

```
h : 1;
l : 1;
o : 1;

o = 1 * true;
while(true) {
    h = *;
    if (h[0]) {
        o = !o;
    } else {
        o = (!o) & (h | !h);
    }
}
```

The HyperLTL formulas used alongside boolean programs have the form `<varName>@<n>`, where `<varName>` is the name of a variable and `<n>` is the boolean variable at the **<n>**th position of the vector assigned to `<varName>`. Listing 3.11 is an example of a property for the system from Listing 3.10.

Listing 3.11: Example of a property for Boolean program

```
forall A. forall B. exists C. (G ({"h@0"_A} <-> {"h@0"_C})) & (G({"l@0"_B} <-> {"
    l@0"_C}))
```

3.3.1.3 Output

The output from AutoHyper is printed to the terminal in the format:

Listing 3.12: AutoHyper Witness

```
======= Witnesses =======
<trace>: ({"<varName>":<value>, ..., "<varName>":<value>} ... {"<varName>":<value>,
     ..., "<varName>":<value>}) ({"<varName>":<value>, ..., "<varName>":<value>}
    ... {"<varName>":<value>, ..., "<varName>":<value>})
...
<trace>: ({"<varName>":<value>, ..., "<varName>":<value>} ... {"<varName>":<value>,
     ..., "<varName>":<value>}) ({"<varName>":<value>, ..., "<varName>":<value>}
    ... {"<varName>":<value>, ..., "<varName>":<value>})
=========================

<satisfiability>
```

when it *satisfies* a formula with an outermost *existential* quantifier (witness) or *dissatisfies* a formula with an outermost *universal* quantifier (counterexample). In the opposite cases, the format is just:

```
<satisfiability>
```

Here, `<trace>` denotes each trace specified in the hyper formula; `<varName>` is the name of each variable; `<value>` is an integer representing the value of the respective variable; and `<satisfiability>` indicates whether the property was satisfied, with `SAT`, or not, with `UNSAT`.

In order to save the output to a file, `> <filePath>` can be added to the end of the command.

3.3.2 HyperQube

HyperQube supports the input systems represented as `NuSMV` models (with most of its features). The HyperLTL formulas must adhere to HyperQube's grammar.

3.3.2.1 Usage

The general usage follows the following format:

```
hyperqb <list of models> <formula> <k> <sem> <mode>
```

where <list of models> is a list of paths to files written in NuSMV format, <formula> is a path to a file written in HyperQube's own grammar for HyperLTL formulas, <k> is a natural number specifying the unrolling bound, <sem> represents the semantics to use, and <mode> is the mode of performing BMC. The supported semantics are -pes, -opt, -hpes, and -hopt: according to Hsu [20], pessimistic semantics are false unless truth is witnessed within the bound, optimistic semantics are true unless falsity is witnessed within the bound, and the halting variants apply the same interpretation before the program halts. The mode can be either -bughunt, which negates the formula and is the default, or -find, which checks the original formula.

3.3.2.2 Input

The system representation format for HyperQube is the same as the format of the NuSMV transition systems from AutoHyper (Section 3.3.1.2).

The HyperLTL formulas for HyperQube are similar to the ones accepted by AutoHyper, with a few key differences: variables are written in the format <vid>[<tid>], conjunction and disjunction are written as /
and
/, respectively, negation is written as , and arithmetic comparisons require asterisks around the expression, in the format *<vid>[<tid>] <arith_comp> <vid>[<tid>]*. Here, <vid> is a variable ID, <tid> is a trace ID, and <arith_comp> is an arithmetic comparison operator.

3.3.2.3 Output

The output from HyperQube is split into two folders: build_today and build_cex (for counterexamples). In the current version, build_cex is always empty, but it is part of their future work to display solver output in a nicely formatted manner. The files are generated inside the build_today folder.

First, there are the I_1.bool, I_2.bool, ..., I_N.bool (for initial) for each of the N models passed as input. They represent the initial state of each trace. Listing 3.13 is an example of one I_N.bool file.

Listing 3.13: I_N.bool file example

```
1 (a/\b/\c)
```

Alongside the files with the initial state are the files R_1.bool, R_2.bool, ..., R_N.bool (for relation), one for each of the N models passed as input. They represent the transition relations for each trace. Listing 3.14 is an example of one R_N.bool file.

Listing 3.14: R_N.bool file example

```
(((a/\b/\c))->(((a'/\~b'/\~c')\/(~a'/\~b'/\~c'))))/\
(((a/\~b/\~c))->(((a'/\b'/\c')\/(~a'/\b'/\c'))))/\
(((~a/\~b/\~c))->(((a'/\b'/\c')\/(~a'/\b'/\c'))))/\
(((~a/\b/\c))->(((a'/\~b'/\~c')\/(~a'/\~b'/\~c'))))
```

Another file present in the folder is `P.hq` (for property). It represents the hyperproperty being evaluated. Listing 3.15 is an example of a `P.hq` file.

Listing 3.15: P.hq file example

```
F(a_A)
```

Another file present in the folder is `QS` (for quantifier structure). It represents the property's quantifier structure. Its format is represented in Listing 3.16.

Listing 3.16: QS file format

```
QS=<qf>...<qf>
```

where `<qf>` is one of `E` (existential quantifier) or `A` (universal quantifier. It depends on the quantifier in the formula, which must match the number of models input.

Finally, the two most important files for future interpretation of the outputs are `HQ.qcir` and `HQ.quabs`. QCIR [79] is an intermediate format to represent QBF problems, which will be passed to the QUABS [80] solver that returns the `HQ.quabs` file.

`HQ.qcir` follows the structure in Listing 3.17.

Listing 3.17: HQ.qcir file format

```
#QCIR-G14
<qf>(<var>, ..., <var>)
<qf>(<var>, ..., <var>)
output(<var>)
<var> = <op>(<var>,...<var>)
...
<var> = <op>(<var>,...<var>)
# <var> : <varName>_<trace>[<state>]
...
# <var> : <varName>_<trace>[<state>]
```

where `qf` is the quantifier (either `exists` or `forall`), `<var>` is an integer that represents a `QCIR` variable, `op` is a boolean operation (either `or` or `and`), `<varName>` is the original name of the variable, `trace` is the trace name and `<state>` is an integer that represents the state number.

In some cases, the original value was an integer, and, to be represented with Boolean variables, it needed to be split into multiple Boolean variables corresponding to the bits of its binary encoding, requiring $\lceil \log_2(n) \rceil$ variables. In these cases, `<varName>` is of the form:

`<varNameBase>_<index>`

where `<varNameBase>` is the actual name of the variable and `<index>` identifies the corresponding bit of the binary encoding, ranging from the least to the most significant bit. The commented lines (starting with `#` map the `QCIR` variables to their original counterpart, indicating their variable name, and to which state and to which trace they refer. Each is assigned to a `QCIR` variable that is used above and is related to a quantifier at the beginning of the file. Then, a list of Boolean operations is performed following the unfolding of `R_N.bool`, assigning the values to intermediate variables that are used in the following operations. The last operation assigns its value to the output variable that matches the variable in the `output(<var>)` line.

`HQ.quabs` follows the structure in Listing 3.18.

Listing 3.18: HQ.quabs file format

```
1 V <-?><var> ... <-?><var> 0
2 r <sat>
```

where `<var>` is the variable number matching the `QCIR` variable, `-?` is an optional `-` that indicates that a variable is negative; if a variable is not preceded by a `-`, then it is positive. `<sat>` is the satisfiability, `SAT` or `UNSAT`.

3.3.3 Discussion

The analysis of AutoHyper and HyperQube reveals their differences and the shared challenges in integrating them into a broader verification pipeline. A first distinction comes in the type of models accepted by each tool. AutoHyper expects a restricted subset of SMV in which the evolution of each variable must be described directly and explicitly. This is more restrictive than allowing transitions to be described declaratively, through constraints over possible current and next states. Since Alloy models are declarative, translating them to AutoHyper's accepted format is more demanding. HyperQube expects a declarative SMV transition system and is therefore closer in style to transition descriptions produced by Alloy, while also supporting more explicit SMV transition descriptions. Supporting multiple backends, therefore, requires handling variations in accepted formats. Another distinction is also a common point between them: both tools produce outputs that differ from their inputs, requiring additional processing to reconstruct traces and enable visualization. Furthermore, both operate at a much lower level of abstraction than Alloy, focusing on Boolean variables and transition systems rather than data structures and first-order operators. This highlights the gap between high-level specification and low-level hyperproperty verification,

motivating the need for an intermediate architecture to translate and interpret information between these layers.

Chapter 4

Solution Design

Analyzing and experimenting with state-of-the-art tools for the automated verification of hyperproperties revealed gaps in these tools. One of the major problems is how counterexamples are presented back to the user. Higher-level tools, like regular Alloy, handle these representations more effectively, but its HyperLTL extension does not currently support the high-level representation of those counterexamples. The solution designed in this chapter builds on the current version of the HyperLTL extension of Alloy by integrating hyperproperty verification not only through HyperQube but also through AutoHyper. Additionally, it returns the counterexamples produced by the low-level solvers to the Alloy visualizer. The implementation of the proposed solution is available in a fork of the already existing HyperLTL extension of Alloy [81].

4.1 Architecture Overview

The architecture of this extension to the Alloy ecosystem is presented in Figure 4.1. In the Alloy Analyzer, the user specifies the model and hyperproperties, as in the current version of the Alloy HyperLTL extension. When the user executes a verification command, it uses an altered version of Electrod to generate intermediary SMV files that could, in principle, already be consumed by the low-level solvers. However, these solvers are not optimized to handle the declarative models generated by Alloy (and AutoHyper does not support declarative SMV at all). Therefore, an intermediate processing step is required: an additional component, HyperSMV, rewrites the generated SMV encoding into a representation better suited to the target solvers. At the lowest level of this architecture, the HyperLTL solvers (currently AutoHyper or HyperQube, but could eventually be any solver that supports standard SMV) will perform model checking on the rewritten encoding. Their outputs are then processed by Inst2SMV, a new component introduced in this architecture. Inst2SMV translates the counterexamples produced by the solvers into SMV instance files, which are then sent back to the Alloy Analyzer via Electrod for visualization.

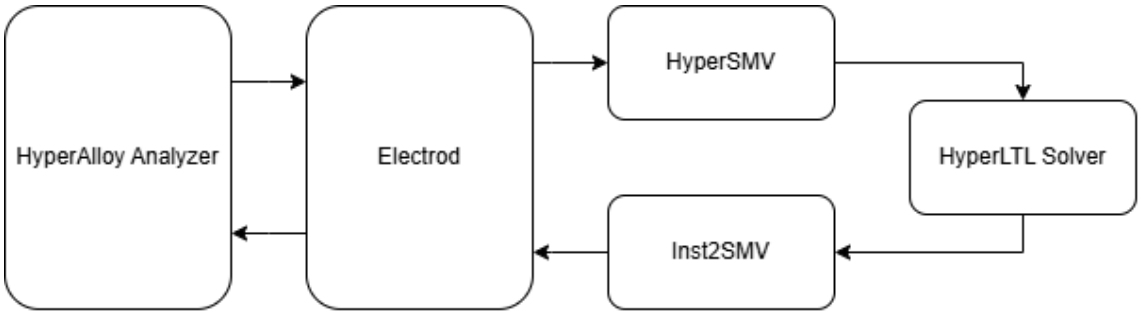


Figure 4.1: Architecture of the solution

4.2 Scope of the Solution

To design the solution, it was first necessary to understand the tools upon which it would be built. This process was detailed in Chapter 3, where AutoHyper and HyperQube were chosen as the low-level solvers, with outputs that could be utilized in the subsequent steps, and Alloy and its extension were studied to understand how they function and how they could be improved, with the inclusion of these other solvers and a visual representation of their output. After these considerations, the architecture illustrated in Figure 4.1 was proposed. It should be noted that the objective of this dissertation was not to develop all components of the pipeline from scratch. Several elements already existed, including the HyperLTL model checkers and the Alloy framework. HyperSMV was not developed for this dissertation, but it was integrated into the pipeline. Instead, the main contribution centers on the design of the architecture that connects these heterogeneous tools, the modification of existing components, and the development of the missing elements needed to support the entire verification workflow, including solver output sanitation and multiple-trace visualization. The following subsections present each segment of the solution in the order of execution. The Table 4.1 summarizes the input and output from each component of the pipeline, listing each component together with the artifacts it consumes and produces. The downward (↓) and upward (↑) arrows distinguish the forward verification flow, from Alloy to the HyperLTL model checkers, from the return flow, where counterexamples are reconstructed and passed back to the Alloy visualization engine.

4.2.1 Model Representation within HyperLTL extension for Alloy

The representation of models within the HyperLTL extension for Alloy was detailed in Section 3.1.1. Unlike the backend HyperLTL model checkers, where the system model and the property are provided as separate inputs and formats, the Alloy extension expresses both the model and the hyperproperty within the same `.als` specification. As a result, the translation process must separate them from a unified representation before they can be processed by the subsequent tools in the pipeline. The Alloy Analyzer in this solution interprets the model defined in the `.als` file and translates it into `ELO` files. The quantified trace variables determine how many model representations must be generated, with each trace being translated independently into its own `ELO` file.

Component	**Input**	**Output**
Alloy ↓	Alloy Model and Hyperproperties (`.als`)	Trace `ELO` Hyperproperty `ELO`
Electrod ↓	Trace `ELO` Hyperproperty `ELO`	Raw Trace `SMV` `output.info` LTLSPEC `SMV`
Alloy ↓	LTLSPEC `SMV`	Hyperproperty (`.hp`)
HyperSMV (AH)	Raw Trace `SMV` Hyperproperty (`.hp`)	Optimized Trace `SMV` (`.exp`) Hyperproperty Formula (`.ah`) Variable Mapping (`.smv.names`)
HyperSMV (HQ)	Raw Trace `SMV` Hyperproperty (`.hp`)	Optimized Trace `SMV` (`.smv`) Hyperproperty Formula (`.hq`) Variable Mapping (`.smv.names`)
AutoHyper	Optimized Trace `SMV` (`.exp`) Hyperproperty Formula (`.ah`)	Satisfiability Possible Counterexample (`cex.ah`)
HyperQube	Optimized Trace `SMV` (`.smv`) Hyperproperty Formula (`.hq`)	Satisfiability Boolean Encoding of the Problem Possible Counterexample (`HQ.qcir`, `HQ.quabs`, `P.hq`, `QS`, `I.bool`, `R.bool`)
Inst2SMV (AH)	Optimized Trace `SMV` (`.exp`) Counterexample (`cex.ah`) Variable Mapping (`.smv.names`)	Sanitized Output (`.out`)
Inst2SMV (HQ)	Counterexample (`HQ.qcir`, `HQ.quabs`) Variable Mapping (`.smv.names`)	Sanitized Output (`.out`)
Electrod ↑	Sanitized Output (`.out`) `output.info`	Electrod `XML`
Alloy ↑	Electrod `XML`	Viz `XML` (`.cnf.xml`)

Table 4.1: Summary of the components in the proposed pipeline and their input and output formats

In addition to those, the hyperproperty expression is extracted separately and translated into an extra ELO file, preserving the association between the formula and the trace variables it refers to. These generated ELO files then become the inputs to Electrod. The trace model files are used to produce the corresponding backend-level transition systems, while the formula file is handled separately so that the hyperproperty can later be reconstructed in the format required by the selected HyperLTL model checker.

4.2.2 ELO to SMV Translation with Electrod

Once each trace is translated to an ELO file, Electrod generates an SMV file for each trace, which consists of a finite-state machine, whose format was described in Section 3.3.1.2. Since Alloy models are declarative, not all constraints can always be encoded directly in the generated SMV finite-state machine. Consequently, Electrod may generate additional LTLSPEC constraints associated with each trace to preserve the semantics of the original Alloy specification. In addition to each SMV file, an .info file is also generated with traceability information. In the original usage of Electrod, it would call the solvers and process their outputs. Since in this pipeline Electrod's forward translation and result-processing phases are split to allow intermediary tools to run between them, variable names and context may be lost due to future optimizations of the SMV files. To reinterpret the information resulting from low-level HyperLTL model checkers, this context is recorded in these .info files to be used later to trace this information. For the ELO file describing the hyperproperty, a different SMV file is generated containing an LTLSPEC, corresponding to an LTL property expressed in SMV. This step reuses the existing Alloy–to–SMV translator for non-hyper LTLSPECs. The resulting LTLSPEC is then sent back to the Alloy Analyzer, which, having access to the original hyperproperty context (namely the quantified trace variables and their occurrences in the hyperformula), reinterprets the specification as a hyperproperty and translates it into a .hp file. The additional LTLSPEC constraints generated for the individual traces are also incorporated into this final hyperproperty specification. This ensures that the backend HyperLTL model checkers reason only about behaviors that satisfy the original Alloy model. This .hp file is subsequently converted into the low-level hyperproperty representation required by the backend HyperLTL model checkers.

