UNIVERSITAS SARAVIENSIS

A Temporal Logic Approach to Information-flow Control

Thesis for obtaining the title of Doctor of Natural Science
of the Faculty of Natural Science and Technology I
of Saarland University

by

Markus N. Rabe

Saarbrücken
February, 2016

Dean of the Faculty	Prof. Dr. Markus Bläser
Day of Colloquium	January 28, 2016
Chair of the Committee	Prof. Dr. Dr. h.c. Reinhard Wilhelm
Reviewers	Prof. Bernd Finkbeiner, Ph.D. Prof. David Basin, Ph.D. Prof. Sanjit A. Seshia, Ph.D.
Academic Assistant	Dr. Swen Jacobs

Abstract

Information leaks and other violations of information security pose a severe threat to individuals, companies, and even countries. The mechanisms by which attackers threaten information security are diverse and to show their absence thus proved to be a challenging problem. *Information-flow control* is a principled approach to prevent security incidents in programs and other technical systems. In information-flow control we define *information-flow properties*, which are sufficient conditions for when the system is secure in a particular attack scenario. By defining the information-flow property only based on what parts of the executions of the system a potential attacker can observe or control, we obtain security guarantees that are independent of implementation details and thus easy to understand. There are several methods available to *enforce* (or *verify*) information-flow properties once defined. We focus on *static* enforcement methods, which automatically determine whether a given system satisfies a given information-flow property for all possible inputs to the system. Most enforcement approaches that are available today have one problem in common: they each only work for one particular programming language or information-flow property.

In this thesis, we propose a temporal logic approach to information-flow control to provide a simple formal basis for the specification and enforcement of information-flow properties. We show that the approach can be used to enforce a wide range of information-flow properties with a single algorithm.

The main challenge is that the standard temporal logics are unable to express information-flow properties. They lack the ability to relate multiple executions of a system, which is essential for information-flow properties. We thus extend the temporal logics LTL and CTL* by the ability to quantify over multiple executions and to relate them using boolean and temporal operators. The resulting temporal logics HyperLTL and HyperCTL* can express many information-flow properties of interest. The extension of temporal logics compels us to revisit the algorithmic problem to check whether a given system (model) satisfies a given specification in HyperLTL or HyperCTL*; also called the *model checking* problem. On the technical side, the main contribution is a model checking algorithm for HyperLTL and HyperCTL* and the detailed complexity analysis of the model checking problem: We give nonelementary lower and upper bounds for its computational complexity, both in the size of the system and the size of the specification. The complexity analysis also reveals a class of specification, which includes many of the commonly considered information-flow properties and for which the algorithm is efficient (in NLOGSPACE in the size of the system). For this class of efficiently checkable properties, we provide an approach to reuse existing technology in hardware model checking for information-flow control. We demonstrate along a case study that the temporal logic approach to information-flow control is flexible and effective. We further provide two case studies that demonstrate the use of HyperLTL and HyperCTL* for proving properties of error resistant codes and distributed protocols that have so far only been considered in manual proofs.

Zusammenfassung

Informationssicherheit stellt eine immer größere Bedrohung für einzelne Personen, Firmen und selbst ganze Länder dar. Ein grundlegender Ansatz zur Vorbeugung von Sicherheitsproblemen in technischen Systemen, wie zum Beispiel Programmen, ist *Informationsflusskontrolle*. In der Informationsflusskontrolle definieren wir zunächst sogenannte Informationsflusseigenschaften, welche hinreichende Bedingungen für die Sicherheit des gegebenen Systems in einem Sicherheitsszenario darstellen. Indem wir Informationsflusseigenschaften nur auf Basis der möglichen Beobachtungen eines Angreifers über das System definieren, erhalten wir einfach zu verstehende Sicherheitsgarantien, die unabhängig von Implementierungsdetails sind. Nach der Definition von Eigenschaften muss sichergestellt werden, dass ein gegebenes System seine Informationsflusseigenschaft erfüllt, wofür es bereits verschiedene Methoden gibt. Wir fokussieren uns in dieser Arbeit auf statische Methoden, welche für ein gegebenes System und eine gegebene Informationsflusseigenschaft automatisch entscheiden, ob das System die Eigenschaft für alle möglichen Eingaben erfüllt, was wir auch das Modellprüfungsproblem nennen. Die meisten verfügbaren Methoden zum Sicherstellen der Einhaltung von Informationsflusseigenschaften teilen jedoch eine Schwäche: sie funktionieren nur für eine einzelne Programmiersprache oder eine einzelne Informationsflusseigenschaft.

In dieser Arbeit verfolgen wir einen Ansatz basierend auf Temporallogiken, um eine einfache theoretische Basis für die Spezifikation von Informationsflusseigenschaften und deren Umsetzung zu erhalten. Wir analysieren den Zusammenhang von der Ausdrucksmächtigkeit von Spezifikationssprachen und dem algorithmischen Problem Spezifikationen für ein System zu überprüfen. Anhand einer Fallstudie im Bereich der Hardwaresicherheit zeigen wir, dass der Ansatz dazu geeignet ist eine breite Palette von bekannten und neuen Informationsflusseigenschaften mittels eines einzelnen Modellprüfungsalgorithmus zu beweisen.

Das Kernproblem hierbei ist, dass wir in den üblichen Temporallogiken Informationsflusseigenschaften nicht ausdrücken können, es fehlt die Fähigkeit mehrere Ausführungen eines Systems miteinander zu vergleichen, was der gemeinsame Nenner von Informationsflusseigenschaften ist. Wir erweitern Temporallogiken daher um die Fähigkeit über mehrere Ausführungen zu quantifizieren und diese miteinander zu vergleichen. Der Hauptbeitrag auf der technischen Ebene ist ein Modellprüfungsalgorithmus und eine detaillierte Analyse der Komplexität des Modellprüfungsproblems. Wir geben einen Modellprüfungsalgorithmus an und beweisen, dass der Algorithmus asymptotisch optimal ist. Die Komplexitätsanalyse zeigt auch eine Klasse von Eigenschaften auf, welche viele der üblichen Informationsflusseigenschaften beinhaltet, und für welche der gegebene Algorithmus effizient ist (in NLOGSPACE in der Größe des Systems). Für diese Klasse von effizient überprüfbaren Eigenschaften diskutieren wir einen Ansatz bestehende Technologie zur Modellprüfung von Hardware für Informationsflusskontrolle wiederzuverwenden. Anhand einer Fallstudie zeigen wir, dass der Ansatz flexibel und effektiv eingesetzt werden kann. Desweiteren diskutieren wir zwei weitere Fallstudien, welche demonstrieren, dass die vorgeschlagene Erweiterung von Temporallogiken auch eingesetzt werden kann, um Eigenschaften für fehlerresistente Kodierungen und verteilte Protokolle zu beweisen, welche bisher nur Abstrakt betrachtet werden konnten.

Acknowledgements

It is impossible to do justice to all those wonderful people who contributed in direct and indirect ways to this thesis. Nevertheless I want to highlight several persons and groups to which I am particularly indebted.

I want to start with expressing my deep gratitude to my advisor Bernd Finkbeiner. I admire his commitment to his students and his ability to provide subtle guidance. He showed a great deal of patience with me and always kept his door open. I cannot imagine a better supervisor.

I want to thank my colleagues, Rayna Dimitrova, Klaus Dräger, Rüdiger Ehlers, Peter Faymonville, Michael Gerke, Swen Jacobs, Felix Klein, Lars Kuhtz, Andrey Kupriyanov, Hans-Jörg Peter, Christa Schäfer, Leander Tentrup, Hazem Torfah, and Martin Zimmermann for the countless shared coffees and inspiring conversations. I am grateful for having worked with Sven Schewe during the first year of my PhD. Even though our common work is not directly a part of this thesis, my research is heavily influenced by him. Thanks also to my doctoral committee for their time and feedback; the external reviewers, David Basin and Sanjit A. Seshia, the committee chair Reinhard Wilhelm, and the academic staff committee member Swen Jacobs.

It would take too many pages to describe what each of my friends means to me, but at least I want to mention their names here. I will always remember my flatmates with whom I shared a home for all those years: Justus, Anika, Claudia, Jana; and of course the co-founders Klaas and Matthias who also happen to be two of my oldest friends. Let us enjoy more roof parties, sunny afternoons on the balcony, and evenings in the kitchen. I am grateful for having met so many wonderful people during conferences, summer schools, and my stay at Cambridge; in particular I want to mention Heidy Khlaaf, Christoph Wintersteiger, and Anton Stefanek. The friends I made in Saarbrücken influenced this thesis in so many ways; Anna Marie, Anne-Christin, Christian, Eva, Fabian, Jonas, Jörg, Kevin, Nikolai, Nora, Richard, and Verena, and also my cinema buddies Ariane and Jana. And finally I want to mention two friends who had big influence on this thesis: Tim and Raphael. Thanks to all these friends for making my time in Saarbrücken so enjoyable and productive, and for helping me during difficult times.

Last but not least, a special thanks to my parents, Winfried and Cornelia, and my sisters, Lena and Gerda. Without their continuous support and their love this thesis would not have been possible.

Contents

Chapter 1

Introduction

1.1 Information Security

Information leaks can have serious implications for individuals and organizations. The loss of personal communication, bank statements, location data, health records, or passwords is suited to severely harm an individuals personal freedom and reputation. On a larger scale, companies are concerned about loosing their users' data as it is suited to deteriorate the trust in the company's services. Even the operation of governmental organizations, like the military or intelligence services, can be severely harmed by information leaks.

The *integrity* of information and services can be an even bigger concern than secrecy. Integrity denotes that information or services cannot be altered maliciously. The widespread use of digital technology in infrastructure, such as power nets, nuclear plants, or (air) traffic, opens the possibility of attacks and it is clear that their malicious manipulation could lead to catastrophes of vast scale. The use of the Stuxnet worm to destroy nuclear facilities in Iran [3] can be seen as a proof of concept for the suitability of such attacks for warfare.

Similar to the integrity of information, it may suffice to interfere with the *availability* of information to cause serious harm. *Distributed denial of service attacks* (DDOS attacks) are a common technique to inhibit the availability of particular websites or internet-based services in the internet. DDOS attacks make use of computer viruses to infect and thereby control large numbers of computers, called botnets, and then bombard a particular internet-based service with vast numbers of meaningless requests issued by the controlled computers. Even though this kind of attack can be easily detected, and is, in fact, observed many thousand times per day, the use of botnets makes it extremely hard to track the source of the attack.

Confidentiality (i.e. the absence of information leaks), integrity, and availability of information constitute the so called *CIA triad* that defines the goals of *information security* [121]. The exact definition of the set of goals is, however, disputed and sometimes also lists closely related concerns like account-ability or authenticity (e.g. [107]). The academic study of information security concerns the analysis of threats to any of these goals and of countermeasures against them. In computer science, we typically study information security for a particular technical system, such as a program, a computer, or network of computers and we also restrict our attention to technical countermeasures. General information security, however, can be addressed on many levels and includes legal and economical aspects [57].

Violations of Information Security

The recent years have shown an alarming number of severe incidents in information security. Through coverage by the media, information security even starts to become a public concern [68, 106, 142, 102, 116, 122, 43, 128, 44, 101, 46, 123]. The vast majority of violations of information security, however, likely remain undisclosed; either because they are not detected in the first place or, in the case of detection, because they are held secret to avoid erosion of the trust in the victim's ability to fend of future attacks.

A fundamental problem in information security is that the causes of security violations are extremely diverse. In fact there are at least hundreds of types of vulnerabilities as reported by the Mitre research center [1]. To give the unfamiliar reader an impression on the range of possible mechanisms, by which a system can be attacked, we review two noteworthy security incidents, discuss problems in hardware security, and information-leaks via variations in execution times.

Heartbleed

Heartbleed[1] is maybe the most severe breach of information security that we have witnessed so far. It enables attackers to extract secret data like the encryption keys from web servers and thereby renders their cryptographic measures useless.

Heartbleed is a vulnerability in OpenSSL, an implementation of the cryptography protocol SSL/TLS. SSL/TLS is used to encrypt the communication between two parties and includes a "heartbeat" request (giving rise to the name of the vulnerability) that allows the communicating parties to check whether the other side is still responsive. Normally the "heartbeat" request consists of a short word, which the other party is supposed to send back, and also a number indicating the length of the word. The implementation of the standard in OpenSSL did not check whether the number indicating the length of the word matched the actual length of the word. Sending a request that indicates a larger word length x than the actual length y of the word (e.g. word: "hat", length of the word: "500" characters) made the other party to answer with the next x characters it stored in memory, instead of the intended y characters. The request could thus be misused to request arbitrary parts of a web

[1] It has become common to give names to major security incidents.

servers memory, which could include, for example, encryption keys or passwords of users. Figure 1.1 depicts how the attack works.

Fixing OpenSSL once the problem was identified was very simple and mainly consists of adding the following two lines of code [60]:

```
if (1 + 2 + payload + 16 > s->s3->rrec.length)
    return 0;
```

This piece of code suppresses the reply to a heartbeat request, whenever the word length indicated in the message is larger than the actual word length.

Heartbleed existed since March 2012 and was discovered only in April 2014 [49]. At the time Heartbleed was revealed many of the most commonly visited web servers were vulnerable to the attack [25]. Even though no attacks based on Heartbleed prior to its release are known, the months after its release showed a number of attacks on web services that did not patch their OpenSSL library timely. For example, Community Health Systems, Inc. reported the loss of 4.5 million patient records from their database due to Heartbleed [32, 133].

Shellshock

Shellshock affected the Bash, a widely used interpreter for the Bash script language [51]. Even though the Bash shell is typically not directly interacting with requests from the internet, it is often used within web servers to process parts of these requests. Sending specially prepared requests made the Bash shell executing arbitrary commands on the web server. For example a request via the hyper text transfer protocol (HTTP), as used by web browsers when a website is opened, could be prepared as follows:

```
GET / HTTP/1.1
Accept-Encoding: gzip,deflate,sdch
Accept-Language: en-US,en;q=0.8,fr;q=0.6
Cache-Control: no-cache
Pragma: no-cache
User-Agent:() { :; }; /bin/cat /etc/passwd
Host: www.victim.com/website.html
```

This HTTP request asks the web server with the URL `www.victim.com` to send the file `website.html`. Additionally, the request indicates several pieces of information, such as the preferred language (`Accept-Language`), possible encodings of the reply (`Accept-Encoding`), and program that sent the request (`User-Agent`). The last field would typically indicate the browser used to open the website, such as `Mozilla/5.0 (Macintosh; Intel Mac OS X 10_9_4) AppleWebKit/537.36 Safari/537.36`. Here, however, the attacker inserted the string `() { :;};` followed by the malicious command:

```
/bin/cat /etc/passwd
```

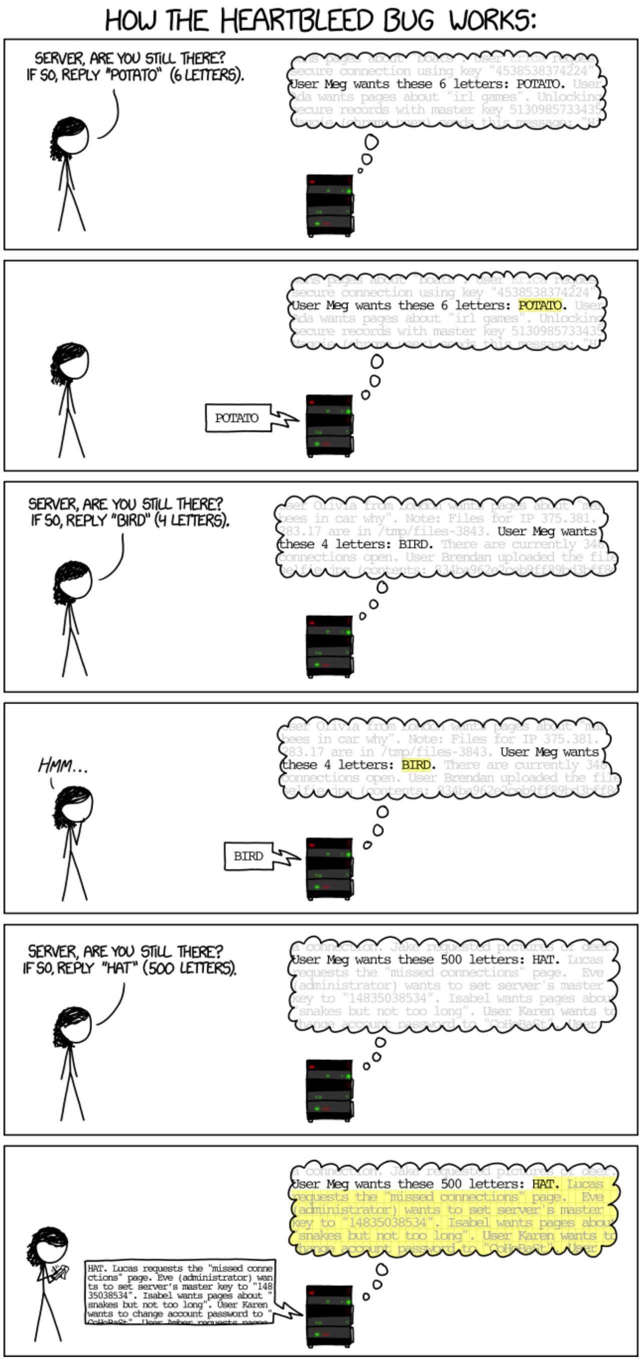

HOW THE HEARTBLEED BUG WORKS:
SERVER, ARE YOU STILL THERE? IF SO, REPLY "POTATO" (6 LETTERS).
ants pages about "boats". User Erica reques secure connection using key "4538538374224" User Meg wants these 6 letters: POTATO. User Ada wants pages about "irl games". Unlocking secure records with master key 5130985733435 Maggie (chrome user) sends this message: "H
ants pages about "boats". User Erica reques secure connection using key "4538538374224" User Meg wants these 6 letters: POTATO. User Ada wants pages about "irl games". Unlocking secure records with master key 5130985733435 Maggie (chrome user) sends this message: "H
POTATO
SERVER, ARE YOU STILL THERE? IF SO, REPLY "BIRD" (4 LETTERS).
User Olivia from London wants pages about "ma bees in car why". Note: Files for IP 375.381. 283.17 are in /tmp/files-3843. User Meg wants these 4 letters: BIRD. There are currently 346 connections open. User Brendan uploaded the file selfie.jpg (contents: 834ba962e2ceb9ff89bd3bff8
HMM...
User Olivia from London wants pages about "ma bees in car why". Note: Files for IP 375.381. 283.17 are in /tmp/files-3843. User Meg wants these 4 letters: BIRD. There are currently 346 connections open. User Brendan uploaded the file selfie.jpg (contents: 834ba962e2ceb9ff89bd3bff8
BIRD
SERVER, ARE YOU STILL THERE? IF SO, REPLY "HAT" (500 LETTERS).
a connection. Jake requested pictures of deer. User Meg wants these 500 letters: HAT. Lucas requests the "missed connections" page. Eve (administrator) wants to set server's master key to "14835038534". Isabel wants pages about "snakes but not too long". User Karen wants to change account password to "CoHoBaSt". User
a connection. Jake requested pictures of deer. User Meg wants these 500 letters: HAT. Lucas requests the "missed connections" page. Eve (administrator) wants to set server's master key to "14835038534". Isabel wants pages about "snakes but not too long". User Karen wants to change account password to "CoHoBaSt". User
HAT. Lucas requests the "missed connections" page. Eve (administrator) wants to set server's master key to "14835038534". Isabel wants pages about "snakes but not too long". User Karen wants to change account password to "CoHoBaSt". User Amber requests pages

Figure 1.1: How the Heartbleed vulnerability was exploited as explained by `http://xkcd.com` [98].

which lets the web server append a particular password file to the answer to the HTTP request.

The attack uses the fact that these items of data are often processed by Bash scripts, by which they are parsed and stored in variables. The following piece of code was used for this task:

```
315  /* Initialize the shell variables from the current environment.
316     If PRIVMODE is nonzero, don't import functions from ENV or
317     parse $SHELLOPTS. */
318  void
319  initialize_shell_variables (env, privmode)
320    char **env;
321    int privmode;
322  {
[...]
329    for (string_index = 0; string = env[string_index++]; )
330    {
[...]
350      /* If exported function, define it now.  Don't import functions
351       from the environment in privileged mode. */
352      if (privmode == 0 &&
353          read_but_dont_execute == 0 &&
354          STREQN ("()␣{", string, 4))
355      {
[...]
361        parse_and_execute (temp_string, name, SEVAL_NONINT|SEVAL_NOHIST);
362        [...]
363      }
364      [...]
365    }
366    [...]
367  }
```

For parsing the said variable definitions, the function `parse_and_execute` is called in line 361. This function, however, does not only parse, but also execute any code contained in the variable `tmp_string`. If the variable contains a single function definition, as it was intended, no problem occurs. The string `() { :; };` that was embedded in the HTTP request, however, ends the function definition. Any commands that come after that string are then simply executed. The problem was fixed by introducing a check making sure that the variable `temp_string` contained only a function definition.

The vulnerability in the code of Bash was introduced in 1992 [109] and was in use until it was detected and published in September 2014. That is, the Shellshock vulnerability existed for more than two decades. Scans revealed that an alarming number of web servers were vulnerable at the time when the problem was published and shortly after its publication, the bug was widely exploited; at peak times over a million individual attacks and probes were detected within 24 hours. The number of compromised systems and the effects can hardly be estimated.

Security-critical Programming Errors

We have seen two major security incidents that stand prototypical for violations of confidentiality (Heartbleed) and violations of integrity (Shellshock) and both were caused by simple *programming errors*. Developers working on security critical implementations, such as cryptographic protocols, are typically highly skilled and very aware of the criticality of their code. They follow the best practices of creating secure code, such as code reviews, and in the cases discussed above the source code is openly available, such that every interested person can analyze the code. Yet mistakes happen frequently and may remain unnoticed for years.

Hardware Security

Security critical programming errors also occur in hardware such as microprocessors. Vendors like Intel and AMD publish lists of errors found in their processors, so called *specification updates*, though typically without detailed information about their impact in case of malicious usage. A study found that these errors include security critical errors, such as bugs that lead to privilege escalation or arbitrary memory access [61]. Attackers that can execute a program on erroneous hardware can use these errors to circumvent otherwise secure protection mechanisms in operating systems or browsers.

A big problem with security critical errors in hardware is that they are particularly hard to fix once the hardware is shipped. Hence most vulnerabilities remain attackable even after they have been detected.

Timing Channels

The exact time at which a computation finishes may also contain information about secret information used during the execution, which gives rise to the term *timing channel*. Several attacks have demonstrated the feasibility of reconstructing secret information from subtle timing dependencies. In a well known work, Brumley and Boneh demonstrated that by statistical analysis of the answering times of a web server the encryption keys of a web server can be extracted [22]. The problem was that the implementation of certain cryptographic operations depended on the key in use.

Also concurrently executed processes that share the same hardware resources, such as the cache infrastructure of a processor and the instruction pipeline, pose a security problem. Depending on which processes running in parallel, the execution times of the processes vary slightly. It has been demonstrated that simple timing dependencies can suffice to reconstruct secret information, such as encryption keys [105].

Countermeasures

Once a security problem has been identified it is typically possible to counter the threat, for example by updating a piece of software. We suggest that the main challenge in information security is thus to find security problems in the first place.

Today, the best practice in finding security problems is to rely on code reviews and testing. *Testing* is somewhat effective at finding programming errors that affect the anticipated behavior of the program, but it is less suited to find bugs for unusual inputs, as it is hard to anticipate all possible kinds of inputs for which the software should fail. Errors like Heartbleed and Shellshock, however, are of that kind. While *code reviews* are effective at identifying security problems [77], they rely on highly trained experts and are therefore costly. Only large software companies or companies specialized to security critical systems are therefore able to implement the thorough procedures needed to ensure a reasonably high level of security for their systems.

More automatic approaches are suited to reduce the cost of security measures and the entry barrier to apply them. Popularly cited countermeasures with a higher degree of automation include fuzz testing, memory-safety measures, and static analysis. The excellent essay by David A. Wheeler elaborates on which of them could have helped to detect or prevent Heartbleed [143]. The issue with the methods that would have prevented Heartbleed, however, is that they would not have helped against Shellshock, not to speak of timing attacks, because the mechanisms through which the information leaked were fundamentally different.

This represents a fundamental problem in information security: There is an endless number of mechanisms by which a system can be compromised, as witnessed by the list of hundreds of types of software vulnerabilities maintained by the Mitre research center [1], but most countermeasures only defend against a particular type of attacks. This calls for a principled approach to information security.

1.2 A Property-oriented Approach

In this thesis, we follow a *property-oriented approach* to information security. Starting from the basic goals of information security, such as confidentiality and integrity, we formulate mathematically precise specifications (properties) of secure system behavior. The idea is that for a given system we want to mathematically prove (or otherwise enforce) that the system satisfies these specifications.

To ensure that the specifications provide general security guarantees - and do not only protect against a specific kind of attack - it is crucial that the specifications are independent of the mechanisms by which information security is violated. We thus consider specifications that only refer to the possible observations of an attacker about the inputs and outputs of the given system. Figure 1.2 depicts this so called *black-box* view on systems: we are only interested in which inputs I are read by the system and which outputs O are produced in response to these inputs. For a web server, for example the inputs and outputs could represent the received and sent network traffic.

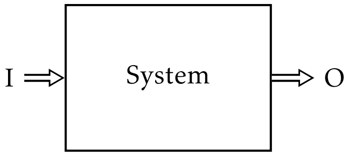


Figure 1.2: The abstract view on a reactive system. Specifications should only refer to the observable behavior of a system. We are indifferent to implementation details, such as whether a system is implemented as software or hardware.

The formalization of information security was started in the early eighties with formal specifications of confidentiality (or interchangeably *secrecy*) such as the notion of *noninterference* for multi-agent systems by Goguen and Meseguer [55]. It assumes that the attacker can observe the so called *public* part of the inputs and the outputs, and that the non-public part of the inputs constitutes the secret information.

Excursion: A Formal Definition of Noninterference

In this excursion we present the simple formal definition of noninterference. To have an idea what we specify, we need to introduce a formal *system model* with a minimum of formal notation.

We assume that the system operates stepwise; in the step with number k, the system reads an input i_k and outputs a value o_k. The system may have internal state, such that its outputs may depend not only on the current input, but also on previous inputs. This could, for example, represent a web server, where the HTTP requests are modeled as inputs, and the websites sent in return are modeled as outputs.

For simplicity let us assume that the input I_k to a system at a particular point of time k consists of the boolean variables $i_1,\dots,i_n$, and likewise the output O consists of the boolean variables $o_1,\dots,o_m$. That is, if i_j is in set I_k then input variable i_j is true at time k. An execution trace, or short *trace*, of a system is a sequence of inputs $I_1, I_2 \dots$ together with a sequence of outputs $O_1, O_2, \dots$ produced by the system on this sequence of inputs. Formally, the trace is defined to be the sequence $(I_1 \cup O_1), (I_2 \cup O_2), \dots$. (By $I_k \cup O_k$ we denote the *union* of the sets I_k and O_k, i.e. the set of elements that contains the elements of both I_k and O_k.) For now, we define systems K by the set of traces they generate, denoted by $\mathrm{Traces}(K)$. (We choose the letter K for systems to be consistent with their more detailed formalization as <u>K</u>ripke structures that follows later.)