4.2.3 Input Sanitation with HyperSMV

HyperSMV is a tool used in this part of the pipeline to optimize the SMV files that will serve as input to the lower-level solvers. HyperLTL model checking can take a long time for large models; therefore, it is crucial that the models arrive in their most simplified form possible. In this process, variable names are simplified, and this mapping is stored in .names files. Trace representations as SMV files are translated to SMV, transition systems, or .exp, explicit state systems, representing the models to be used by HyperQube and AutoHyper, respectively. The LTLSPEC translated to an .hp file is also taken by HyperSMV to generate .ah or .hq files, depending on whether the

verification will be performed by AutoHyper or HyperQube. These files contain a representation of the hyperproperty for evaluation by the low-level solver.

4.2.4 Verification with HyperLTL Solvers

The set of `.exp` and `.ah` or `SMV` and `.hq` files are then passed to the corresponding low-level HyperLTL model checker, which verifies whether the property is satisfied or unsatisfied. For satisfied existential properties and unsatisfied universal properties, an example or a counterexample is returned, containing the representation of the traces. Neither solver provides these examples and counterexamples for the input `SMV` model. Rather, each solver produces witnesses in its own internal format, tied to solver-specific encodings and abstraction levels. These formats were described in detail in Sections 3.3.1.3 and 3.3.2.3.

4.2.5 Output Sanitation with Inst2SMV

The outputs produced by HyperLTL model checkers are not directly intelligible to users nor compatible with the format expected by Electrod. In particular, HyperQube ultimately returns the result of a QBF solver, in which traces are represented using low-level Boolean variables that no longer preserve the structure of the original model. To pass these outputs back to Electrod for transmission to Alloy, these traces need to be converted to the same `SMV` format that Electrod can interpret. In case of an unsatisfied existential property or a satisfied universal property, only UNSAT/SAT needs to be transmitted. In the opposite examples, the traces in the outputs must be sanitized to meet the `SMV` format. The `.names` file will be used here to trace back the variable names. This process will be detailed in Section 5.1.

4.2.6 SMV to XML Translation with Electrod

This step is very similar to the one performed at this stage by regular Alloy. Electrod takes the `SMV` instances and the `.info` files and converts them into an Electrod `eXtensible Markup Language (XML)` file that Alloy can translate to a `.cnf.xml` file, detailing all the information about that trace: the number of states, where it loops, and variable information in each step.

4.2.7 Witness and Counterexample Visualization

The original Alloy Analyzer contains a trace visualizer that already displays a single trace from one of these `XML` files. But now there will be a trace for each invalid outermost existential quantifier in the hyperproperty, resulting in multiple `XML` files for visualization. Each trace, in theory, may have a varying number of states and different looping segment lengths. To support these differences, it will be necessary to extend the existing codebase to support synchronized multi-trace visualization. To account for all traces, each is rendered in its own row, providing a clear visual separation between executions while preserving their temporal alignment. This synchronization

is maintained between traces with different lengths or loop sizes by halting their progression until all traces reach the end of the loop.

4.3 Experimental Process and Evaluation Design

Chapter 5 focuses on the details of the pipeline components implemented in this work, namely, solver output sanitation, multiple-trace visualization, and the integration of the entire pipeline into Alloy. The experimental process focuses on evaluating the ability to provide meaningful, comprehensible, and complete feedback to the user during hyperproperty verification, encompassing the entire round-trip, starting and ending on Alloy. More specifically, the evaluation considers whether the pipeline correctly reconstructs counterexamples involving multiple traces, whether all relevant information produced by the verification backend is preserved during the successive translation stages, and whether the resulting visualization remains understandable as the number and complexity of traces increase. In addition, the evaluation examines the overhead introduced by the additional processing stages, such as the time required to sanitize solver output, reconstruct Alloy instances, and generate visualizations, although performance is considered a secondary criterion rather than the primary focus of this work.

4.3.1 Benchmarks

Many of the benchmarks in AutoHyper's and HyperQube's repositories are relatively simple and expressed directly at a low level. Since this dissertation focuses on bridging the gap between high-level specification and low-level hyperproperty verification, the goal of the evaluation is to demonstrate that the proposed pipeline correctly supports the complete verification workflow, from Alloy models to backend verification and back to high-level feedback. To this end, the experimental analysis focuses on a small number of representative case studies in which a higher-level modeling language is justified and enables a detailed qualitative discussion. In particular, two `robot` benchmarks from AutoHyper's and HyperQube's repositories are selected:

- `robot_shortest`
- `robot_two`

They also illustrate the need for higher-level modeling, as the original authors provide an automated Python script to generate the corresponding low-level models in the HyperQube repository [82]. The `robot_shortest` example is simpler and can serve as a running example. In contrast, the `robot_two` example adds complexity to the process and enables the visualization of two different traces. Together, these benchmarks allow evaluating the proposed pipeline on both single- and multiple-trace scenarios, demonstrating that the generated feedback remains consistent and understandable across different verification settings.

4.3.2 Evaluation Methodology

Each benchmark is evaluated according to the process described below, with the corresponding results presented in Chapter 6.

1. A high-level explanation of the Alloy model and the intended hyperproperty.
2. A description of the translation process from Alloy to the lower-level intermediate representations used by the backend tools.
3. An analysis of the verification result, including solver feedback.
4. A high-level interpretation of the generated counterexample, showing how it relates to the original Alloy model and hyperproperty.
5. A report of the execution time of the verification process.

This process enables evaluating whether the feedback produced is consistent, understandable, and useful from the user's perspective. In addition to the qualitative analysis of the generated feedback, the evaluation considers the pipeline's performance across different model sizes. For this purpose, the `robot_shortest` benchmark is evaluated using grids of different dimensions and obstacle configurations. This allows observing how the execution time evolves as the size of the generated model increases.

Chapter 5

Solution Implementation

Following the design presented in Chapter 4, this chapter describes the implementation of the proposed solution. Since the proposed solution is too broad for the scope of this work, it was decided to focus on developing key components (solver output, sanitation, and multiple-trace visualization) while aligning with the defined formats of the other components to promote interoperability. This solution builds on the existing HyperLTL extension of Alloy and the supporting verification infrastructure. The implementation presented in this chapter is limited to the components developed as part of this work, namely the Inst2SMV sanitation stage, the multiple-trace visualization support, and the integration of all components into the Alloy verification pipeline. The implementation of the remaining infrastructure, including the HyperLTL language support and the modified Electrod backend, was developed independently and is therefore outside the scope of this chapter. Although the implementation of the individual components was generally straightforward once their requirements were understood, integrating them into a complete execution pipeline proved considerably more challenging. The implementation required understanding the outputs produced by the verification backends, ensuring compatibility between the intermediary representations exchanged by each stage, coordinating the generation and consumption of the auxiliary files required throughout the verification process, and identifying failures throughout the pipeline so that the corresponding components could be adapted accordingly. This chapter details how the low-level outputs from the HyperLTL solvers were translated into a uniform intermediary representation using Inst2SMV, as described in Section 4.2.5; how multiple traces were incorporated into the Alloy visualizer, as described in Section 4.2.7; and finally how all these components were integrated into Alloy's pipeline, following the architecture proposed in Section 4.1.

5.1 Solvers Output Sanitation

HyperLTL model checkers, such as AutoHyper and HyperQube, generate outputs that are not immediately usable within Alloy or Electrod. Their internal representations vary significantly in structure, as seen in Section 3.3.1.3 and in Section 3.3.2.3, respectively. In particular, AutoHyper returns witness blocks, while HyperQube produces a witness via a `QuAbS` file; however, this

information cannot be processed directly without its corresponding `QCIR` file. To unify these heterogeneous representations, a sanitation tool was implemented. Its purpose is to convert solver-level outputs into a common intermediate `SMV` output format that Electrod can process and then translate into Alloy's `XML` trace format. The format of that `SMV` instance file is demonstrated in Listing 5.1.

Listing 5.1: Format of SMV instance file generated after sanitation

```
Trace Description: LTL Counterexample/Witness
Trace Type: Counterexample/Witness
  -> State: 1.<stateID> <-
    <var> = <val>
    ...
    <var> = <val>
  ...
  -> State: 1.<stateID> <-
    <var> = <val>
    ...
    <var> = <val>
```

At this stage, HyperSMV sanitized the variable names. During that process, `.names` files were recorded, containing mappings of unsanitized names to their sanitized counterparts. The format of those `.names` files is demonstrated in Listing 5.2.

Listing 5.2: Format of `.names` file

```
<unsanitized_var> := <sanitized_var>
...
<unsanitized_var> := <sanitized_var>
```

The running example in Listing 5.3 will be used throughout this section to demonstrate the sanitation algorithms. The model describes a simple machine with three state variables: `water` (units of water), `input` (either a request `Req` or a refill `Fill`), and `output` (either `Coffee` or `Tea`). The machine initializes with two units of water and no pending input or output. On each step, a `Req` input consumes one unit of water and produces a drink; a `Fill` input when the tank is empty restores it to two units; otherwise the water level is unchanged. A mutant of this machine differs only in its refill behavior: rather than always restoring the water level to two, it may restore it to either one or two units. The hyperproperty under verification, `PotentiallyKillable`, asks whether this mutant is distinguishable from the original — that is, whether there exists a run of the mutant such that every run of the original machine receiving the same inputs eventually produces a different output.

Listing 5.3: Snippet from mutations_sat_unsat.als

```
open util/natural
trace sig Machine {
  var water : one Natural,
  var input : lone Input,
  var output : lone Output,
}
enum Output { Coffee, Tea }
enum Input { Req, Fill }
pred machine[m:Machine] {
  no m.output
  no m.input
  m.water = inc[One]
  always {
    m.input = Req and gt[m.water,Zero] implies some m.output'
    not (m.input = Req and gt[m.water,Zero]) implies no m.output'
    m.input = Fill and m.water = Zero implies m.water' = inc[One]
    m.input = Req and gt[m.water,Zero] implies m.water' = dec[m.water]
    not (m.input = Fill and m.water = Zero) and not (m.input = Req and gt[
    m.water,Zero]) implies m.water' = m.water
  }
}
pred mutant[m:Machine] {
…
  always {
…
    m.input = Fill and m.water = Zero implies m.water' in One + inc[One]
    // Difference
…
  }
}
pred same_input[m1,m2:Machine] {
  m1.input = m2.input
}
pred same_output[m1,m2:Machine] {
  m1.output = m2.output
}
run PotentiallyKillable {
  some m1:Machine | mutant[m1] and all m2:Machine |
     (machine[m2] and always same_input[m1,m2]) implies
       eventually not same_output[m1,m2]
} for 10 steps, 3 Natural, 0 Int expect 1
```

5.1.1 AutoHyper Output Sanitation

AutoHyper's output begins with a fixed header indicating the start of witness descriptions, followed by one line per trace, a line indicating the end of the witness block, and a final line indicating whether the property is satisfiable. Each trace line contains two components: a prefix and a loop, each represented as sets of variable–value pairs. This format was described in more detail in Section 3.3.1.3. Since it follows this specific format, the script in Listing 5.4 was developed to convert it to the SMV output format.

Listing 5.4: Pseudocode for `inst2smv_ah`

```
FUNCTION inst2smv_ah(run_dir, input_ah, stem):
  lines ← read all lines from input_ah, skipping leading digit lines
  IF witness header not found: print "No witnesses found." and RETURN
  // Determine SAT/UNSAT from the line two positions after
      "========================="
  sat ← "Witness" if "SAT", else "Counterexample"
  FOR EACH trace line (between witness header and "
      ========================="):
    // Split into trace identifier, prefix, and loop
    trace, prefix, loop ← parse(line)
    // If model is explicit-state, expand state numbers to variable-value
        maps using stem-trace-ah.exp; otherwise, prefix/loop are already
        variable maps
    IF prefix[0] ≠ "{":
      exp_states ← \{state\_num → \{var:val,...\} : line ∈ stem-trace-ah.
          exp\}
      prefix, loop ← expand(prefix, exp_states), expand(loop, exp_states)
    // Normalize and parse into lists of (variable, value) tuples
    prefix_states ← parseStates(prefix)
    loop_states   ← parseStates(loop)
    // Load sanitized → unsanitized name map from stem.smv.names
    names_dict ← \{sanitized → unsanitized : line ∈ stem.smv.names\}
    // Write output to stem-trace.out
    WRITE "Trace Description: LTL " + sat
    WRITE "Trace Type: "            + sat
    FOR i, state IN enumerate(prefix_states):
      WRITE "-> State: 1." + (i+1) + " <-"
      FOR (var, val) IN state:
        WRITE unsanitize(var, names_dict) + " = " + val
    FOR i, state IN enumerate(loop_states):
      IF i = 0: WRITE "-- Loop starts here"
      WRITE "-> State: 1." + (i + 1 + |prefix_states|) + " <-"
      FOR (var, val) IN state:
```

```
29          WRITE unsanitize(var, names_dict) + " = " + val
```

Listing 5.5 shows the raw output produced by AutoHyper for the running example. The single witness line for trace `A` contains a prefix of 24 state numbers and a loop of 9 state numbers.

Listing 5.5: Snippet from cex.ah for mutations_sat_unsat.als

```
1 ...
2 ======= Witnesses =======
3 A: (0 0 1 3 9 10 12 18 19 20 9 10 12 18 19 18 18 19 20 0 1 3 9 10) (12 18 19 20 0 1
      3 9 10)
4 ==========================
5 UNSAT
```

The counterexample follows the state order from Listing 5.5. The state numbers must be resolved against the `PotentiallyKillable-A-ah.exp` file, shown in Listing 5.6. Each entry maps a state number to a set of variable-value pairs. In this case, the prefix starts with state 0, so `a` and `b` will both be 0; the prefix ends with state 10, so `a` will be 1 and `b` will be 0; the loop starts with state 12, in which `a` will be 0 and `b` will be 1.

Listing 5.6: Snippet from PotentiallyKillable-A-ah.exp

```
1  Variables: ("a" Int)  ("b" Int)
2  Init: 0
3  --BODY--
4  State: 0 {("a" 0)  ("b" 0)}
5  0  1  2
6  ...
7  State: 10 {("a" 1)  ("b" 0)}
8  12  13  14  15  16  17
9  ...
10 State: 12 {("a" 0)  ("b" 1)}
11 18  19  20
12 ...
13 State: 20 {("a" 2)  ("b" 0)}
14 0  1  2  9  10  11
15 --END--
```

The sanitized variable names `a` and `b` used in the explicit-state system are then mapped back to their original Electrod identifiers using the `PotentiallyKillable-A.smv.names` file, shown in Listing 5.7. This means that `a`, `b`, and `c`, refer respectively to the input, the output and the water level.

Listing 5.7: PotentiallyKillable-A.smv.names

```
1 __this##Machine#input-__  := a
2 __this##Machine#output-__ := b
3 __this##Machine#water-__  := c
```

Applying this mapping and writing the expanded states in order yields the SMV trace in Listing 5.8. The first 24 states correspond to the prefix and the remaining 9, beginning after the loop marker, correspond to the loop.

Listing 5.8: Snippet from PotentiallyKillable-A.out via AutoHyper

```
1  Trace Description: LTL Counterexample
2  Trace Type: Counterexample
3    -> State: 1.1 <-
4      __this##Machine#input-__  = 0
5      __this##Machine#output-__ = 0
6    ...
7    -> State: 1.24 <-
8      __this##Machine#input-__  = 1
9      __this##Machine#output-__ = 0
10 -- Loop starts here
11   -> State: 1.25 <-
12     __this##Machine#input-__  = 0
13     __this##Machine#output-__ = 1
14   ...
15   -> State: 1.33 <-
16     __this##Machine#input-__  = 1
17     __this##Machine#output-__ = 0
```

5.1.2 HyperQube Output Sanitation

HyperQube's output differs substantially from AutoHyper's, as it is distributed across multiple files, as described in Section 3.3.2.3. Not all of those files are necessary for translating the output into the SMV format. Only QuAbS and QCIR files are essential for this step. The QuAbS file contains the value of the Boolean variables corresponding to the Boolean variables of a certain state of a certain trace, or an integer value split into several Boolean assignments, representing the various bits of a binary number. These variables are encoded in the QCIR file. Since these files follow these specific formats, the following script was developed to translate them into an SMV file:

Listing 5.9: Pseudocode for `inst2smv_hq`

```
FUNCTION inst2smv_hq(run_dir, input_qcir, input_quabs, stem):
  // Parse variable encoding from QCIR comment lines
  // (skip line 0 and all non-comment lines before the comment block)
  // Each comment: # <var_num> : <name>_<trace>[<state>]
  // Split name/trace at the last "_" before "["
  variables ← \{var\_num → \{name, trace, state, value: None\}
                 : comment line ∈ input\_qcir\}
  traces    ← set of all trace identifiers found above
  max_state ← maximum state number found above
  // Parse QuAbS output: assign TRUE/FALSE to each variable number;
  // "V" lines hold assignments (leading "-" means FALSE), "r" lines hold
      verdict
  FOR EACH line IN input_quabs:
    IF line starts with "V":
      FOR EACH token IN line[1:].split():
        variables[|token|].value ← FALSE if token is negative, else TRUE
    IF line starts with "r":
      sat ← "Witness" if "SAT", else "Counterexample"
  // Reconstruct bitvectors: variables whose name ends in "_<digit>"
      encode an integer; accumulate 2^suffix for each TRUE bit, then replace
      all bit-parts with a single variable holding the decimal value
  bitvectors ← \{(trace, state, base\_name) → ∑_{b=TRUE} 2^suffix\}
  FOR EACH key IN bitvectors:
    remove all bit-part variables sharing that (trace, state, base_name)
    insert new variable "bv_trace_state_base" with value = bitvectors[key]
  // Load sanitized → unsanitized name map from stem.smv.names
  names_dict ← \{sanitized → unsanitized : line ∈ stem.smv.names\}
  // Write one output file per trace
  FOR EACH trace IN traces:
    OPEN stem-trace.out FOR WRITING
    WRITE "Trace Description: LTL " + sat
    WRITE "Trace Type: "            + sat
    FOR state IN 0 .. max_state:
      IF state = max_state: WRITE "-- Loop starts here"
      WRITE "-> State: 1." + (state+1) + " <-"
      FOR EACH var IN variables WHERE var.trace = trace
                                AND var.state = state
                                AND var.value ≠ None:
        WRITE unsanitize(var.name, names_dict) + " = " + var.value
```

Listing 5.10 shows the `HQ.quabs` file produced by HyperQube for the running example. The `V` line assigns Boolean values to each variable: a positive token means `TRUE` and a negative token means `FALSE`.

Listing 5.10: HQ.quabs for mutations_sat_unsat.als

```
V -2 8 14 -20 26 32 -38 -1 -7 -13 19 -25 -31 -37 -4 -10 -16 -22 -28 34 -40 -3 -9 15
     21 -27 -33 -39 -6 -12 18 -24 30 -36 -42 5 11 -17 -23 -29 -35 -41 0
r UNSAT
```

The variable numbers in the `V` line are decoded using the comment block at the bottom of the `HQ.qcir` file, shown in Listing 5.11. Each comment maps a variable number to a name of the form `name_bit_trace[state]`, where `name` is the sanitized variable name, `bit` is the bit position, `trace` is the trace identifier, and `state` is the state index.

Listing 5.11: Snippet from HQ.qcir for mutations_sat_unsat.als

```
...
# 2 : a_0_A[0]
...
# 38 : a_0_A[6]
# 1 : a_1_A[0]
...
# 37 : a_1_A[6]
# 4 : b_0_A[0]
...
# 40 : b_0_A[6]
# 3 : b_1_A[0]
...
# 39 : b_1_A[6]
# 6 : c_0_A[0]
...
# 42 : c_0_A[6]
# 5 : c_1_A[0]
...
# 41 : c_1_A[6]
```

The integer value of each variable at each state is reconstructed by accumulating $2^{\texttt{bit}}$ for each bit set to `TRUE`. For instance, at state 0 of trace `A`: tokens `-2` and `-1` set both bits of `a` to `FALSE`, and likewise for `b`, giving `a = b = 0`; token `-6` sets `c_0_A[0]` to `FALSE` and token `5` sets `c_1_A[0]` to `TRUE`, giving `c` $= 2^1 =$ `2`. At state 1: token `8` sets `a_0_A[1]` to `TRUE` and token `-7` sets `a_1_A[1]` to `FALSE`, giving `a` $= 2^0 =$ `1`; `b` remains 0 for the same reason as before; token `11` sets `c_1_A[1]` to `TRUE` and token `-12` sets `c_0_A[1]` to `FALSE`, giving `c` $= 2^1 =$ `2`. At state 6, all tokens are negative, so `a = b = c = 0`.

The same `PotentiallyKillable-A.smv.names` file from Listing 5.7 is then used to map `a`, `b`, and `c` back to their original Electrod identifiers. Applying this mapping and writing the reconstructed states in order yields the SMV trace in Listing 5.12. Since state 6 is the maximum state index, it is preceded by the loop marker.

Listing 5.12: Snippet from PotentiallyKillable-A.out via HyperQube

```
Trace Description: LTL Counterexample
Trace Type: Counterexample
  -> State: 1.1 <-
    __this##Machine#input-__  = 0
    __this##Machine#output-__ = 0
    __this##Machine#water-__  = 2
  -> State: 1.2 <-
    __this##Machine#input-__  = 1
    __this##Machine#output-__ = 0
    __this##Machine#water-__  = 2
  ...
-- Loop starts here
  -> State: 1.7 <-
    __this##Machine#input-__  = 0
    __this##Machine#output-__ = 0
    __this##Machine#water-__  = 0
```

5.2 Multiple Trace Visualization

Once the solver outputs are sanitized and translated into Alloy's XML format through Electrod, they must be visualized. Hyperproperties involve multiple traces; however, the original Alloy visualizer was designed to display a single trace, so new capabilities had to be added to support them.

5.2.1 VizTrace

The original Alloy visualizer internally had a list of states for the resulting trace. Since hyperproperties involve multiple traces, a redesign of the internal representation was necessary to support them. A new class, `VizTrace`, was introduced to serve as a list of various `VizState`, the Alloy visualizer class for each state in a trace. Instead of storing all states in a global list, each `VizTrace` maintains its own ordered list of states, its prefix length, and its loop length. A list of traces was stored instead of the original list of states.

5.2.2 XML Loading

Previously, Alloy expected all information to be described by a single XML file. These XML files contain information regarding a single trace. With the introduction of multiple traces, multiple XML files had to be loaded, so instead of loading one file from the expected folder, all files within that folder with a `.cnf.xml` extension are loaded, each becoming a `VizTrace`. The parsing logic is the same as in the original Alloy: it extracts the prefix, loop, and state sequence independently for each file.

5.2.3 Graph Rendering

The multiple traces produced by the verification backend are transformed into independent Alloy visualizations, allowing users to inspect the graphical representation of each execution instead of the underlying textual traces. Each trace is represented by a `VizTrace` instance, which serves as the input to the rendering process. The method responsible for rendering the graph was updated to accept a `VizTrace` instance and draw the corresponding state graph. The visualization panel is divided, allowing multiple traces to be viewed simultaneously. Each trace is rendered independently but aligned temporally. Different traces may have different prefix and loop sizes, so when a shorter trace reaches its end, it will loop back while the longer trace continues forward. The abstract trace overview displayed above each graph was also adapted to support multiple traces. Rather than presenting a single execution, a separate overview is generated for each trace, showing its prefix and loop structure. The currently displayed state is highlighted independently in each overview, allowing users to track the evolution of every trace. The user interface was adapted so that advancing to the next state updates all traces simultaneously. The Theme Editor also works for multiple traces by altering the theme of all traces, maintaining visual consistency across all traces. Figure 5.1 shows two traces being displayed in the Alloy visualizer.

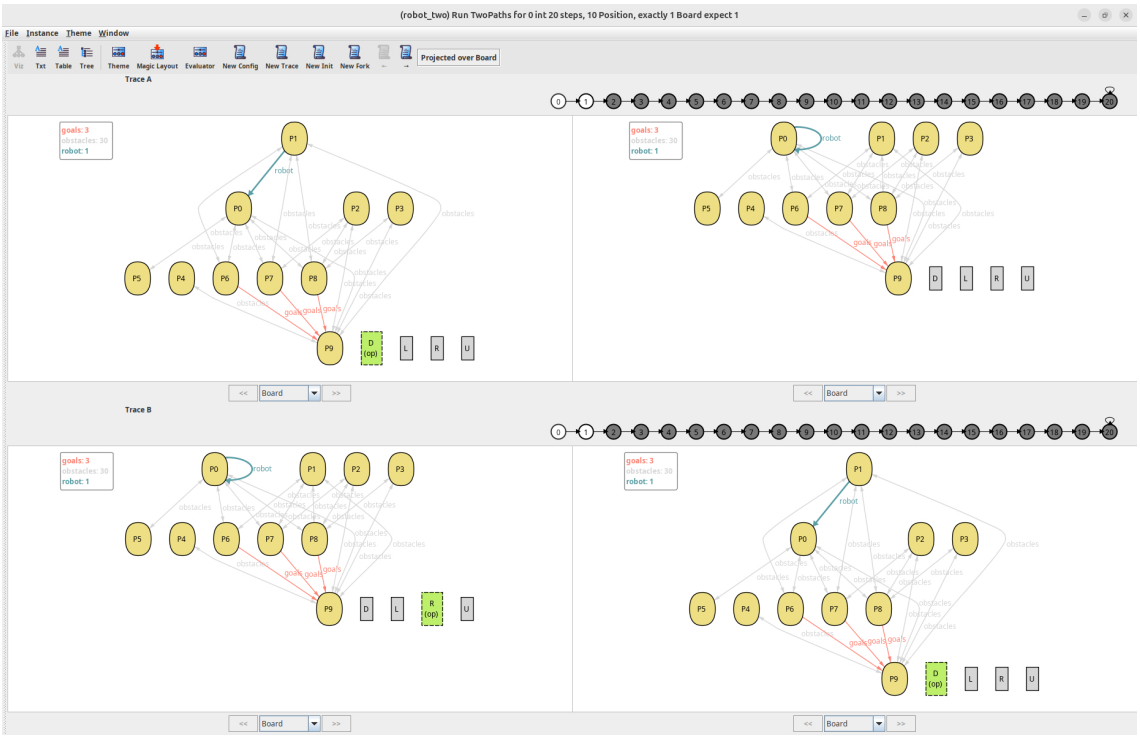


Figure 5.1: Example of two traces being rendered in the visualizer

Figure 5.1 illustrates the visualization of two traces. The upper row contains the abstract trace overviews, where the prefix and loop of each execution are represented. The highlighted node indicates the state currently displayed in the graph below. The lower rows show the Alloy instances corresponding to the selected state of each trace. Advancing to the next state updates

both traces simultaneously, while each trace independently follows its own loop structure once the end of its prefix is reached.

5.3 Integration

The proposed solution was integrated into Alloy by extending its original verification pipeline while preserving its overall structure and user interaction model. Alloy orchestrates this pipeline automatically, invoking each external tool in sequence, managing temporary directories and intermediate files, and ensuring that each stage's outputs are correctly forwarded to the next. After the Alloy model and hyperproperties are submitted to the Alloy Analyzer, the specification is first translated by Alloy, then processed by a modified version of Electrod, which generates the intermediate representations required for HyperLTL verification. This process also records structural information about relations, bounds, and instance mappings that is later reused during back-translation. These representations are then optimized by HyperSMV, producing inputs for the selected HyperLTL backend, either AutoHyper or HyperQube. Once the solver terminates, its output files are forwarded to the output sanitizer introduced in this work. This module parses the solver-specific formats (witness blocks in AutoHyper and `QuAbS`/`QCIR` pairs in HyperQube), reconstructs concrete variable assignments using the `.names` mappings generated by HyperSMV, and produces a uniform `SMV` trace description. Using the structural information previously recorded by Electrod, the sanitized trace is first converted into an intermediate XML representation and then refined into Alloy-level instances in the final `XML` trace format. Finally, the Alloy visualizer loads all generated `.cnf.xml` files as independent traces, instantiates one `VizTrace` object per file, and renders them synchronously using the extended multi-trace visualization infrastructure.

Chapter 6

Evaluation

This chapter evaluates the proposed Alloy extension's ability to provide meaningful, comprehensible, and complete feedback for verifying hyperproperties. Rather than focusing exclusively on performance metrics, the evaluation centers on the quality of the verification results and, in particular, the usefulness of the generated counterexample representations for end users. The evaluation is conducted using two representative examples inspired by AutoHyper's and HyperQube's benchmark repositories: `robot_shortest` and `robot_two`. These examples were selected to illustrate both simple and more complex scenarios, including properties that require the analysis and visualization of multiple traces. For each case study, the verification pipeline is exercised using both AutoHyper and HyperQube as backend solvers, enabling a comparison of their intermediate representations, solver outputs, and resulting counterexample visualizations. The chapter follows a structured evaluation methodology. Each example begins with a presentation of the Alloy model and the corresponding hyperproperty, followed by an analysis of the intermediate representations generated during translation, the verification results reported by each backend, and the final counterexample visualization. Finally, the chapter concludes with a discussion that relates the observed results to the research questions and hypothesis defined in Chapter 1, highlighting the strengths, limitations, and impact of the proposed approach.

6.1 Example `robot_shortest`

The `robot_shortest` benchmark is a simple example to illustrate the complete verification and counterexample reconstruction pipeline. The model describes a robot navigating a finite grid with fixed obstacles and a single goal position, and the hyperproperty asserts the existence of a trace with a minimal path to the goal.

6.1.1 Alloy Model and Property

Listing 6.1 shows a snippet from the model and the property representations in the HyperLTL extension of Alloy.

Listing 6.1: Snippet from robot_shortest.als

```
open util/ordering[Position]
sig Position {}
trace sig Board {
  obstacles : Position → Position,    -- obstacles as set of position pairs
  goals : Position → Position,        -- goals as set of position pairs
  var robot : Position → Position, } -- robot position as set of position
    pairs
pred moveleft[b:Board] {
  some next.(b.robot) and no next.(b.robot) & b.obstacles   -- guard: move
     if position some and not obstacle
  b.robot' = next.(b.robot) }                    -- effect: move robot
pred moveright[b:Board] {…}
pred moveup[b:Board] {…}
pred movedown[b:Board] {…}
pred stutter[b:Board] {
  b.robot' = b.robot }                        -- do nothing
let p0[] { first }
let p1[] { first.next }
let p2[] { p1.next }
…
let p8[] { p7.next }
let p9[] { last }
pred board[b:Board] {
  no b.obstacles & b.goals                    -- goals do not overlap with
    obstacles
  b.goals = last → last                  -- fix a goal at top-right
  b.obstacles =                         -- fix some obstacles
  p5 → p0 + p7 → p0 + … + p2 → p9 + p4 → p9
  always one b.robot
  b.robot = first → first                    -- fix starting position at
    bottom-left
  always {                            -- allowed events
    moveleft[b] or moveright[b] or moveup[b] or movedown[b] or stutter[b]
    }
  eventually b.robot in b.goals }               -- the robot eventually
    reaches a goal
pred Path {
  some b1:Board | board[b1] and all b2:Board | board[b2] implies always (
    b2.robot in b2.goals implies b1.robot in b1.goals) }
run Path for 10 Position, 20 steps, 0 Int expect 1
```

The model represents a robot moving on a 10x10 two-dimensional grid, where each two-dimensional position is a pair of ordered signatures `Position`. The signature `Board` contains

a set of `obstacles`, a set of `goals`, and a `robot`. Each obstacle, each goal, and the robot are also represented by a pair of `Position`. Movement is modeled using four directional predicates (`moveleft`, `moveright`, `moveup`, and `movedown`), each guarded to ensure that the robot moves only to valid, non-obstacle positions. A `stutter` action is also permitted, allowing the robot to remain in the same position. At every step, exactly one robot position is enforced. The goal is fixed at the top-right of the grid, while the robot has a fixed starting position at the bottom-left. The obstacles are spread at fixed positions across the grid.

The hyperproperty under verification is defined by the predicate `Path`. It expresses a shortest-path condition: there exists an execution such that, whenever another execution has already reached the goal, this execution has also reached the goal. The property quantifies over two traces and requires that whenever one trace reaches the goal, the other must also be in a goal state at that step. This formulation makes `Path` a hyperproperty rather than a simple temporal property, as it compares multiple executions of the same system. Consequently, it cannot be directly verified using standard LTL model checking alone and requires a backend capable of HyperLTL verification.

6.1.2 AutoHyper

After running Electrod, the pipeline branches into AutoHyper- and HyperQube-centric approaches; therefore, it is necessary to separate results into the two implementations. This subsection focuses on the AutoHyper pipeline, including intermediate representations, verification results, and counterexample visualizations, while Section 6.1.3 focuses on HyperQube's.

6.1.2.1 Intermediate Representations and Verification Results

The Alloy analyzer generates `ELO` files of the model for Electrod to use. Electrod generates `SMV` files and a `.hp` file from the `ELO` files, alongside an `.info` file for backtracking purposes. All these files are very extensive and difficult for humans to read. HyperSMV generates `-ah.exp` files from the `SMV` files, alongside a `formula.ah` file for the hyperproperty, and a `.names` file for backtracking purposes. Listing 6.2 shows the `.names` file for the `robot_shortest` model.

Listing 6.2: Path.smv.names (AutoHyper)

```
1 __this##Board#robot-__-__ := a
```

This means that the robot's position on the board will be denoted by `a` until the counterexample is reconstructed as an SMV instance.

The Listing 6.3 shows a snippet from an `.-ah.exp` file for the `robot_shortest` model.