We now present a modern version of noninterference, adapted to our system model. We assume that the public inputs I_p are a subset of the input variables I and the public outputs O_p are a subset of the output variables O. In formal notation noninterference is then defined as follows:

$$\forall t, t' \in \mathsf{Traces}(K): \ t =_{I_p} t' \implies t =_{O_p} t' \ ,$$

where $t =_{I_p} t'$ and $t =_{O_p} t'$ denotes the stepwise equality of the public inputs and public outputs, respectively. The definition reads as follows: all pairs of traces of a system that have the same sequence of public inputs must also have the same public outputs. That is, confidentiality is defined as the public output's independence of the secret information (i.e. the non-public inputs).

The insight behind the definition of noninterference is that, to precisely specify confidentiality for a given system, we only need to identify which parts of the inputs and outputs of the system are directly observable by the attacker. In particular, this definition guarantees the confidentiality of a system independent from the mechanism by which information may be leaked. The only assumption is that the model of the system truthfully represents what the attacker can observe about the system. Based on this insight, the field of *information-flow control* emerged. In information-flow control we discuss noninterference-like specifications, which we call *information-flow properties*, and ways to prove or enforce that a system satisfies the specification [55, 87, 89, 85, 119].

Since then a plenitude of information-flow properties were proposed, including different generalizations confidentiality, such as noninference [89] and generalized noninterference [87]. In some scenarios, we also need to allow for exceptions under which information may be leaked, so called *declassification policies* [120]. Consider a program that is supposed to show a particular website only after the user enters the correct password. The password checker necessarily needs to provide different answers depending on whether the provided password matches the correct password. This scenario includes two declassifications: (1) upon entering an incorrect password, the fact that the correct password is different to the provided password needs to be revealed, and (2) upon entering the correct password, the previously secret website becomes accessible. A large variety of different scenarios where declassification is needed and properties that match the needs of particular scenarios were discussed and classified in the literature [120].

Enforcement of Security Policies

Specifying secure behavior alone does not prevent security incidents. We also need to *enforce* that a system adheres to a given information-flow property. The enforcement of an information-flow property before a program is executed is called *static enforcement*. The most commonly considered static enforcement approaches are *language-based* information-flow control and enforcement based on *program analysis*.

In language-based information-flow control we consider programming languages that come with a *security type system* [140, 119]. A security type system assigns a *security type* to every occurrence of every variable in a program. The available security types depend on the security type system, but often consist of the two types *secret*, indicating that it may contain secret information, and *public*, indicating that it certainly contains no secret information. The security type system *accepts* a program, if the observable outputs (or observable variables) are typed *public*. The security type systems are designed in a way such that a program that is accepted by the security type system is guaranteed to satisfy a specific information-flow property—and there are different type systems for different information-flow properties. The rules according to which it is decided which security type is given to the occurrences of each variable depend on the language and the property. As an example, consider the following piece of code:

```
int z = x + y;
```

This statement introduces a variable z and assigns to it the sum of the variables x and y. Assume that variable x has the security type *secret* and that variable y has the security type *public*. For typical programming languages and information-flow properties, the variable z would then be assigned the security type *secret*, as it may contain secret information—in particular in case y is 0, z is equal to x after the assignment.

More intricate cases to consider in security type systems include *control-flow dependencies*:

```
if (x == 5) {
   z = 0;
} else {
   z = 1;
}
```

This piece of code assigns the variable z the values 0 or 1 depending on whether x is equal to 5. Assuming the variable x to have the security type *secret*, typical security type systems would also give the security type *secret* to the variable z, as variable z now contains the information whether or not the *secret* variable x is 5. Control-flow dependencies are representative for a large number of intricate information flows.

Designing a security type system requires to think of all possible ways in which information can propagate from one variable to another variable. While most of the approaches in language-based information-flow control are not available for real programming languages it has been demonstrated that they can be extended to real programming languages such as Java [100] or Verilog [149].

A problem with security type systems is that they may not be precise. In the case of the if-statement above, we may know from the context of the code that x cannot have value 5. In this case, security type systems would still assign z the security type *high*, since it *abstracts* from the actual semantics of the program to simplify the analysis. In this way it can happen that perfectly secure programs are not accepted by a security type system. That is, security type systems are “inherently imprecise” [12].

As an alternative to security type systems also program analysis [36, 82, 6, 58, 72, 71] and manual (or semi-automatic) proofs [66, 99] has been proposed as a means to statically enforce information-flow properties. Program analyses for information-flow properties, similar to security type systems, abstract from the exact semantics of the program and may thus give false alarms in the analysis.

Lastly, it has been observed that instead of analyzing the original program for the information-flow property, we can analyze a transformed program for a more simple property—a property over single execution traces [12, 129, 13, 11]. The transformation, called *self-composition*, creates two copies of the original system that run in parallel. A run of the self-composed system corresponds to two runs of the original system. Hence we can express certain information-flow properties of the original system as properties over single executions of the self-composed system. The advantage over the previously considered techniques for enforcing information-flow properties is self-composition enables us to reuse existing theories and tools for the analysis of single-trace properties, and hence profits from the precision and scalability of the available approaches. First approaches demonstrated the feasibility of model checking information-flow properties [63].

An alternative to the static enforcement approach is to enforce the compliance with an information-flow property at the runtime of a program, which is called *dynamic enforcement*. That is, a specially programmed *monitor* observes the run of the program and raises an alarm (or stops the execution of the monitored program) whenever a violation of a specific information-flow property is detected [42, 37]. This thesis focusses on static enforcement, though an extension to dynamic enforcement seems possible [104].

Challenges in Information-flow Control

Despite its promise to provide general security guarantees, information-flow control has not been widely adapted in the practice of information security so far. One major problem shared by most of the approaches in information-flow control is that they focus on the enforcement of a single and fixed information-flow property. For example, these methods are so specialized that even the fact whether the termination is observable by the attacker is hard-coded in the enforcement approach. Most of the approaches also do not admit flexible declassification methods, such that they are not applicable for simple scenarios like the password checker.

However, even minor changes of the information-flow property require a redesign of the security type system or the program analysis technique and also non-trivial changes in the proof of soundness of the technique. We thus need approaches that support a wide range of information-flow properties.

We are not the first to observe this deficiency of existing enforcement methods. In an invited talk on information-flow control at the ETAPS conferences in 2014 David Maziéres stated that one of the next challenges in information-flow control is to design specification languages that allow us to easily specify information-flow properties [53].

First steps in this direction were taken with the development of specification frameworks for information-flow control. Early works, such as the *Modular Assembly Kit for Security Properties* (MAKS) by Mantel [85, 86], provided the basis for comparing various information-flow properties. More recent works on specification frameworks focus on generality: Clarkson and Schneider propose to consider a class of properties they coin *hyperproperties*. Hyperproperties only refer to the traces of a system, but may relate multiple traces [31]. They show that the notion of hyperproperties includes many of the known information-flow properties.

Unfortunately, these specification frameworks do not come with approaches for the automatic enforcement of properties. What we need are specification languages for information-flow properties that enable the automatic enforcement of the specified properties.

1.3 Temporal Logics

Temporal logics provide a unique connection between specification languages and the enforcement of specifications. In this context, the static enforcement of properties is usually called *verification* or *model checking*. The origins of temporal logics lie in the philosophical inquiry into the reasoning about temporal relationships, but their use in computer science was soon discovered by Pnueli [110]. He proposed what today is called the *linear-time temporal logic* (LTL). LTL is now the standard logic to describe properties of individual traces, called *trace properties*, such as *invariants* ("$x > y$ is true for all times") or *response* properties ("every event a is followed by an event b").

LTL is built from simple logical operations such as the *globally* operator $\square\varphi$ and the *eventually* operator $\diamondsuit\varphi$. The globally operator $\square\varphi$ expresses that its subformula φ has to hold for all future times, i.e. φ is invariant. The eventually operator $\diamondsuit\varphi$ expresses that its subformula φ has to hold for some time in the future. Using these simple building blocks, we can express more complex properties, such as that an event a has to hold infinitely often: The formula $\square\diamondsuit a$ expresses that for all times, there is a point further in the future where a holds. If there were only finitely many events a, the subformula $\diamondsuit$ would be violated after the last occurrence of a.

The semantics of LTL is not concerned with the meaning of events such as a or expressions like "$x > y$." Statements that can be interpreted independent of time (i.e. are either true or false at any point of time) are considered as *atomic propositions* and LTL only defines their temporal relationship.

LTL specifies properties based on single traces. A system satisfies an LTL property, if all its traces, considered individually, satisfy the property. As its name suggests this logic thus corresponds to the philosophical view that time is linear—by considering a fixed trace, we implicitly fix the complete future of the computation. The rivaling philosophical view is that of *branching time*, which assumes that at any point in the execution different futures are still possible. This view led to the so called *branching-time temporal logics*, such as the *computation tree logic* CTL [29] and the more general CTL* [41]. CTL* extends LTL by existential and universal quantification over traces, such as "there is a trace where event *a* occurs eventually". Combining quantification over traces with temporal operators we can express more complex properties, such as $\mathrm{A}\square(\mathrm{E}\diamondsuit\mathrm{off})$, which expresses that along all computations (A) and for all points of time ($\square$), there exists a continuation of the computation (E) where eventually ($\diamondsuit$) the atomic proposition "off" holds. In other words, *there is always a way to shut the system off.* (This does not mean, however, that the system is necessarily shut off eventually.)

From the beginnings of temporal logics in computer science the *model checking problem* was one of the defining research questions. The model checking problem is to decide whether a given system satisfies a given property specified in a temporal logic. Temporal logics are considered to provide a unique understanding of the relationship between specifications and the problem to check a system for accordance with a specification. Many variants of temporal logics were proposed with the aim to study how the expressiveness of temporal logics interacts with our ability to automatically check whether a system satisfies a given specification.

The first model checking algorithms considered a system as an *automaton* (also called a labeled graph) that consists of a list of states with a designated initial state and a list of transitions between the states. The states may further have labels that indicate, e.g. whether an atomic proposition such as "$x > 5$" holds in this state. For a graphical representation of an automaton take a glance at Fig. 2.1. The problem to check whether a simple property like "for all times it holds $x > 5$," that is $\square x > 5$ in LTL notation, is then simply to determine whether there is a path from the initial state to a state where "$x > 5$" does *not* hold using the transitions of the automaton. If such a path is found, the system violates the property. It is clear that this search can be done automatically. The connection to automata was discovered early and led to a well-understood branch of automata-theory [144, 138, 76].

Even though these algorithms provide a beautiful theoretical basis and were proven to be optimal in terms of worst-case complexity, they only allow us to check small systems. Larger system show the so called *state-space explosion* problem. That is, the number of states of a system grows exponentially with the number of variables. The run-times of algorithms that consider individual states, so called *explicit-state* algorithms, thus also grow exponentially and quickly become ineffective.

Relief was brought by *symbolic* model checking algorithms that try to avoid considering systems as a list of states and transitions, and instead operate on the level of a succinct representation of the system, such as *binary decision diagrams* [90] or boolean logic [16]. In the worst-case, symbolic algorithms cannot perform faster than the previous explicit-state algorithms, but in practice they often work for systems that simply could not be checked by explicit-state algorithms. A series of developments in *symbolic* model checking algorithms enabled checking substantial pieces of software and hardware [91, 9, 28, 111, 39, 20, 33, 34]. In this way, implementation errors can now be found effectively and early in the design process. An extension of LTL, the *property specification language* (PSL) [2], is used to find design flaws in hardware modules, such as CPUs.

Challenges in Temporal Logics

It would be desirable to use temporal logics also for the specification of information-flow properties. This would link the problem to enforce information-flow properties to the well understood theory on model checking and it would allow us to leverage the existing model checking algorithms for finding violations of information-flow properties in software and hardware.

It turns out, however, that information-flow properties are not expressible in the commonly considered temporal logics and thus the model checking approach did not seem to be applicable to information security [5]. LTL considers traces individually and hence it is not surprising that information-flow properties cannot be expressed using LTL. At a first glance, the quantification over executions in CTL and CTL* seems to be a reasonable candidate to express properties relating multiple traces. Both logics restrict the semantics, however, to always refer only to one execution at a time. In the example property $\mathsf{A}\square(\mathsf{E}\diamond\mathsf{off})$ discussed above, the subformula $\diamond\mathsf{off}$ only refers to the execution quantified by the inner quantifier E and CTL* offers no means to refer to the execution quantified by the outer quantifier A. In fact, it has been observed that noninterference is not even an ω-regular tree language [5], which is a class of properties that includes most temporal logics.

This raises the question how temporal logics can be extended to enable expressing information-flow properties and other hyperproperties of interest. First, the study of such extensions will lead to a better understanding of the interplay between the expressiveness of specification languages and the complexity of the model checking problem. Second, temporal logics capable of expressing information-flow properties may also lead to very practical impact as it may provide new options to ensure information security—in particular in the design of hardware, where temporal logic model checking is widely applied already.

Excursion: Epistemic temporal logic

Epistemic temporal logic extends the commonly considered temporal logics by the ability to specify the *knowledge* of *agents* [135, 45]. The *knowledge operator* $\mathcal{K}_A\varphi$ expresses that agent A knows the fact φ, which is again a temporal formula. It is assumed that every agent can only observe a certain part of the system. For example the property $\Diamond\mathcal{K}_A a$ expresses that eventually agent A *knows* that a holds. For knowing that a holds, however, agent A does not have to be able to observe it. The agent may just deduce from other observations that the fact must hold.

Similar to information-flow properties, knowledge is formalized using pairs of traces: Agent A knows fact φ at some time i in a given trace t, if φ holds for t at time i and for all traces t' that look the same as t to agent A the fact φ holds at time i as well.

It turns out that certain information-flow properties can be expressed in epistemic temporal logics [10]. These encodings are, however, fairly complex. For example, the encoding of a simplified variant of noninterference states that for all possible *values* v of the secret the attacker must not know whether the actual secret does not have the value v. In formal notation that is $\Box(\forall v : \neg\mathcal{K}(h \neq v))$. Besides making use of the quantification over a set of values, which would lead to a formula of exponential size in the number of variables, the encoding is very different from the original formulation of noninterference - and with its double negation arguable more complex. It is thus questionable whether epistemic temporal logic is suited to serve as a specification language for information-flow properties and other hyperproperties of interest. We believe further that the study of the interplay between the expressive power and the complexity of the algorithmic problems in the area of hyperproperties demands to look beyond epistemic temporal logic and study alternative mechanisms to specify relations over multiple executions.

1.4 Contributions

In this thesis, we consider the use of temporal logics for information-flow control and ask which classes of properties that relate multiple executions of a system can be automatically enforced. For that we consider an extension of temporal logics that enables the quantification over multiple traces and that extends the atomic propositions with the ability to refer to *any* of those traces—not only the “current” trace. We propose two temporal logics: HyperLTL and HyperCTL*. Both logics extend the LTL syntax by the trace quantifiers $\forall\pi.\varphi$ and $\exists\pi.\varphi$, and they require to index each atomic proposition by a variable π indicating the trace it refers to. HyperLTL differs from HyperCTL* in that it requires that the trace quantifiers occur only in the beginning of the formula, while HyperCTL* allows us to use quantifiers also inside boolean and temporal operators.

Noninterference can then be specified as follows in HyperLTL (assuming that the public inputs consist of a single input variable i and the public outputs consist of a single output variable o):

$$\forall\pi.\forall\pi'.\ \Box(i_\pi = i_{\pi'}) \implies \Box(o_\pi = o_{\pi'})$$

The two quantifiers in the beginning of the formula, i.e. $\forall\pi.\forall\pi'.$, state that the remaining formula must hold for all pairs of traces, which we assign the names π and π'. The remaining formula is an implication ($\implies$) that states that if $\square(i_\pi = i_{\pi'})$ holds, then also the formula $\square(o_\pi = o_{\pi'})$ must hold. The first subformula $\square(i_\pi = i_{\pi'})$ states that for all times the public input variable i has the same values on the traces π and π'. Analogously $\square(o_\pi = o_{\pi'})$ denotes that the public output variable o has the same value on the traces π and π' for all times. That is, the specification above states that all pairs of execution traces that have the same public input, the public output must be equal. We prove in Theorem 3.3.1 that the formulation above exactly reflects the original definition of noninterference as proposed by Goguen and Meseguer [55].

Semantics

The most immediate question that arises from an extension of temporal logics is how it affects the expressiveness. We start with a practical perspective and determine that we can express different information-flow properties and their declassification policies in Chapter 3. The encodings are simple and often close to the original formulation in the literature. That is, we argue that with HyperLTL and HyperCTL* we can express information-flow properties in a *natural* way—the proposed temporal logics apparently provide a suitable level of abstraction for information-security experts. We also provide natural encodings of hyperproperties from other fields, including error resistant codes (the distance of code words) and in distributed computing (symmetries in protocols), suggesting that the proposed extensions have applications beyond information security.

A fundamental question that arises from extensions of temporal logics is how it affects their classification as linear-time temporal logic or branching-time temporal logic [54, 103]. Considering HyperLTL and HyperCTL* as extensions of LTL and CTL*, we come to the conclusion that the extension of temporal logics to hyperproperties is orthogonal to the classification in linear-time and branching-time logics. HyperLTL remains in the linear-time setting and HyperCTL* remains in the branching-time setting (Chapter 3 and Chapter 4). We argue that the logic HyperCTL* combines the unique abilities to specify how information enters a system (branching-time), and how information flows through and leaves a system (information-flow).

Lastly, a comparison to existing temporal logics that relate multiple executions is in order. While in general the expressiveness of HyperLTL and epistemic temporal logics is incomparable, we show in Chapter 7 that with minor assumptions HyperLTL can be shown to be more expressive than the epistemic temporal logic. We also consider the relation to fixed point logics, such as the polyadic modal μ-calculus [7].

Algorithmics

Extending temporal logics compels us to revisit the question whether any property expressible in the new temporal logics can be model checked. We give a positive answer to this question by giving a model checking algorithm for HyperLTL and HyperCTL* in Chapter 5. As it is common in the analysis of temporal logics we focus on systems with finitely many states. The algorithmic insights for finite-state systems have repeatedly led to advances in the analysis of more general system models.

In the worst-case the algorithm has *nonelementary* run-time and space requirements. That is, the run-time and space requirements of the algorithm grow faster than any exponential function, even any nesting of exponential functions. This suggests that the algorithm is highly impractical and our complexity analysis even shows that there cannot be a substantially better model checking algorithm in general. Our complexity analysis, however, also reveals a structure behind the complexity issues and draws a more positive picture of the model checking problems of HyperLTL and HyperCTL*.

We identify *quantifier alternations* in the formula as the main source of computational complexity. A use of a universal quantifier $\forall\pi.$ inside an existential quantifier $\exists\pi.$ or vice versa is considered a quantifier alternation. For example, the formula $\forall\pi.\exists\pi'.\forall\pi''.\varphi$ has (at least) 2 quantifier alternations. When we bound the number of quantifier alternations in the formula we can give better bounds on the run-time and space requirements of the algorithm. We show that the class of model checking problems for formulas with k quantifier alternations is equivalent to the class of problems solvable by a nondeterministic Turing machine with $g(k,n)$ memory cells, where n is the size of the formula and where the function $g(k,n)$ represents a *tower of exponentials* of height k. That is $g(k,n)$ is defined to be n for $k = 0$ and for a height $k > 0$ the tower of exponentials is recursively constructed as follows:

$$g(k,n) = 2^{g(k-1,n)}$$

When we measure the complexity in the size m of the system instead of the size of the formula, the complexity is $g(k-1,m)$. That is, quantifier alternations in the formula still play a decisive role, but the tower of exponentials has one level less.

In particular, these results imply that for formulas without quantifier alternations, the so called *alternation-free fragment*, the model checking problem is PSPACE-complete in the size of the formula and NLOGSPACE-complete in the size of the system. This coincides with the complexities of the model checking problems of LTL and CTL*.

Practical Challenges

It turns out that many of the information-flow properties can be described without using quantifier alternations and thus are in the alternation-free fragment. *Symbolic* model checking algorithms for this fragment would thus enable the automatic enforcement of a large class of information-flow policies. In Chapter 6 we ask how to devise symbolic model checking algorithms for the alternation-free fragments of HyperLTL and HyperCTL*. Considering the more widespread use of temporal logic model checking in the hardware industry, as opposed to the software industry, we focus on hardware model checking in this chapter.

We propose an approach that allows us to reuse existing symbolic hardware model checkers. Given a hardware module to check and an alternation-free HyperLTL or HyperCTL* formula, our algorithm compiles a new hardware module. The new hardware module is then checked for a simple property using any of the existing hardware model checkers. From their answer we can derive whether the original hardware module satisfied the HyperLTL or HyperCTL* formula. This approach has several advantages over devising specialized symbolic algorithms for this problem: (1) The concept can be adopted easily in other tool chains, because it requires comparably little engineering effort, (2) we can easily evaluate how the various symbolic approaches to hardware model checking compare for the new class of properties, and (3) our approach likely profits from future improvements to symbolic model checking algorithms.

Along three case studies, we demonstrate that the proposed approach enables us to check real hardware modules for interesting hyperproperties. We consider a variety of information-flow properties of an I2C bus master implementation, symmetries in the bakery protocol for mutual exclusion, and the correctness of error resistant encoders and decoders.

Summary

To summarize, we propose the use of temporal logics as specification languages for information-flow control to provide a formal basis and to overcome the inflexibility of current enforcement approaches. We propose a simple extension to temporal logics that can express various information-flow properties and other hyperproperties of interest, but that still allows us to give effective model checking algorithms. We study the expressiveness of the proposed temporal logics HyperLTL and HyperCTL*, propose a model checking algorithm, and give a detailed analysis of the complexity of the model checking problem. The proposed approach is directly applicable in hardware security: We demonstrate how to leverage existing model checking technology for checking information-flow properties for real hardware modules.

1.5 Publications and Collaborations

This thesis is based on the following peer reviewed publications:

[30] Michael Clarkson, Bernd Finkbeiner, Masoud Koleini, Kristopher K. Micinski, Markus N. Rabe, and César Sánchez. “Temporal Logics for Hy-

perproperties.” In Proceedings of the 3rd Conference on Principles of Security and Trust (POST), pages 265-284, 2014.

[38] Rayna Dimitrova, Bernd Finkbeiner, Máté Kovács, Markus N. Rabe, and Helmut Seidl. “Model Checking Information Flow in Reactive Systems.” In Proceedings of the 13th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI), pages 169-185, 2012.

[47] Bernd Finkbeiner and Markus N. Rabe. “The Linear-Hyper-Branching Spectrum of Temporal Logics.” *it-Information Technology*, 56, no. 6, pages 273-279, 2014.

[48] Bernd Finkbeiner, Markus N. Rabe, and César Sánchez. “Algorithms for Model Checking HyperLTL and HyperCTL*.” In Proceedings of the 27th International Conference on Computer Aided Verification (CAV), pages 30-48, 2015.

Further, the thesis contains material published in the following report:

[115] Markus N. Rabe, Peter Lammich, and Andrei Popescu. “A shallow embedding of HyperCTL*.” Archive of Formal Proofs, April 2014. `http://afp.sf.net/entries/HyperCTL.shtml`, Formal proof development.

Chapter 2

Systems and Properties

In this chapter, we discuss the system model used throughout the thesis and we introduce different notions of properties. In the introduction we relied on the view that a system is defined by the set of executions (traces) it can produce, but for the discussion of richer classes of properties and also for the analysis of systems it will be of importance how the set of traces is generated. While there are many ways to represent systems (programming languages, circuits, . . .), we emphasize simplicity in this work and hence choose *Kripke structures* as a unified system model. A Kripke structure consists of a collection of states, transitions between states, and labels that describe the properties of states.

Each Kripke structure generates a set of execution traces, but there may be different Kripke structures generating the same set of traces. We can thus distinguish systems even when they share the same set of traces. Branching-time properties, for example, may also specify the *branching structure*, that is the points of time when two traces split and take different states in the system (though may still share the same sequence of labels). These are the points of time when inputs or decisions in the system take place.

In this chapter we provide the formal definition of Kripke structures, before we discuss, along an example, how Kripke structures relate to real software and hardware. Then we turn to a discussion on the different terms of properties. We define trace properties, hyperproperties, and branching-time properties and we relate them to the spectrum of process equivalences as defined by van Glabbeek [54]. We discover a tight connection between the process equivalence defining linear-time semantics - trace equivalence - and hyperproperties.

This discussion gives also rise to the structure of this thesis: We start with temporal logics for the linear-time end of the spectrum (Chapter 3), before we discuss branching-time temporal logics (Chapter 4). Only after the semantical discussion, we turn to the algorithmic verification in Chapter 5 and Chapter 6. We wrap up with a detailed comparison to related temporal logics and fixed point calculi in Chapter 7.

2.1 Kripke Structures

A *Kripke structure* is a tuple $K = (S, s_0, \delta, \mathsf{AP}, L)$ consisting of a set of states S, an initial state s_0, a transition function $\delta : S \rightarrow 2^S$, a set of *atomic propositions* AP, and a *labeling function* $L : S \rightarrow 2^{\mathsf{AP}}$. We require that each state has a successor, that is $\delta(s) \neq \emptyset$, to ensure that every execution of a Kripke structure can always be continued to infinity. The *size* of a Kripke structure is defined as the cardinality of its set of states $|S|$. (The restriction to a single initial state here will prove convenient for the definitions of the logics. We can easily encode multiple initial states in the first transition of the system.)

A *path* of a Kripke structure is an infinite sequence $s_0 s_1 \ldots \in S^\omega$ such that s_0 is the initial state of K and $s_{i+1} \in \delta(s_i)$ for all $i \in \mathbb{N}$. By $\mathsf{Paths}(K,s)$ and $\mathsf{Paths}^*(K,s)$ we denote the set of all paths of K starting in state $s \in S$ and the set of their suffixes, respectively. An *execution trace*, or simply *trace*, of a path $\sigma = s_0 s_1 \ldots$ is the sequence of labels $l_0 l_1 \ldots$ with $l_i = L(s_i)$ for all $i \in \mathbb{N}$. $\mathsf{Traces}(K,s)$ (and $\mathsf{Traces}^*(K,s)$) is the set of all (suffixes of) traces of paths of a Kripke structure K starting in state s. The sets of traces and paths starting in the initial state of a system are denoted with $\mathsf{Traces}(K)$ and $\mathsf{Paths}(K)$, respectively.

For traces $t = a_0 a_1 \ldots$ and paths $p = p_0 p_1 \ldots$ we denote the n-th element with $t(n)$ and $p(n)$, respectively. The n-th suffix of a trace or path , written $t[n,\infty]$ and $p[n,\infty]$, is defined as the subsequence of the trace or path starting from the n-th element.