Listing 6.3: Snippet from Path-A-ah.exp

```
1 Variables: ("a" Int)
```

```
Init: 0
--BODY--
State: 0 {("a" 0)}
0  1  2
State: 1 {("a" 1)}
0  1  3  4
State: 2 {("a" 10)}
0  2  4  5
State: 3 {("a" 2)}
1  3  6  7
State: 4 {("a" 11)}
1  2  4  7  8
...
State: 67 {("a" 78)}
63  64  65  67
State: 68 {("a" 89)}
63  65  68  70
State: 69 {("a" 98)}
65  66  69  70
State: 70 {("a" 99)}
68  69  70
--END--
```

This file lists all possible states in this model, detailing the robot's reachable positions, i.e., positions that are not obstacles or orthogonally surrounded by obstacles/board limits. Their state number will be used to understand the counterexample returned by AutoHyper. The part in brackets and parentheses indicates which position corresponds to that state, using a 2-digit integer in which each digit is one of the coordinates. The positions where the first coordinate is 0 are represented as a single-digit number, i.e., `0` for `00`.

And Listing 6.4 shows the `formula.ah` file for the property expressed in the `Path` predicate.

Listing 6.4: formula.ah for robot_shortest

```
forall A. exists B.
(F ({(("a"_A = 0) | ("a"_A = 1) | ("a"_A = 2) | ("a"_A = 3) | ("a"_A = 4) | ("a"_A
    = 5) | ("a"_A = 6) | ("a"_A = 7) | ("a"_A = 8) | ("a"_A = 9) | ("a"_A = 10) | (
    "a"_A = 11) | ("a"_A = 12) | ("a"_A = 14) | ("a"_A = 17) | ("a"_A = 20) | ("a"
    _A = 21) | ("a"_A = 22) | ("a"_A = 23) | ("a"_A = 24) | ("a"_A = 25) | ("a"_A =
     27) | ("a"_A = 30) | ("a"_A = 32) | ("a"_A = 37) | ("a"_A = 40) | ("a"_A = 41)
     | ("a"_A = 42) | ("a"_A = 43) | ("a"_A = 44) | ("a"_A = 47) | ("a"_A = 48) | (
    "a"_A = 51) | ("a"_A = 52) | ("a"_A = 53) | ("a"_A = 55) | ("a"_A = 56) | ("a"
    _A = 57) | ("a"_A = 58) | ("a"_A = 59) | ("a"_A = 60) | ("a"_A = 61) | ("a"_A =
     62) | ("a"_A = 65) | ("a"_A = 69) | ("a"_A = 71) | ("a"_A = 72) | ("a"_A = 73)
     | ("a"_A = 75) | ("a"_A = 76) | ("a"_A = 77) | ("a"_A = 78) | ("a"_A = 79) | (
    "a"_A = 80) | ("a"_A = 81) | ("a"_A = 82) | ("a"_A = 83) | ("a"_A = 84) | ("a"
    _A = 85) | ("a"_A = 86) | ("a"_A = 87) | ("a"_A = 88) | ("a"_A = 89) | ("a"_A =
```

```
    90) | ("a"_A = 92) | ("a"_A = 93) | ("a"_A = 94) | ("a"_A = 96) | ("a"_A = 97)
    | ("a"_A = 98))} & {("a"_B = 99)}))
```

This formula essentially means that for all traces `A` there exists a trace `B` in which the robot reaches the goal (position 99) earlier, i.e., it is still in another of the valid positions. Therefore, this formula corresponds to the negation of the original `Path` property, which asserts the existence of a trace that reaches the goal no later than any other trace. This way, a counterexample to the backend formula can be interpreted as a witness for the original property.

AutoHyper generates a `cex.ah` file from the `-ah.exp` files and the `formula.ah`. Listing 6.5 shows a `cex.ah` file for the `robot_shortest` model and hyperproperty.

Listing 6.5: cex.ah for robot_shortest

```
1 ======= Witnesses =======
2 A: (0 1 3 6 10 13 18 22 28 35 38 42 46 50 54 58 63 68) (70)
3 ==========================
4
5 UNSAT
```

It lists the sequence of states from Listing 6.3, which forms a path that does not satisfy the hyperproperty. The property being unsatisfied means that in trace A, there exists a path along which the robot reaches the goal position in the same step as in trace B, meaning it takes the shortest path.

Inst2SMV translates `cex.ah` into an instance file for the counterexample. Listing 6.6 shows a snippet from the `.out` file.

Listing 6.6: Snippet from Path-A.out (AutoHyper)

```
1  Trace Description: LTL Counterexample
2  Trace Type: Counterexample
3    -> State: 1.1 <-
4      __this##Board#robot-__-__ = 0
5    -> State: 1.2 <-
6      __this##Board#robot-__-__ = 1
7    -> State: 1.3 <-
8      __this##Board#robot-__-__ = 2
9    -> State: 1.4 <-
10     __this##Board#robot-__-__ = 3
11 ...
12   -> State: 1.16 <-
13     __this##Board#robot-__-__ = 69
14   -> State: 1.17 <-
15     __this##Board#robot-__-__ = 79
16   -> State: 1.18 <-
```

```
17      __this##Board#robot-__-__ = 89
18  -- Loop starts here
19    -> State: 1.19 <-
20      __this##Board#robot-__-__ = 99
```

It reaches the goal in state 19, having taken the shortest path, since the Manhattan distance between positions `0` and `99` is 19.

6.1.2.2 Counterexample Visualization

Using Alloy's visualizer, you can observe the robot's path without digging into intermediate files. Figure 6.1 shows the first state from the counterexample. It shows the positions of the obstacles, the goal, and the robot using relations between each coordinate. The robot starts in `P0->P0` and the goal is in `P9->P9`. The right panel shows the next step, where all the information is the same except for the robot position, which is now in `P0->P1`. Figure 6.2 shows the last state of the prefix (on the left) and the loop state (on the right). In the last prefix state, the robot position is now `P8->P9` and on the loop state it reaches `P9->P9`, meaning it reached the goal position.

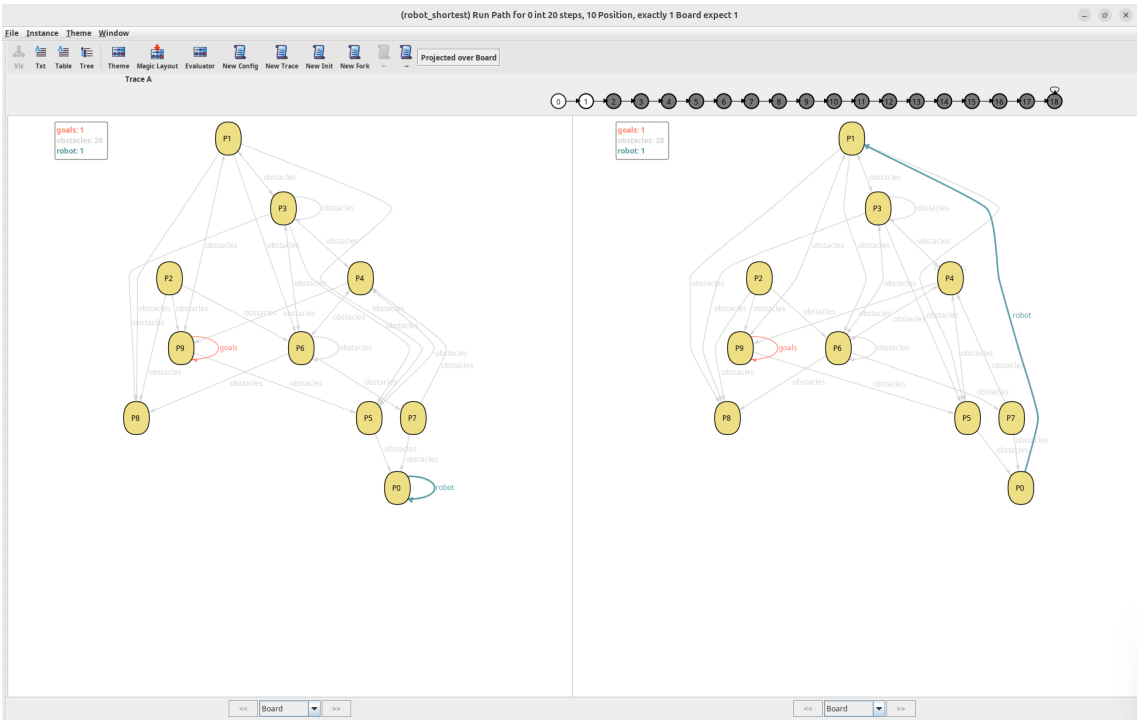


Figure 6.1: Initial state from the counterexample from AutoHyper for robot_shortest

6.1.3 HyperQube

As mentioned in Section 6.1.2, this subsection focuses on the HyperQube pipeline and its intermediate representations, verification results, and counterexample visualizations.

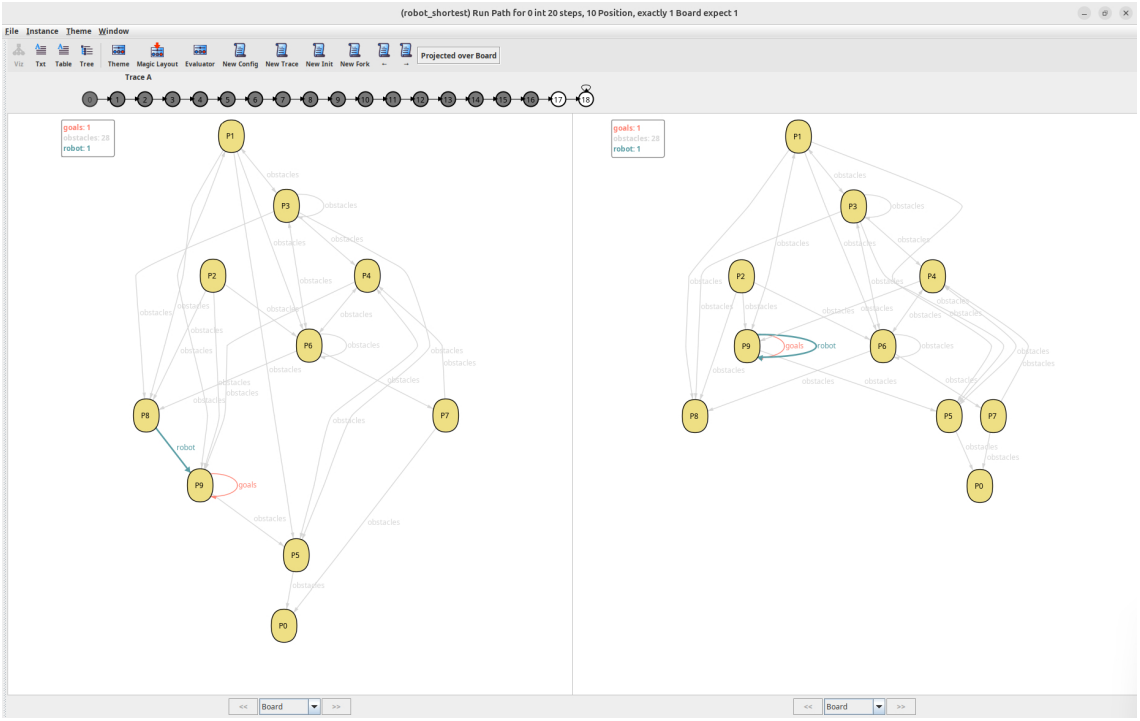


Figure 6.2: Last prefix state and loop state from the counterexample from AutoHyper for robot_shortest

6.1.3.1 Intermediate Representations and Verification Results

The pipeline only branches after Electrod, so the `ELO`, `SMV`, `.hp`, and `.info` files are the same as the ones mentioned in Section 6.1.2.1. From this point, HyperSMV generates `-hq.smv` files from the `SMV` files, alongside a `formula.hq` file for the hyperproperty, and a `.names` file for backtracking purposes. Listing 6.7 shows the `.names` file for the `robot_shortest` model.

Listing 6.7: Path.smv.names (HyperQube)

```
1 __this##Board#robot-__-__ := a
```

This means that the robot's position on the board will be denoted by `a` until the counterexample is reconstructed as an SMV instance.

The Listing 6.8 shows a snippet from one of the `.-hq.exp` files for the `robot_shortest` model.

Listing 6.8: Snipet from Path-A-hq.smv

```
1 MODULE main
2 VAR
3 a : 0..99;
```

```
4 DEFINE
5 K0 := (! (a = 99));
6 INIT
7 ((! (a = 97)) & (! (a = 35)) & (! (a = 50)) & (! (a = 20)) & ... & (! (a = 60)) &
      (! (a = 58)) & (! (a = 8)) & (! (a = 61)))
8 TRANS
9 ((((next (a = 37)) <-> (a = 37)) & ((next (a = 84)) <-> (a = 84)) & ((next (a = 94)
      ) <-> (a = 94)) & ((next (a = 51)) <-> (a = 51)) & ... & ((next (a = 20)) <-> (
      a = 20)) & ((next (a = 74)) <-> (a = 74)) & ((next (a = 25)) <-> (a = 25)) & ((
      next (a = 97)) <-> (a = 97))) | ... | (((next (a = 14)) <-> (a = 15)) & ((next
      (a = 43)) <-> (a = 44)) & (! (a = 35)) & (! (next (a = 79))) & ... & ((next (a
      = 20)) <-> (a = 21)) & ((next (a = 80)) <-> (a = 81)) & ((next (a = 12)) <-> (a
       = 13)) & ((next (a = 44)) <-> (a = 45))))
```

This file defines the position as the numbers from 0 to 99, makes a definition for this trace so that the robot doesn't reach position 99, lists all possible initial values of `a`, i.e., initial positions, and all possible transitions, i.e., the robot's valid movements (including staying in the same place). The initial position is defined by negating that it can be in any other position, leaving only (a = 0). The possible transitions exclude all positions occupied or orthogonally blocked by obstacles. The positions are represented by 2-digit integers, with each digit corresponding to a coordinate. The positions where the first coordinate is 0 are represented as a single-digit number, i.e., `0` for `00`.

The Listing 6.9 shows an even smaller snippet from the other `.-hq.exp` file for the `robot_shortest` model. It is important to show the difference in the definition of `K0` and `T2`. `T2` is defined as the robot reaching position 99.

Listing 6.9: Snipet from Path-B-hq.smv

```
1 MODULE main
2 VAR
3 a : 0..99;
4 DEFINE
5 T2 := (a = 99);
6 INIT
7 ((! (a = 97)) & (! (a = 35)) & ... & (! (a = 8)) & (! (a = 61)))
8 TRANS
9 ((((next (a = 37)) <-> (a = 37)) & ((next (a = 84)) <-> (a = 84)) & ... & ((next (a
       = 25)) <-> (a = 25)) & ((next (a = 97)) <-> (a = 97))) | ... | (((next (a =
      14)) <-> (a = 15)) & ((next (a = 43)) <-> (a = 44)) & ... & ((next (a = 12))
      <-> (a = 13)) & ((next (a = 44)) <-> (a = 45))))
```

And Listing 6.10 shows the `formula.hq` file for the property expressed in the `Path` predicate.

Listing 6.10: formula.hq for robot_shortest

```
forall A. exists B.
(((F (T2[B])) /\ (F (T2[B] /\ K0[A]))) \/ (G (K0[A])))
```

This formula means that for all traces `A` there exists a trace `B` such that either in trace `B` the robot reaches the goal before the robot in trace `A`, or in trace `A` the robot never reaches the goal. This means that, if the goal is reachable, there exists a trace `B` in which the robot takes the shortest path to reach the goal.

HyperQube generates a `HQ.quabs` and a `HQ.qcir` files from the `-hq.smv` files and the `formula.hq`. Listing 6.11 shows the `HQ.quabs` file for the `robot_shortest` model and hyperproperty.

Listing 6.11: HQ.quabs for robot_shortest

```
V 1 81 89 97 105 113 121 129 137 -145 -153 9 -161 17 25 33 41 49 57 65 73 -8 88 96
    -104 -112 120 128 136 144 152 160 16 168 -24 32 -40 48 -56 64 72 80 -7 -87 95
    -103 111 119 -127 135 -143 151 159 -15 167 23 31 -39 -47 55 63 -71 79 -6 86 94
    -102 -110 -118 126 134 -142 -150 -158 -14 -166 -22 -30 38 46 54 62 -70 -78 -5
    -85 93 -101 109 117 -125 133 141 -149 -157 -13 -165 -21 -29 -37 -45 -53 -61 -69
     77 -4 -84 -92 100 108 116 -124 -132 140 -148 -156 -12 -164 -20 -28 -36 -44 -52
     -60 68 76 -3 83 91 99 107 115 -123 -131 -139 147 155 -11 163 -19 -27 -35 -43
    -51 -59 -67 -75 -2 -82 -90 -98 -106 -114 122 130 138 146 154 -10 162 -18 -26
    -34 -42 -50 -58 -66 -74 0
r UNSAT
```

Listing 6.12 shows a snippet from the `HQ.qcir` file for the `robot_shortest` model and hyperproperty.

Listing 6.12: Snippet from HQ.qcir for robot_shortest

```
#QCIR-G14
forall(1, 81, 89, 97, ..., 50, 58, 66, 74)
exists(169, 249, 257, 265, ..., 218, 226, 234, 242)
output(337)
339 = or(-1,2,3,4,5,6,7,8)
343 = and(1,-2,-3,-4,-5,-6,-7,-8)
...
9819 = and(9820,19301)
337 = or(338,9819)
# 1 : K0_A[0]
# 81 : K0_A[10]
...
# 65 : K0_A[8]
# 73 : K0_A[9]
# 8 : a_0_A[0]
```

```
# 88 : a_0_A[10]
...
# 72 : a_0_A[8]
# 80 : a_0_A[9]
# 7 : a_1_A[0]
# 87 : a_1_A[10]
...
# 66 : a_6_A[8]
# 74 : a_6_A[9]
# 169 : T2_B[0]
# 249 : T2_B[10]
...
# 233 : T2_B[8]
# 241 : T2_B[9]
# 176 : a_0_B[0]
# 256 : a_0_B[10]
...
# 240 : a_0_B[8]
# 248 : a_0_B[9]
# 175 : a_1_B[0]
# 255 : a_1_B[10]
...
# 234 : a_6_B[8]
# 242 : a_6_B[9]
```

The values in `HQ.quabs` indicate the value of the bit for the corresponding HyperQube state in `HQ.qcir` as described in Section 5.1.2. In this form, it is difficult to understand its meaning. Inst2SMV then calculates the values of `a` for each state and generates the instance file for the counterexample. Listing 6.13 shows a snippet from the `.out` file.

Listing 6.13: Snippet from Path-A.out (HyperQube)

```
Trace Description: LTL Counterexample
Trace Type: Counterexample
  -> State: 1.1 <-
    __this##Board#robot-__-__ = 0
  -> State: 1.2 <-
    __this##Board#robot-__-__ = 1
  -> State: 1.3 <-
    __this##Board#robot-__-__ = 2
  -> State: 1.4 <-
    __this##Board#robot-__-__ = 3
...
  -> State: 1.18 <-
    __this##Board#robot-__-__ = 89
  -> State: 1.19 <-
    __this##Board#robot-__-__ = 99
```

```
16   -> State: 1.20 <-
17     __this##Board#robot-__-__ = 99
18 -- Loop starts here
19   -> State: 1.21 <-
20     __this##Board#robot-__-__ = 99
```

As with AutoHyper, it reaches the goal in state 19, but there are two additional states (20 and 21) where it remains at position 99. The loop only starts in state 21.

6.1.3.2 Counterexample Visualization

Figure 6.3 shows the first state from the counterexample. It shows the positions of the obstacles, the goal, and the robot using relations between each coordinate. The robot starts in `P0->P0` and the goal is in `P9>P9`. The right panel shows the next step, where all the information is the same except for the robot position, which is now in `P0->P1`. Figure 6.4 shows the last state of the prefix (on the left) and the loop state (on the right). In the last prefix state, the robot position is now `P9->P9` and not `P8->P9` like in Section 6.1.2.2 because of the two extra states.

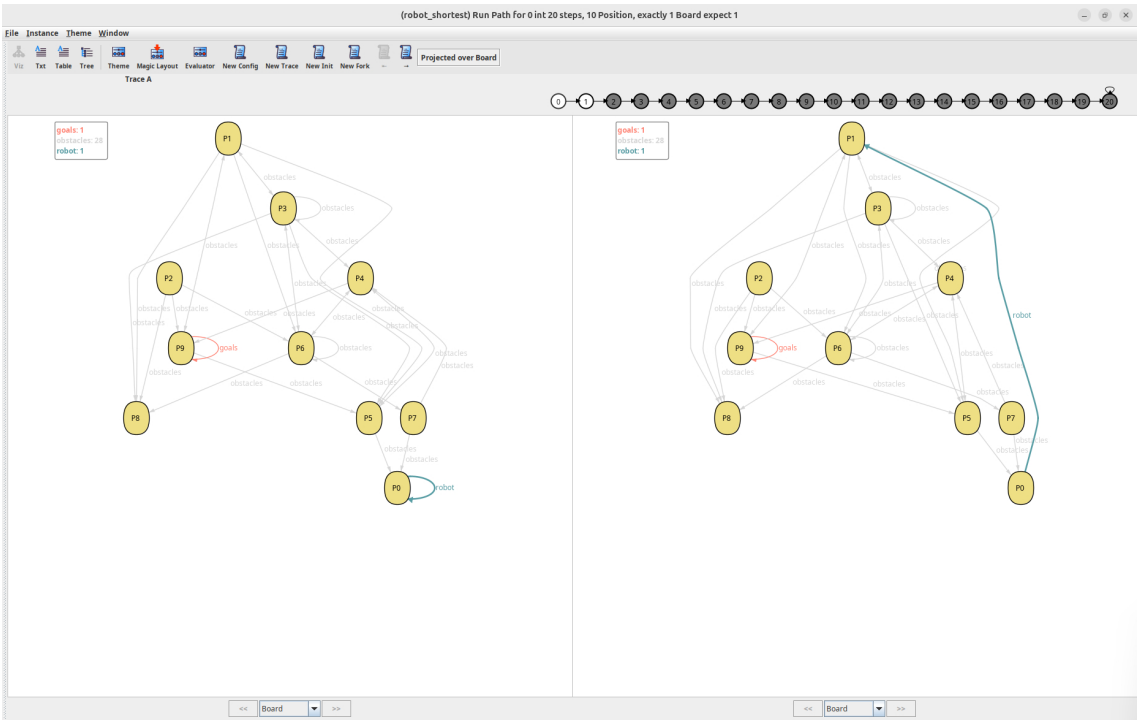


Figure 6.3: Initial state from the counterexample from HyperQube for robot_shortest

6.1.4 Comparison

The tool was able to reconstruct a valid counterexample using either AutoHyper or HyperQube. The main difference between their counterexamples was that AutoHyper started the loop as soon as the robot reached the goal, whereas HyperQube reached the goal and waited for two states

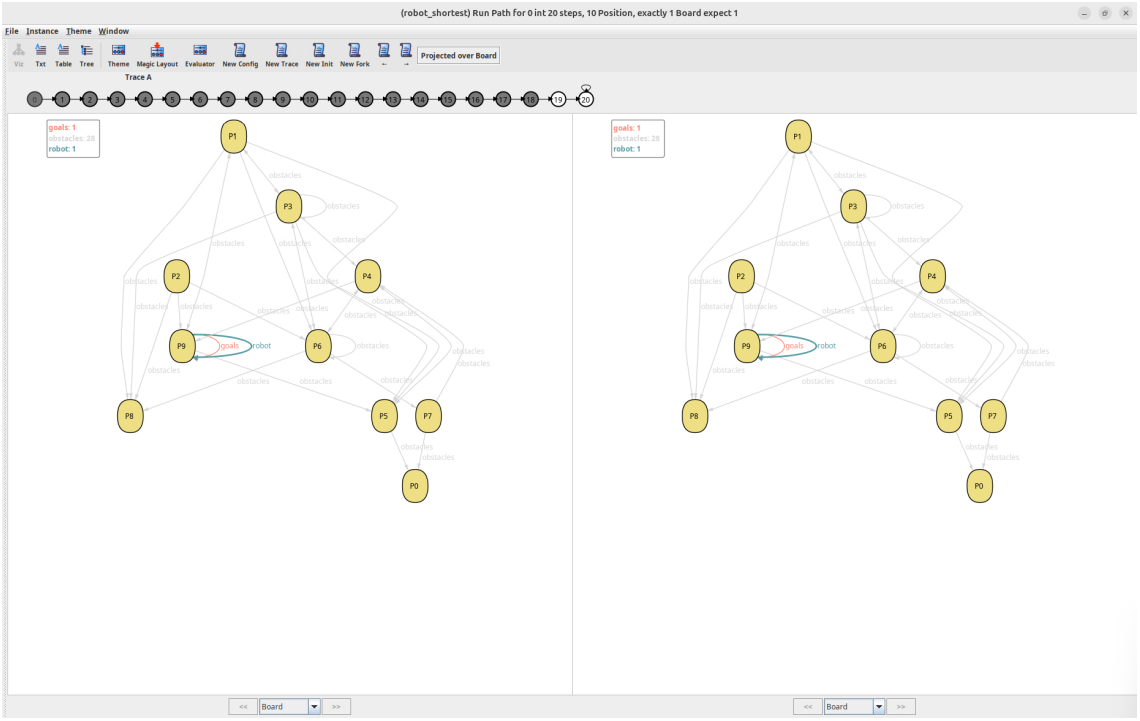


Figure 6.4: Last prefix state and loop state from the counterexample from HyperQube for robot_shortest

before entering the loop, resulting in two extra states, since both counterexamples reach the goal through the same path.

6.2 Example `robot_two`

The `robot_two` benchmark is a more complex example that illustrates the verification of more complex systems and also allows visualization of two different traces. It uses the model `robot_robust`, which is essentially the same as the one in Section 6.1, except that there are multiple initial positions and goals. The hyperproperty asserts that there are two different paths to the goal.

6.2.1 Alloy Model and Property

Listing 6.14 shows a snippet from the model and the property representations in the HyperLTL extension of Alloy.

Listing 6.14: Snippet from robot_robust.als

```
1 open util/ordering[Position]
2 sig Position {}
3 abstract sig Op {}
```

```
4  one sig L, R, U, D extends Op {}
5  trace sig Board {
6    obstacles : Position → Position,    -- obstacles as set of position pairs
7    goals : Position → Position,        -- goals as set of position pairs
8    var robot : Position → Position,    -- robot position as set of position
       pairs
9    var op : lone Op }
10 pred moveleft[b:Board] {
11   b.op = L
12   some next.(b.robot) and no next.(b.robot) & b.obstacles   -- guard: move
       if position some and not obstacle
13   b.robot' = next.(b.robot) }                    -- effect: move robot
14 pred moveright[b:Board] {…}
15 pred moveup[b:Board] {…}
16 pred movedown[b:Board] {…}
17 pred stutter[b:Board] {
18   no b.op
19   b.robot' = b.robot }                      -- do nothing
20 let p0[] { first }
21 …
22 let p9[] { last }
23 pred board[b:Board] {
24   b.goals = p6 → p9 + p7 → p9 + p8 → p9          -- fix some goals
25   b.obstacles =                             -- fix some obstacles
26       p5 → p0 + p6 → p0 + … + p3 → p9 + p4 → p9
27   always one b.robot
28   b.robot in p0 → p0 + p1 → p0 + p2 → p0         -- fix some positions
29   always {                              -- allowed events
30     (moveleft[b] or moveright[b] or moveup[b] or movedown[b] or stutter[b
       ]) } }
31 pred same_behaviour[b1,b2:Board] {
32   b1.op = b2.op }
33 pred TwoPaths {
34   some b1:Board | (board[b1] and eventually b1.robot in b1.goals) and some
       b2:Board | (board[b2] and eventually b2.robot in b2.goals) and not (
      always same_behaviour[b1,b2]) }
35 run TwoPaths for 10 Position, 20 steps, 0 Int expect 1
```

This model also represents a robot moving on a 10x10 two-dimensional grid, as in Section 6.1.1. One difference in this model is that the `Board` has an extra field, `op` (operation). This can be `L`, `R`, `U`, and `D`, which accompanies `moveleft`, `moveright`, `moveup`, and `movedown`, respectively. This will be used to verify the predicate `same_behaviour`, which compares whether the robot performs the same operation in both traces at the same state. Another major difference between the models is that `robot_robust` has three initial positions (p0 -> p0, p1 -> p0, and p2

-> p0) and three goal positions (p6 -> p9, p7 -> p9, and p8 -> p9).

The hyperproperty under verification is defined by the predicate `TwoPaths`. It checks whether two successful traces reach a goal but differ in their sequences of operations. The property quantifies over two traces and requires that, at least once, they perform different operations.

6.2.2 AutoHyper

Just like in `sec:example-robot-shortest`, after running Electrod, the pipeline branches into AutoHyper- and HyperQube-centric approaches, so it will also be necessary to separate the results from the two implementations. This subsection focuses on the AutoHyper pipeline, while Section 6.2.3 focuses on HyperQube's.

6.2.2.1 Intermediate Representations and Verification Results

The Alloy analyzer generates `ELO` files, which Electrod used to generate `SMV` files, a `.hp`, and an `.info` file. Just like in the `robot_shortest` example, these files are very extensive and difficult to read. HyperSMV generates `-ah.exp` files from the `SMV` files, alongside a `formula.ah` file for the hyperproperty, and a `.names` file for backtracking purposes. Listing 6.15 shows the `.names` file for the `robot_robust` model.

Listing 6.15: TwoPaths.smv.names (AutoHyper)

```
1 __this##Board#op-__ := a
2 __this##Board#robot-__-__ := b
```

This means that the operation performed in that state will be denoted by `a`, and the robot's position on the board will be denoted by `b` until the counterexample is reconstructed as an SMV instance.

The Listing 6.16 shows a snippet from an `.-ah.exp` file for the `robot_robust` model.

Listing 6.16: Snippet from TwoPaths-A-ah.exp

```
1  Variables: ("a" Int)
2  Init: 0  1  2  4  5  6  7  8  9  10  11
3  --BODY--
4  State: 0 {("a" 0)}
5  0  6  9
6  State: 1 {("a" 0)}
7  1  4  7  10
8  State: 2 {("a" 0)}
9  2  5  8  11
10 State: 4 {("a" 1)}
11 0  6  9
```

```
...
State: 344 {("a" 4)}
325  326  327  328  329
State: 345 {("a" 0)}
345  346  349
State: 346 {("a" 1)}
340  341  342  344
State: 349 {("a" 4)}
335  336  338  339
--END--
```

This file lists all possible states in this model and details the operation executed at each state (left, right, up, or down). Their state number will be used to understand the counterexample returned by AutoHyper. The part in brackets and parentheses indicates which operation is taken.

And Listing 6.17 shows the `formula.ah` file for the property expressed in the `Path` predicate.

Listing 6.17: formula.ah for robot_two

```
forall A. forall B.
(G {((("a"_B = 0) & ("a"_A = 0)) | (("a"_B = 1) & ("a"_A = 1)) | (("a"_B = 2) & ("a
    "_A = 2)) | (("a"_B = 3) & ("a"_A = 3)) | (("a"_B = 4) & ("a"_A = 4)))})
```

This formula means that for all traces `A` and for all traces `B`, the operation taken in a state in both states will always be the same. This formula does not mention the robot reaching the goal position.

AutoHyper generates a `cex.ah` file from the `-ah.exp` files and the `formula.ah`. Listing 6.18 shows a snippet from the `cex.ah` file for the `robot_robust` model with the `robot_two` hyperproperty.

Listing 6.18: Snippet from the cex.ah for robot_two

```
======= Witnesses =======
A: (0 0 0 0 ... 24 9 24 9) (24 0 0 0 ... 24 9 24 9)
B: (6 1 4 0 ... 134 109 79 54) (29 1 4 0 ... 134 109 79 54)
=========================

UNSAT
```

It lists the sequence of states from Listing 6.3, but in this case, it is unintelligible because each trace has more than 8000 states, making it impossible for a human to understand the output.

Inst2SMV translates `cex.ah` into an instance file for the counterexample. Listing 6.19 shows a snippet from the `.out` file.

Listing 6.19: Snippet from TwoPaths-A.out (AutoHyper)

```
Trace Description: LTL Counterexample
Trace Type: Counterexample
  -> State: 1.1 <-
    __this##Board#op-__ = 0
  -> State: 1.2 <-
    __this##Board#op-__ = 0
...
  -> State: 1.4495 <-
    __this##Board#op-__ = 4
  -> State: 1.4496 <-
    __this##Board#op-__ = 3
-- Loop starts here
  -> State: 1.4497 <-
    __this##Board#op-__ = 4
  -> State: 1.4498 <-
    __this##Board#op-__ = 0
...
  -> State: 1.8991 <-
    __this##Board#op-__ = 4
  -> State: 1.8992 <-
    __this##Board#op-__ = 3
```

Since `twopaths-a-ah-exp` lacked the position for each state, the counterexample also lacks the position, making this counterexample incomprehensible as well.

6.2.2.2 Counterexample Visualization

Electrod generated the XML file, but Alloy failed to generate the respective `.cnf.xml` file, possibly due to the lack of the robot position in the intermediate files.

6.2.3 HyperQube

As mentioned in Section 6.2.2, this subsection focuses on the HyperQube's verification results and counterexample visualizations.

6.2.3.1 Intermediate Representations and Verification Results

The pipeline only branches after Electrod, so the `ELO`, `SMV`, `.hp`, and `.info` files are the same as the ones mentioned in Section 6.2.2.1. From this point, HyperSMV generates `-hq.smv` files from the `SMV` files, alongside a `formula.hq` file for the hyperproperty, and a `.names` file for backtracking purposes. Listing 6.20 shows the `.names` file for the `robot_robust` model.