A Simple Example

We consider the Kripke structure K^*:

- $S = \{s_0, s_0', s_1, s_2, s_3\}$
- s_0 is the initial state.
- $\delta(s_0) = \{s_0', s_1\}$
- $\delta(s_0') = \{s_0', s_1\}$
- $\delta(s_1) = \{s_0', s_2\}$
- $\delta(s_2) = \{s_0', s_3\}$
- $\delta(s_3) = \{s_0, s_0'\}$
- $\mathsf{AP} = \{a, r\}$
- $L(s_0) = \{a\}$
- $L(s_0') = \{a, r\}$
- $L(s_i) = \emptyset$ for all $i \in \{1, 2, 3\}$

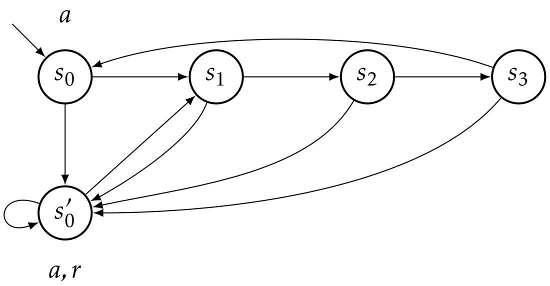


Figure 2.1: A graphical representation of the Kripke structure K^*.

As a mathematical definition of Kripke structures is sometimes hard to read, we often depict Kripke structures as labelled graphs as shown in Fig. 2.1. The Kripke structure K^* represents a resettable counter modulo 4. Consider r to be the input and a to be the output of the system. Without giving the reset input, the system is supposed to produce the output a every fourth step. By giving the reset input, the system gives the output a immediately and resets its counter to 0. As Kripke structures have no explicit notion of inputs and outputs, both kinds of signals are represented as atomic propositions. Each state is labelled by the inputs that were given in the last step and the outputs that the system produced. In particular this means that each state has two successors: one where the r is set and another one where r is not set.

The Kripke structure K^* generates an infinite set of execution traces. For the input sequence that never sets the signal r, the output of the system is a in the first step and then exactly every four steps. The path taken in the Kripke structure would be an infinite repetition of the states s_0, s_1, s_2, and s_3, in this order. For each infinite sequence of input signals (where in each step r is either set or not set) this Kripke has a unique execution trace.

In case of incomplete or underspecified systems, however, this may not be the case and the Kripke structure may offer different possible executions for the same input sequence. We call this *nondeterministic* behavior, or short nondeterminism. Nondeterminism is a useful tool for modeling systems; when we do not know how exactly a system behaves, or if we intentionally want to abstract from details to keep the model simple, we can include a nondeterministic choice of behavior in the Kripke structure.

From Software and Hardware to Kripke Structures

Pieces of software and hardware can be seen as succinct representations of Kripke structures. By a hardware module, we typically understand a *circuit* (or a program-like description of a circuit in a language like Verilog or VHDL) that consists of a certain number of memory cells, that is *latches*, and a combinational part that determines, for a given input and state of the latches, the outputs and the next state of the latches. Given a hardware module, that is a circuit, we consider the set of all possible values of the memory cells in the circuit as the set of states of the Kripke structure. The transitions between the states are then determined by the combinational part.

Given a program we typically consider the set of all possible values of the variables as the set of states of the Kripke structure. For that purpose we also consider the program counter as a variable. The transitions of a program are then determined by considering for each state the effect of the current program statement.

While for hardware circuits the translation into Kripke structures is fairly unique, for software there are more degrees of freedom in the translation. In particular there are different options available for modeling how the variables are initialized, how inputs are read, and how programs terminate. By defining temporal logics on Kripke structures we abstract from these modeling choices, which allows us to focus on the fundamental questions of how the expressiveness of specification languages interacts with the model checking problem.

2.2 Properties

In this thesis we consider *properties* of systems. Each Kripke structure K either satisfies a given property P, denoted $K \models P$, or violates it. A property can be represented by the set of systems it satisfies—the *characteristic set* of the property.

Analogously, properties of traces are either satisfied or violated by any given trace and we can represent them by their characteristic set of traces. We will denote properties of traces with *trace properties* to clearly distinguish them from properties of systems. Trace properties can be interpreted as system properties, by requiring that all traces of a given system satisfy the trace property. As an exercise, we formalize the use of trace properties as system properties as follows: Let $T(\mathsf{AP}) = (2^{\mathsf{AP}})^\omega$ be the set of all traces over a given set of atomic propositions AP. Then a (the characteristic set of a) trace property $P \in 2^{T(\mathsf{AP})}$ is said to be satisfied by a Kripke structure K, if the traces of K are a subset of P.

Earlier studies identified a beautiful structure in trace properties. As is well known from topology, every set can be represented as the intersection of an open and a closed set. For the characteristic set of trace properties this decomposition corresponds to the notions of *liveness trace properties* (e.g. $\Diamond a$) and *safety trace properties* (e.g. $\Box a$) [4]. This is an insight of great practical relevance, as safety trace properties are trace properties that can be refuted by a finite (prefix of a) trace and liveness trace properties are the trace properties that need an infinite trace to be refuted.

System Equivalences

A different approach to the classification of properties is to ask what features of systems we want to be able to specify and to what features of systems we want to be indifferent. The *induced process equivalence* of a given class of properties is the equivalence that distinguishes two systems exactly if there exists a property in the class that is satisfied by one system but not by the other. Van Glabbeek studied classes of properties defined by process semantics and the equivalences they induce [54]. In the same way classes of properties generated by temporal logics can be analyzed for their induced process equivalences [8, 103].

The coarsest of the typically considered process equivalence of the spectrum is trace equivalence. Two systems are considered *trace equivalent*, if the sets of traces they generate are equal. Trace equivalence gives rise to the notion linear-time properties, which abstract from the state space and the inner structure of a system, and only consider the execution traces a system generates. In this thesis we consider a property as a *linear-time property* (not to be confused with the commonly used term *linear property*, which we call trace property in this work), if all pairs of trace equivalent Kripke structures K and K' either both satisfy the property or both violate the property. That is, it is the most general term of properties that still induces trace equivalence. It is clear that trace properties are a subset of the linear-time properties.

Bisimulation (or bisimilarity) is the finest of the typically considered process equivalences and corresponds to the branching-time view. Two processes are *bisimilar* if there exists a relation between their states that relates their initial states and that ensures that for every pair of related states, every transition from one state can be simulated by a transition from the other state such that the successors are related again. Analogous to linear-time properties we define *branching-time properties* to be the most general class of properties that still induces bisimulation. That is, a property is a branching-time property, if all pairs of bisimilar Kripke structures K and K' either both satisfy the property or both violate the property.

Hyperproperties

Information-flow properties relate multiple traces and thus fall outside of the class of trace properties, which consider traces in isolation. Consider the secrecy (confidentiality) property *noninterference* [55].[1]

Given a Kripke structure K and a partition of propositions into low and high input propositions I_l and I_h, and low and high output propositions O_l and O_h. While the *low* propositions are considered to be observable by the attacker, the *high* propositions model the secret. The set of high input propositions I_h is defined to be *noninterferent* with the low output propositions O_l, if for all traces $t = t_0t_1t_2\ldots$ and $u = u_0u_1u_2\ldots$ with $t_i \cap I_l = u_i \cap I_l$ for all $i \in \mathbb{N}$,

[1] We consider a modern presentation of noninterference that is also referred to as *observational determinism* [117, 148].

it holds that $t_i \cap O_l = u_i \cap O_l$ for all $i \in \mathbb{N}$. That is, O_l may not depend on the high input I_h and thus the attacker cannot obtain any information about the secret.

To formally prove that noninterference is not a trace property, we give a proof by contradiction. Suppose that there is a trace property φ that expresses noninterference. The set of traces T that satisfy the formula cannot be the full set of traces (with respect to some non-empty set of atomic propositions), because in that case all Kripke structures would satisfy the property. We pick a trace t that is *not* in T and consider a Kripke structure that only allows for trace t. Since this Kripke structure only has a single trace, it obviously satisfies noninterference; but since that trace is not in T, it violates φ, contradicting our assumption that φ expresses noninterference.

Clarkson and Schneider proposed the notion of hyperproperties to provide a new class of properties that includes many information-flow properties. They define a *hyperproperty* $H \in 2^{2^{T(\mathsf{AP})}}$ to be a set of sets of traces. A system K satisfies a hyperproperty H, if its set of traces is an element of H, that is $\mathsf{Traces}(K) \in H$. Similar to trace properties, hyperproperties can be classified into safety hyperproperties and liveness hyperproperties [31]. For example, for a given classification of inputs and outputs into public and secret noninterference is the hyperproperty that consists of all sets of traces that satisfy the noninterference condition:

$$\left\{T \subseteq T(\mathsf{AP}) \mid \forall t, t' \in T :\ t =_{I_p} t' \implies t =_{O_p} t'\right\}$$

The definition of the class of hyperproperties compels us to compare it to the known classes of properties. Hyperproperties clearly are a subclass of the linear-time properties: a pair of trace equivalent systems clearly satisfies the same set of hyperproperties. Given two systems K and K' with $\mathsf{Traces}(K) = \mathsf{Traces}(K')$ then clearly $\mathsf{Traces}(K) \in H$ iff $\mathsf{Traces}(K') \in H$ for any hyperproperty H. A close look reveals that the converse is true as well. Given two systems K and K' with $\mathsf{Traces}(K) \neq \mathsf{Traces}(K')$ then there is a hyperproperty H that is satisfied by K but violated by K': Choose $H = \{\mathsf{Traces}(K)\}$.

Theorem 2.2.1. *A property is a hyperproperty iff it is a linear-time property.*

This comes at no surprise. Already Clarkson and Schneider observed that hyperproperties are the most general class of properties that can be defined over traces [31]. The formulation via trace equivalence, however, lets us relate hyperproperties to the branching-time view: there are branching-time properties that are not hyperproperties. However, if we assume the modeling principles proposed by Nain and Vardi, which let trace equivalence and bisimulation collapse [103], hyperproperties are models of all branching-time properties. This can be seen as a strengthening of Clarkson and Schneider's observation that "hyperproperties can be models for branching-time temporal predicates" [31].

The Spectrum of Temporal Logics

The classifications of LTL as linear-time temporal logic, and CTL and CTL* as branching-time temporal logics is based on the process equivalences they induce: LTL induces trace equivalence and both CTL and CTL* induce bisimulation [8]. The first two chapters of this thesis follow the classification of temporal logics into linear-time and branching-time. In Chapter 3 we start with extending LTL to enable expressing information-flow properties while staying within the linear-time properties. In Chapter 4 we then discuss the effect of extending branching-time temporal logics. In particular, we identify a class of information-flow properties, which are not linear-time properties, i.e. are not hyperproperties.

Chapter 3

Linear-time Temporal Logics

In this chapter, we study linear-time temporal logics as specification languages. We review the well-known linear-time temporal logic (LTL) before we discuss an extension of LTL, HyperLTL. HyperLTL enables us to express hyperproperties that are not trace properties but remains a linear-time temporal logic. Along a number of encodings of common information-flow properties and hyperproperties from other fields we demonstrate the versatility of the logic. We also prove that the original definition of Goguen and Meseguer's noninterference can be encoded in HyperLTL.

3.1 Linear-time Temporal Logics

Linear-time temporal logic is the most commonly studied specification language in the linear-time setting [110]. LTL specifies properties of single traces such as invariants, ("for all times variable x is greater than 5") or eventualities ("eventually variable x is greater than 5"). For example the property "event a is always followed by event b" would be $\Box(a \Rightarrow \Diamond b)$ in LTL syntax. LTL is generated by the following grammar:

$$\varphi \ ::= \ a \mid \neg\varphi \mid \varphi \wedge \varphi \mid \bigcirc\varphi \mid \varphi\,\mathcal{U}\,\varphi$$

where a is an *atomic proposition*, $\neg$ is the *negation* operator, $\wedge$ is *conjunction*, $\bigcirc$ denotes *next*, and $\mathcal{U}$ is the *until* operator. We also consider the usual derived Boolean operators and the derived temporal operators *eventually* $\Diamond\varphi \equiv \mathrm{true}\,\mathcal{U}$ φ and *globally* $\Box\varphi \equiv \neg\Diamond\neg\varphi$.

LTL is interpreted over traces t:

$$\begin{array}{lll}
t \models a & \text{iff} & a \in t(0) \\
t \models \neg\varphi & \text{iff} & t \not\models \varphi \\
t \models \varphi_1 \wedge \varphi_2 & \text{iff} & t \models \varphi_1 \wedge t \models \varphi_2 \\
t \models \bigcirc\varphi & \text{iff} & t[1,\infty] \models \varphi \\
t \models \varphi_1\,\mathcal{U}\,\varphi_2 & \text{iff} & \text{for some } i \geq 0 : \ t[i,\infty] \models \varphi_2 \text{ and} \\
 & & \quad \text{for all } 0 \leq j < i : \ t[j,\infty] \models \varphi_1
\end{array}$$

LTL formulas define, by definition, trace properties. A system K is then defined to satisfy an LTL formula φ, denoted denoted $K \models \varphi$, if all execution traces of the system satisfy the property.

3.2 HyperLTL

Even though most information-flow properties are linear-time properties (hyperproperties) LTL cannot express them, as LTL can only express trace properties but, as we discussed in Chapter 2, information-flow properties involves the comparison of two or more execution traces. We thus propose to extend LTL with *trace quantifiers*, which allow us to *relate* multiple execution traces. In the resulting logic, HyperLTL [30], we can then express noninterference as follows:

$$\forall\pi.\forall\pi'.\,\square(I_{l,\pi} = I_{l,\pi'}) \Rightarrow \square(O_{l,\pi} = O_{l,\pi'})\,,$$

where I_l and O_l are the sets of propositions for the public inputs and outputs as introduced in the definition of noninterference, and $I_{l,\pi} = I_{l,\pi'}$ and $O_{l,\pi} = O_{l,\pi'}$ express that at the current point of time the paths π and π' are equal on the atomic propositions I_l and O_l respectively. The formula states that when for all pairs of traces π and π' and for all times, indicated by the "box"-operator $\square$, the low inputs on π and π' are equal, then also the low outputs shall be equal for all times.

The formulas of HyperLTL are generated by the following grammar, where $a \in \mathsf{AP}$ and π ranges over trace variables:

$$\begin{array}{lllllllllll} \varphi & ::= & \mathsf{true} & \mid & a_\pi & \mid & \neg\varphi & \mid & \varphi\vee\varphi & \mid & \varphi\wedge\varphi \\ & \mid & \bigcirc\varphi & \mid & \varphi\,\mathcal{U}\,\varphi & \mid & \varphi\,\mathcal{R}\,\varphi & \mid & \exists\pi.\,\varphi & \mid & \forall\pi.\,\varphi \end{array}$$

We require that HyperLTL formulas are closed and in prenex form. A formula is in *prenex form* if it starts with a quantifier prefix and continues with a quantifier-free subformula. Quantifiers $\forall\pi.\psi$ *bind* all occurrences of path variables in their subformula ψ, and path variables that are not bound by quantifiers are called *free*. A formula is *closed*, if it does not contain any free path variables.

As we will frequently specify the comparison among traces, we introduce the equality on sets of atomic propositions as syntactic sugar and define $A_\pi = A_{\pi'}$ as $\bigwedge_{a\in A} a_\pi \leftrightarrow a_{\pi'}$. We also introduce the derived temporal operators $\Diamond\varphi := \mathsf{true}\,\mathcal{U}\,\varphi$, $\square\varphi := \neg\Diamond\neg\varphi$, and $\varphi_1\,\mathcal{W}\,\varphi_2 := \varphi_1\,\mathcal{U}\,\varphi_2 \vee \square\varphi_1$, as well as the boolean connectives $\varphi_1 \Rightarrow \varphi_2 := \neg\varphi_1 \vee \varphi_2$, $\varphi_1 \Leftrightarrow \varphi_2 := \varphi_1 \Rightarrow \varphi_2 \,\wedge\, \varphi_2 \Rightarrow \varphi_1$, and $\varphi_1\ \mathsf{xor}\ \varphi_2 := \neg(\varphi_1 \Leftrightarrow \varphi_2)$.

Semantics. In the following we define the semantics for the operators a_π, $\neg\varphi$, $\varphi_1 \vee \varphi_2$, $\bigcirc\varphi$, $\varphi_1\,\mathcal{U}\,\varphi_2$, and $\exists\pi.\,\varphi$. The other operators are defined via the following equalities: $\forall\pi.\,\varphi = \neg\exists\pi.\,\neg\varphi$, $\varphi_1 \wedge \varphi_2 = \neg(\neg\varphi_1 \vee \neg\varphi_2)$, and $\varphi_1\,\mathcal{R}\,\varphi_2 = \neg(\neg\varphi_1\,\mathcal{U}\,\neg\varphi_1)$. These derived operators are kept in the syntax to guarantee the existence of equivalent formulas in negation normal form.

Let K be a Kripke structure and let s_0 be its initial state. The semantics of HyperLTL is given via trace assignments $\Pi : N \to \mathsf{Traces}^*(K, s_0)$ of a set of trace variables N to suffixes of *traces*. We use $\Pi[i, \infty]$ for the trace assignment that assigns to each trace variable π the suffix $\Pi(\pi)[i, \infty]$. We define the validity of a formula as follows:

$$
\begin{array}{lll}
\Pi \models_K a_\pi & \text{iff} & a \in \Pi(\pi)(0) \\
\Pi \models_K \neg\varphi & \text{iff} & \Pi \not\models_K \varphi \\
\Pi \models_K \varphi_1 \vee \varphi_2 & \text{iff} & \Pi \models_K \varphi_1 \text{ or } \Pi \models_K \varphi_2 \\
\Pi \models_K \bigcirc\varphi & \text{iff} & \Pi[1, \infty] \models_K \varphi \\
\Pi \models_K \varphi_1 \,\mathcal{U}\, \varphi_2 & \text{iff} & \text{for some } i \geq 0 : \ \Pi[i, \infty] \models_K \varphi_2 \text{ and} \\
 & & \quad \text{for all } 0 \leq j < i : \ \Pi[j, \infty] \models_K \varphi_1 \\
\Pi \models_K \exists\pi.\, \varphi & \text{iff} & \text{for some } t \in \mathsf{Traces}(K, s_0) : \ \Pi[\pi \mapsto t] \models_K \varphi
\end{array}
$$

Validity on a Kripke structure K, written $K \models \varphi$, is defined as $\emptyset \models_K \varphi$, where $\emptyset$ is the empty assignment.

The extension of LTL by the ability to quantify over multiple traces compels us to ask whether HyperLTL still is a linear-time logic. Given that LTL is a sublogic of HyperLTL, it is clear that its induced equivalence is at least as fine as trace equivalence. HyperLTL also cannot distinguish more than trace equivalence, because its semantics refers to the system only in terms of the set of traces starting from the initial state.

Theorem 3.2.1. *HyperLTL induces trace equivalence.*

HyperLTL is thus a logic for linear-time properties, or hyperproperties. This observation is helpful to understand the limits of the expressiveness of HyperLTL. For example, we can immediately conclude that it is impossible to specify when a nondeterministic choice happens - the standard example to separate linear-time logics from branching-time logics. We refer to Chapter 4 for a detailed discussion.

3.3 Applications in Information-flow Control

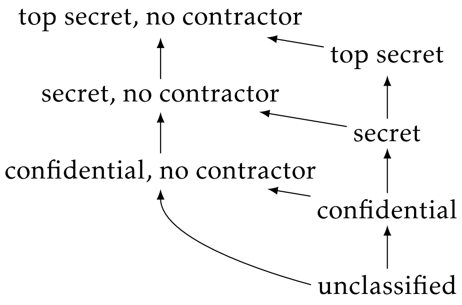
We have already seen that we can express noninterference [55] in HyperLTL in a natural way: $\forall\pi.\forall\pi'.\ \square(I_{l,\pi} = I_{l,\pi'}) \Rightarrow \square(O_{l,\pi} = O_{l,\pi'})$. In the following, we discuss various encodings of information-flow properties, properties of distributed systems and of coding theory.

The property-oriented approach to information-flow control considers system properties that restrict the *information* that the system reveals to an attacker. The information visible to the attacker can be specified as a *relation over the executions* of the system. The inputs and outputs of the system are typically partitioned into *high* and *low*, which gives rise to an *attacker model*: The attacker is modeled as a an (implicit) entity that can observe the low outputs and that has full control over the low inputs. We assume that the attacker can repeatedly execute the system with various low inputs of his or her choice.

Since there is no notion of inputs and outputs in Kripke structures, we assume a partitioning of the atomic propositions into inputs I and outputs O for the purpose of the following example encodings. To simplify the discussion further, we assume input-enabled systems in this section: A Kripke structure is *input-enabled* in some set of sets of propositions $I \subseteq 2^{\mathsf{AP}}$, if for all states s of the Kripke structure and all $a \in I$ there is a successor state s' with label $L(s') \cap I = a$. Input-enabled Kripke structures model systems whose behavior is modeled for all inputs and where the last input is visible in each state. Common reactive system, such as circuits or reactive programs, can be interpreted as input-enabled Kripke structures.

Security Lattices

A classical security model is that of multiple levels of secrecy [15, 35]. For example, consider a simplified model of the "United States government classification system" that includes the following classifications: "unclassified", "confidential", "secret", "top secret", and "no contractor". When the security levels do not form a linear hierarchy (e.g. the classification "confidential, no contractor" is incomparable to the classification "secret"), but are only partially ordered, the possible classifications of data form a security lattice. A *security lattice* $(L, \leq)$ consists of a set of security levels L and a partial order $\leq$ on L. The security lattice for the example classification system can be depicted as follows (transitive edges are not drawn):



Starting from a classification of inputs and outputs of the system, we can easily express a security lattice as the conjunction over multiple instances of noninterference (or a different basic notion of secrecy). Each conjunct restricts a security level to only depend on the information in lower security levels:

$$\bigwedge_{\lambda\in L} \forall\pi.\forall\pi'.\ \square(I_{\leq\lambda,\pi} = I_{\leq\lambda,\pi'}) \Rightarrow \square(O_{\lambda,\pi} = O_{\lambda,\pi'})\ ,$$

where $I_{\leq\lambda}$ are the inputs classified on the security level λ or a lower classified security level, and O_λ are the outputs classified on security level λ.

Declassification

Noninterference forbids the leakage of *any* information about the secret. Many security critical systems, however, are intended reveal secret information partially or under certain conditions. For example, a piece of software that checks passwords must reveal whether the entered password is correct or not. The term *declassification* summarizes the various ways by which the degree of intended information release can be specified; see [120] for a survey.

For example, we can release predefined facts about the secret (the "what-dimension" [120]):

$$\forall\pi.\forall\pi'.\,\square(I_{l,\pi} = I_{l,\pi'} \;\wedge\; \varphi) \Rightarrow \square(O_{l,\pi} = O_{l,\pi'})\;,$$

where φ is a formula (typically not containing temporal operators) describing the fact that may be released. In our password checker example discussed in the introduction, $\varphi := I_{l,\pi} = I_{h,\pi} \leftrightarrow I_{l,\pi'} = I_{h,\pi'}$ describes that the information whether the public input is equal to the secret input (i.e. the password) may be released.

We can also release information when a certain condition ψ is met (the "when-dimension" [120]):

$$\forall\pi.\forall\pi'.\,(O_{l,\pi} = O_{l,\pi'})\,\mathcal{W}\,(I_{l,\pi} \neq I_{l,\pi'} \;\vee\; \psi)\;.$$

For example, a system that protects a secret until the correct password is entered could be specified by defining $\psi := I_{l,\pi} = I_{h,\pi}$. (Consider that for $\varphi :=$ false the property is equivalent to noninterference, due to our assumption of input enabledness.)

Quantitative Information Flow

In cases where leaks cannot be prevented absolutely, we may still be able to limit the *quantity* of the lost information [64, 27, 70, 126, 146]. Different notions of entropy have been proposed to measure the amount of leaked information. Two definitions of entropy that are argued to be appropriate for security considerations are min entropy [126] and minimal guessing entropy [70]. For our discussion on quantitative information flow properties, we assume that the secrets I_h is chosen uniform at random and that for a fixed low input I_l, we obtain a unique probability distribution over the traces of the system.

Min entropy measures the expected likelihood to guess the secret after observing the run of the system. The min entropy of I_h in the observable output O_l is the logarithm of the number of different observations in O_l [126]. In HyperLTL, we can express that the min entropy is bounded by n bits by requiring that for any set of 2^{n+1} traces of the system, at least one pair has equal observations:

$$\forall\pi_1.\;\ldots\;.\;\forall\pi_{2^{n+1}}.\;\bigvee_{i\neq j}\square(O_{l,\pi_i} = O_{l,\pi_j})\;.$$

To incorporate public input that is controllable by the attacker, we can refine the property by as follows:

$$\forall\pi_1.\ \dots\ .\ \forall\pi_{2^{n+1}}.\ \Big(\bigwedge_{i\neq j}\square(I_{l,\pi_i}=I_{l,\pi_j})\Big)\Rightarrow\bigvee_{i\neq j}\square(O_{l,\pi_i}=O_{l,\pi_j})\ .$$

Minimal guessing entropy is an alternative definition of entropy that measures the worst-case likelihood to guess the secret after observing the run of the system [70]. Bounding the minimal guessing entropy by n bits boils down to the requirement that for every observation that the system may produce, there must be at least 2^n secrets that produce the observation. In HyperLTL, we formalize this as follows:

$$\forall\pi.\ \exists\pi_1.\ \dots\ .\ \exists\pi_{2^n}.\ \Big(\bigwedge_{i}\square(O_{l,\pi}=O_{l,\pi_i})\Big)\wedge\bigwedge_{i\neq j}\diamondsuit(I_{h,\pi_i}\neq I_{h,\pi_j})\ .$$

Noninterference and Event-based System Models

Noninterference, introduced by Goguen and Meseguer in 1982 [55], is seen as the starting point for information-flow security. They state secrecy in terms of groups of "users" that may issue commands to the system and that may make observations. The commands of one group of users are considered to be the secret and a second group of users is considered to be a group of attackers. The original definition of noninterference then requires that the attackers observations remain unchanged when removing the secret user's commands.

Steps in the system model are not necessarily observable. Sequences of observations are identified only after contracting equal observations: observations $o_1o_1o_2$ and $o_1o_2o_2$ are considered to be indistinguishable. This represents the assumption that the attackers cannot tell, whether or not the secret users performed a command in case its execution does not affect the attackers' observations.

At a first glance, the observational equivalence required for our formalization of noninterference seems different in nature, as it involves the lock-step comparison of traces. Two (finite prefixes of) traces with the observations $o_1o_1o_2$ and $o_1o_2o_2$ are hence considered distinguishable by the attackers in our setting.

A closer look reveals, however, that this intuition is misleading. We show in this subsection that noninterference, as formalized in HyperLTL above, is equivalent to the original definition of noninterference for a natural translation of their system model in Kripke structures that allows the system to stutter [115]. The key idea is that quantifying over two traces of the stuttering system generates all possible alignments of pairs of traces of the non-stuttering version of the system. It is then sufficient to synchronize the traces by the common choices of low inputs. This suggests that HyperLTL may also serve as a specification language for information-flow properties for event-based system models.

Encoding GM's System Model. The system model used in [55], which we will refer to as *deterministic state machines*, is a tuple $(S, U, C, Out, \mathsf{out}, \mathsf{do}, s_0)$, where S is some set of states with an initial state s_0, C is a set of commands that are issued by users $u \in U$, and Out is a set of observations. The evolution of a deterministic state machine is governed by the transition function $\mathsf{do} : S \times U \times C \to S$ and there is a separate observation function $\mathsf{out} : S \times U \to Out$ that for each user indicates what he can observe.