Listing 6.20: TwoPaths.smv.names (HyperQube)

```
__this##Board#op-__ := a
__this##Board#robot-__-__ := b
```

This means that the operation performed in that state will be denoted by `a`, and the robot's position on the board will be denoted by `b` until the counterexample is reconstructed as an SMV instance, as it was in Section 6.2.3.1.

The Listing 6.21 shows a snippet from one of the `.-hq.exp` files for the `robot_robust` model.

Listing 6.21: Snipet from TwoPaths-A-hq.smv

```
MODULE main
VAR
a : 0..4;
b : 0..99;
DEFINE
K0 := (! ((b = 89) | (b = 79) | (b = 69)));
INIT
((! (b = 2)) & (! (b = 54)) & ... & (! (b = 41)) & (! (b = 45)))
TRANS
((((next (b = 27)) <-> (b = 26)) & ((next (b = 78)) <-> (b = 77)) & ... & ((next (b
     = 66)) <-> (b = 65)) & ((next (b = 73)) <-> (b = 72))) | ... | (((next (b =
    30)) <-> (b = 20)) & ((next (b = 15)) <-> (b = 5)) & ... & (! (b = 29)) & (! (
    next (b = 1)))))
```

This file defines the position as the numbers from 0 to 99, makes a definition for this trace so that the robot doesn't reach positions 69, 79, or 89, lists all possible initial values of `b`, i.e., initial positions, and all possible transitions, i.e., the robot's valid movements (including staying in the same place), paired up with the position after said movement. The initial position is defined by negating that it can be in any other position, leaving only (a = 0), (a = 10), and (a = 20). The possible transitions exclude all positions occupied or orthogonally blocked by obstacles. The positions are represented by 2-digit integers, with each digit corresponding to a coordinate. The positions where the first coordinate is 0 are represented as a single-digit number, i.e., `0` for `00`. The operations range from `0` to `4`, being `0` the stutter, `1`, left, `2`, right, `3`, up, and `4`, down. The other `.-hq.exp` file is almost the same except for `K0`, which is called `K1`.

Listing 6.22 shows the `formula.hq` file for the property expressed in the `TwoPaths` predicate.

Listing 6.22: formula.hq for robot_two

```
forall A. forall B.
((G ((*a[A] = 2* <-> *a[B] = 2*) /\ (*a[A] = 1* <-> *a[B] = 1*) /\ (*a[A] = 3* <->
    *a[B] = 3*) /\ (*a[A] = 4* <-> *a[B] = 4*))) \/ (G (K0[A])) \/ (G (K1[B])))
```

This formula means that for all traces A and for all traces B, the operation taken in a state in both states will always be the same, or the robot from one of the traces will never be in a goal position. This formula corresponds to the negation of the original TwoPaths property, so a counterexample to the backend formula can be interpreted as a witness to the original property.

HyperQube generates a HQ.quabs and a HQ.qcir files from the -hq.smv files and the formula.hq. Listing 6.23 shows a snippet from the HQ.quabs file for the robot_robust model and the robot_two hyperproperty.

Listing 6.23: Snippet from the HQ.quabs for robot_two

```
V 1 -111 -122 -133 -144 -155 -166 -177 -188 -199 -210 12 -221 23 34 -45 -56 ...
    -247 -456 -258 -269 -280 -291 -302 -313 -324 -335 0
r UNSAT
```

Listing 6.24 shows a snippet from the HQ.qcir file for the robot_robust model and hyperproperty robot_two.

Listing 6.24: Snippet from HQ.qcir for robot_two

```
#QCIR-G14
forall(1, 111, 122, 133, 144, 155, ..., 434, 445, 247, 456, 258, 269, 280, 291,
    302, 313, 324, 335)
output(463)
467 = or(-1,-2,3,4,5,6,-7,8,-9,10,11)
468 = or(-1,-2,3,4,5,6,7,8,9,10,11)
...
99777 = or(99778,100785)
463 = or(464,99777)
# 1 : K0_A[0]
...
# 100 : K0_A[9]
# 232 : K1_B[0]
...
# 331 : K1_B[9]
# 4 : a_0_A[0]
...
# 103 : a_0_A[9]
# 235 : a_0_B[0]
...
# 334 : a_0_B[9]
# 3 : a_1_A[0]
...
# 332 : a_2_B[9]
# 11 : b_0_A[0]
...
```

```
# 110 : b_0_A[9]
# 242 : b_0_B[0]
...
# 341 : b_0_B[9]
# 10 : b_1_A[0]
...
# 335 : b_6_B[9]
```

The values in `HQ.quabs` indicate the value of the bit for the corresponding HyperQube state in `HQ.qcir` as described in Section 5.1.2. In this form, it is difficult to understand its meaning. Inst2SMV then calculates the values of `a` and `b` for each state and generates the instance file for the counterexample. Listings 6.25 and 6.26 shows snippets from the `.out` files of both traces.

Listing 6.25: Snippet from TwoPaths-A.out (HyperQube)

```
Trace Description: LTL Counterexample
Trace Type: Counterexample
  -> State: 1.1 <-
    __this##Board#op-__ = 0
    __this##Board#robot-__-__ = 20
  -> State: 1.2 <-
    __this##Board#op-__ = 2
    __this##Board#robot-__-__ = 20
  -> State: 1.3 <-
    __this##Board#op-__ = 1
    __this##Board#robot-__-__ = 30
  -> State: 1.4 <-
    __this##Board#op-__ = 4
    __this##Board#robot-__-__ = 20
  -> State: 1.5 <-
    __this##Board#op-__ = 0
    __this##Board#robot-__-__ = 0
...
  -> State: 1.20 <-
    __this##Board#op-__ = 0
    __this##Board#robot-__-__ = 0
-- Loop starts here
  -> State: 1.21 <-
    __this##Board#op-__ = 0
    __this##Board#robot-__-__ = 0
```

Listing 6.26: Snippet from TwoPaths-B.out (HyperQube)

```
Trace Description: LTL Counterexample
Trace Type: Counterexample
```

```
  -> State: 1.1 <-
    __this##Board#op-__ = 2
    __this##Board#robot-__-__ = 0
  -> State: 1.2 <-
    __this##Board#op-__ = 2
    __this##Board#robot-__-__ = 10
  -> State: 1.3 <-
    __this##Board#op-__ = 0
    __this##Board#robot-__-__ = 20
  -> State: 1.4 <-
    __this##Board#op-__ = 4
    __this##Board#robot-__-__ = 20
  -> State: 1.5 <-
    __this##Board#op-__ = 0
    __this##Board#robot-__-__ = 0
  -> State: 1.20 <-
    __this##Board#op-__ = 0
    __this##Board#robot-__-__ = 0
-- Loop starts here
  -> State: 1.21 <-
    __this##Board#op-__ = 0
    __this##Board#robot-__-__ = 0
```

They start differently, but then, in state 5, they return to position 0 (from position 20, which should be impossible) and remain there until the loop, failing to reach the goal position.

6.2.3.2 Counterexample Visualization

Figure 6.5 shows the first state from the counterexample. Both traces show the positions of the obstacles, the goal, and the robot using relations between each coordinate. In contrast to `robot_shortest`, this visualization also shows the operations and indicates which operation is performed at each step. The robot starts in `P2->P0`, in trace A, and in `P0->P0`, in trace B. The goals are in `P6>P9`, `P7>P9`, and `P8>P9` in both traces. In trace A, no operation is executed, while in trace B, the robot performs the `R` operation. The right panel shows the next step: the robot's position in trace A remains unchanged (because no operation was performed), and in trace B it is now `P10->P0`, having just moved to the right. Figure 6.6 shows the last state of the prefix (on the left) and the loop state (on the right).

6.2.4 Comparison

The tool failed to reconstruct a valid counterexample with either AutoHyper or HyperQube, for different reasons. AutoHyper returns a huge, unintelligible counterexample and, in addition, lacks the robot's position, causing the pipeline to be interrupted because the low-level counterexample was missing information. Using HyperQube, it generates counterexamples that can be visualized,

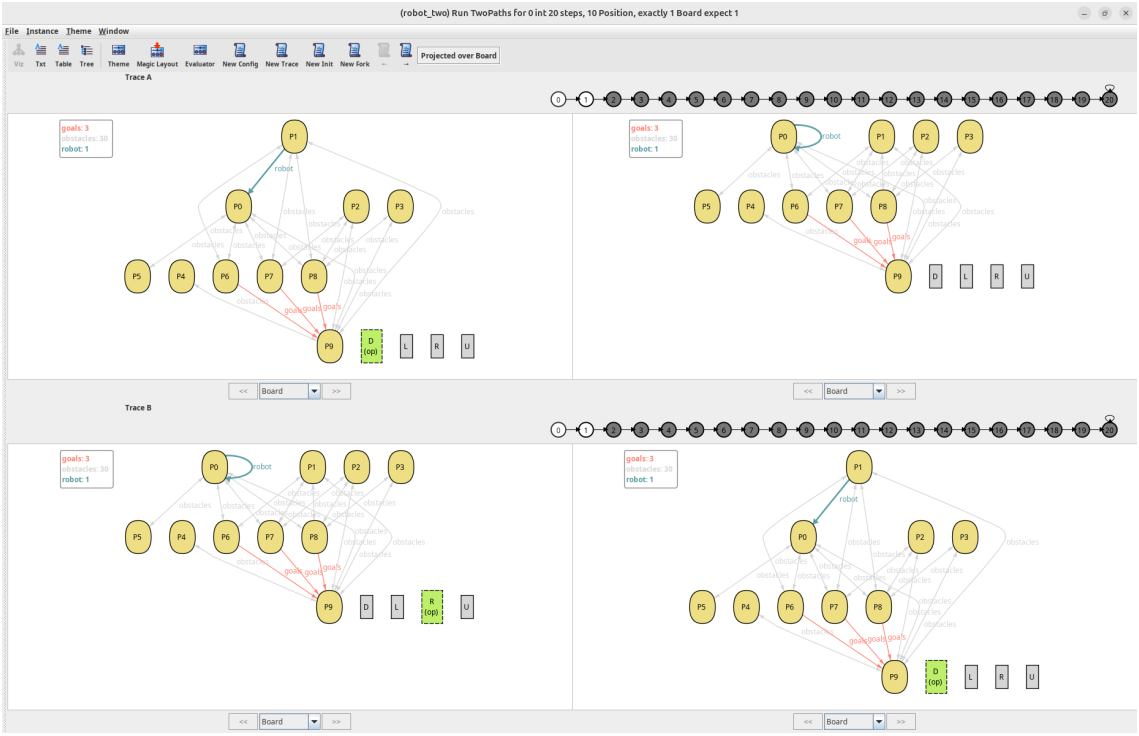


Figure 6.5: Initial state from the counterexample from HyperQube for robot_two

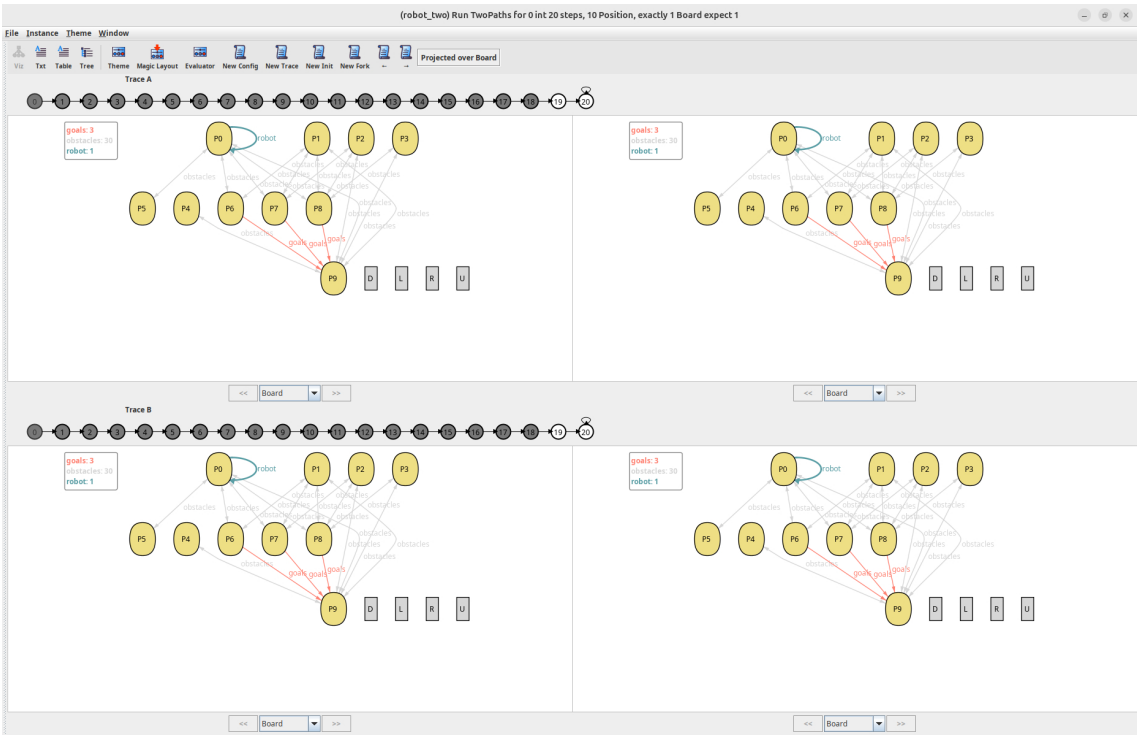


Figure 6.6: Last prefix state and loop state from the counterexample from HyperQube for robot_two

but they do not represent a realistic trace because, at some point, the robot stays at position `0`, even though it was not next to that position in the previous step.

6.3 Discussion

The previous sections presented the pipeline's behavior in two examples, highlighting how it takes the high-level model, runs the verification pipeline, and displays a visualization of the counterexample returned by the low-level tools, albeit with varying degrees of success. This section discusses the differences in abstraction levels among the input models, hyperproperties, and output instances. Besides that, it analyses how different tools perform as the model size scales. Finally, it discusses the trade-offs between the two backend tools, considering their scalability and the conclusiveness of their verification results.

6.3.1 Input and Output Abstraction

In the proposed approach, only a single Alloy specification is required, containing both the system model and the hyperproperty. The model is described in relational terms, making it easier to express the problem concisely. This contrasts with the lower-level models used by the backend solvers and the instances they return. AutoHyper and HyperQube do not operate directly over the Alloy model, as they require explicit-state or SMV representations, coupled with a separate file for the HyperLTL formula. Besides being split into different files, these files are much harder to write, inspect, or change manually because they encode the systems as states, transitions, and simple variables rather than the relational structures from Alloy. This abstraction gap also applies to backend tools' outputs, as they return instances in their own format, which requires effort to make them minimally understandable. This is the main reason why there is a necessity to translate them back to the Alloy abstraction level.

6.3.2 Performance and Scalability

Besides the abstraction level of the inputs and outputs, another important aspect to evaluate is how the pipeline behaves as the model size increases. For this purpose, the performance evaluation focuses on the `robot_shortest` example, as both backends can produce reconstructible counterexamples, making it suitable for comparing their behavior as the model size increases. This test is performed by increasing the board size and the number of obstacles. The creation of the expanded grids was assisted by Large Language Models (LLMs). Table 6.1 compares the average execution times for each model and each backend tool, measured in seconds. Table 6.2 compares the average execution times for different grid sizes (10x10, 20x20, and 40x40) for each backend tool, measured in seconds. These are the times from the last Alloy call (when generating the hyperproperty from `LTLSPEC`) until the counterexample is ready for visualization. This is because the first steps in the pipeline count as different solves, and since they do not return the solution, their time is not accounted for internally.

Table 6.1: Execution times for each benchmark (seconds)

Tool	**robot_shortest**	**robot_two**
AutoHyper	3.33s	N/A
HyperQube	4.85s	11.67s

For the smaller models, AutoHyper is slightly faster than HyperQube, and the executions for `robot_shortest` are also faster than the executions for `robot_two`. Execution times for `robot_two` using AutoHyper could not be recorded because the runs failed to complete. As the grid size increases, AutoHyper's runtime becomes much larger than HyperQube's, to the point that AutoHyper's execution times for 20x20 were too large to record, indicating that HyperQube's bounded approach scales better than AutoHyper's complete approach.

Beyond execution time, the model and property file sizes (Tables 6.3 to 6.6) provide a complementary view of scalability. The raw model and property sizes are identical between the two tools for a given configuration, as both consume the same input before backend-specific processing; the difference emerges only after optimization. On the model side, AutoHyper consistently produces smaller optimized models than HyperQube whenever it completes (e.g., 2.4 kB vs. 23.0 kB at 10×10, and 15.3 kB vs. 94.6 kB at 20×20), reflecting its more aggressive minimization. However, this advantage is contingent on AutoHyper being able to finish: at 40×40, its optimization step times out entirely, mirroring its behavior on execution time, while HyperQube still returns a bounded optimized model (391.0 kB). On the property side, the trend is reversed: HyperQube's optimized property size remains nearly constant (75 B–179 B) across all benchmarks and grid sizes, consistently smaller than AutoHyper's, whose optimized property size grows with the grid (1.1 kB at 10×10, 5.7 kB at 20×20) and also times out at 40×40. Taken together, these results suggest that HyperQube's bounded approach does not necessarily yield smaller artifacts than AutoHyper's at smaller scales, but it degrades more gracefully: it keeps producing usable, size-bounded output as the grid grows, whereas AutoHyper's outputs are typically more compact when it succeeds but the pipeline itself becomes intractable past a certain size.