We define standard notions on sequences of users and events. For $w \in (U \times C)^*$ and $G \subseteq U$ let $|w|_G$ denote the projection of w to the tuples (u, c) with $u \in G$, i.e. those commands that are issued by the users in G. (Goguen and Meseguer called this the *purge* operation [55].) Further, we extend the transition function do to sequences, $\mathsf{do}(s, (u, c).w) = \mathsf{do}(\mathsf{do}(s, u, c), w)$, where the dot indicates concatenation. Finally, we extend the observation function out to sequences w, indicating the observation *after* w: $\mathsf{out}(w, G) = \mathsf{out}(\mathsf{do}(s_0, w), G)$. *GM-noninterference* is then defined as a property on deterministic state machines $M = (S, U, C, \mathsf{do}, \mathsf{out})$. A set of users $G_H \subseteq U$ does not interfere with a second group of users $G_L \subseteq U$, if:

$$\forall w \in (U \times C)^*.\ \mathsf{out}(w, G_L) = \mathsf{out}(|w|_{U \setminus G_H}, G_L)\ .$$

That is, GM-noninterference asks whether the same output would have been produced by the system, if all actions issued by users in G_H were removed.

To show that GM-noninterference can be expressed in HyperLTL, we need to translate deterministic state machines to Kripke structures. We give an intuitive translation that indicates in every state which observations can be made for the different users and which action was issued last by which user. Given a deterministic state machine $M = (S, U, C, Out, \mathsf{out}, \mathsf{do}, s_0)$, we construct the Kripke structure $K(M) = (S', s_0', \delta, \mathsf{AP}, L)$, where $S' = S \times U \times C\ \cup\ S$, $s_0' = s_0$, $\mathsf{AP} = U \times C\ \cup\ U \times Out$, and the labeling function is defined as $L((s, u, c)) = \{(u, c)\}\ \cup\ \{(u', \mathsf{out}(s, u')) \mid u' \in U\}$ for tuple states and $L(s) = \{(u', \mathsf{out}(s, u')) \mid u' \in U\}$ for all other states. The transition function is defined as follows:

$$\begin{array}{lcl} \delta(s) & = & \{s\} \cup \{(s', u', c') \mid \mathsf{do}(s, u', c') = s'\} \\ \delta((s, u, c)) & = & \{s\} \cup \{(s', u', c') \mid \mathsf{do}(s, u', c') = s'\} \end{array}$$

The states without a user-command pair indicate states in which no command was issued in the last step. Except for the initial state they are only reachable via the newly introduced stuttering transitions. Consider that $K(M)$ is *input-enabled* in the set $I = \{(u, c) \mid u \in U,\ c \in C\}$.

Theorem 3.3.1. *Let M be a deterministic state machine, let G_L and G_H be two disjoint groups of users, and let $I_l = \{(u, c) \mid u \in U \setminus G_H,\ c \in C\}$ and $O_l = \{(u, o) \mid u \in G_L,\ o \in Out\}$ be the inputs and observations as modeled in the Kripke structure. Then $K(M) \models \forall\pi.\forall\pi'.\ \square(I_{l,\pi} = I_{l,\pi'}) \Rightarrow \square(O_{l,\pi} = O_{l,\pi'})$ holds if, and only if, G_H does not interfere with G_L.*

Proof. Given a sequences $w \in (U \times C)^*$ of length n with $w = (u_1, c_1), \ldots, (u_n, c_n)$, the path prefix $p(w) = s_0', ((w|_1), u_1, c_1), \ldots, ((w|_n), u_n, c_n)$ can be extended to a path in $\mathsf{Paths}(K(M), s_0)$ (i.e. it *exists*), where $w|_i$ is the prefix of w up to and including position i (i.e. $w|_n = w$). Note that path $p(w)$ has length $n+1$. The labels $l_0, l_1, \ldots, l_n$ along path $p(w)$ represent the observations $\mathsf{out}(w|_i, U)$ and the users commands, i.e. $l_0 = \{(u, \mathsf{out}(s_0, u)) \mid u \in U\}$ and $l_i = \{(u, \mathsf{out}(w|_i, u)) \mid u \in U\} \cup \{(u(i), c(i))\}$ for all $0 < i \leq n$.

We can pump all paths of $K(M)$ at any point: if $p = s_0, \ldots, s_{i-1}, s_i, s_{i+1}, \ldots$ is a path in $\mathsf{Paths}(K(M), s_0)$, then also $p' = s_0, \ldots, s_{i-1}, s_i, s_i, s_{i+1}, \ldots$ is a path in $\mathsf{Paths}(K(M), s_0)$ for any i. In particular, we know that p and its stuttered version p' show the same final observation, as seen from all observers.

Consider any finite sequence $w \in (U \times C)^*$ in the given deterministic state machine M. If M violates noninterference, then the *last* observations of the users G_L disagree for w and $|w|_{U \setminus G_H}$, i.e. $\mathsf{out}(w, G_L) = \mathsf{out}(|w|_{U \setminus G_H}, G_L)$. The path prefixes $p(w)$ and $p(|w|_{U \setminus G_H})$ necessarily have different labels in the last position, but they are not be compared to each other in the HyperLTL property, as they don't have the same length.

We now insert for every position in w that was deleted by the projection $|w|_{U \setminus G_H}$ a stuttering step in $p(|w|_{U \setminus G_H})$. We obtain a path $p''(|w|_{U \setminus G_H})$ that has again length $n+1$ and that shows the same low inputs, i.e. $I_{l,\pi} = I_{l,\pi'}$ is not violated up to position n. Further, as $K(M)$ is input-enabled, we can extend $p(w)$ and $p''(|w|_{U \setminus G_H})$ in a way such that $\Box(I_{l,\pi} = I_{l,\pi'})$ holds (for all times). Their observation at position n, however, are still different, such that the HyperLTL property is violated.

Now assume the HyperLTL property $\forall \pi. \forall \pi'. \ \Box(I_{l,\pi} = I_{l,\pi'}) \Rightarrow \Box(O_{l,\pi} = O_{l,\pi'})$ is violated. Let p and p' be two paths that have the same low inputs but different low observations and let n be the first position such that their low observations disagree. We consider their prefixes p_n and p_n' up to and including position n and remove all stuttering steps to obtain q_n and q_n'. The last state of prefix q_n still has the same observation as the last state of p_n and equivalently the last state of q_n' has the same observation as the last state of p_n'. We consider the command sequences w of q_n and w' of q_n' and their common subsequence $|w|_{U \setminus G_H}$ of commands of users in $U \setminus G_H$. GM-noninterference requires that both, w and w', must lead to the same observation as $|w|_{U \setminus G_H}$, but w and w' have different observations (in their last states). □

Security Definitions for Nondeterministic Systems

The system model used to define Goguen and Meseguer's noninterference only allowed for deterministic transition relations when all agents fixed their transitions. A common concern of many pieces of work following the seminal work of Goguen and Meseguer is the extension of noninterference to nondeterministic systems (e.g. [147, 85, 148, 88]). How should nondeterministic choices in the transition relation be treated, if they are not resolved upon fixing an input? The notions proposed in the literature for this case differ in their assumptions on the source of nondeterminism.

Observational determinism [148, 117] treats nondeterminism in the same way as secrets, representing the worst-case assumption that the nondeterminism may depend on the secret information. In particular, unspecified behavior in the low behavior leads to a violation of observational determinism, and hence the property is not necessarily applicable on abstract models. We consider observational determinism to be a modern formalization of noninterference and thus the two notions coincide in this thesis.

Noninference, in contrast, treats nondeterminism as an existential choice and requires that for every trace there must be an alternative trace that receives a dummy value λ as the secret input but produces the same output [89]:

$$\forall \pi . \exists \pi' .\, \square(I_{l,\pi} = I_{l,\pi'}) \;\wedge\; \square(I_{h,\pi'} = \lambda_{\pi'}) \;\wedge\; \square(O_{l,\pi} = O_{l,\pi'}) \,.$$

Noninference is satisfied for strictly more systems compared to noninterference, but noninference is not necessarily preserved under refinement. That is, it is unclear which security guarantees hold for implementations of an abstract model satisfying noninference.

Generalized noninterference tries to find a middle ground by requiring that for all traces and all alternative secrets, there is an execution trace that combines the alternative secret with the low input and low observation of the first trace [87, 89]. Thus, any secret can be an explanation for every observation.

$$\forall \pi . \forall \pi' .\, \exists \pi'' .\, \square(I_{l,\pi''} = I_{l,\pi}) \;\wedge\; \square(I_{h,\pi''} = I_{h,\pi'}) \;\wedge\; \square(O_{l,\pi''} = O_{l,\pi}) \,.$$

Integrity

Besides secrecy, information-flow security also considers the *integrity* of systems. Integrity requires that the high behavior of the system is not influenced by the low inputs that are potentially controlled by an attacker. Shellshock [51], POODLE [96, 50] and Stuxnet [3] are well-known incidents that affected the integrity of systems. On the technical level, integrity is similar to secrecy: It can be specified as a relation over the executions of the system requiring that all pairs of traces having the same high inputs, but possibly different low inputs, have the same high outputs. The following HyperLTL property specifies integrity:

$$\forall \pi . \forall \pi' .\, \square(I_{h,\pi} = I_{h,\pi'}) \Rightarrow \square(O_{h,\pi} = O_{h,\pi'}) \,,$$

Analogous to the discussion of declassification, we can also express exceptions to integrity in HyperLTL.

3.4 Applications in Distributed Systems

Mutual exclusion protocols are a classical object of study in distributed systems. The typical fairness properties for which mutual exclusion protocols have been analyzed can be seen as coarse abstractions of what is really expected from mutual exclusion protocols: symmetric access to the shared resource. HyperLTL enables a finer grained analysis of the symmetry between the processes by permitting to express, for example, that switching the actions and roles between two components in a trace results in another legal trace, in which the access to the shared resource is switched accordingly. That is, in a setting with two components that signal requests via the propositions r_1 and r_2, and receive grants via the propositions g_1 and g_2, the property can be expressed as follows:

$$\forall\pi.\forall\pi'.\ \square(r_{1,\pi} \leftrightarrow r_{2,\pi'} \wedge r_{2,\pi} \leftrightarrow r_{1,\pi'}) \Rightarrow \square(g_{1,\pi} \leftrightarrow g_{2,\pi'} \wedge g_{2,\pi} \leftrightarrow g_{1,\pi'})$$

It is well-known that mutual exclusion protocols cannot fulfill this strict version of symmetry, since both processes may request the resource at the same time and the protocol has to resolve this conflict in some (non-symmetric) way [83]. In Chapter 6, we study under which circumstances an implementation of a mutual exclusion protocol guarantees symmetric access.

3.5 Applications in Error Resistant Codes

Error resistant codes enable the transmission of data over noisy channels. While the correct operation of encoder and decoders is crucial for communication systems, the formal verification of their functional correctness has received little attention. A typical model of errors bounds the number of flipped bits that may happen for a given code word length. Then, error correction coding schemes must guarantee that all code words have a minimal Hamming distance. Alternation-free HyperLTL can specify that all code words produced by an encoder have a minimal Hamming distance of d:

$$\forall\pi.\forall\pi'.\Diamond(\bigvee_{a\in I} a_\pi \neq a_{\pi'}) \Rightarrow \neg\mathrm{Ham}_O(d-1,\pi,\pi')$$

where I are the inputs denoting the data, O denote the code words, and the predicate $\mathrm{Ham}_O(d,\pi,\pi')$ is defined as $\mathrm{Ham}_O(-1,\pi,\pi') := \mathsf{false}$ and:

$$\mathrm{Ham}_O(d,\pi,\pi') \ := \ \Big(\bigwedge_{a\in O} a_\pi = a_{\pi'}\Big)\mathcal{W}\Big(\bigvee_{a\in O} a_\pi \neq a_{\pi'} \ \wedge \ \bigcirc\mathrm{Ham}_O(d-1,\pi,\pi')\Big)$$

Note that this formalization is independent of the encoding scheme and can hence be applied to different coding schemes, as we demonstrate in Chapter 6.

Symmetries in the Golay code. The Golay code [56] is a block-code that operates on data words of 12 bits and uses code words of 23 bits (in some variants 24 bits). It was used by NASA during the first years of the Voyager missions [112], and was object of study in many works, also due to its tight links to group theory.

The code enjoys interesting properties, such as symmetries in the space of code words. For example, a code word of the Golay code that is inverted bitwise, is also a code word:

$$\forall\pi.\exists\pi'.\,\square(C_{\pi}\ \mathsf{xor}\ C_{\pi'})\ .$$

Even though many of the properties considered in this section have been analyzed so far on abstract models, little attention has been paid to the verification on the implementation level. In Chapter 6 we show that their formulation HyperLTL formulas enables the automatic verification of industrial hardware designs.

Chapter 4

Branching-time Temporal Logics

Additional to the traces a system generates, the branching-time view considers the *branching structure* of the system's behavior. The branching structure describes *when* nondeterministic choices are made, that is when information enters the system [92]. In this chapter we study temporal logics for branching-time in the light of information-flow properties.

Branching-time temporal logics like CTL and CTL* introduce use path quantifiers to distinguish Kripke structures with different branching structures. We determine that the extension of LTL to HyperLTL and the extension of LTL to CTL* introduces different kinds of expressiveness. That is, the expressiveness of HyperLTL and CTL/CTL* is incomparable. This raises the question what the combination of both kinds of expressiveness yields. We examine the extension of CTL and CTL* by the quantification over named *paths*, resulting in the logic HyperCTL*. The difference to HyperLTL is that the path quantifiers of HyperCTL* can be used inside temporal operators. HyperCTL* can thus reason both about the branching-structure of a system, which are the points where information enters the system, and the information-flows within the system.

We describe the class of *temporal information-flow properties*, which describes information-flow requirements that change over time. Temporal information-flow properties specify both aspects - the information-flow and the branching structure - and therefore fall outside HyperLTL and even outside hyperproperties (as hyperproperties do not include branching-time properties; see Section 2.2). We introduce a sublogic of HyperCTL*, which we call SecLTL, to ease the specification of (temporal) information-flow properties. Syntactically, SecLTL extends LTL by the *hide* operator, $\mathcal{H}_{H,O}\varphi$, which expresses that the secret input H that is read at the current point of time remains secret with respect to observations in O until a release condition φ becomes true. We also discuss how SecLTL expresses standard information-flow properties like noninterference and information-flow properties with declassification policies.

4.1 CTL and CTL*

CTL* is generated by the following grammar of state formulas Φ and path formulas φ:

$$\begin{array}{lllllllllll}
\Phi & ::= & a & \mid & \neg\Phi & \mid & \Phi\wedge\Phi & \mid & \mathrm{A}\,\varphi & \mid & \mathrm{E}\,\varphi \\
\varphi & ::= & \Phi & \mid & \neg\varphi & \mid & \varphi\wedge\varphi & \mid & \bigcirc\varphi & \mid & \varphi\,\mathcal{U}\,\varphi
\end{array}$$

Again, we consider the usual derived Boolean and temporal operators. CTL is the sublogic of CTL* where every temporal operator is immediately preceded by a path quantifier. CTL* state formulas Φ are interpreted over states and path formulas φ are interpreted over paths of *a given Kripke structure* and thus have access to more information than LTL formulas. For a given Kripke structure $K = (S, s_0, \delta, \mathsf{AP}, L)$ and a state $s \in S$, the semantics of CTL* state formulas is defined as follows:

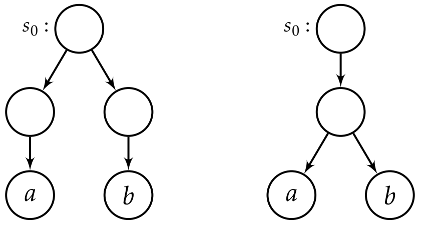
$$\begin{array}{lll}
s \models_K a & \text{iff} & a \in L(s) \\
s \models_K \neg\Phi & \text{iff} & s \not\models_K \Phi \\
s \models_K \Phi_1 \wedge \Phi_2 & \text{iff} & s \models_K \Phi_1 \text{ and } s \models_K \Phi_2 \\
s \models_K \mathrm{A}\,\varphi & \text{iff} & \forall p \in \mathsf{Paths}(K,s) : p \models_K \varphi \\
s \models_K \mathrm{E}\,\varphi & \text{iff} & \exists p \in \mathsf{Paths}(K,s) : p \models_K \varphi
\end{array}$$

The semantics of path formulas closely resembles the interpretation of LTL formulas. For a given Kripke structure $K = (S, s_0, \delta, \mathsf{AP}, L)$ and a path $p \in \mathsf{Paths}^*(K, s_0)$, we define:

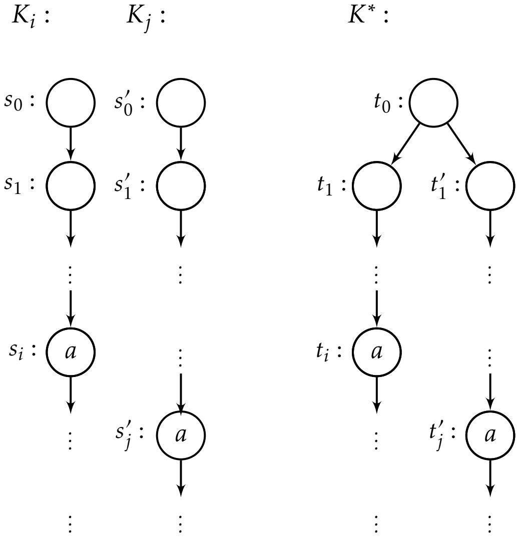
$$\begin{array}{lll}
p \models_K \Phi & \text{iff} & p(0) \models_K \Phi \\
p \models_K \neg\varphi & \text{iff} & s \not\models_K \varphi \\
p \models_K \varphi_1 \wedge \varphi_2 & \text{iff} & s \models_K \varphi_1 \text{ and } s \models_K \varphi_2 \\
p \models_K \bigcirc\varphi & \text{iff} & t[1,\infty] \models \varphi\text{, and} \\
p \models_K \varphi_1\,\mathcal{U}\,\varphi_2 & \text{iff} & \text{for some } i \geq 0 :\ t[i,\infty] \models \varphi_2 \text{ and} \\
 & & \quad\text{for all } 0 \leq j < i :\ t[j,\infty] \models \varphi_1
\end{array}$$

We say that a Kripke structure K with initial state s_0 satisfies a CTL* formula Φ, denoted $K \models \Phi$ if $s_0 \models_K \Phi$.

CTL and CTL* can distinguish trace-equivalent Kripke structures that differ in their branching structure. For example, the CTL formula $\mathrm{A}\bigcirc(\mathrm{E}\bigcirc a) \wedge \mathrm{E}\bigcirc b$ distinguishes the following pair of Kripke structures:



CTL and CTL* cannot, however, express noninterference, despite their ability to quantify over paths. Alur et al. observed that noninterference is not a regular tree property and hence is not expressible in CTL* [5]. The fact can also be shown by the following direct argument: Consider a family of observationally deterministic Kripke structures $K_1, K_2, \dots,$ where each K_i consists of a single branch from the initial state that only has one label a at step i:



All members of this family trivially satisfy noninterference. We pick a pair K_i and K_j with $i \neq j$ of Kripke structures such that s_1 and s'_1 satisfy the same subformulas of φ. (We can treat path formulas as state formulas as each path uniquely corresponds to a certain state.) Such a pair of Kripke structures must exist as φ has finitely many subformulas and the family of Kripke structures is infinite. We "merge" K and K' into one Kripke structure K^*, such that they share only the initial state as depicted above. By construction, states s_1, s'_1, t_1, and t'_1 all fulfill the same subformulas. Both, t_0 and s_0, have the same label (i.e. none) and all their successors satisfy the same subformulas of φ. Hence, they also satisfy the same subformulas of φ. In particular K^* satisfies φ but not noninterference, which contradicts the assumption.

This shows already that HyperLTL and CTL/CTL* are incomparable in terms of expressiveness.

A Branching-time Information-flow Property

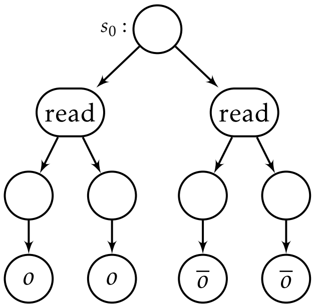
This raises the immediate question of what could be gained by the *combination* of the two types of expressiveness. Is there an information-flow property that requires branching-time aspects? To develop some intuition, let us consider the following example programs and their computation trees:

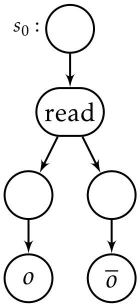
Program 1:

```
bool y;
bool x = read();
output(y);
```

Program 2:

```
bool x = read();
output(x);
```





We assume program semantics where the uninitialized variables, like variable y in Program 1, have a nondeterministic value that is fixed in the first step of the program. The computation tree of Program 1 thus branches in the initialization step into the cases where y is true and where y is false. In the second step of both Program 1 and Program 2 some input is read and stored in variable x. Here the programs branch again into the cases true and false.

Let us assume a security scenario in which the input is secret and should not be observable in the output. Then Program 1 is intuitively secure as the output is *independent* from the secret input (in particular it is chosen before the secret). Program 2, on the other hand, is obviously insecure.

We observe that the two programs are trace equivalent: they both admit exactly the two execution traces $\emptyset\{\mathrm{read}\}\emptyset\{o\}\emptyset^{\omega}$ and $\emptyset\{\mathrm{read}\}\emptyset\{\overline{o}\}\emptyset^{\omega}$. To develop an information-flow property that distinguishes the two cases, we thus have to look beyond hyperproperties.

The requirement needed to distinguish the two computation trees is that the outputs may depend on the choice of y, but not on the secret. This is a branching-time property: only the paths of the subtrees that are rooted in some state where the read-operation is executed, but not all paths in the computation tree, must have the same outputs. While CTL* can surely distinguish the two computation trees, as they are not bisimilar, the property cannot be expressed in CTL*. (The argument is the same as for noninterference: the *class of systems* that give arbitrary output over an unknown number steps could not be correctly classified in CTL*.)

We observe that the branching-time aspect helps us to isolate the *origin* of the relevant information. We are not the first ones to make this observation; Robin Milner formulated that "information enters a non-deterministic process in finite quantities throughout time", and that the branching-time view allows us to observe "in which states, and in what ways" this happens and he attributed this insight to Carl-Adam Petri [92]. Information-flow properties, in contrast, specify the *wherabouts* of information. While branching-time and hyperproperties are thus already useful individually, it is the combination that allows us to track information all the way from the point of entry, via some nondeterministic choice, to the point of exit through some externally observable variable.

4.2 HyperCTL*

Extending the path quantifiers of CTL* by *path variables* leads to the logic HyperCTL*, which subsumes both HyperLTL and CTL*. HyperCTL* allows us to express the property, "all paths rooted in some state where a read-operation is to be executed next must have the same output", discussed in the previous example:

$$\forall\pi.\, \Box(\mathrm{read}_\pi \Rightarrow \forall\pi'.\, \Box(o_\pi \leftrightarrow o_{\pi'}))$$

The formulas of HyperCTL* are generated by the following grammar:

$$\begin{array}{lllllllllll} \varphi & ::= & \mathsf{true} & \mid & a_\pi & \mid & \neg\varphi & \mid & \varphi\vee\varphi & \mid & \varphi\wedge\varphi \\ & \mid & \bigcirc\varphi & \mid & \varphi\,\mathcal{U}\,\varphi & \mid & \varphi\,\mathcal{R}\,\varphi & \mid & \exists\pi.\,\varphi & \mid & \forall\pi.\,\varphi \end{array}$$

We require that temporal operators only occur inside the scope of path quantifiers, and that HyperCTL* formulas are closed, which is defined analogous to closed HyperLTL formulas. In contrast to HyperLTL, we do not require formulas to be in prenex form. We introduce the derived operators $\Diamond\varphi = \mathsf{true}\,\mathcal{U}\,\varphi$, $\Box\varphi = \neg\Diamond\neg\varphi$, and $\varphi_1\,\mathcal{W}\,\varphi_2 = \varphi_1\,\mathcal{U}\,\varphi_2 \vee \Box\varphi_1$.

The semantics of HyperCTL* is given in terms of assignments of variables to *paths*, which are defined analogous to trace assignments. As for HyperLTL, we define the semantics only for the core set of operators and define the remaining operators with the equalities: $\forall\pi.\,\varphi = \neg\exists\pi.\,\neg\varphi$, $\varphi_1\wedge\varphi_2 = \neg(\neg\varphi_1 \vee \neg\varphi_2)$, and $\varphi_1\mathcal{R}\varphi_2 = \neg(\neg\varphi_1\mathcal{U}\neg\varphi_1)$. Given a Kripke structure $K = (S, s_0, \delta, \mathsf{AP}, L)$ and a special name $\varepsilon \in N$, the validity of HyperCTL* formulas is defined as follows:

$$\begin{array}{lll} \Pi \models_K a_\pi & \text{iff} & a \in L\big(\Pi(\pi)(0)\big) \\ \Pi \models_K \neg\varphi & \text{iff} & \Pi \not\models_K \varphi \\ \Pi \models_K \varphi_1 \vee \varphi_2 & \text{iff} & \Pi \models_K \varphi_1 \text{ or } \Pi \models_K \varphi_2 \\ \Pi \models_K \bigcirc\varphi & \text{iff} & \Pi[1,\infty] \models_K \varphi \\ \Pi \models_K \varphi_1\,\mathcal{U}\,\varphi_2 & \text{iff} & \text{for some } i \ge 0 :\ \Pi[i,\infty] \models_K \varphi_2 \text{ and} \\ & & \quad\text{for all } 0 \le j < i :\ \Pi[j,\infty] \models_K \varphi_1 \\ \Pi \models_K \exists\pi.\,\varphi & \text{iff} & \text{for some } p \in \mathsf{Paths}(K, \Pi(\varepsilon)(0)) :\ \Pi[\pi \mapsto p, \varepsilon \mapsto p] \models_K \varphi \end{array}$$

The variable ε denotes the path most recently added to Π (i.e., closest in scope to π). For the empty assignment $\Pi = \emptyset$, we define $\Pi(\varepsilon)(0)$ to yield the initial state of K. A Kripke structure $K = (S, s_0, \delta, \mathsf{AP}, L)$ satisfies a HyperCTL* formula φ, denoted with $K \models \varphi$, iff $\emptyset \models_K \varphi$.

Induced process equivalence. When we restrict HyperCTL* to a single path variable (thus all path quantifiers have to use the same path variable), its formulas directly correspond to formulas in CTL*. CTL* can hence be seen as a sublogic of HyperCTL*. Since CTL* induces bisimulation [8], the induced equivalence of HyperCTL* must be at least as fine as bisimulation equivalence. A *bisimulation* for a pair of Kripke structures $K = (S, s_0, \delta, \mathsf{AP}, L)$ and $K' = (S', s'_0, \delta', \mathsf{AP}', L')$ is an equivalence relation $R \subseteq S \times S'$ on their states, such that it holds for all pairs $(s,s') \in R$ that $L(s) = L'(s')$ and for all successors $t \in \delta(s)$ of s, there exists a successor $t' \in \delta'(s')$ of s' such that $(t,t') \in R$, and vice versa. Two Kripke structures $K = (S, s_0, \delta, \mathsf{AP}, L)$ and $K' = (S', s'_0, \delta', \mathsf{AP}', L')$ are called *bisimulation equivalent* (or *bisimilar*), iff there exists a bisimulation $R \subseteq S \times S'$ and $(s_0, s'_0) \in R$.