6.3.3 Backend Tradeoff

The results also highlight an important trade-off between the two backend tools. AutoHyper performs complete verification of the generated explicit-state models. This means that its answers are conclusive with respect to the translated model: if the property is satisfied or violated, the result refers to all reachable behaviors of the model. However, this completeness comes at a cost. As shown in Table 6.2, AutoHyper's execution time increases significantly when the size of the

Table 6.2: Execution times for different grid sizes (seconds)

Tool	**10×10**	**20×20**	**40×40**
AutoHyper	3.33s	91.86s	timeout
HyperQube	4.85s	24.02s	174.00s

Table 6.3: Model sizes for each benchmark

Tool	**robot_shortest**		**robot_two**	
	raw	**opt.**	**raw**	**opt.**
AutoHyper	272.0 kB	2.4 kB	297.5 kB	12.8 kB
HyperQube	272.0 kB	23.0 kB	297.5 kB	23.2 kB

Table 6.4: Property sizes for each benchmark

Tool	**robot_shortest**		**robot_two**	
	raw	**opt.**	**raw**	**opt.**
AutoHyper	68.5 kB	1.1 kB	33.1 kB	176 B
HyperQube	68.5 kB	75 B	33.1 kB	179 B

Table 6.5: Model sizes for different grid sizes

Tool	**10×10**		**20×20**		**40×40**	
	raw	**opt.**	**raw**	**opt.**	**raw**	**opt.**
AutoHyper	272.0 kB	2.4 kB	1.5 MB	15.3 kB	6.4 MB	timeout
HyperQube	272.0 kB	23.0 kB	1.5 MB	94.6 kB	6.4 MB	391.0 kB

Table 6.6: Property sizes for different grid sizes

Tool	**10×10**		**20×20**		**40×40**	
	raw	**opt.**	**raw**	**opt.**	**raw**	**opt.**
AutoHyper	68.5 kB	1.1 kB	375.0 kB	5.7 kB	1.6 MB	timeout
HyperQube	68.5 kB	75 B	375.0 kB	75 B	1.6 MB	75 B

board increases, which makes it less scalable for larger instances. HyperQube, on the other hand, follows a bounded approach. Instead of exploring the entire state space, it searches for witnesses or counterexamples within a fixed number of states. This makes it more scalable in some cases, as observed when increasing the grid size, but it also affects the conclusiveness of the result. This trade-off is also reflected in the artifact sizes reported in Section 6.3.2, though its direction differs between models and properties. For optimized models, AutoHyper's complete approach tends to produce smaller artifacts when it succeeds, as its exhaustive analysis of the state space allows for more aggressive minimization; however, this same exhaustiveness is what causes it to time out on larger instances, at which point no optimized model is produced at all. HyperQube's bounded approach, in contrast, consistently returns an optimized model regardless of grid size, even if that model is comparatively larger, since it constrains its search rather than attempting to minimize over the full state space. For optimized properties, the pattern reverses: HyperQube's property sizes remain small and nearly constant across all grid sizes, while AutoHyper's grow with the grid and eventually time out as well. Taken together, these results show that AutoHyper's completeness does not uniformly translate into more compact output, and that HyperQube's bounded approach is the more consistently scalable of the two, both in execution time and in its ability to produce a usable artifact at every grid size tested. The interpretation of the result depends on whether the property is existential or universal. For existential properties, finding a witness within the selected bound is conclusive, since the witness corresponds to an actual behavior of the model. However, failing to find such a witness only shows that none exists up to the chosen bound; one may still exist at a greater depth. Conversely, for universal properties, finding a counterexample within the bound is conclusive because it demonstrates a genuine violation of the property. If no counterexample is found, the result remains valid only up to that bound, as it may still appear beyond the explored depth.

Chapter 7

Discussion

This chapter discusses the results obtained throughout the dissertation and relates them back to the problem, objectives, research questions, and hypothesis introduced in Chapter 1. It begins by revisiting the original problem and the goals that guided the work. Then, it summarizes the proposed solution and its main contributions, followed by a discussion of the conclusions drawn from the evaluation. Finally, it outlines possible directions for future work.

7.1 Problem and Objectives

This dissertation addresses the gap between high-level specification and low-level verification of hyperproperties. Hyperproperties are relevant for expressing security and privacy requirements that relate multiple executions of a system, but existing model checkers for HyperLTL generally operate over low-level representations and produce outputs that are difficult to interpret directly, so even when a verification result is obtained, understanding it remains challenging. The main objective of this work was therefore to investigate how an Alloy-based architecture could bridge this gap. This motivated the formulation of two research questions that guided this work, presented in Section 1.4: how can high-level hyperproperty specifications be effectively translated into low-level formal representations for the back-end model checker for HyperLTL (RQ1), and how can low-level feedback from verification tools be abstracted back into high-level human-readable outputs (RQ2)? In this sense, the discussion of the proposed solution and evaluation results should be understood in relation to the hypothesis presented in Section 1.5: an architecture can be designed to describe a pipeline to formally express hyperproperties at a high level, translate them into low-level formal representations to be verified with existing model checkers for HyperLTL, and present their feedback in a human-friendly way.

7.2 Proposed Solution and Contributions

The solution proposed in this dissertation consists of a verification pipeline [81] that connects high-level hyperproperty specification in Alloy with existing low-level model checkers for Hy-

perLTL. The architecture follows a pipeline in which the original Alloy specification is translated through a sequence of intermediate tools, including Electrod and HyperSMV, until it reaches a format accepted by the selected backend solver (AutoHyper or HyperQube, at the moment). After verification, the result produced by the backend is processed in reverse by Inst2SMV and Electrod until it reaches Alloy again, which can produce visualizable counterexamples. The main contributions of this work include Inst2SMV, which is the first step in reconstructing the counterexamples from the backend solvers, the modification of Alloy's visualization, enabling multiple trace visualization, and, primarily, the pipeline itself, because the most important part was joining all the aforementioned pieces to enable the full pipeline.

7.3 Evaluation Conclusions

The evaluation showed that the proposed pipeline can bridge the gap between high-level Alloy specifications and existing HyperLTL model checkers, recovering backend results in a form that can be interpreted at the Alloy level. The analysis throughout the evaluation shows that the architecture can generate necessary intermediate representations, call different backends, and transform verification feedback into visualizable counterexamples; however, there are still cases where it cannot do so. Regarding RQ1, the evaluation shows that high-level hyperproperty specifications can be translated into low-level formal representations accepted by backend model checkers for HyperLTL. It works for some examples, demonstrating that the pipeline can generate the required intermediate files and adapt the original Alloy-level specification to the formats expected by AutoHyper and HyperQube, but not working for all examples dampens its utility, as it is not generalized to all models and properties. Regarding RQ2, the evaluation shows that low-level feedback from verification tools can be abstracted back into high-level, human-readable outputs when the backend provides sufficient information to reconstruct the corresponding traces. In successful cases, Inst2SMV and Electrod enable the transformation of backend-level counterexamples into Alloy-level temporal instances, allowing them to be inspected via Alloy's visualizer. However, the evaluation also shows that this reconstruction is fragile. Some backend results could not be reconstructed into visualizable counterexamples. This occurred either because the backend did not produce a counterexample, or because, due to a limitation in the current interaction with HyperSMV, the optimized SMV model omitted variables required for trace reconstruction. Consequently, the resulting counterexample did not contain sufficient information to be translated back into an Alloy execution and displayed by the visualization component. From a performance perspective, the results indicate that the pipeline is suitable for demonstrating the approach's feasibility, but scalability remains a concern. Execution times increase with model size, especially for AutoHyper and complete model checking. HyperQube with bounded model checking scales better at the cost of possible inconclusiveness. This trade-off extends beyond execution time to the size of the generated artifacts. AutoHyper's optimized models tend to be smaller than HyperQube's when it is able to complete, reflecting its exhaustive minimization over the full state space, but it produces no optimized artifact at all once execution times out, whereas HyperQube consistently

returns a bounded artifact regardless of grid size. Conversely, HyperQube's optimized properties remain small and largely constant in size across all grid sizes, while AutoHyper's grow with the model and eventually fail to complete as well. This is the main trade-off between the two backend tools. Overall, the evaluation supports the hypothesis introduced in Section 1.5. The results show that it is possible to design an architecture that expresses hyperproperties at a high level, translates them into low-level representations for existing HyperLTL model checkers, and reconstructs the resulting feedback into a more human-friendly form. At the same time, the evaluation shows that this is not always guaranteed for every model, property, or backend output, meaning that robustness and scalability remain important challenges.

7.4 Future Work

The limitations identified throughout the evaluation suggest several possible directions for future work. One important direction is to improve the robustness of the reconstruction process, making the tool applicable to a broader range of models and HyperLTL hyperproperties expressed in Alloy. Scalability is also an important area for future improvement. The performance results show that execution times increase with model size, especially when using complete model checking. Future work could explore optimizations in the translation process, reduce unnecessary intermediate steps, or investigate backend configurations that improve performance. Another point that could be improved is cleaning the modified Alloy codebase. Some secondary features, such as the Evaluator and the new configurations, are not fully optimized for multiple traces. Besides those, the Visualizer is not yet ready to handle executing multiple Alloy commands at once. The way the pipeline is invoked within Alloy could also be refined. And some parameters from HyperSMV/AutoHyper/HyperQube are hard-coded in Alloy's options and could be made more configurable. Finally, another improvement to consider in the future is integrating new backend HyperLTL model checkers as they are developed.

7.5 Final Remarks

This dissertation demonstrated the feasibility of using Alloy as a high-level front-end for hyperproperty verification through existing HyperLTL model checkers. The proposed pipeline demonstrates how to connect high-level specifications to low-level backend solvers and reconstruct portions of their feedback as Alloy-level visualizations. However, the approach remains dependent on external tools, and counterexample reconstruction is not yet robust for all cases. Nevertheless, the implemented architecture supports the dissertation's hypothesis and provides a foundation for future work on more usable verification environments for hyperproperties.

Bibliography

[1] Daniel Jackson. "Alloy: A Language and Tool for Exploring Software Designs". In: *Communications of the ACM* 62.9 (2019), pp. 66–76. DOI: 10.1145/3338843. URL: https://alloytools.org.

[2] Ricardo Carvalho et al. "Verification of System-Wide Safety Properties of ROS Applications". In: *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2020, pp. 7249–7254. DOI: 10.1109/IROS45743.2020.9341085. URL: https://doi.org/10.1109/IROS45743.2020.9341085.

[3] Michael R. Clarkson and Fred B. Schneider. "Hyperproperties". In: *Journal of Computer Security* 18.6 (2010), pp. 1157–1210.

[4] B. Finkbeiner, M. N. Rabe, and C. Sánchez. "Algorithms for model checking HyperLTL and HyperCTL*". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 9206. 2015, pp. 30–48. DOI: 10.1007/978-3-319-21690-4_3. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-84950983917&doi=10.1007%2f978-3-319-21690-4_3&partnerID=40&md5=60ae980d8413404dd84f4504e314bee0.

[5] Rui Daniel Queirós Faria Gonçalves. "Verificação de segurança para design de software confiável". MA thesis. FCUP, 2024.

[6] Matthew J Page et al. "The PRISMA 2020 statement: an updated guideline for reporting systematic reviews". In: *BMJ* 372 (2021). DOI: 10.1136/bmj.n71. eprint: https://www.bmj.com/content/372/bmj.n71.full.pdf. URL: https://www.bmj.com/content/372/bmj.n71.

[7] Elsevier. *Scopus*. 2024. URL: https://www.scopus.com.

[8] IEEE. *IEEE Xplore*. 2024. URL: https://ieeexplore.ieee.org.

[9] Elsevier. *ScienceDirect*. 2024. URL: https://www.sciencedirect.com.

[10] ACM. *ACM Digital Library*. 2024. URL: https://dl.acm.org.

[11] Clarivate Analytics. *EndNote*. Version 21. 2023. URL: https://endnote.com.

[12] S. Agrawal and B. Bonakdarpour. "Runtime Verification of k-Safety Hyperproperties in HyperLTL". In: *2016 IEEE 29th Computer Security Foundations Symposium (CSF)*. 2016, pp. 239–252. ISBN: 978-1-5090-2607-4. DOI: 10.1109/CSF.2016.24.

[13] B. Finkbeiner et al. “RVHyper: A runtime verification tool for temporal hyperproperties”. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 10806 LNCS. 2018, pp. 194–200. DOI: 10.1007/978-3-319-89963-3_11. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85045836026&doi=10.1007%2f978-3-319-89963-3_11&partnerID=40&md5=0ffef792736dec9cc7fc2abb39b67d11.

[14] T. Dardinier, A. Li, and P. Müller. “Hypra: A Deductive Program Verifier for Hyper Hoare Logic”. In: *Proceedings of the ACM on Programming Languages* 8.OOPSLA2 (2024). DOI: 10.1145/3689756. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85206919963&doi=10.1145%2f3689756&partnerID=40&md5=d0e891cbbabea2b08feab5eb8abd569b.

[15] M. R. Clarkson et al. “Temporal logics for hyperproperties”. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 8414 LNCS. 2014, pp. 265–284. DOI: 10.1007/978-3-642-54792-8_15. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-84900551171&doi=10.1007%2f978-3-642-54792-8_15&partnerID=40&md5=aaea957c8dded42b7328e07d960fd669.

[16] E. Bartocci et al. “Mining Hyperproperties using Temporal Logics”. In: *ACM Transactions on Embedded Computing Systems* 22.5 s (2023). DOI: 10.1145/3609394. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85171751823&doi=10.1145%2f3609394&partnerID=40&md5=d74b4bdaa99b7b57d8171c7d7279f327.

[17] T. Horak et al. “Visual Analysis of Hyperproperties for Understanding Model Checking Results”. In: *IEEE Transactions on Visualization and Computer Graphics* 28.1 (2022), pp. 357–367. DOI: 10.1109/TVCG.2021.3114866. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85118646966&doi=10.1109%2fTVCG.2021.3114866&partnerID=40&md5=0a060411167884ebb117977e803dc42f.

[18] N. Coenen et al. “Explaining Hyperproperty Violations”. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 13371 LNCS. 2022, pp. 407–429. DOI: 10.1007/978-3-031-13185-1_20. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85135863580&doi=10.1007%2f978-3-031-13185-1_20&partnerID=40&md5=4c6b18a59e66ee957c57a8ffcc2832e9.

[19] R. Beutner and B. Finkbeiner. “AutoHyper: Explicit-State Model Checking for HyperLTL”. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 13993 LNCS. 2023, pp. 145–163. DOI: 10.1007/978-3-031-30823-9_8. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85161438369&doi=10.1007%2f978-3-031-30823-9_8&partnerID=40&md5=049ced57c9dd1fba4d5235b773f6793e.

[20] T. H. Hsu, C. Sánchez, and B. Bonakdarpour. "Bounded Model Checking for Hyperproperties". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 12651 LNCS. 2021, pp. 94–112. DOI: 10.1007/978-3-030-72016-2_6. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85150217740&doi=10.1007%2f978-3-030-72016-2_6&partnerID=40&md5=886cea7ec983417d448d08c6858a853e.

[21] T. H. Hsu et al. "Efficient Loop Conditions for Bounded Model Checking Hyperproperties". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 13993 LNCS. 2023, pp. 66–84. DOI: 10.1007/978-3-031-30823-9_4. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85161444432&doi=10.1007%2f978-3-031-30823-9_4&partnerID=40&md5=a3007a0d66e3a275bb0e39432de3163e.

[22] S. Itzhaky, S. Shoham, and Y. Vizel. "Hyperproperty Verification as CHC Satisfiability". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 14577 LNCS. 2024, pp. 212–241. DOI: 10.1007/978-3-031-57267-8_9. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85192176155&doi=10.1007%2f978-3-031-57267-8_9&partnerID=40&md5=b228075cce63f5efb48d6b5fafe614b3.

[23] Arthur Correnson et al. "Finding ∀∃ Hyperbugs using Symbolic Execution". In: *Proc. ACM Program. Lang.* 8.OOPSLA2 (2024), Article 321. DOI: 10.1145/3689761. URL: https://doi.org/10.1145/3689761.

[24] R. Beutner. "Automated Software Verification of Hyperliveness". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 14571 LNCS. 2024, pp. 196–216. DOI: 10.1007/978-3-031-57249-4_10. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85192252065&doi=10.1007%2f978-3-031-57249-4_10&partnerID=40&md5=dffcd692affdac341977ec695b4ae0c7.