Before we show that the equivalence induced by HyperCTL* is not finer than bisimulation, we need to lift bisimulation to paths and path assignments. In the following, let K and K' be Kripke two structures. A pair of paths $p \in \mathsf{Paths}^*(K, s_0)$ and $p' \in \mathsf{Paths}^*(K', s'_0)$ with $p = s_0 s_1 \ldots$ and $p' = s'_0 s'_1 \ldots$ is called *bisimilar*, written $p \sim p'$, if there is a bisimulation $\sim$ on the states of K and K' such that $s_j \sim s'_j$ for all $j \in \mathbb{N}$. A pair of path assignments $\Pi : N \to \mathsf{Paths}^*(K, s_0)$ and $\Pi' : N \to \mathsf{Paths}^*(K', s'_0)$ is called *bisimilar*, written $\Pi \sim \Pi'$, if they bind the same set of variables, $\Pi^{-1}(\mathsf{Paths}^*(K, s_0)) = \Pi'^{-1}(\mathsf{Paths}^*(K', s'_0)) = N$, and for all $\pi \in N$ it holds $\Pi(\pi) \sim \Pi'(\pi)$. In the special case of two empty path assignments, the path assignments are bisimilar iff there is a bisimulation for K and K' such that their initial states s_0 and s'_0 are bisimilar.

We show by induction on the formula structure that a HyperCTL* formula has the same value in all bisimilar path assignments. This implies that HyperCTL* cannot distinguish two bisimilar Kripke structures, because the empty assignments $\emptyset$ are bisimilar for all bisimilar Kripke structures. Bisimilar path assignments satisfy, by definition, the same atomic propositions. The path quantifier $\exists\pi.\varphi$ selects a new path starting in the state $\Pi(\varepsilon)(0)$ and $\Pi'(\varepsilon)(0)$, respectively. Because these states are bisimilar, there is a pair of bisimilar paths starting in these states [8, Lemma 7.5]. Hence, the path assignments $\Pi[\pi \to p]$ and $\Pi'[\pi \to p']$ are again bisimilar and, by induction hypothesis, $\exists\pi.\varphi$ has the same value in $\Pi[\pi \to p] \models \varphi$ and $\Pi'[\pi \to p'] \models \varphi$. For the temporal operators $\bigcirc$ and $\mathcal{U}$, we note that suffixes from identical positions of bisimilar paths are bisimilar again; hence, suffixes of bisimilar path assignments are bisimilar again. Therefore, by induction hypothesis, $\bigcirc\varphi$, and likewise $\varphi_1 \,\mathcal{U}\, \varphi_2$, has the same value in bisimilar path assignments.

Theorem 4.2.1. *HyperCTL* induces bisimulation.*

The ability to express hyperproperties is thus orthogonal to the ability to express branching-time properties. In Fig. 4.1 we classify the expressiveness of the standard temporal logics LTL, CTL, and CTL* as well as HyperLTL and HyperCTL* along two axes. The linear-branching axis organizes the logics according to the induced process equivalence and therefore puts LTL and HyperLTL, and CTL* and HyperCTL* into the same group. The “hyper” axis

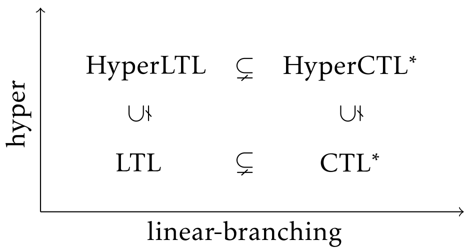


Figure 4.1: Linear-hyper-branching spectrum of temporal logics

classifies the logics according to their expressiveness with respect to properties that relate multiple paths, which separates HyperLTL and HyperCTL* from LTL and CTL*.

Relation to HyperLTL

Syntactically HyperCTL* extends HyperLTL by formulas that are not in prenex form; that is we can now use path quantifiers inside temporal operators. While the semantics of HyperLTL is based on trace assignments, the semantics of HyperCTL* is based on path assignments and thus carry additional information about the states visited during the execution. It is easy to see that quantifier-free subformulas of HyperCTL* only depend on the sequence of labels and thus have the same interpretation as in HyperLTL.

Lemma 4.2.2. *Given a Kripke structure K with labelling function L, a path assignment Π, a trace assignment Π' with $\Pi'(\pi)(i) = L(\Pi(\pi)(i))$ for all π and i, and a quantifier-free subformula φ, then the interpretations of φ in HyperLTL and HyperCTL* agree:*

$$\Pi \models_K \varphi \quad \textit{iff} \quad \Pi' \models_K \varphi \,.$$

Since path quantifiers only accesses the first state of the most recently quantified path and it is clear that the paths bound by two path quantifiers that are not separated by temporal operators (i.e. where no Next, Until, or Release occurs on the unique path between the to operators in the syntax tree of the formula) always agree on their first state. In particular, when no path quantifier occurs inside temporal operators the expressiveness of HyperCTL* reduces to the expressiveness of HyperLTL.

Theorem 4.2.3. *For all HyperCTL* formulas φ where no path quantifier occurs inside Next, Until, or Release, there is a HyperLTL formula φ' that satisfies the same Kripke structures K.*

Proof. Given a HyperCTL* formula φ, we rename the path variables such that every path variable occurs in at most one the path quantifier. We can then pull the quantifiers to the front of the formula to obtain the formula φ'. As Kripke structures have a unique initial state, quantifying over all traces of the

Kripke structure is the same as quantifying over all traces starting in the first state of the most recently quantified path ($\Pi(\varepsilon)(0)$) when the path quantifiers do not occur inside temporal operators. □

Satisfiability

The satisfiability problem, that is to ask for the existence of a model for a given formula, is a common problem analyzed in computational logics. The *finite-state satisfiability problem* for HyperCTL* is to determine the existence of a finite model, while the general *satisfiability problem* for HyperCTL* asks for the existence of a possibly infinite model. That is, the satisfiability problem for HyperCTL* is unlike the satisfiability problem for LTL, where the satisfiability problem asks for the existence of a trace that satisfies the property. Instead, the satisfiability problem of HyperCTL* is closely related to the reactive synthesis problem for LTL. We can express on which inputs the outputs may depend and thus it is not surprising that the problem is undecidable. The general satisfiability problem may be even harder: we give a reduction from the Halting problem for Turing machines with an oracle for the Turing machine Halting problem, which defines the complexity class Σ_1^1 from the arithmetical hierarchy.

Theorem 4.2.4. *The finite-state satisfiability problem for HyperCTL* is undecidable and the satisfiability problem is hard for* Σ_1^1.

Proof. We give a reduction from the synthesis problem for LTL specifications in a distributed architecture consisting of two processes with disjoint sets of variables. The synthesis problem consists on deciding whether there exist transition systems for the two processes with input variables I_1 and I_2, respectively, and output variables O_1 and O_2, respectively, such that the synchronous product of the two transition systems satisfies a given LTL formula φ. This problem is hard for Σ_1^0 if the transition systems are required to be finite, and hard for Σ_1^1 if infinite transition systems are allowed (Theorems 5.1.8 and 5.1.11 in [118]).

To reduce the synthesis problem to HyperCTL* satisfiability, we construct a HyperCTL* formula ψ as a conjunction $\psi = \psi_1 \wedge \psi_2 \wedge \psi_3$. The first conjunct ensures that φ holds on all paths: $\psi_1 = \forall\pi.[\varphi]_\pi$. The second conjunct ensures that every state of the model has a successor for every possible input: $\forall\pi.\square \bigwedge_{I\subseteq I_1\cup I_2} \exists\pi'.\bigcirc \bigwedge_{i\in I} i \;\; \bigwedge_{i\notin I} \neg i$. The third conjunct ensures that the output in O_1 does not depend on I_2 and the output in O_2 does not depend on I_1: $\psi_3 = \forall\pi.\forall\pi'.\left(\pi =_{I_1} \pi' \rightarrow \pi =_{O_1} \pi'\right) \wedge \left(\pi =_{I_2} \pi' \rightarrow \pi =_{O_2} \pi'\right)$. The distributed synthesis problem has a (finite) solution iff the HyperCTL* formula ψ has a (finite) model. □

Logics with an undecidable satisfiability problem can still be of practical use. In particular for temporal logics, the model checking problem, that is to decide $K \models \varphi$ for a given Kripke structure K and a given formula φ, is highly relevant for the functional verification of software and hardware. In Chapter 5 we determine that the model checking problem is decidable.

4.3 SecLTL

The usability of a logic is not only determined by its expressiveness, but also by its simplicity. This is particularly important in security, because a users lack of understanding of the conditions under which his or her system is secure easily opens the possibility for attacks. In this section, we discuss an attempt to tackle this problem.

Nested path quantifiers and temporal operators are likely too complex to be used in a specification language for a wider audience. The temporal logic SecLTL [38], a fragment of HyperCTL*, simplifies the specification of many information-flow properties. SecLTL is based on LTL syntax and introduces the *Hide* modality $\mathcal{H}_{H,O}\varphi$, which expresses the absence of an information flow between two sets H and O of atomic propositions until the *release condition* φ is met. The property discussed in the example in Section 4.1 can also be expressed in SecLTL:

$$\square(\mathsf{read} \Rightarrow \mathcal{H}_{\{\},\{o\}}\mathsf{false})$$

To give SecLTL formulas an intuitive meaning, we restrict our discussion to Kripke structures that are input-enabled for 2^I, where $I \subset \mathsf{AP}$ is the set of *input propositions*.

The formulas of SecLTL are generated by the following grammar:

$$\varphi \ ::= \ a \mid \neg\varphi \mid \varphi \wedge \varphi \mid \bigcirc\varphi \mid \varphi\,\mathcal{U}\,\varphi \mid \mathcal{H}_{H,O}\varphi\,,$$

where $H \subseteq I$ and $O \subseteq \mathsf{AP}$.

Given a SecLTL formula φ, we define the validity on Kripke structures K that are input-enabled for I, denoted $K \models \varphi$, via a translation to HyperCTL*: Select two distinct path variables π and π', prepend a universal quantifier $\forall\pi$, index all atomic propositions in φ with the path variable π, and we replace any Hide operator $\mathcal{H}_{H,O}\psi$ by

$$\forall\pi'.\ \bigcirc\Big((I\setminus H)_\pi = (I\setminus H)_{\pi'} \ \wedge\ \bigcirc\square\ I_\pi = I_{\pi'}\Big) \implies \Big(O_\pi = O_{\pi'}\ \mathcal{W}\ [\psi]\Big)\,,$$

where $[\psi]$ denotes the formula that results from ψ when through indexing all atomic propositions with π and replacing the hide operators by the formula above. Consider that the outermost "Next" operator ($\bigcirc$) of the antecedent in the translation from SecLTL to HyperCTL* only compensates the unit delay of the input: in each state, we can only see the *last* input.

The Hide operator thus compares all those paths to the "main path" that have the same input I, except for the *first* valuation of H. Let us take another look at the example in Section 4.1 to see how this enables simple specifications of interesting information flow properties. The system considered in the example had a single input that is represented by the nondeterministic choice after the read-operation of which the result is stored in variable x. The

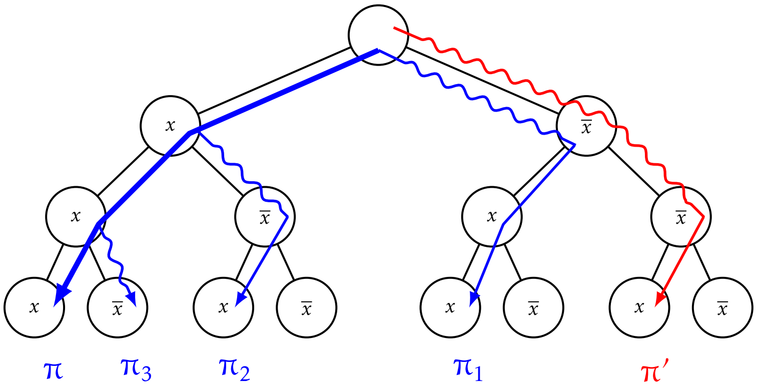


Figure 4.2: A sketch of a computation tree of a Kripke structure that is input-enabled in an atomic proposition x. Consider the SecLTL formula $\square\mathcal{H}_{\{x\},O}\mathsf{false}$ applied to the blue execution path π. The other three paths marked blue indicate the paths that π is compared to. Those segments on paths π_1, π_2, and π_3 that are drawn with a squiggly line indicate the unique points where paths π_i differ from π. The red execution path π' is not *directly* compared to π according to the semantics of the Hide operator.

SecLTL formula $\square(\mathsf{read} \Rightarrow \mathcal{H}_{\{\},\{o\}}\mathsf{false})$ translates[1] to the HyperCTL* formula $\forall\pi.\ \square(\mathsf{read}_\pi \implies \forall\pi'.\ \square\ o_\pi \leftrightarrow o_{\pi'})$, which is exactly the formulation of the property in HyperCTL* in Section 4.1. This exactly matches our intuition for the example, where we wanted to characterize the secret as the single point in the computation tree where we branch for the input. But how does this relate to the standard information-flow policies like noninterference?

In contrast to noninterference, *a single application* of a Hide operator considers only the first input in the specified input proposition as the secret. Using temporal connectives, however, it is easy to declare any branching in the input propositions H as a secret. The formula $\square\mathcal{H}_{H,O}\mathsf{false}$ expresses that for all executions π and all points in time i, the alternative executions that differ from π only in the branching in the inputs H at time i produce the same sequence of observations O. We thus compare executions only to those executions that differ in one point of time, but not multiple points in time.

Consider the simple computation tree in Fig. 4.2. For the SecLTL formula $\square\mathcal{H}_{H,O}\mathsf{false}$, the execution path π is only compared to the executions π_1, π_2, and π_3. The red execution path π' is not *directly* compared to π according to the semantics of the Hide operator, as there are two points in time, i.e. step 1 and step 2, in which their input differs.

Nevertheless, the formula $\square\mathcal{H}_{H,O}\mathsf{false}$ is equivalent to noninterference for input-enabled Kripke structures. The input-enabledness of the Kripke structure guarantees that every sequence of inputs results in a path. Since SecLTL

[1] We also applied the simplification $\varphi\,\mathcal{W}\,\mathsf{false} = \square\varphi$.

formulas are implicitly quantified over all executions, also the execution path π_1 has to satisfy the path formula $\square\mathcal{H}_{H,O}\mathsf{false}$, which involves the comparison to path π'. If π is observationally equivalent to π_1 and π_1 is observationally equivalent to π', then also π and π' must be observationally equivalent.

Theorem 4.3.1. *For Kripke structures K that are input-enabled in I, the SecLTL formula $\square\mathcal{H}_{H,O}\mathsf{false}$ expresses that O is observationally deterministic in $I \setminus H$. That is,*

$$K \models \square\mathcal{H}_{H,O}\mathsf{false} \quad \textit{iff} \quad K \models \forall\pi.\forall\pi'.\, \square((I\setminus H)_\pi = (I\setminus H)_{\pi'}) \Rightarrow \square(O_{l,\pi} = O_{l,\pi'})\,.$$

Proof. The formula $\varphi = \square\mathcal{H}_{H,O}\mathsf{false}$ is clearly weaker than noninterference. To prove the other direction, consider an input-enabled Kripke structure, and two execution paths p and p' that have the inputs in $I \setminus H$, i.e. the same low inputs. We have to show that φ is violated, if p and p' differ in the valuations of O.

Assume that $i \in \mathbb{N}$ is the first position at which $p(i) \neq p'(i)$ and consider the finite set $D \subset \mathbb{N}$ of positions smaller than i at which p and p' differ in the valuation in H. Let $j_1 < j_2 < \cdots < j_d$ be elements of D in increasing order. We construct a sequence of paths $p_{j_1}, p_{j_2}, \ldots, p_{j_d}$, such that each p_{j_k} has the same high input as p except for the positions $j_1, j_2, \ldots, j_k$, where it has the high inputs of p'. We additionally require $p' = p_{j_d}$.

The formula φ requires that each pair of paths $(p_{j_k}, p_{j_{k+1}})$ and also (p, p_{j_1}) have the same low observations. As the equivalence of low observations on paths is an equivalence relation, p and p' must be in the same equivalence class. □

4.4 Applications: Temporal Information-flow

HyperCTL* and SecLTL enable the specification of information-flow requirements that change over time. Because of their branching-time nature we can also precisely classify single inputs as high or low. The Hide modality of SecLTL eases the specification of such that we can formulate understandable security properties for complex scenarios. In this section, we demonstrate this by specifying information-flow requirements in a conference management system; a system in which multiple users cooperate to submit, review, discuss, and select scientific documents.

Besides the static requirement that the reviewer's identities must remain secret, conference management systems show intricate information-flow requirements that change over time. Properties of interest are, for example, that (4.1) *"the final decision of the program committee remains secret until the notification is sent to the authors"* and that (4.2) *"all intermediate decisions of the program committee are never revealed to the author"*. Their formalization in SecLTL is intuitive:

$$\square(\bigcirc\text{last\_decision} \Rightarrow \mathcal{H}_{H,O}\ \text{notification\_phase}) \tag{4.1}$$

$$\square(\neg\bigcirc\text{last\_decision} \Rightarrow \mathcal{H}_{H,O}\mathsf{false}) \tag{4.2}$$

where H is defined as the set of propositions that concern the acceptance and rejectance decisions of the program committee, as well as the contributions to the discussion, and O is defined as the set of propositions visible to the authors.

Here, we assume that the point of time of the last decision can be characterized by the atomic proposition "last_decision". In case the system does not indicate this fact by a dedicated atomic proposition, we could easily adapt the property by characterizing this point with a temporal formula. For example, we could specify the last decision as the last time a new decision is entered before the notification phase: $\bigcirc(\neg\text{new\_decision}\ \mathcal{U}\ \text{notification\_phase})$.

This shows the strength of the approach to use temporal logics for the specification of information-flow requirements: in the common situation that a static property such as noninterference is not applicable, a logic like SecLTL or HyperCTL* allows us to precisely formulate the circumstances under which the information-flow requirements are lifted. The formulations of information-flow properties for conference management systems in SecLTL [38] have inspired a large-scale case study of the construction and verification of a secure conference management system [67] that included the verification of the properties above.

Chapter 5

Algorithmic Verification

The aim of algorithmic verification is to automatically verify the correctness of a system or otherwise to reveal a flaw in its design. Thereby it promises to improve the quality of software and hardware and to simplify the development process, in which the validation, e.g. via testing and code reviews, accounts for a significant portion of the development time. The automatic analysis of software and hardware systems is undecidable in general [134], but the problem has attracted a lot of research since the early days of computer science.

Automata-theoretic model checking is a fundamental approach to the automatic analysis of software and hardware systems that focuses on the decidability and complexity of the analysis of restricted classes of systems, such as finite state systems, which may represent hardware systems or abstractions of software systems. Instead of considering fixed properties, model checking typically considers specification languages such as temporal logics. While the model checking problem for the standard temporal logics LTL and CTL/CTL* has been well-studied [110, 29, 41, 76], a comparable algorithmic theory is not available for information-flow properties. After the seminal work of Goguen and Meseguer proposing noninterference [55] as a fundamental notion of secrecy, research on the automatic analysis of secrecy properties has mostly focused on language-based techniques that merely *approximate* the property [119]. To check notions of secrecy without approximation, Barthe et al. [12] proposed *self-composition*, a system transformation that reduces information-flow properties to trace properties [129]. On a self-composed system, the requirements to check observational determinism can be formulated in standard temporal logics [63]. So far, however, the verification of information-flow properties via self-composition has only been considered for isolated properties.

This chapter develops a general verification approach for arbitrary properties specified in HyperLTL and HyperCTL* and thereby represents the center piece of this thesis. Via an alternating automaton construction, we give a model checking algorithm for finite state systems (Section 5.2). We then generalize the model checking algorithm to formulas with quantifier alternations (Section 5.3). A detailed complexity analysis of the model checking problem, both in the size of the formula and the size of the Kripke structure to check, shows that the algorithm is optimal in terms of complexity (Section 5.6). While the algorithm is efficient for alternation-free formulas, it is non-elementary in the general case. We conclude this chapter with a discussion of cases when the exponential blow-up caused by quantifier-alternations can be avoided (Section 5.7).

5.1 Alternating Büchi Automata

We start with a brief review of alternating automata. Given a finite set Q, $\mathbb{B}(Q)$ denotes the set of Boolean formulas over Q and $\mathbb{B}^+(Q)$ the set of positive Boolean formulas, that is, formulas that do not contain negation. The satisfaction of a formula $\theta \in \mathbb{B}(Q)$ by a set $Q' \subseteq Q$ is denoted by $Q' \models \theta$.

Definition 5.1.1 (Alternating Büchi automata). *An* alternating Büchi automaton *(on words) is a tuple $\mathcal{A} = (Q, q_0, \Sigma, \rho, F)$, where Q is a finite set of states, $q_0 \in Q$ is the initial state, Σ is a finite alphabet, $\rho : Q \times \Sigma \to \mathbb{B}^+(Q)$ is a transition function that maps a state and a letter to a positive Boolean combination of states, and $F \subseteq Q$ are the accepting states.*

A run of an alternating automaton is a Q-labeled tree. A *tree* T is a subset of $\mathbb{N}^*_{>0}$ such that for every *node* $\tau \in \mathbb{N}^*_{>0}$ and every positive integer $n \in \mathbb{N}_{>0}$, (i) if $\tau \cdot n \in T$ then $\tau \in T$ (i.e., T is prefix-closed), and (ii) for every $0 < m < n$, $\tau \cdot m \in T$. The root of T is the empty sequence ε and for a node $\tau \in T$, $|\tau|$ is the length of the sequence τ, in other words, its distance from the root. A *run* of $\mathcal{A}$ on an infinite word $\pi \in \Sigma^\omega$ is a Q-labeled tree (T, r) such that $r(\varepsilon) = q_0$ and for every node τ in T with children $\tau_1, \ldots, \tau_k$ the following holds: $1 \leq k \leq |Q|$ and $\{r(\tau_1), \ldots, r(\tau_k)\} \models \rho(q, \pi[i])$, where $q = r(\tau)$ and $i = |\tau|$. A run r of $\mathcal{A}$ on $\pi \in \Sigma^\omega$ is *accepting* whenever for every infinite path $\tau_0\tau_1\ldots$ in T, there are infinitely many i with $r(\tau_i) \in F$. We say that π is accepted by $\mathcal{A}$ whenever there is an accepting run of $\mathcal{A}$ on π, and denote with $\mathcal{L}_\omega(\mathcal{A})$ the set of infinite words *accepted* by $\mathcal{A}$.

If the transition function of an alternating automaton does not contain any conjunctions, we call the automaton *nondeterministic*. The transition function ρ of a nondeterministic automaton thus identifies a disjunction over a set of successor states. Such a transition function can also be stated as a function $\rho : Q \times \Sigma \to 2^Q$ identifying the successors. Our model checking algorithm relies on the standard translation for alternation removal due to Miyano and Hayashi:

Theorem 5.1.2 ([95]). *Let $\mathcal{A}$ be an alternating Büchi automaton with n states. There is a nondeterministic Büchi automaton $\mathsf{MH}(\mathcal{A})$ with $2^{\mathcal{O}(n)}$ states that accepts the same language.*

Proof. Let $\mathcal{A} = (Q, q_0, \Sigma, \rho, F)$ be an alternating Büchi automaton. We construct a nondeterministic Büchi automaton $\mathcal{N} = (Q^\mathcal{N}, q_0^\mathcal{N}, \Sigma, \rho^\mathcal{N}, F^\mathcal{N})$ as follows: $Q^{\mathcal{N}} = 2^Q \times 2^Q$, $q_0^{\mathcal{N}} = (\{q_0\}, \emptyset)$, $F^{\mathcal{N}} = \{(R, \emptyset) \mid R \subseteq Q\}$. The transition relation is defined as $\rho^{\mathcal{N}}((R_1, R_2), a) = \{(R_1', R_1' \setminus F) \mid R_1' \models \bigwedge_{q \in R_1} \rho(q, a)\}$, if $R_2 = \emptyset$, and

$$\begin{aligned}\rho^{\mathcal{N}}((R_1, R_2), a) = \{(R_1', R_2' \setminus F) \mid \quad & R_2' \subseteq R_1', \\ & R_1' \models \textstyle\bigwedge_{q \in R_1} \rho(q, a), \\ & R_2' \models \textstyle\bigwedge_{q \in R_2} \rho(q, a)\}\end{aligned}$$

if $R_2 \neq \emptyset$. Furthermore, we may assume that the sets R_1 are minimal, that is that $\forall R_1'' \subset R_1'.\ R_1'' \not\models \bigwedge_{q \in R_1} \rho(q, a)$. □

5.2 Model Checking the Alternation-Free Fragment

We present a model checking algorithm for the alternation-free fragments of HyperLTL and HyperCTL*. For the purpose of the following discussion, we can consider HyperLTL as the prenex fragment of HyperCTL* and thus focus on HyperCTL*. The alternation-free fragment is expressive enough to capture a broad range of other information-flow properties, like declassification mechanisms, quantitative noninterference, and information-flow requirements that change over time [30, 38]. The case studies in Chapter 6 illustrate that this fragment also captures properties in application domains beyond information-flow security.

Definition 5.2.1 (Alternation-free HyperCTL*). *A HyperCTL* formula* φ *in negation normal form is* alternation-free, *if* φ *contains only quantifiers of one type. Additionally, we require that no existential quantifier occurs in the left subformula of an until operator or in the right subformula of a release operator, and, symmetrically, that no universal quantifier occurs in the right subformula of an until operator or in the left subformula of a release operator.*

Similar to the automata-theoretic approach to LTL properties [97, 137], we construct an alternating automaton bottom up from the formula, but handling multiple path quantifiers. For alternation-free HyperCTL*, the quantifiers may occur inside temporal operators (with the restrictions in Def. 5.2.1) as long as there is no quantifier alternation.

Let K be a Kripke structure $K = (S, s_0, \delta, \mathsf{AP}, L)$. To check the satisfaction of a HyperCTL* formula φ by K, we translate φ into a K-equivalent alternating automaton $\mathcal{A}_\varphi$. The construction of $\mathcal{A}_\varphi$ proceeds inductively following the structure of φ, as follows. Assume that φ is in negation normal form and starts with an existential quantifier, and consider a subformula ψ of φ. Let n be the number of path quantifiers occurring on the path from the root of the syntax tree of φ to ψ, and let these path quantifiers bind the variables $\pi_1, \dots, \pi_n$. The alphabet Σ of $\mathcal{A}_\psi$ is S^n, the set of n-tuples of states of K. We say that a language $L \subseteq (S^n)^\omega$ is *K-equivalent* to ψ, if all sequences of state tuples $(s_0^0, \dots, s_n^0)(s_0^1, \dots, s_n^1)\dots$ in L correspond to a path assignment Π satisfying ψ. That is, for all $(s_0^0, \dots, s_n^0)(s_0^1, \dots, s_n^1)\dots \in L$ it holds $\Pi \models_K \psi$ for the path assignment $\Pi(\pi_i) = s_i^0 s_i^1 \dots$ (for all $i \leq n$). An automaton is K-equivalent to ψ if its language is K-equivalent to ψ.

For atomic propositions, Boolean connectives, and temporal operators, our construction follows the standard translation from LTL to alternating automata [97, 137]. Let $\mathcal{A}_{\psi_1} = (Q_1, q_{0,1}, \Sigma_1, \rho_1, F_1)$ and $\mathcal{A}_{\psi_2} = (Q_2, q_{0,2}, \Sigma_2, \rho_2, F_2)$ be the alternating automata for the subformulas ψ_1 and ψ_2:

$\psi = a_{\pi_k}$ $\qquad \mathcal{A}_\psi = (\{q_0\}, q_0, \Sigma, \rho, \emptyset)$,

where $\rho(q_0, \vec{s}) = (a \in L(\vec{s}\rceil_k))$

$\psi = \neg a_{\pi_k}$ $\qquad \mathcal{A}_\psi = (\{q_0\}, q_0, \Sigma, \rho, \emptyset)$,

where $\rho(q_0, \vec{s}) = (a \notin L(\vec{s}\rceil_k))$

$\psi = \psi_1 \vee \psi_2$ $\qquad \mathcal{A}_\psi = (Q_1 \uplus Q_2 \uplus \{q_0\}, q_0, \Sigma, \rho, F_1 \uplus F_2)$,

where $\rho(q_0, \vec{s}) = \rho_1(q_{0,1}, \vec{s}) \vee \rho_2(q_{0,2}, \vec{s})$
and $\rho(q, \vec{s}) = \rho_i(q, \vec{s})$ for $q \in Q_i, i \in \{1, 2\}$

$\psi = \psi_1 \wedge \psi_2$ $\qquad \mathcal{A}_\psi = (Q_1 \uplus Q_2 \uplus \{q_0\}, q_0, \Sigma, \rho, F_1 \uplus F_2)$,

where $\rho(q_0, \vec{s}) = \rho_1(q_{0,1}, \vec{s}) \wedge \rho_2(q_{0,2}, \vec{s})$
and $\rho(q, \vec{s}) = \rho_i(q, \vec{s})$ for $q \in Q_i, i \in \{1, 2\}$

$\psi = \bigcirc \psi_1$ $\qquad \mathcal{A}_\psi = (Q_1 \uplus \{q_0\}, q_0, \Sigma, \rho, F)$,

where $\rho(q_0, \vec{s}) = q_{0,1}$
and $\rho(q, \vec{s}) = \rho_1(q, \vec{s})$ for $q \in Q_1$

$\psi = \psi_1 \,\mathcal{U}\, \psi_2$ $\qquad \mathcal{A}_\psi = (Q_1 \uplus Q_2 \uplus \{q_0\}, q_0, \Sigma, \rho, F)$,

where $\rho(q_0, \vec{s}) = \rho_2(q_{0,2}, \vec{s}) \vee (\rho_1(q_{0,1}, \vec{s}) \wedge q_0)$
and $\rho(q, \vec{s}) = \rho_i(q, \vec{s})$ for $q \in Q_i, i \in \{1, 2\}$

$\psi = \psi_1 \,\mathcal{R}\, \psi_2$ $\qquad \mathcal{A}_\psi = (Q_1 \uplus Q_2 \uplus \{q_0\}, q_0, \Sigma, \rho, F \uplus \{q_0\})$,

where $\rho(q_0, \vec{s}) = \rho_2(q_{0,2}, \vec{s}) \wedge (\rho_1(q_{0,1}, \vec{s}) \vee q_0)$
and $\rho(q, \vec{s}) = \rho_i(q, \vec{s})$ for $q \in Q_i, i \in \{1, 2\}$

For a quantified subformula $\psi = \exists \pi. \psi_1$, we have to reduce the alphabet $\Sigma_{\psi_1} = S^{n+1}$ to $\Sigma = S^n$. The language for formula ψ contains exactly those sequences σ of state tuples, such that there is a path p through the Kripke structure K for which σ extended by p is in $\mathcal{L}(\mathcal{A}_{\psi_1})$. Let $\mathcal{N}'_{\psi_1} = (Q', q'_0, \Sigma, \rho', F')$ be the nondeterministic automaton $\mathcal{N}'_{\psi_1} = \mathsf{MH}(\mathcal{A}_{\psi_1})$ constructed from $\mathcal{A}_{\psi_1}$ by the construction in Theorem 5.1.2, and let $\mathcal{A}_\psi = (Q'', q''_0, \Sigma_\psi, \rho'', F'')$ be constructed from $\mathcal{N}'_{\psi_1}$ and the Kripke structure $K = (S, s_0, \delta, \mathsf{AP}, L)$ as follows:

$\psi = \exists \pi. \psi_1$ $\qquad \mathcal{A}_\psi = (Q' \times S \uplus \{q''_0\}, q''_0, \Sigma_\psi, \rho'', F' \times S)$,

where $\rho''(q''_0, \vec{s}) = \{(q', s') \mid q' \in \rho'(q'_0, \vec{s} + \vec{s}\rceil_n), s' \in$
$\delta(\vec{s}\rceil_n)\}$
and $\rho''((q, s), \vec{s}) = \{(q', s') \mid q' \in \rho'(q, \vec{s} + s), s' \in \delta(s)\}$

For the case that $n = 0$ we define that $\vec{s}|_n$ is the initial state s_0 of K.

This completes the construction for the alternation-free fragment. Its correctness can be shown by structural induction.

Proposition 5.2.2. *Let φ be a HyperCTL* formula and $\mathcal{A}_\varphi$ the alternating automaton obtained by the previous construction. Then, φ and $\mathcal{A}_\varphi$ are K-equivalent.*

So far, we only considered alternation-free formulas that start with existential quantifiers. To decide $K \models \varphi$ for an arbitrary φ, we first transform φ in a Boolean combination over a set X of quantified subformulas. Each element ψ' of X is now in the form $\exists\pi.\varphi$ for which we apply the construction above. Since ψ' is of the form $\exists\pi.\psi_1$, $\mathcal{A}_{\psi'}$ is a nondeterministic Büchi automaton. We apply a standard nonemptiness test to determine the answer to the model checking problem [139].

Theorem 5.2.3. *The model checking problem for the alternation-free fragment of HyperCTL* is PSPACE-complete in the size of the formula and NLOGSPACE-complete in the size of the Kripke structure.*

Proof. The alternating automaton $\mathcal{A}_{\psi_1}$ is a tree with self-loops, when we consider automata created for quantified subformulas as leafs of the tree. By structural induction, we show that the size of $\mathcal{A}_{\psi'}$ for an alternation-free formula ψ' is polynomial in $|\psi'|$ and in $|K|$ and that sub-automata for quantified subformulas are not reachable via actions that are self-loops with conjunctions.

Base case: for atomic propositions and negated atomic propositions, the induction hypothesis is fulfilled.

Induction step: Let $\psi = \exists\pi.\ \psi_1$. Only Until operators and Release operators in the formula lead to nodes that have two transitions, one with a self-loop and one without self-loops. By the restrictions in the definition of the alternation-free fragment, we guarantee that automata of quantified subformulas are not reachable via transitions with self-loops that contain conjunctions.

Conjunctive transitions that are not part of loops or self-loops only lead to a polynomial increase in size during nondeterminization. Emptiness of nondeterministic Büchi automata is in NLOGSPACE [139], so the upper bound of the theorem follows. Since HyperCTL* subsumes LTL, the lower bound for LTL model checking [124] implies the lower bound for HyperCTL*. □

5.3 From Alternation-free Formulas to Full HyperCTL*

The construction from the previous subsection can be extended to full HyperCTL* by adding a construction for *negated* quantified subformulas. We compute an automaton for the complement language, based on the following theorem:

Theorem 5.3.1 ([75])**.** *For every alternating Büchi automaton $\mathcal{A} = (Q, q_0, \Sigma, \rho, F)$, there is an alternating Büchi automaton $\overline{\mathcal{A}}$ with $O(|Q|)^2$ states that accepts the complemented language: $\mathcal{L}_\omega(\overline{\mathcal{A}}) = \overline{\mathcal{L}_\omega(\mathcal{A})}$.*

We extend the previous construction with the following case:

$$\varphi = \neg\exists\pi.\psi_1 \qquad \overline{\mathcal{N}'_{\psi_1}}, \quad \text{where } \mathcal{N}'_{\psi_1} = \mathsf{MH}(\mathcal{A}_{\psi_1}) \text{ via Theorem 5.1.2}$$

We capture the complexity of the resulting model checking algorithm in terms of the alternation depth of the HyperCTL* formula. The formulas with alternation depth 0 are exactly the alternation-free formulas.

Definition 5.3.2. *The* alternation depth *of a HyperCTL* formula in negation normal form is the highest number of alternations from existential to universal and universal to existential quantifiers along any of the paths of the formula's syntax tree. Existential quantifiers in the left subformula of an until operator or in the right subformula of a release operator, and, symmetrically, universal quantifiers in the right subformula of an until operator or in the left subformula of a release operator also count as alternation.*

For example, let ψ be a formula without additional quantifiers, then

- $\exists\pi.\ \psi$ has alternation depth 0,
- $\forall\pi_1.\exists\pi.\ \psi$ has alternation depth 1,
- $\exists\pi.\ \Diamond\exists\pi'.\ \psi$ has alternation depth 0,
- $\exists\pi.\ \Box\exists\pi'.\ \psi$ has alternation depth 1,
- $(\forall\pi.\ \psi) \wedge (\exists\pi.\ \psi)$ has alternation depth 0.

Note that the above definition of alternation depth refines the definition in [30], which would classify the formula $\exists\pi.\ \Diamond\exists\pi'.\ \psi$ to have alternation depth 1. The new definition of alternation depth also allows us to unify the algorithms of HyperCTL* and SecLTL and the analysis of their upper bounds (see Section 5.7).

Let $g_c(k,n)$ be a tower of exponentiations of height k, defined simply as $g_c(0,n) = n$ and $g_c(k,n) = c^{g_c(k-1,n)}$. We define NSPACE$(g(k,n))$ to be the languages that are accepted by a nondeterministic Turing machine that runs in SPACE $O(g_c(k,n))$ for some $c > 1$. For convenience, we define NSPACE$(g(-1,n))$ to be NLOGSPACE.

Proposition 5.3.3. *Let K be a Kripke structure and φ a HyperCTL* formula with alternation depth k. The alternating automaton $\mathcal{A}_\varphi$ resulting from the previous construction has $O(g(k+1,|\varphi|))$ and $O(g(k,|K|))$ states and can be constructed in NSPACE$(g(k,|\varphi|))$ and NSPACE$(g(k-1,|K|))$.*

Proof. We compute the number of states of $\mathcal{A}_\varphi$ by induction on the alternation depth of φ.

- **Base case**: In the base case, φ has alternation depth 0, and φ does not contain quantifier alternations. By induction on the structure of φ the number of states of $\mathcal{A}_\varphi$ is linear in $|\varphi|$ if φ has no quantifiers, and single-exponential if φ contains at least one quantifier. The base case of the atomic propositions and the induction step for the Boolean connectives and temporal operators satisfy the claim. The first occurrence of a quantifier requires a nondeterminization via the Miyano-Hayashi construction, and therefore results in a single-exponential number of states. The interesting case is a quantified formula $\varphi = \exists\pi.\ \psi_1$, where ψ_1 contains again at least one quantified subformula $\exists\pi'.\ \psi_2$, but no such quantified subformula occurs after a negation, in the left subformula of an until, or in the right subformula of a release. The repeated nondeterminization does not cause a further exponential increase in the number of states, because the automaton $\mathcal{A}_{\exists\pi'.\psi_2}$ for the subformula $\exists\pi'.\ \psi_2$ is nondeterministic and forms a closed subautomaton of $\mathcal{A}_{\psi_1}$, i.e. has no transitions to states outside $\mathcal{A}_{\exists\pi'.\psi_2}$. Each state of $\mathcal{A}_{\exists.\pi'.\psi_2}$ therefore occurs at most once in the state sets R, R' of the Miyano-Hayashi construction and the total number of states of $\mathcal{N}'_{\psi_1}$ is linear in the number of states of $\mathcal{A}_{\exists\pi'.\psi_2}$.

- **Inductive step** For the inductive step, we prove, again by structural induction on φ, that the number of states of $\mathcal{A}_\varphi$ is linear in the number of states of $\mathcal{A}_\psi$ for any subformula that does not occur in φ after a negation, in the left subformula of an until, or in the right subformula of a release. As in the base case, the repeated nondeterminization in quantified subformulas of φ that contain ψ does not cause a further exponential increase in the number of states, because the nondeterministic automaton $\mathcal{A}_{\exists\pi'.\psi_2}$ for the subformula $\exists\pi'.\ \psi$ forms a closed subautomaton of $\mathcal{A}_\varphi$, i.e. has no transitions to states outside $\mathcal{A}_{\exists\pi'.\psi_2}$. For subformulas ψ that occur in φ directly after the first negation, as the first left subformula of an until, or as the first right subformula of a release, the number of states of $\mathcal{A}_\varphi$ is single exponential in the number of states of $\mathcal{A}_\psi$ due to the Miyano-Hayashi construction. □

Theorem 5.3.4. *Given a Kripke structure K and a HyperCTL* formula φ with alternation depth k, we can decide whether $K \models \varphi$ in NSPACE$(g(k, |\varphi|))$ and in NSPACE$(g(k-1, |K|))$.*

Proof. Since φ is a Boolean combination of a set X of quantified subformulas, we first check, as discussed in the previous subsection, for each quantified formula $\psi \in X$ whether $K \models \psi$ holds. We then determine, independently of $|K|$ and in linear space in $|\varphi|$, the truth value of the Boolean combination. To determine for a subformula $\psi \in X$ whether $K \models \psi$ holds, we construct the automaton $\mathcal{A}_\psi$ for ψ. Since ψ is of the form $\exists\pi.\psi'$, $\mathcal{A}_\psi$ is a nondeterministic automaton with, by Proposition 5.3.3, $O(g(k, (|\varphi| \cdot \log|K|)))$ states. Since the nonemptiness problem for nondeterministic Büchi automata is in NLOGSPACE [139], the nonemptiness of $\mathcal{A}_\psi$ can be determined in space $O(g(k-1, (|\varphi| \cdot \log|K|)))$. □

Theorem 5.3.4 subsumes the result for the alternation-free fragment:

Corollary 5.3.5. *The model-checking problem of the alternation-free fragment of HyperCTL* is in PSPACE in the size of the formula and in NLOGSPACE in the size of the Kripke structure.*

5.4 Extended Path Quantification

We generalize the quantification over paths of a single Kripke structure to the quantification over paths from multiple Kripke structures. Besides the unification of QPTL with HyperLTL and HyperCTL*, this allows us to simplify the proof of Theorem 5.6.4.

We introduce *extended path quantifiers*, that extend the syntax of the path quantifier by an additional parameter $\exists_K \pi.\ \varphi$ to indicate the Kripke structure from which the new execution path should be chosen. We generalize the Kripke structure in the validity relation $\Pi \models_K \varphi$ to an *environment* Γ that maps names to Kripke structures. Given a Kripke structure environment Γ for which $\Gamma(\kappa)$ is defined, we extend the semantics of HyperLTL for extended path quantifiers as follows:

$$\Pi \models_\Gamma \exists_\kappa \pi.\ \varphi \quad \text{iff} \quad \exists t \in \mathsf{Traces}^*\Big(\Gamma(\kappa), s_0(\Gamma(\kappa))\Big) : \Pi[\pi \mapsto t] \models_\Gamma \varphi\,,$$

and for HyperCTL* we define

$$\Pi \models_\Gamma \exists_\kappa \pi.\ \varphi \quad \text{iff} \quad \exists p \in \mathsf{Paths}^*\Big(\Gamma(\kappa), s_0(\Gamma(\kappa))\Big) : \Pi[\pi \mapsto p, \varepsilon \mapsto p] \models_\Gamma \varphi$$

where $s_0(\Gamma(\kappa))$ is the initial state of $\Gamma(\kappa)$. To resolve the question which Kripke structure the first path starts in, we require that HyperLTL and HyperCTL* formulas with extended path quantifiers start with an extended path quantifier. We say that a Kripke structure environment Γ is valid for a HyperLTL formula or a HyperCTL* formula φ with extended path quantifiers, denoted $\Gamma \models \varphi$, if $\emptyset \models_\Gamma \varphi$.

Trace equivalence. In HyperLTL with extended path quantifiers we can easily express trace-based relations between Kripke structures with a common alphabet AP, such as language inclusion, that is $\mathsf{Traces}(K_1, s_0) \subseteq \mathsf{Traces}(K_2, s_0')$:

$$\forall_{\kappa_1} \pi. \exists_{\kappa_2} \pi'.\ \bigwedge_{a \in AP} a_\pi \leftrightarrow a_{\pi'}$$

and trace equivalence, that is $\mathsf{Traces}(K_1, s_0) = \mathsf{Traces}(K_2, s_0')$:

$$\forall_{\kappa_1} \pi. \exists_{\kappa_2} \pi'.\ \bigwedge_{a \in AP} a_\pi \leftrightarrow a_{\pi'} \quad \wedge \quad \forall_{\kappa_2} \pi. \exists_{\kappa_1} \pi'.\ \bigwedge_{a \in AP} a_\pi \leftrightarrow a_{\pi'}$$

Model checking of concurrent processes. Given n Kripke structures, we can express that their common execution that synchronize on the actions labeled S satisfy an LTL property φ:

$$\forall_{\kappa_1}\pi_1.\ \dots\forall_{\kappa_n}\pi_n.\ \Box \bigwedge_{i,j} S_{\kappa_i} = S_{\kappa_j} \;\Rightarrow\; \varphi$$

Elimination of Extended Path Quantifiers

Model checking HyperLTL and HyperCTL* with extended path quantifiers is not harder than model checking HyperLTL and HyperCTL*.

Theorem 5.4.1. *Let φ be a HyperLTL (HyperCTL*) formula with extended path quantifiers and let Γ be an environment binding the Kripke structure names occurring in φ to the finite Kripke structures $K_1, \dots K_n$. We can construct in LOGSPACE a HyperLTL (HyperCTL*) formula φ'* without *extended path quantifiers and a single Kripke structure K such that $\Gamma \models \varphi$ iff $K \models \varphi'$. Further the size of φ' is in $O(|\varphi|)$ and the size of K is in $O(|K_1| + \cdots + |K_n|)$.*

Proof. Let φ be a HyperLTL or HyperCTL* formula with extended path quantifiers and let Γ be an environment that binds the Kripke structure names occurring in φ. Let $K_1, \dots, K_n$ be the Kripke structures that occur in the formula with the names $\kappa_1, \dots, \kappa_n$. We denote the components of the Kripke structures by indices: $K_i = (S_i, s_{0,i}, \delta_i, \mathsf{AP}_i, L_i)$, and we assume their sets of states to be disjoint. We construct $K = (S, s_0, \delta, \mathsf{AP}, L)$ as follows:

$$\begin{array}{lcl}
S & = & \{s_0\} \uplus \bigcup_i S_i \\
\delta(s_0) & = & \bigcup_i s_{0,i} \\
\delta(s) & = & \delta_i(s) \uplus \bigcup_j \delta_j(s_{0,j}), \text{ for } s \in S_i \\
\mathsf{AP} & = & \{\kappa_1, \dots, \kappa_n\} \uplus \bigcup \mathsf{AP}_i \\
L(s_0) & = & \emptyset \\
L(s) & = & \{\kappa_i\} \uplus L_i(s) \text{ for } s \in S_i
\end{array}$$

We obtain the formula φ' from φ by replacing every extended path quantifier $\exists_{\kappa_i}\pi.\ \psi$ with $\exists\pi.\ \bigcirc\Box\kappa_{i,\pi} \wedge [\psi]_{\pi,s_{0,i}}$, where $[\psi]_{\pi,s_{0,i}}$ is the formula that results when we "unroll" the until and release operators in ψ once and then replace the atomic propositions referring to π in the propositional part. That is, we apply the following rules:

$$\begin{array}{lcll}
[a_{\pi'}]_{\pi,s_{0,i}} & := & a \in L_i(s_{0,i}) & \text{for } \pi = \pi' \\
[a_{\pi'}]_{\pi,s_{0,i}} & := & a_{\pi'} & \text{for } \pi \neq \pi' \\
[\neg\psi']_{\pi,s_{0,i}} & := & \neg[\psi']_{\pi,s_{0,i}} & \\
[\psi_1 \vee \psi_2]_{\pi,s_{0,i}} & := & [\psi_1]_{\pi,s_{0,i}} \vee [\psi_2]_{\pi,s_{0,i}} & \\
[\bigcirc\psi']_{\pi,s_{0,i}} & := & \bigcirc\psi' & \\
[\psi_1 \,\mathcal{U}\, \psi_2]_{\pi,s_{0,i}} & := & [\psi_2]_{\pi,s_{0,i}} \vee [\psi_1]_{\pi,s_{0,i}} \wedge \bigcirc(\psi_1 \,\mathcal{U}\, \psi_2) & \\
[\psi]_{\pi,s_{0,i}} & := & \psi & \text{otherwise}
\end{array}$$

In particular, in case φ was a HyperLTL formula the transformation maintains the number and polarity of the quantifiers. □

Checking whether two Kripke structures are trace equivalent is PSPACE-complete [62, Section 11.3.5]. The encoding of trace equivalence in HyperLTL hence yields a lower bound of the model checking complexity in the size of the Kripke structure.

Corollary 5.4.2. *The model checking problem for HyperLTL is PSPACE-hard in the size of the Kripke structure.*

We will refine this result in Section 5.6 to a non-elementary lower bound in both the size of the formula and the size of the Kripke structure with matching upper bounds.

5.5 Quantification over Propositions

Quantified propositional temporal logic (QPTL) [125] extends LTL with *quantification over propositions*. Quantification over a proposition $\exists a.\varphi$ fixes an interpretation of the proposition *for all points of time*. This enables us to express arbitrary ω-regular properties and it adds a high degree of compactness. While the concepts of quantification are different, they share similarities on the algorithmic level. We show that the satisfiability problem of QPTL can be reduced to the model checking problem of HyperCTL* and thereby obtain a lower bound on the complexity.

QPTL formulas are generated by the following grammar, where $a \in \mathsf{AP}$:

$$\varphi \;::=\; a \mid \neg\varphi \mid \varphi \vee \varphi \mid \bigcirc\varphi \mid \Diamond\varphi \mid \exists a.\, \varphi\,,$$

QPTL formulas are interpreted over traces with all operators inheriting the LTL semantics except $\exists a.\varphi$:

$$t \models \exists a.\, \varphi \quad \text{iff} \quad \exists t' \in (2^{\mathsf{AP}})^{\omega}.\; t =_{\mathsf{AP}\setminus a} t' \;\wedge\; t' \models \varphi\,,$$

where $t =_{\mathsf{AP}\setminus a} t'$ denotes that for all points in time i it holds $(t(i) \setminus a) = (t'(i) \setminus a)$.

The *satisfiability problem of QPTL* is to determine, for a given QPTL formula φ, the existence of a trace $t \in (2^{\mathsf{AP}})^{\omega}$ sucht that $t \models \varphi$. A Kripke structure K satisfies a QPTL formula φ, denoted $K \models \varphi$, if all traces of K from the initial state satisfy the formula. In particular, QPTL expresses trace properties, which gives us the following theorem:

Theorem 5.5.1. *QPTL does not subsume HyperLTL.*

Also the converse statement is not true. Adding quantification over propositions to LTL extends the expressiveness from the ω-star-free languages to the ω-regular languages [65, 79, 131, 125, 132]. For example, the ω-regular trace property over $\Sigma = \{a, b\}$ that every second element is b, which is $(ab + bb)^{\omega}$ in ω-regular expression syntax, is not ω-star-free.

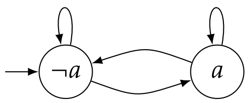
Theorem 5.5.2. *HyperCTL* does not subsume QPTL.*

Proof. Consider the class of linear Kripke structures. A Kripke structure is *linear*, if all states have exactly one successor. In particular, properties of linear Kripke structures can be seen word languages. Every HyperCTL* formula is equivalent to its LTL interpretation on this class of Kripke structures. We obtain the *LTL interpretation* of a HyperCTL* formula, by syntactically removing all quantifiers and all path variables from the atomic propositions. The equivalence stems from the fact that quantifiers always bind a unique path in linear Kripke structures. QPTL can express any ω-regular language [125], while LTL, and thus HyperCTL* over linear Kripke structures, can only express the ω-star-free languages. □

Hence, quantification over propositions and quantification over paths are orthogonal extensions to temporal logics. In the first publications on HyperLTL and HyperCTL* [30], we presented a slightly different semantics for HyperLTL and HyperCTL* that naturally unifies the two concepts. The semantics allowed us to refer to propositions that not part of the Kripke structure that were then interpreted in the same was as quantified propositions, which lead to the result that HyperCTL* subsumes QPTL. In the subsequent works [47, 48] and in this thesis, however, we dropped this feature for the sake of simplicity. The only result that is affected by this change is the ability to express ω-regular languages and the inclusion of epistemic temporal logics. We show how the more elegant unification in the extended path quantification semantics enables both results again.

Encoding QPTL via Extended Path Quantification

Quantification over propositions can now be easily encoded as the quantification over the paths in a fixed Kripke structure $K_a = (\{s_0, s_1\}, s_0, \delta, \{a\}, L)$ with $\delta(s_i) = \{s_0, s_1\}$ for all $i \in \{0,1\}$, and $L(s_0) = \emptyset$ and $L(s_1) = \{a\}$:



To encode the satisfiability problem of a QPTL formula we additionally consider the trivial Kripke structure K_{AP} generating all traces $(2^{\mathsf{AP}})^\omega$. Given a QPTL formula φ we first transform it into prenex normal form [125], prepend a universal path quantifier $\forall_{\kappa_{\mathsf{AP}}}\pi$, we index all atomic propositions that are not bound by a quantifier by π and then we replace every quantifier over propositions $\exists a$ ($\forall a$) by an extended path quantifier $\exists_{\kappa_a}\pi'$ ($\forall_{\kappa_a}\pi'$), where π' is a fresh path variable, and replace every atomic proposition a bound by that quantifier by $\bigcirc a'_\pi$. Checking the satisfiability of φ is equivalent to checking the resulting HyperLTL formula with extended path quantification over the Kripke structures K_{AP} and K_a.

Theorem 5.5.3. *For every QPTL formula φ there is a HyperLTL formula φ' with extended path quantifiers such that $\exists t.\ t \models \varphi$ iff $\{\kappa_a \mapsto K_a, \kappa_{\mathsf{AP}} \mapsto K_{\mathsf{AP}}\} \models \varphi'$.*

EQCTL* and QCTL*. The logic EQCTL* [74, 76] extends CTL* by the *existential* quantification over propositions. QCTL* [52] additionally allows for negated existential (i.e. universal) quantifiers. Like the satisfiability problem of HyperLTL, the satisfiability of QCTL* is undecidable. The proof, however, relies on a different technique [52]. To the best of the author's knowledge, no model checking algorithm has been proposed for QCTL*, but it is straightforward to derive such an algorithm via HyperCTL* with extended path quantifiers, using a similar encoding as for QPTL. The model checking problem for QCTL, a restricted version of QCTL* that builds on CTL instead of CTL*, allows for a comparably low worst-case complexity of the model checking problem [108]. It is open how these results relate to the model checking problems of HyperLTL and HyperCTL*.