[25] N. Coenen et al. "Verifying Hyperliveness". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 11561 LNCS. 2019, pp. 121–139. DOI: 10.1007/978-3-030-25540-4_7. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85069805360&doi=10.1007%2f978-3-030-25540-4_7&partnerID=40&md5=e3431c24145a99dfa7b494dba551f958.

[26] L. Lamport and F. B. Schneider. "Verifying Hyperproperties With TLA". In: *2021 IEEE 34th Computer Security Foundations Symposium (CSF)*. 2021, pp. 1–16. ISBN: 978-1-7281-7607-9. DOI: 10.1109/CSF51468.2021.00012.

[27] Oyendrila Dobe et al. *Lightweight Verification of Hyperproperties*. 2023. DOI: 10.1007/978-3-031-45332-8_1. URL: https://doi.org/10.1007/978-3-031-45332-8_1.

[28] M. Assaf et al. "Hypercollecting semantics and its application to static analysis of information flow". In: *ACM SIGPLAN Notices* 52.1 (2017), pp. 874–887. DOI: 10.1145/3009837.3009889. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85119339908&doi=10.1145%2f3009837.3009889&partnerID=40&md5=8174ed82ec8d9ad178cfd7560cfab58c.

[29] Amir Pnueli. "The temporal logic of programs". In: *18th annual symposium on foundations of computer science (sfcs 1977)*. ieee. 1977, pp. 46–57.

[30] J. A. Goguen and J. Meseguer. "Security Policies and Security Models". In: *1982 IEEE Symposium on Security and Privacy*. 1982, pp. 11–11. DOI: 10.1109/SP.1982.10014.

[31] J. McLean. "A general theory of composition for trace sets closed under selective interleaving functions". In: *Proceedings of 1994 IEEE Computer Society Symposium on Research in Security and Privacy*. 1994, pp. 79–93. DOI: 10.1109/RISP.1994.296590.

[32] S. Zdancewic and A.C. Myers. "Observational determinism for concurrent program security". In: *16th IEEE Computer Security Foundations Workshop, 2003. Proceedings.* 2003, pp. 29–43. DOI: 10.1109/CSFW.2003.1212703.

[33] D. McCullough. "Noninterference and the composability of security properties". In: *Proceedings. 1988 IEEE Symposium on Security and Privacy*. 1988, pp. 177–186. DOI: 10.1109/SECPRI.1988.8110.

[34] A. Sabelfeld and D. Sands. "Dimensions and principles of declassification". In: *18th IEEE Computer Security Foundations Workshop (CSFW'05)*. 2005, pp. 255–269. DOI: 10.1109/CSFW.2005.15.

[35] David Clark, Sebastian Hunt, and Pasquale Malacaria. "Quantified Interference for a While Language". In: *Electronic Notes in Theoretical Computer Science* 112 (2005), pp. 149–166. ISSN: 1571-0661. DOI: https://doi.org/10.1016/j.entcs.2004.01.018.

[36] Michael R. Clarkson, Andrew C. Myers, and Fred B. Schneider. "Quantifying information flow with beliefs". In: *J. Comput. Secur.* 17.5 (2009), pp. 655–701. ISSN: 0926-227X.

[37] J.W. Gray. "Toward a mathematical foundation for information flow security". In: *Proceedings. 1991 IEEE Computer Society Symposium on Research in Security and Privacy*. 1991, pp. 21–34. DOI: 10.1109/RISP.1991.130769.

[38] Boris Köpf and David Basin. "An information-theoretic model for adaptive side-channel attacks". In: *Proceedings of the 14th ACM Conference on Computer and Communications Security*. New York, NY, USA: Association for Computing Machinery, 2007, pp. 286–296. ISBN: 9781595937032. DOI: 10.1145/1315245.1315282.

[39] Geoffrey Smith. “On the Foundations of Quantitative Information Flow”. In: *Foundations of Software Science and Computational Structures*. Ed. by Luca de Alfaro. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 288–302. ISBN: 978-3-642-00596-1.

[40] Hirotoshi Yasuoka and Tachio Terauchi. “On Bounding Problems of Quantitative Information Flow”. In: *Computer Security – ESORICS 2010*. Ed. by Dimitris Gritzalis, Bart Preneel, and Marianthi Theoharidou. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 357–372. ISBN: 978-3-642-15497-3.

[41] Edmund M. Clarke and E. Allen Emerson. “Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic”. In: *Logic of Programs, Workshop*. Berlin, Heidelberg: Springer-Verlag, 1981, pp. 52–71. ISBN: 354011212X.

[42] E. Allen Emerson and Joseph Y. Halpern. ““Sometimes” and “not never” revisited: on branching versus linear time temporal logic”. In: *J. ACM* 33.1 (Jan. 1986), pp. 151–178. ISSN: 0004-5411. DOI: 10.1145/4904.4999. URL: https://doi.org/10.1145/4904.4999.

[43] R. Beutner and B. Finkbeiner. “HYPERATL*: A LOGIC FOR HYPERPROPERTIES IN MULTI-AGENT SYSTEMS”. In: *Logical Methods in Computer Science* 19.2 (2023). DOI: 10.46298/LMCS-19(2:13)2023. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85161431720&doi=10.46298%2fLMCS-19%282%3a13%292023&partnerID=40&md5=308795bbef7242fbc02e22baa32fbf01.

[44] J. O. Gutsfeld et al. “Temporal Team Semantics Revisited”. In: *Proceedings - Symposium on Logic in Computer Science*. 2022. DOI: 10.1145/3531130.3533360. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85136992511&doi=10.1145%2f3531130.3533360&partnerID=40&md5=42435e3a5e2d71cf758cf8757efeb785.

[45] N. Deka et al. “Deciding Branching Hyperproperties for Real Time Systems”. In: *2024 IEEE 37th Computer Security Foundations Symposium (CSF)*. 2024, pp. 65–79. ISBN: 979-8-3503-6203-9. DOI: 10.1109/CSF61375.2024.00032.

[46] Hsi-Ming Ho, Ruoyu Zhou, and Timothy M. Jones. “Timed hyperproperties”. In: *Information and Computation* 280 (2021), p. 104639. ISSN: 0890-5401. DOI: https://doi.org/10.1016/j.ic.2020.104639. URL: https://www.sciencedirect.com/science/article/pii/S0890540120301279.

[47] M. Waga and André É. “Hyper Parametric Timed CTL”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 43.11 (2024), pp. 4286–4297. ISSN: 1937-4151. DOI: 10.1109/TCAD.2024.3443704.

[48] G. de Giacomo et al. “HyperLDLf: a Logic for Checking Properties of Finite Traces Process Logs”. In: *IJCAI International Joint Conference on Artificial Intelligence*. 2021, pp. 1859–1865. DOI: 10.24963/ijcai.2021/256. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85120722905&doi=10.24963%2fijcai.2021%2f256&partnerID=40&md5=a1c69014b3ec70abb26a127f82ddd794.

[49] S. K. Muduli, G. Takhar, and P. Subramanyan. “HyperFuzzing for SoC Security Validation”. In: *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. 2020, pp. 1–9. ISBN: 978-1-6654-2324-3.

[50] N. Coenen et al. “Smart Contract Synthesis Modulo Hyperproperties”. In: *2023 IEEE 36th Computer Security Foundations Symposium (CSF)*. 2023, pp. 276–291. ISBN: 979-8-3503-2192-0. DOI: 10.1109/CSF57540.2023.00006.

[51] Y. Wang et al. “Statistical Model Checking for Hyperproperties”. In: *Proceedings - IEEE Computer Security Foundations Symposium*. Vol. 2021-June. 2021. DOI: 10.1109/CSF51468.2021.00009. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85104205312&doi=10.1109%2fCSF51468.2021.00009&partnerID=40&md5=7c5e77e7fd8eb5d4aad1e6d1e0d3b824.

[52] Y. Wang et al. “Statistical verification of hyperproperties for cyber-physical systems”. In: *ACM Transactions on Embedded Computing Systems*. Vol. 18. 2019. DOI: 10.1145/3358232. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85073146582&doi=10.1145%2f3358232&partnerID=40&md5=a7604b552fdd508dcf856c2211bb806a.

[53] B. Finkbeiner et al. “Realizing ω-regular Hyperproperties”. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 12225 LNCS. 2020, pp. 40–63. DOI: 10.1007/978-3-030-53291-8_4. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85089233127&doi=10.1007%2f978-3-030-53291-8_4&partnerID=40&md5=4b7ca10bc74633b84014f1b094ae4828.

[54] Raven Beutner and Bernd Finkbeiner. “Software Verification of Hyperproperties Beyond k-Safety”. In: *Computer Aided Verification*. Ed. by Sharon Shoham and Yakir Vizel. Cham: Springer International Publishing, 2022, pp. 341–362. ISBN: 978-3-031-13185-1.

[55] Jan Baumeister et al. “A Temporal Logic for Asynchronous Hyperproperties”. In: *Computer Aided Verification: 33rd International Conference, CAV 2021, Virtual Event, July 20–23, 2021, Proceedings, Part I*. Berlin, Heidelberg: Springer-Verlag, 2021, pp. 694–717. ISBN: 978-3-030-81684-1. DOI: 10.1007/978-3-030-81685-8_33. URL: https://doi.org/10.1007/978-3-030-81685-8_33.

[56] Andreas Bauer, Martin Leucker, and Christian Schallhart. “Runtime Verification for LTL and TLTL”. In: *ACM Trans. Softw. Eng. Methodol.* 20.4 (2011). ISSN: 1049-331X. DOI: 10.1145/2000799.2000800.

[57] Reiner Hähnle and Marieke Huisman. “Deductive Software Verification: From Pen-and-Paper Proofs to Industrial Tools”. In: *Computing and Software Science: State of the Art and Perspectives*. Ed. by Bernhard Steffen and Gerhard Woeginger. Cham: Springer International Publishing, 2019, pp. 345–373. ISBN: 978-3-319-91908-9. DOI: 10.1007/978-3-319-91908-9_18.

[58] Ana Paula Ludtke Ferreira. “Model Checking”. In: *2011 Workshop-School on Theoretical Computer Science*. 2011, pp. 9–14. DOI: 10.1109/WEIT.2011.34.

[59] Markus N. Rabe. *MCHyper: a model checker for hyperproperties*. http://www.react.uni-saarland.de/tools/mchyper/. 2015.

[60] Bernd Finkbeiner, Christopher Hahn, and Marvin Stenger. “EAHyper: Satisfiability, Implication, and Equivalence Checking of Hyperproperties”. In: *Computer Aided Verification*. Ed. by Rupak Majumdar and Viktor Kunčak. Cham: Springer International Publishing, 2017, pp. 564–570. ISBN: 978-3-319-63390-9.

[61] Tomáš Babiak et al. “The Hanoi Omega-Automata Format”. In: *Computer Aided Verification*. Ed. by Daniel Kroening and Corina S. Păsăreanu. Cham: Springer International Publishing, 2015, pp. 479–486. ISBN: 978-3-319-21690-4.

[62] Alessandro Cimatti et al. “NuSMV 2: An OpenSource Tool for Symbolic Model Checking”. In: *Computer Aided Verification*. Ed. by Ed Brinksma and Kim Guldstrand Larsen. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 359–364. ISBN: 978-3-540-45657-5.

[63] Raven Beutner and Bernd Finkbeiner. “A Temporal Logic for Strategic Hyperproperties”. In: *32nd International Conference on Concurrency Theory (CONCUR 2021)*. Ed. by Serge Haddad and Daniele Varacca. Vol. 203. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021, 24:1–24:19. ISBN: 978-3-95977-203-7. DOI: 10.4230/LIPIcs.CONCUR.2021.24.

[64] Raven Beutner and Bernd Finkbeiner. “Model Checking Omega-Regular Hyperproperties with AutoHyperQ”. In: *Proceedings of 24th International Conference on Logic for Programming, Artificial Intelligence and Reasoning*. Ed. by Ruzica Piskac and Andrei Voronkov. Vol. 94. EPiC Series in Computing. EasyChair, 2023, pp. 23–35. DOI: 10.29007/1xjt. URL: /publications/paper/d1VW.

[65] Arie Gurfinkel. “Program Verification with Constrained Horn Clauses (Invited Paper)”. In: *Computer Aided Verification*. Ed. by Sharon Shoham and Yakir Vizel. Cham: Springer International Publishing, 2022, pp. 19–29. ISBN: 978-3-031-13185-1.

[66] Raven Beutner. *ForEx: Automated Software Verification of Hyperliveness*. Dec. 2023. DOI: 10.5281/zenodo.10436583. URL: https://doi.org/10.5281/zenodo.10436583.

[67] Axel Legay and Sean Sedwards. “On Statistical Model Checking with PLASMA”. In: *2014 Theoretical Aspects of Software Engineering Conference*. 2014, pp. 139–145. DOI: 10.1109/TASE.2014.20.

[68] Emina Torlak and Daniel Jackson. “Kodkod: A relational model finder”. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 4424 LNCS (2007). Cited by: 342, pp. 632–647. DOI: 10.1007/978-3-540-71209-1_49. URL: https://www.scopus.com/inward/record.

uri?eid=2-s2.0-37149013755&doi=10.1007%2f978-3-540-71209-1_49&partnerID=40&md5=c6d4e4e6256ed5f8b76889c99c361e55.

[69] A. Cimatti et al. "NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking". In: *Proc. International Conference on Computer-Aided Verification (CAV 2002)*. Vol. 2404. LNCS. Copenhagen, Denmark: Springer, July 2002.

[70] Roberto Cavada et al. "The nuXmv Symbolic Model Checker". In: *CAV*. 2014, pp. 334–342.

[71] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. Revised edition. MIT Press, 2011.

[72] Alcino Cunha et al. *Practical Alloy: A Hands-on Guide to Formal Software Design*. https://practicalalloy.github.io/. 2025.

[73] David Chemouil and Julien Brunel. *electrod - Formal analysis for the Electrod formal specification language*. Jan. 2021. URL: https://ocaml.org/p/electrod/latest.

[74] Nuno Macedo et al. "Lightweight specification and analysis of dynamic systems with rich configurations". In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE 2016. Seattle, WA, USA: Association for Computing Machinery, 2016, pp. 373–383. ISBN: 9781450342186. DOI: 10.1145/2950290.2950318. URL: https://doi.org/10.1145/2950290.2950318.

[75] Alexandre Duret-Lutz et al. "From Spot 2.0 to Spot 2.10: What's New?" In: *Proceedings of the 34th International Conference on Computer Aided Verification (CAV'22)*. Vol. 13372. Lecture Notes in Computer Science. Springer, Aug. 2022, pp. 174–187. DOI: 10.1007/978-3-031-13188-2_9.

[76] Yong Li et al. "Roll 1.0: ω-Regular language learning library". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 11427 LNCS (2019). Cited by: 8; All Open Access, Hybrid Gold Open Access, pp. 365–371. DOI: 10.1007/978-3-030-17462-0_23. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85064895658&doi=10.1007%2f978-3-030-17462-0_23&partnerID=40&md5=24f9e0fba06a6c1bc52f63e3ae9cbf

[77] Kyveli Doveri et al. "Inclusion Testing of Büchi Automata Based on Well-Quasiorders". In: *32nd International Conference on Concurrency Theory (CONCUR 2021)*. Vol. 203. Cited by: 6. 2021. DOI: 10.4230/LIPIcs.CONCUR.2021.3. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85115341164&doi=10.4230%2fLIPIcs.CONCUR.2021.3&partnerID=40&md5=e5e9358d81ee13bb686f2946015dde

[78] Kyveli Doveri, Pierre Ganty, and Nicolas Mazzocchi. "FORQ-Based Language Inclusion Formal Testing". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 13372 LNCS (2022). Cited by: 7, pp. 109–129. DOI: 10.1007/978-3-031-13188-2_6. URL: https://

www.scopus.com/inward/record.uri?eid=2-s2.0-85133522705&doi=10.1007%2f978-3-031-13188-2_6&partnerID=40&md5=dd89567548b03112c2d11f8d56f557f

[79] Charles Jordan, Will Klieber, and Martina Seidl. "Non-CNF QBF solving with QCIR". In: *Proceedings of the AAAI 2016 Workshop on Beyond NP*. Vol. WS-16-01 - WS-16-15. Cited by: 25. 2016, pp. 320–326. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85021955805&partnerID=40&md5=9237e621ca35593e905db795a19153c

[80] Jesko Hecking-Harbusch and Leander Tentrup. "Solving QBF by abstraction". In: *Proceedings of the 21st International Conference on Theory and Applications of Satisfiability Testing (SAT 2018)*. Vol. 277. Cited by: 17; All Open Access, Gold Open Access. 2018, pp. 88–102. DOI: 10.4204/EPTCS.277.7. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85056273847&doi=10.4204%2fEPTCS.277.7&partnerID=40&md5=9680e9d6334b6ffd816c1a488ff1ce76.

[81] Miguel Montes. *HyperPardinus*. https://github.com/MiguelLPMM/HyperPardinus. 2026.

[82] Tzu-Han Hsu, Borzoo Bonakdarpour, and César Sánchez. *HyperQB*. https://github.com/TART-MSU/HyperQB. 2021.