5.6 Lower Bounds

In this section, we give lower bounds to the runtime and space requirements of the algorithm that match the upper bounds. We characterize the complexity both in the size of the formula and in the size of the Kripke structure.

The lower bound in the size of the formula is given by the encoding of QPTL into HyperLTL given in Section 5.5. By combining Theorem 5.4.1 and Theorem 5.5.3 we obtain a lower bound in the size of the formula. As the transformations in the proofs of the theorems maintain the number of quantifier alternations, we can even state the stronger result that the complexity grows with every quantifier alternation.

Theorem 5.6.1. *The model checking problem for HyperCTL* with quantifier alternation depth k is hard for NSPACE($g(k, |\varphi|)$).*

The lower bound in the size of the Kripke structure is of particular importance for this work. Not only are Kripke structures often much larger than the formulas, lower bounds in the complexity also provide an interesting argument for the comparison of two specification logics. Consider two logics $\mathcal{L}$ and $\mathcal{L}'$ with model checking problems that are complete for two complexity classes in the size of the Kripke structure to check $\mathcal{C}$ and $\mathcal{C}'$ with $\mathcal{C} \lneq \mathcal{C}'$. Then there are properties expressible in $\mathcal{L}'$ that characterize a language, i.e. a class of Kripke structures, that is only checkable in $\mathcal{C}'$, but not in C. Hence $\mathcal{L}'$ cannot be contained in $\mathcal{L}$.

We now establish that the complexity in the size of the Kripke structure also grows by one exponent for each additional quantifier alternation in the formula. The proof builds on a technique developed by Stockmeyer [127] and its adaption to the proof of the computational hardness of QPTL [125]. Our proof extends the technique by making the formula independent of the size of the input to the Turing machine. We also simplify the encoding compared to the encoding in QPTL [125], by making use of the Until operator, which is not available in QPTL.

We start with expressing yardsticks in HyperLTL. A *yardstick* is a subformula that defines a distance of length $h_c(k,n)$ with

$$h_c(0,n) = n \quad \text{and} \quad h_c(k,n) = h_c(k,n-1) \cdot c^{h_c(k,n-1)}$$

In particular we will use yardsticks to define the length of the tape available to a Turing machine. Clearly it holds $h_c(k,n) \geq g_c(k,n)$.

Lemma 5.6.2. *Let $c \in \mathbb{R}$, let x and y be atomic propositions, and let π be a trace variable. For all $k \in \mathbb{N}$ there is a HyperLTL subformula $\varphi_{c,k}(x_\pi, y_{\pi'})$ with extended path quantifiers and alternation depth k that has an unbound trace variable π, such that for all $n \in \mathbb{N}_{\geq 1}$ there is a Kripke structure $K(n)$, such that $\Pi \models_\Gamma \varphi_{c,k}(x_\pi, y_{\pi'})$, where $\Pi = \{\pi \mapsto t, \pi' \mapsto t'\}$ and $\Gamma = \{\kappa \mapsto K(n),\ \kappa_a \mapsto K_a\}$, holds iff*

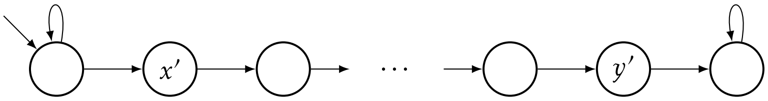
1. *x occurs exactly once on t, and*
2. *y occurs exactly once on t', and*
3. *y occurs exactly on t' exactly $h_c(k,n)$ steps after x occurs on t, that is $x \in t(i)$ iff $y \in t'(i + h_c(k,n))$.*

The length of φ is in $O(c \cdot k)$ and the size of K is in $O(n)$. Further, the Kripke structure $K(n)$ is constructible in space $O(\log n)$.

Proof. We prove the statement by induction over k. For $k = 0$, we choose the following HyperLTL subformula with alternation depth 0:

$$\varphi_{c,k}(x_\pi, y_{\pi'}) \ := \ \exists_\kappa \pi''.\ \Diamond x'_{\pi''} \ \wedge \ \Box(x_\pi \leftrightarrow \bigcirc x'_{\pi''} \ \wedge \ y_{\pi'} \leftrightarrow \bigcirc y'_{\pi''})$$

and we choose $K(n)$ as follows:



where the states labeled x' and y' are exactly n steps apart. As our Kripke structures are defined to have a single initial state, we are forced to use the Next operator in front of the atomic propositions referring to π'' in order to leave both options for x on path π in the first position. Note that $K(n)$ can be constructed in $O(\log n)$ space and that the formula is independent of n.

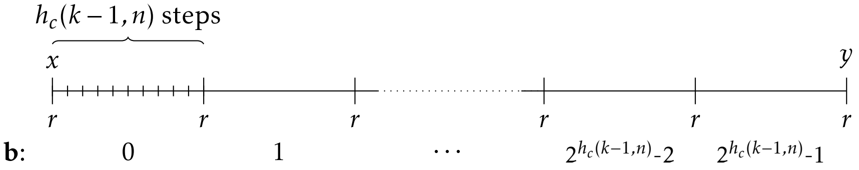
Induction step ($k > 0$). To create the yardstick of length $h_c(k,n) = h_c(k-1,n) \cdot c^{h_c(k-1,n)}$ from the yardstick of length $h_c(k-1,n)$, we encode a counter with $h_c(k-1,n) \cdot \lceil \log_2 c \rceil$ bits. The bits of the counter are represented as *quantified propositions*, as in the logic QPTL [125]. We have seen in Subsection 5.5 that quantification over propositions can be encoded into HyperLTL with extended path quantifiers. In the following, we thus use quantifiers over propositions, e.g. $\exists b.\ \psi(b)$, as syntactic sugar for extended trace quantifiers

$\exists_{K_a}\pi_b.\ \psi(\bigcirc a_{\pi_b})$, with a fresh trace variable π_b, interpreted over the Kripke structure K_a.

One valuation of the counter will then be spread over $h_c(k-1,n)$ positions and thus we need to introduce $\lceil \log_2 c \rceil$ atomic propositions $b_1, \ldots, b_{\lceil \log_2 c \rceil}$ via quantifiers over propositions. For simplicity we restrict the discussion to $c = 2$, such that we can represent the counter with (consecutive valuations of) a single proposition b. The extension of the formula below to arbitrary c is straight-forward. We use little endian encoding for the valuations of the counter, i.e. the least significant bit occurs first on the trace.

We introduce an additional quantified proposition r that helps us to separate the counter valuations. Every occurrence of r indicates that a new valuation of the counter starts (the least significant bit).

The figure below depicts the yardstick of level k. The values **b** indicate the valuation of the counter.



We construct $\varphi_{c,k}(x_\pi, y_{\pi'})$ as follows:

$$
\begin{aligned}
&\exists b.\exists r.\forall x'.\forall y'.\ \varphi_{c,k-2}(x', y') \ \Rightarrow \\
&\qquad (\neg x_\pi\, \mathcal{U}\, (x_\pi \wedge \bigcirc\Box\neg x_\pi)) \wedge (\neg y_{\pi'}\, \mathcal{U}\, (y_{\pi'} \wedge \bigcirc\Box\neg y_{\pi'})) \wedge && (1)\\
&\qquad \Box(x_\pi \vee y_{\pi'} \Rightarrow r) \wedge && (2)\\
&\qquad \Box(x' \Rightarrow (\neg y' \wedge \neg r)\, \mathcal{U}\, (r \wedge \neg y' \wedge \bigcirc(\neg r\, \mathcal{U}\, y'))) \wedge && (3)\\
&\qquad \Box(x_\pi \Rightarrow (\neg b\, \mathcal{U}\, (\neg b \wedge \bigcirc r))) \wedge && (4)\\
&\qquad \Box(\bigcirc(\neg r\ \mathcal{U}\, y_{\pi'}) \Rightarrow b) \wedge && (5)\\
&\qquad \Box(r \wedge \neg x_\pi \ \Rightarrow\ (\bigcirc\neg r)\, \mathcal{U}\, b) \wedge && (6)\\
&\qquad \Big((\Box(x' \Rightarrow b) \leftrightarrow \Box(y' \Rightarrow b)) \leftrightarrow \Box(r \Rightarrow \neg x'\, \mathcal{U}\, (\neg x' \wedge (\neg b \vee \bigcirc r)))\Big) && (7)
\end{aligned}
$$

The subformulas (1) to (7) express the requirements of $\varphi_{c,k}(x, y)$ as follows:

(1) x_π and $y_{\pi'}$ occur exactly once.

(2) r is aligned with x_π and with $y_{\pi'}$.

(3) r occurs exactly once between x' and y'. As x' and y' are universally quantified and are exactly $h_c(k-1,n)$ steps apart, r must occur at exactly every $h_c(k-1,n)$ steps.

(4) The counter must be initialized with 0. That is, the first $h_c(k-1,n)$ occurrences of b starting from the point where x_π holds, must be 0.

(5) The last counter value must be $2^{h_c(k-1,n)} - 1$, as we assumed $c = 2$.

(6) The counter is never 0 except for the first valuation.

(7) Consecutive counter valuations must increment by one. Given consecutive counter valuations $\alpha = \alpha_0, \alpha_1, \ldots, \alpha_{h_c(k-1,n)}$ and $\beta = \beta_0, \beta_1, \ldots, \beta_{h_c(k-1,n)}$, where α_0 and β_0 are the least significant bits, we know that $\beta = \alpha + 1 \mod 2^{h_c(k-1,n)}$, iff for all i:

$$\alpha_i = \beta_i \text{ iff } \exists j < i \text{ such that } \alpha_j = 0\,.$$

Like subformula (3), subformula (7) relies on the yardstick of level $k-1$. The sequences over propositions x' and y', such that each of them occurs only once and y' occurs exactly $h_c(k-1,n)$ steps after x', indicate two corresponding positions in consecutive counter valuations. If, and only if, they are the same, we require that there must be a less significant bit b in the first counter valuation, i.e. at a position before x', that is 0.

Formally, the formula $\varphi_{c,k}(x,y)$ is a HyperCTL* formula and not yet in HyperLTL, as the subformula $\varphi_{c,k-1}(x',y')$ contains quantifiers. Through Theorem 4.2.3 we obtain an equivalent HyperLTL formula. As the subformula $\varphi_{c,k-1}(x',y')$ starts with an existential quantifier and occurs negated, the number of quantifier alternations grows (just) by one with every new level of the yardstick.

The formula is thus independent of n, but it uses extended path quantifiers over the Kripke structures K_a and $K(n)$. By Theorem 5.4.1 the model checking problem with extended path quantifiers can be reduced in LOGSPACE to checking a single Kripke structure of size $|K_a| + |K(n)|$. □

Lemma 5.6.3. *Given $k \in \mathbb{N}$, every language in NSPACE($g(k-1,n)$) is LOGSPACE reducible to the model checking problem of a fixed HyperLTL formula with quantifier alternation depth k.*

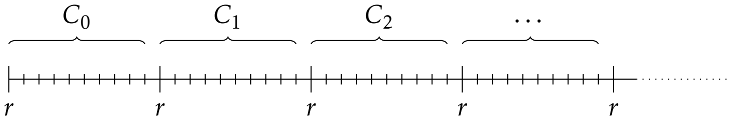
Proof. For $k = 0$ the statement follows from Theorem 5.2.3. For the case $k > 0$ we encode a $h_c(k-1,n)$-space bounded Turing machine M into a HyperLTL formula $\psi(M)$ of alternation depth k. The input word I to the Turing machine will be encoded as a Kripke structures $K(I)$ of linear size in the input. If and only if the Turing machine terminates on (accepts) the input, the HyperLTL formula holds on the Kripke structure.

A *Turing machine* $M = (Q, \Sigma, \delta, q_0, q_A)$ consists of a finite set of states Q, an alphabet Σ, a transition relation $\delta : Q \times \Sigma \to 2^{Q \times \Sigma \times \{\text{left},\text{right}\}}$, an initial state q_0, and an accepting state q_A. A Turing machine operates with a read/write-head on a tape of cells that is infinite in one direction, say to the right, and each cell of the tape contains a single symbol from Σ. Initially the Turing machine starts in the initial state q_0 with the head on the leftmost cell, and the input of length n is given as the sequence of symbols in the first n cells. All other cells initially contain the blank symbol $\# \in \Sigma$. In each step, the Turing machine changes its state, writes to the current cell, and moves the head left or right on the tape as indicated by the transition relation. For the current state q and the content σ of the cell at the current head position, the Turing machine nondeterministically picks one of the successor states q', a symbol

$\sigma' \in \Sigma$ to write, and moves the head in the direction dir $\in$ {left, right} such that $(q', \sigma', \mathrm{dir}) \in \delta(q, \sigma)$. A *configuration* of a Turing machine consists of the current state, the head position, and the tape. We say that a Turing machine *accepts* an input, if there is a finite sequence of configurations $C_0, C_1, \dots, C_m$ that satisfies the step relation of the Turing machine and ends in the accepting state q_A.

An $h_c(k-1,n)$*-space bounded* Turing machine is restricted to a tape with $h_c(k-1,n)$ cells. We represent configurations of an $h_c(k-1,n)$-space bounded Turing machine as sequences of length $h_c(k-1,n)$ over the alphabet $\Sigma \cup (\Sigma \times Q)$, where exactly one position is from $\Sigma \times Q$ and all other positions are from Σ. Given some input word $i_1, \dots, i_n$, the initial configuration C_0 is thus the sequence $(q_0, i_1), i_1, \dots, i_n$ followed by $h_c(k-1,n) - n$ occurrences of #.

We encode a configuration of an $h_c(k-1,n)$-space bounded Turing machine as $h_c(k-1,n)$ consecutive valuations of (quantified) propositions $P = \{\sigma_P \mid \sigma \in \Sigma\} \cup \{q_P \mid q \in Q\}$ that include one proposition for every state in Q and for every alphabet symbol. The computation of the Turing machine is then encoded as a sequence of configurations. Similar to the encoding of the yardstick $\varphi_{c,k}(x,y)$, we use an additional proposition $r \in \mathsf{AP}$ to indicate the beginning of each configuration.



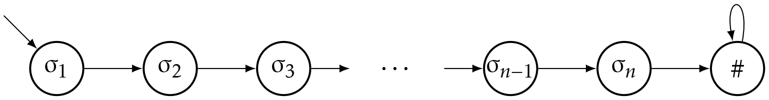
Given an $h_c(k-1,n)$-space bounded Turing machine $M = (Q, \Sigma, \delta, q_0, q_A)$, we choose the formula $\psi_{c,k}(M)$ as follows:

$$\begin{aligned}
&\exists P.\exists_{K(I)}\pi_I.\exists r.\forall x.\forall y.\ \varphi_{c,k-1}(x,y) \Rightarrow \\
&\quad r \wedge \Box(x \Rightarrow (\neg y \wedge \neg r)\ \mathcal{U}\ (r \wedge \neg y \wedge \bigcirc(\neg r\, \mathcal{U}\, y))) \wedge && (1)\\
&\quad \textstyle\bigwedge_{\sigma\in\Sigma}(\sigma_P = \sigma_{\pi_I} \wedge \bigcirc\neg r)\ \mathcal{U}\ (\sigma_P = \sigma_{\pi_I} \wedge \bigcirc r) \wedge && (2)\\
&\quad q_{0,P} \wedge \textstyle\bigwedge_{q\in Q\setminus q_0}\neg q_P \wedge \bigcirc((\bigwedge_{q\in Q}\neg q_P)\,\mathcal{U}\, r) \wedge && (3)\\
&\quad \Diamond q_{A,P} \wedge && (4)\\
&\quad \textstyle\bigwedge_{p_1,p_2,p_3\in 2^P} \Box\big(x \Rightarrow p_1 \wedge \bigcirc(p_2 \wedge \neg r) \wedge \bigcirc\bigcirc(p_3 \wedge \neg r)\big) \Rightarrow && (5.1)\\
&\qquad\qquad \textstyle\bigvee_{(p_1',p_2',p_3')\in\Delta(p_1,p_2,p_3)} \Box\big(y \Rightarrow p_1' \wedge \bigcirc p_2' \wedge \bigcirc\bigcirc p_3'\big) && (5.2)
\end{aligned}$$

The subformulas describe an accepting computation of M as follows:

(1) Proposition r occurs in the first state and then every $h_c(k-1,n)$ steps.

(2) The tape of the initial configuration is equal to the first $h_c(k-1,n)$ labels of the unique trace of the Kripke structure $K(I)$, which for a given input $I = \sigma_1, \dots, \sigma_n$ is defined as follows:



(3) The Turing machine starts in the initial state q_A. All other tape positions do not contain labels for the state of the Turing machine. That is, the head position is indicated by the position of the state relative to the tape.

(4) The Turing machine reaches an accepting configuration.

(5) Consecutive configurations must follow the step relation of the Turing machine. As the Turing machine can move its head only by one position in every step, all but three consecutive tape cells can potentially be affected: the current cell, and the left or the right cell (to which the head is moved). This allows us to characterize the step relation δ as a relation Δ over triples of tape cells [127]: A triple $(p'_1, p'_2, p'_3) \in \Sigma \cup (\Sigma \times Q)$ is in $\Delta(p_1, p_2, p_3)$ iff there is a configuration C and a successor configuration C', such that p_1, p_2, and p_3 form consecutive symbols in the configuration C starting at position i and p'_1, p'_2, and p'_3 form consecutive symbols in the successor configuration C' also starting at position i.

In (5.1) we assume that the p_1, p_2, and p_3 form consecutive (sets of) symbols starting at a position marked with x, which is required to be at least two states before the start of a new configuration (indicated by r). Subformula (5.2) asserts that one of the successor triples $(p'_1 a, p'_2, p'_3) \in \Delta(p_1, p_2, p_3)$ occurs at the corresponding position marked with y in the consecutive configuration.

By Lemma 5.6.2 the subformula $\varphi_{c,k-1}(x,y)$ has alternation depth $k-1$ and starts with an existential quantifier. Since $\varphi_{c,k-1}(x,y)$ occurs under one negation in a universally quantified subformula, the formula $\psi_{c,k}(M)$ has quantifier alternation depth k. We eliminate the extended path quantifiers by Theorem 5.4.1 and bring the formula $\psi_{c,k}(M)$ into prenex form by Theorem 4.2.3. This maintains the quantifier alternation depth of k.

The family of Kripke structures for which the model checking problem of the formula $\psi_{c,k}(M)$ represents the halting problem for M on input I of length $|I| = n$ results from K_a, $K(n)$, and $K(I)$ through the construction in Theorem 5.4.1. □

Together with the algorithm presented in Section 5.3, we obtain a precise characterization of the complexity in the size of the Kripke structure to check.

Theorem 5.6.4. *The model checking problem for HyperLTL and HyperCTL* formulas with alternation depth k is NSPACE($g(k-1,n)$)-complete in the size of the Kripke structure.*

In particular this means that we can express strictly more properties with every new quantifier alternation.

Corollary 5.6.5. *The hierarchy of fragments with fixed alternation depths is strict.*

5.7 Efficient Fragments

Though the model checking problems of HyperLTL and HyperCTL* is non-elementary in general, it is may be feasible for many applications, as many properties fall into the alternation-free fragment. For these encodings, the model checking problem is only NLOGSPACE in the size of the Kripke structure. In fact, we demonstrate in the next and final chapter that the alternation-free fragment of HyperCTL* can be efficiently checked in practice.

In this section we show that beyond the quantifier alternation depth there are other syntactic fragments that limit the complexity. We show that the model checking problem of SecLTL is in PSPACE, both in the size of the formula and the size of the Kripke structure. As a second example, we consider CTL*, whose model checking problem is not harder than LTL model checking, despite the presence of quantifier alternations. We observe that CTL* formulas can be seen as HyperCTL* formulas in which the quantifiers are restricted to be closed subformulas and derive a criterion that can be used to significantly speed up the algorithm.

SecLTL

We defined SecLTL as a fragment of HyperCTL* that uses only two path quantifiers and at most a single quantifier alternation. The model checking problem of SecLTL is hence in EXPSPACE in the size of the formula and PSPACE in the size of the Kripke structure. The bounds on the space requirements in the size of the formula can be improved to PSPACE, when we consider that the inner path quantifiers in SecLTL formulas (for the "alternative paths") have a fixed subformula up to renaming. This matches the lower bound that we know through the subsumption of LTL in SecLTL.

Theorem 5.7.1. *The model checking problem for SecLTL is in PSPACE, both in the size of the formula and in the size of the Kripke structure.*

Corollary 5.7.2. *The inclusion of SecLTL in HyperCTL* is strict.*

When, however, is it possible to check SecLTL formulas efficiently in the size of the Kripke structure? We use the definition of alternation-free HyperCTL* formulas to derive a SecLTL fragment. Hide operators are the only operator to introduce additional paths in SecLTL, and they only introduce universally quantified formulas. Negated hide operators necessarily lead to formulas that fall outside the alternation-free fragment. If we restrict the Hide operator to occur only under an even number of negations and that, in the negation normal form of a formula, occur only on the left side of Until operators and on the right side of Release operators, we obtain the efficiently checkable fragment called *restricted SecLTL*. It is easy to see that a SecLTL formula is in the restricted fragment, iff it has alternation depth 0.

Corollary 5.7.3. *The model checking problem $K \models \varphi$ for formulas φ in restricted SecLTL is in PSPACE in both $|\varphi|$ and in NLOGSPACE in $|K|$.*

CTL* and Closed Subformulas

Common algorithms for CTL* use the fact that the evaluation of CTL* state formulas, including the quantifiers A and E, only depends on the current state. Subformulas ψ that are state formulas can thus be eliminated by annotating all states s of the Kripke structure to check with the truth value of $s \models \psi$ and replacing ψ by a fresh atomic proposition indicating the annotated value. Thereby the CTL* model checking problem is reduced to a sequence of LTL model checking problems.

We observe that, in the automaton construction for HyperCTL*, this principle can be applied to *closed HyperCTL* subformulas*, which include all CTL* state formulas. Closed HyperCTL* subformulas depend only on the current state of the most recently quantified path and can therefore be eliminated in the same way as CTL* state formulas. In particular, this provides a means to avoid the expensive complementation operation for quantifier alternations where the inner quantifier is a closed subformula.

Chapter 6

Symbolic Verification and Case Studies

In this chapter we demonstrate that the automaton-based construction from Chapter 5 for alternation-free formulas translates into a practical verification approach. The key to practical verification is to avoid listing the states of the Kripke structure and the nondeterministic Büchi automata explicitly, but instead work with their symbolic (succinct) representation. Popular symbolic representations of state spaces are binary decision diagrams [90] and boolean logic [16]. Even though the worst-case complexities still hold for *symbolic model checking algorithms*, they can often avoid the exploration of all states and thereby enable the verification of systems that could not be analyzed by explicit-state methods. Today a variety of different methods are available, each with different strengths and weaknesses. SAT-based bounded model checking [16], interpolation [91], and IC3 [20].

While model checking of software remains a challenging problem, the development of hardware components, such as micro processors, heavily relies on model checking already. In the following, we discuss an approach to leverage existing hardware model checking technology for the verification of circuits for alternation-free HyperCTL* formulas (Section 6.1). Along several case studies we demonstrate the feasibility of the approach for industrial-size hardware components (Section 6.2).

6.1 Symbolic Model Checking of Circuits

In this section we translate the automaton-based construction for alternation-free formulas from Section 5.2 into a practical verification algorithm for circuits. Given a circuit C and an alternation-free formula φ the algorithm produces a new circuit C_φ that is linear in the size of C and also linear in the size of φ. The compactness of the encoding builds on the ability of circuits to describe systems of exponential size with a linear number of binary variables. The circuit C_φ is then checked for fair reachability to determine the validity of $C \models \varphi$. This check can be done with of-the-shelf model checkers leveraging modern hardware verification technology [23, 16, 21].

A *circuit*[1] $C = (X,\ \mathit{init},\ I,\ O,\ T)$ consists of a set X of binary variables (latches with unit delay), a condition $\mathit{init} \in \mathbb{B}(X)$ characterizing a non-empty set of initial states of X, a set of input variables I, a set of output variables O, and a transition relation $T \in \mathbb{B}(X \times I \times O \times X)$. We require that T is input-enabled and input-deterministic. That is, for all $x \subseteq X$, $i \subseteq I$, there is exactly one $o \subseteq O$ and one $x' \subseteq X$ such that $T(x, i, o, x')$ holds. We denote a subset of X as a *state* of circuit C, indicating exactly those latches that are set to 1. The size of a circuit C, denoted $|C|$, is defined as the number of latches $|X|$.

A circuit C can be interpreted as a Kripke structure K_C of potentially exponential but finite size. The state space of K_C is $S = s_0\ \cup\ 2^X \times 2^I \times 2^O \times 2^X$, where s_0 is a fresh initial state. The transition relation distinguishes the initial step of the computation: $s' \in \delta(s_0)$ iff there is a circuit state $x \subseteq X$ with $\mathit{init}(x)$ and $x = s'|_X$ such that $T(x, s'|_I, s'|_O, s'|_X)$, where $s'|_I$, $s'|_O$, $s'|_X$, and $s'|_{X'}$ are the projections to variables I, O, the first copy of X, and the second copy of X respectively. For subsequent steps of computation we define $s' \in \delta(s)$ whenever $T(s|_X, s'|_I, s'|_O, s'|_{X'})$ and $s|_{X'} = s'|_X$. That is, the first copy of the state latches X denotes the *previous* state, whereas the second copy X' of the state latches denotes the *current* state. The labelling function of K_C maps each state s to the set $s|_X \uplus s|_I \uplus s|_O$. That is, the alphabet AP_{K_C} is $I \uplus O \uplus X$. The semantics of HyperCTL* on a circuit C is defined using the associated Kripke structure K_C. We write $C \models \varphi$ whenever $K_C \models \varphi'$, where φ' is obtained by replacing all atomic propositions a_π by $\bigcirc a_\pi$. This leads to a natural semantics on circuits: the atomic propositions always refer to the *current* value of the latches, the *next* input, and the *next* output.

Given a circuit C and an alternation-free HyperCTL* formula φ, we reduce the model checking problem $C \models \varphi$ to finding a computation path in a circuit C_φ that does not visit a bad state and satisfies a conjunction of strong fairness (or compassion) constraints $F = \{f_1, \ldots, f_k\}$. A strong fairness constraint f of a circuit consists of a tuple (a_1, a_2) of atomic propositions and a path p satisfies f, if a_1 holds only finitely often or a_2 holds infinitely often on p. We build C_φ bottom up following the formula structure. Without loss of generality, we assume that φ contains only existential quantifiers and is in negation normal form. Let ψ be a subformula of φ that occurs under n quantifiers. Let ψ be a subformula of φ that occurs under n quantifiers. Let $C_{\psi_1} = (X_{\psi_1}, \mathit{init}_{\psi_1}, I_{\psi_1}, O_{\psi_1}, T_{\psi_1})$, $C_{\psi_2} = (X_{\psi_2}, \mathit{init}_{\psi_2}, I_{\psi_2}, O_{\psi_2}, T_{\psi_2})$ be the circuits, and let F_{ψ_1} and F_{ψ_2} be the fairness constraints for the subformulas ψ_1 and ψ_2. For LTL operators, the construction resembles the standard translation from

[1] Our definition of circuits can be considered as a model of and-inverter graphs in the Aiger standard [17], where the gate list is abstracted to a transition relation.

LTL to circuits [69, 26]. We construct C_ψ and F_ψ as follows:

$$\begin{array}{ll}
\psi = a_{\pi_k} & C_\psi = (\emptyset,\ \mathsf{true},\ I_\psi,\ \{o_\psi\},\ o_\psi \leftrightarrow a_{\pi_k}),\\[4pt]
 & F_\psi = \emptyset\\[8pt]
\psi = \neg a_{\pi_k} & C_\psi = (\emptyset,\ \mathsf{true},\ I_\psi,\ \{o_\psi\},\ o_\psi \ \mathsf{xor}\ a_{\pi_k}),\\[4pt]
 & F_\psi = \emptyset\\[8pt]
\psi = \psi_1 \vee \psi_2 & C_\psi = (X_{\psi_1} \uplus X_{\psi_2},\ \mathit{init}_{\psi_1} \wedge \mathit{init}_{\psi_2},\\
 & \qquad\quad I_{\psi_1} \cup I_{\psi_2},\ O_{\psi_1} \uplus O_{\psi_2} \uplus \{o_\psi\},\\
 & \qquad\quad (o_\psi \leftrightarrow (o_{\psi_1} \vee o_{\psi_2})) \wedge T_{\psi_1} \wedge T_{\psi_2}),\\[4pt]
 & F_\psi = F_{\psi_1} \cup F_{\psi_2}\\[8pt]
\psi = \psi_1 \wedge \psi_2 & C_\psi = (X_{\psi_1} \uplus X_{\psi_2},\ \mathit{init}_{\psi_1} \wedge \mathit{init}_{\psi_2},\\
 & \qquad\quad I_{\psi_1} \cup I_{\psi_2},\ O_{\psi_1} \uplus O_{\psi_2} \uplus \{o_\psi\},\\
 & \qquad\quad (o_\psi \leftrightarrow o_{\psi_1} \wedge o_{\psi_2}) \wedge T_{\psi_1} \wedge T_{\psi_2}),\\[4pt]
 & F_\psi = F_{\psi_1} \cup F_{\psi_2}\\[8pt]
\psi = \bigcirc \psi_1 & C_\psi = (X_{\psi_1} \uplus \{x_\psi\},\ \mathit{init}_{\psi_1}, I_{\psi_1} \uplus \{i_\psi\},\ O_{\psi_1} \uplus \{o_\psi, b_\psi\},\\
 & \qquad\quad T_{\psi_1} \wedge (o_\psi \leftrightarrow i_\psi) \wedge (x'_\psi \leftrightarrow i_\psi) \wedge (\neg b_\psi \leftrightarrow (o_{\psi_1} \leftrightarrow x_\psi))),\\[4pt]
 & F_\psi = F_{\psi_1}\\[8pt]
\psi = \psi_1\, \mathcal{U}\, \psi_2 & C_\psi = (X_{\psi_1} \uplus X_{\psi_2} \uplus \{x_\psi\},\ \mathit{init}_{\psi_1} \wedge \mathit{init}_{\psi_2},\\
 & \qquad\quad I_{\psi_1} \cup I_{\psi_2} \uplus \{i_\psi\},\ O_{\psi_1} \uplus O_{\psi_2} \uplus \{o_\psi, b_\psi\},\\
 & \qquad\quad T_{\psi_1} \wedge T_{\psi_2} \wedge (o_\psi \leftrightarrow x_\psi) \wedge (x'_\psi \leftrightarrow i_\psi) \wedge\\
 & \qquad\quad (\neg b_\psi \leftrightarrow (\neg x_\psi \vee o_{\psi_2} \vee o_{\psi_1} \wedge x'_\psi))),\\[4pt]
 & F_\psi = F_{\psi_1} \cup F_{\psi_2} \cup \{(x_\psi, o_{\psi_2})\}\\[8pt]
\psi = \psi_1\, \mathcal{R}\, \psi_2 & C_\psi = (X_{\psi_1} \uplus X_{\psi_2} \uplus \{x_\psi\},\ \mathit{init}_{\psi_1} \wedge \mathit{init}_{\psi_2},\\
 & \qquad\quad I_{\psi_1} \cup I_{\psi_2} \uplus \{i_\psi\},\ O_{\psi_1} \uplus O_{\psi_2} \uplus \{o_\psi, b_\psi\},\\
 & \qquad\quad T_{\psi_1} \wedge T_{\psi_2} \wedge (o_\psi \leftrightarrow x_\psi) \wedge (x'_\psi \leftrightarrow i_\psi) \wedge\\
 & \qquad\quad (\neg b_\psi \leftrightarrow (\neg x_\psi \vee o_{\psi_1} \wedge o_{\psi_2} \vee o_{\psi_2} \wedge x'_\psi))),\\[4pt]
 & F_\psi = F_{\psi_1} \cup F_{\psi_2}\\[8pt]
\psi = \exists \pi.\ \psi_1 & C_\psi = (X_{\psi_1} \uplus X_n,\ \mathit{init}_{\psi_1} \wedge (n = 1 \Rightarrow \mathit{init}(X_n)),\\
 & \qquad\quad I_{\psi_1} \setminus X_n,\ (O_{\psi_1} \setminus O_n) \uplus \{o_\psi\},\\
 & \qquad\quad T_{\psi_1} \wedge T(X_n) \wedge (\neg b_\psi \leftrightarrow ((o_\psi \leftrightarrow o_{\psi_1}) \wedge (X_n = X_{n-1})))),\\[4pt]
 & F_\psi = F_{\psi_1}
\end{array}$$

Here $I_\psi = \bigcup_{i \leq n} I_i \uplus O_i \uplus X_i$; $\mathit{init}(X_n)$ is the initial condition applied to copy X_n of the latches; and likewise $T(X_n)$ is the transition relation of C applied to the copy X_n. We use $X_n = X_{n-1}$ to denote the expression that all latches in X_n are equal to their counterparts in X_{n-1} (at the current point of time). It is easy to verify that the transition relation is input-enabled and input-deterministic.

Proposition 6.1.1. *Given a circuit C and an alternation-free formula φ with k quantifiers, the size of the circuit C_φ is at most $|C| \cdot k + |\varphi|$.*

For each subformula ψ of φ, the output o_ψ in the circuit C_φ indicates that ψ must hold for the current computation path, and the latch x_ψ represent the requirements on the future of the computation that arise from the output o_ψ. The output b_ψ indicates that the requirements for subformula ψ are violated and a *bad state* is entered. We thus say that a computation of such a circuit C_φ is *accepting*, if o_φ holds in the first step, none of the outputs b_ψ holds along the computation for any subformula ψ of φ, and every fairness constraint in F_φ is satisfied.

Proposition 6.1.2. *Let C be a circuit and let φ be an alternation-free HyperCTL* formula in negation normal form that has only existential quantifiers. $C \models \varphi$ holds iff there is an accepting computation for the circuit C_φ.*

Accepting computations of C_φ directly correspond to accepting runs of the alternating automata $\mathcal{A}_\varphi$ constructed in Chapter 5. The circuit construction additionally guarantees that every *suffix* of a computation that corresponds to an accepting run in $\mathcal{A}_\varphi$, again shows itself is an accepting computation for C_φ (i.e. shows the output o_φ in the first step of the suffix, ...).

The proof proceeds by structural induction on the structure of the formula. The base cases (the proposition and the negated proposition) are trivially fulfilled. For each of the remaining cases we use the property that the sub-circuitC_{ψ_1} (and likewise C_{ψ_2}) corresponds to the language of $\mathcal{A}_{\psi_1}$ (or $\mathcal{A}_{\psi_2}$) whenever we require its propositions o_{ψ_1} (or o_{ψ_2}). The transition relation T for every case is then a direct translation of the transitions of the alternating automata construction in Chapter 5.

The only significant difference to the automaton construction is the lack of nondeterminism in the system model - instead we make use of the existential choice of inputs in circuits. Nondeterministic choices are necessary whenever we formulate proof obligations for the future of the computation. Consider the formula $\exists\pi. (\bigcirc a_\pi) \vee (\bigcirc b_\pi)$. Intuitively, when analyzing the first step of the computation, we cannot know which of the two disjuncts holds and thus have to guess for each of the disjuncts whether it is the case. For each subformula ψ that starts with a temporal operator we thus introduce an additional input i_ψ to indicate the *guess* that this subformula holds. Every guess is stored in the additional variable x_ψ. The additional variables thus represent *proof obligations* for the future of the computation. (In the automaton construction of Chapter 5 the *states* of the alternating automaton correspond to the proof obligations.) The transition relation is then designed to indicate in the output b_ψ whether the proof obligation is violated.

The search for accepting computations of circuits can be performed by standard hardware model checkers. Below we report on experiments using the circuit encoding of alternation-free HyperCTL*.

					Verification time in s			
		Model	#Latches	#Gates	IC3	INT	BMC	
IF1	(NI1)	I2C Master	254	1207	95.17	1.13	0.07	×
IF2	(NI2)				53.08	1.16	0.08	×
IF3	(NI3)				168.96	1.38	-	✓
IF4	(NI4)				438.41	1.01	0.09	×
IF5	(NI5)				717.74	8.31	0.77	×
IF6	(NI6)				186.20	1.10	0.07	×
IF7	(NI7)				TO	6.82	0.55	×
IF8	(NI8)				1557.14	2.92	0.16	×
IF9	(NI2′)	Ethernet	21093	70837	TO	155.77	6.27	×
Sym1	(S1)	Bakery	46	1829	6.34	0.88	0.08	×
Sym2	(S2)				168.59	464.52	7.00	×
Sym3		Bakery.a	47	1588	69.12	TO	71.92	×
Sym4	(S3)	Bakery.a.n	47	1618	26.31	4.75	0.39	×
Sym5		Bakery.a.n.s	47	1532	66.41	TO	-	✓
Sym6	(S4)				16.83	TO	-	✓
Sym7	(S5)	Bakery.a.n.s.5proc	90	3762	97.45	TO	-	✓
Sym8	(S6)				13.59	TO	-	✓
Sym9	(S7)	Bakery.a.n.s.7proc	136	6775	312.53*	TO	-	✓
Huff1	(HD1)	Huffman_enc	19	571	3.08	37.19	-	✓
Huff2	(HD2)				0.62	0.09	0.02	×
8b10b_1	(HD1)	8b10b_enc	39	271	0.32	0.09	0.02	×
8b10b_2	(HD1′)				1.19	9.06	-	✓
8b10b_3	(HD2′)				0.03	0.04	0.02	×
8b10b_4	(HD1″)	8b10b_dec	19	157	0.05	0.09	-	✓
Hamm1	(HD1$_1$)	Hamming_enc	27	47	0.02	0.04	0.02	×
Hamm2	(HD1$_2$)				0.02	0.03	0.02	×
Hamm3	(HD1$_3$)				0.03	0.04	0.02	×
Hamm3′	(HD1$'_3$)				7.34	0.18	-	✓
Hamm4	(HD1$_4$)				66.93	0.10	-	✓
Hamm5	(HD2$_1$)				11.83	1.31	-	✓
Hamm6	(HD2$_2$)				14.44	0.78	-	✓
Hamm7	(HD3)				12.23	1.25	-	✓

Table 6.1: Experimental results for the case studies.

6.2 Case Studies and Experimental Results

The practical model checking algorithm from Section 6.1 is implemented in a tool called MCHyper [114]. We rely on standard hardware synthesis tools to compile VHDL and Verilog files into Aiger circuits [17] to which we the apply the tool. Given an Aiger circuit C and a formula φ MCHyper produces the circuit C_φ that can then be checked by hardware model checkers.

For the experiments, we used the ABC model checker [21] as the backend verification engine. ABC is well-suited for our experiments, because it provides many of the modern verification algorithms, including IC3 [20]/PDR [40], interpolation (INT) [91], and SAT-based bounded model checking (BMC) [16]. Table 6.1 shows the verification times for the various circuits and properties considered in our case studies. The running times are as reported by ABC on an Intel Core i5 processor, model 4278U, with 2.6 GHz. In all verification runs except for the entry marked with *, we used the default settings of ABC. The symbol ✓ in the last column indicates that an invariant was found, and × that a (counter-) example path was found. The running times for BMC are only reported for (counter-) examples.

The experimental results show that our approach enables the verification of hyperproperties for hardware modules with hundreds or even thousands of latches. Further, the table shows that bounded model checking is particularly efficient in finding counterexamples, and for cases where an invariant was needed, the relative performance of IC3/PDR vs. interpolation was inconclusive. In addition to benchmarking, our goal for these case studies has been to explore the versatility of alternation-free HyperCTL* model-checking and the potential of our prototype tool. In the following subsections, we report on the setup and results of the case studies, as well as on the verification workflow from a user perspective. Our case studies come from three different areas: information flow, symmetry, and error resistant codes.

Case Study 1: Information Flow Properties of I2C

Our first case study investigates the information flow properties of an I2C bus master. I2C is a widely used bus protocol that connects multiple components in a master-slave topology. Even though the I2C bus has no security features, it has been used in security-critical applications, such as the smart cards of the German public health insurance, which led to exploits in the years between 1995 and 2013 [130]. We analyzed a I2C bus master implementation from the open source repository `opencores.org`. The setup consits of one *master*, one *controller*, and up to eight *slaves*. The master communicates to the slaves via two physical wires, the clock line (SCL) and the data line (SDA). The interface of the master towards the *controller* consists of 8 bit wide words for input and output of data, a 3-bit wide address to encode slave numbers, a system clock input, and several reset and control signals. We checked the I2C bus master implementation against the information flow properties shown in Table 6.2.

(NI1)	$\forall\pi.\forall\pi'.\ \ \square(\overline{\text{ADDR\_I}}_\pi = \overline{\text{ADDR\_I}}_{\pi'}) \Rightarrow \square(\text{SDA\_O}_\pi = \text{SDA\_O}_{\pi'})$
(NI2)	$\forall\pi.\forall\pi'.\ \ \square(\overline{\text{DAT\_I}}_\pi = \overline{\text{DAT\_I}}_{\pi'}) \Rightarrow \square(\text{SDA\_O}_\pi = \text{SDA\_O}_{\pi'})$
(NI3)	$\forall\pi.\forall\pi'.\ \ \square(\neg\text{WEn} \wedge \overline{\text{DAT\_I}}_\pi = \overline{\text{DAT\_I}}_{\pi'}) \Rightarrow$ $\square(\text{SDA\_O}_\pi = \text{SDA\_O}_{\pi'})$
(NI4)	$\forall\pi.\forall\pi'.\ \ \square(\overline{\{\text{SDA\_I,SCL\_I}\}}_\pi = \overline{\{\text{SDA\_I,SCL\_I}\}}_{\pi'}) \Rightarrow$ $\square(\text{DAT\_O}_\pi = \text{DAT\_O}_{\pi'})$
(NI5)	$\forall\pi\ \ \square(\text{SDA\_Enable} \Rightarrow \mathcal{H}_{\{\text{SDA\_I,SCL\_I}\},\{\text{DAT\_O}\}}\mathsf{false})$
(NI6)	$\forall\pi.\forall\pi'.\ \ \square(\overline{\text{SDA\_I}}_\pi = \overline{\text{SDA\_I}}_{\pi'}) \Rightarrow \square(\text{SDA\_O}_\pi = \text{SDA\_O}_{\pi'})$
(NI7)	$\forall\pi.\forall\pi'.\ \ \square(\overline{\text{DAT\_I}}_\pi = \overline{\text{DAT\_I}}_{\pi'}) \Rightarrow$ $(\square(I_\pi = I_{\pi'}) \Rightarrow \Diamond\square(\text{SDA\_O}_\pi = \text{SDA\_O}_{\pi'}))$
(NI8)	$\forall\pi.\forall\pi'.\ \ \square(\overline{\{\text{SDA\_I,SCL\_I}\}}_\pi = \overline{\{\text{SDA\_I,SCL\_I}\}}_{\pi'}) \Rightarrow$ $(\square(I_\pi = I_{\pi'}) \Rightarrow \Diamond\square(\text{DAT\_O}_\pi = \text{DAT\_O}_{\pi'}))$

Table 6.2: Information flow properties for the verification of the I2C bus master. In this list of properties, $P_\pi = P_{\pi'}$ is defined as $\bigwedge_{a\in P} a_\pi = a_{\pi'}$. $\overline{P}_\pi = \overline{P}_{\pi'}$ is defined as $(I \setminus P)_\pi = (I \setminus P)_{\pi'}$ where $P \subseteq \mathsf{AP}$ and $I \subseteq \mathsf{AP}$ are the inputs of the circuit.

From the controller to the bus. Property (NI1) states that there is no information flow with respect to the address to which the I2C master intends to send data, and (NI2) with respect to the data words themselves. Both information flows are present and obviously intended, and our tool reports the violation. We tried to bound the information flow between the first valuation of the 3 bit wide address input and the bus data by encoding [30] the quantitative information-flow property. While the information flow of 3 bit could be determined (QNI1), checking the upper bound of $\log 9 \approx 3.17$ bit (QNI2) led to a timeout. Property (NI3) states that when the *write enable* bit is not set, no information should flow from the controller inputs to the bus. This property is satisfied by the implementation.

From the bus to the controller. Property (NI4) claims the absence of information flow from the slaves to the controller, which is again legitimately violated by the implementation. Property (NI5) refines (NI4) to determine whether the flow can still happen when we only consider information received on SDA *while* the master sends data too. The branching time operator $\mathcal{H}$ in (NI5), called the Hide operator $\mathcal{H}_{I,O}\varphi$, is borrowed from the logic SecLTL [38] and expresses that information from the inputs I do not interfere with the outputs O. The Hide operator is easily expressible in HyperCTL* [30]. Property (NI5) is violated by the implementation, because the concurrent transmission of data on the bus by multiple masters can bring I2C into arbitration mode and changes the interpretation of information sent over the bus later.

Long-term information flow: Properties (NI7) and (NI8) claim that the information flows from (NI1) and (NI4) cannot happen for an arbitrary delay. These properties are violated, which indicates that information may not be eventually forgotten by the I2C master.

All properties on the I2C Master were easily analyzed by the model checker. In order to determine if our approach scales to even larger designs, we checked an adapted version of property (NI2) on an Ethernet IP core with 21093 latches. The counterexample was still found within seconds.

Case Study 2: Symmetry in Mutual Exclusion Protocols

In our second case study, we investigate symmetry properties of mutual exclusion protocols. Mutual exclusion is a classical problem in distributed systems, for which several solutions have been proposed and analyzed. Violation of symmetry indicates that some clients have an unfair advantage over the other clients.

Our case study is based on a Verilog implementation of the Bakery protocol [80] from the VIS verification benchmark. The Bakery protocol works as follows. When a process wants to access the critical section it draws a "ticket", i.e., it obtains a number that is incremented every time a ticket is drawn. If there is more than one process who wishes to enter the critical section, the process with the smallest ticket number goes first. When two processes draw tickets concurrently, they may receive tickets with the same number, so ties among processes with the same ticket must be resolved by a different mechanism, for example by comparing process IDs. The Verilog implementation has an input *select* to indicate the process ID that runs in the next step, and an input *pause* to indicate whether the step is stuttering. Each process n has a program counter $pc(n)$. When process n is selected, the statement corresponding the program counter $pc(n)$ is executed.

We are interested in the following HyperLTL property:

(S1)	$\forall\pi.\forall\pi'.\quad \Box(\mathit{sym}(\mathit{select}_\pi, \mathit{select}_{\pi'}) \wedge \mathit{pause}_\pi = \mathit{pause}_{\pi'}) \Rightarrow$ $\Box(pc(0)_\pi = pc(1)_{\pi'} \wedge pc(1)_\pi = pc(0)_{\pi'})$

where $\mathit{sym}(\mathit{select}_\pi, \mathit{select}_{\pi'})$ means that process 0 is selected on path π when process 1 is selected on path π' and vice versa. Property (S1) states that, for every execution, there is another execution in which the *select* inputs corresponding to processes 0 and 1 are swapped and the outcome (i.e., the sequence of program counters of the processes) is also swapped. It is well known that it is impossible to accomplish mutual exclusion in an entirely symmetric fashion [84]. It is therefore not surprising that the implementation indeed violates Property (S1).

Inspecting the counterexample revealed, however, that the symmetry is broken even before the critical section is reached: if a non-existing process ID is selected by the variable *select*, process 0 proceeds instead. Property (S2) excludes paths on which a non-existing process ID is selected. The model-checker produced a counterexample in which processes 0 and 1 tried to access the critical section, but were treated differently.

(S2)	$\forall\pi.\forall\pi'.\quad \Box\big(\mathsf{sym}(select_\pi, select_{\pi'}) \wedge$ $pause_\pi = pause_{\pi'} \wedge$ $select_\pi < 3 \wedge select_{\pi'} < 3\big) \Rightarrow$ $\Box\big(pc(0)_\pi = pc(1)_{\pi'} \wedge pc(1)_\pi = pc(0)_{\pi'}\big)$

Next, we parameterized the necessary symmetry breaking in the system. We introduced additional inputs indicating which process may move, in case of a tie of the tickets and extended the property by the assumption that the symmetry is broken symmetrically.

(S3)	$\forall\pi.\forall\pi'.\quad \Box\big(\mathsf{sym}(select_\pi, select_{\pi'}) \wedge$ $pause_\pi = pause_{\pi'} \wedge$ $select_\pi < 3 \wedge select_{\pi'} < 3 \wedge$ $\mathsf{sym}(sym\_break_\pi, sym\_break_{\pi'})\big) \Rightarrow$ $\Box\big(pc(0)_\pi = pc(1)_{\pi'} \wedge pc(1)_\pi = pc(0)_{\pi'}\big)$

Property (S3) is still violated by the implementation: the order in which the processes were checked depends on the process IDs and causes delays in how the program counters evolve. After contracting the comparison of process IDs into a single step, property (S3) became satisfied.

In further experiments, we changed the structure of property from the form (S3) $\forall\pi.\forall\pi'.\ \Box\varphi \Rightarrow \Box\psi$ to (S7) $\forall\pi.\forall\pi'.\ \psi\,\mathcal{W}\,\neg\varphi$, which removes the liveness part of the property, while maintaining the semantics (for input-deterministic and input-enabled systems). This change significantly reduced the verification times and enabled the verification of the protocol for up to 7 participants.

Case Study 3: Error Resistant Codes

Error resistant codes enable the transmission of data over noisy channels. While the correct operation of encoder and decoders is crucial for communication systems, the formal verification of their functional correctness has received little attention. A typical model of errors bounds the number of flipped bits that may happen for a given code word length. Then, error correction coding schemes must guarantee that all code words have a minimal Hamming distance. Alternation-free HyperCTL* can specify that all code words produced by an encoder have a minimal Hamming distance of d:

(HDd)	$\forall\pi.\forall\pi'.\Diamond\big(\bigvee_{a\in I} a_\pi \neq a_{\pi'}\big) \Rightarrow \neg\mathrm{Ham}_O(d-1,\pi,\pi')$

where I are the inputs denoting the data, O denote the code words, and the predicate $\mathrm{Ham}_O(d,\pi,\pi')$ is defined, like in Chapter 3, as $\mathrm{Ham}_O(-1,\pi,\pi') =$ false and:

$$\mathrm{Ham}_O(d,\pi,\pi') := \Big(\bigwedge_{a\in O} a_\pi = a_{\pi'}\Big)\,\mathcal{W}\,\Big(\bigvee_{a\in O} a_\pi \neq a_{\pi'} \;\wedge\; \bigcirc \mathrm{Ham}_O(d-1,\pi,\pi')\Big).$$

We started with two simple encoders that are not intended to provide error resistance: a Huffman encoder from the VIS benchmarks, and an 8bit-10bit encoder from opencores.org that guarantees that the difference between the number of 1s and the number of 0s in the codeword is bounded by 2. As expected, encoders provide a Hamming distance of 1 (Huff1 and 8b10b_2), but not more (Huff2 and 8b10b_3). The experiments on these simple encoders were useful to determine the configuration of the command signals that enable the transmission of data. For example, checking the plain property as specified above for the 8bit-10bit encoder reveals that the reset signal must be set to false before sending data (8b10b_1). Similarly, for the 8bit-10bit *decoder*, we checked whether all codewords of Hamming distance 1 produce different outputs (8b10b_4).

Next, we considered an encoder for the 7-4-Hamming code, which encodes blocks of 4 bits into codewords of length 7, and guarantees a Hamming distance of 3. We started with finding out in which configuration the encoder actually sends encoded data (Hamm1 to Hamm4). With Hamm3 we discovered that the implementation deviates from the specification because the reset signal for the circuit is active high, instead of active low as specified. In Hamm3, we fixed the usage of the reset bit. We then scaled the specification to Hamming distances 2 and 3 (Hamm5 to Hamm7).

Chapter 7

Related Logics

In this Chapter, we study the relation of HyperLTL and HyperCTL* to other logics that are able to express certain information-flow properties. The comparison of logics (or specification languages) is difficult, as they often have very different aims. Some logics are designed for the largest possible expressiveness, others restrict the expressiveness on purpose to obtain algorithmic results. Also theoretical purity and usability can be conflicting goals in the design of logics. It is therefore futile to find the *best* logic and we focus on establishing the basic facts about the relations to other logics.

First, we consider the epistemic temporal logic that concern the specification of the knowledge of agents or absence thereof. The concept of knowledge is also formalized as a property over pairs of executions and it has been observed that we can encode certain information-flow properties via epistemic temporal logics [136, 10]. We show that with extended path quantification HyperLTL includes epistemic temporal logics and we explain how results by Bozzelli et al. [19], who showed that HyperLTL and epistemic temporal logics are incomparable in their expressiveness, fit in the picture (Section 7.1). Second, we consider fixed point calculi that are suited to relate multiple execution traces (Section 7.2). We show that the expressiveness of HyperCTL* is incomparable to both the polyadic modal μ-calculus [7], the higher-dimensional μ-calculus [81]. Holistic hyperproperties, a fixed point based logic for hyperproperties, aims for generality and includes much more than HyperCTL* [94, 93]. We show that this comes at the cost of an undecidable model checking problem and that its sublogic *incremental hyperproperties*, for which model checking is again decidable, does not include HyperLTL.

7.1 Epistemic Temporal Logic

Epistemic temporal logic, or the logic of knowledge and time, extends temporal logics by the *knowledge operator* $\mathcal{K}_A\varphi$ that expresses that agent A knows fact φ. Knowledge is defined via the quantification over all traces that are observationally equivalent for agent A. Thereby, epistemic temporal logic allows us to express certain information-flow properties [10, 24, 136]. In this section, we consider the epistemic temporal logic LTL_K that extends LTL with the knowledge operator [135, 45]:

$$\varphi \ ::= \ a_\pi \mid \neg\varphi \mid \varphi \wedge \varphi \mid \bigcirc\varphi \mid \varphi\, \mathcal{U}\, \varphi \mid \mathcal{K}_A\varphi$$

Agents are only referred to as entities with a particular observational power, and hence we define $A \subseteq \mathsf{AP}$ to be the set of atomic propositions this agent can observe. Again, we introduce the derived temporal operators $\diamondsuit\varphi$ and $\square\varphi$ as well as the derived boolean operators $\varphi \vee \varphi$ and $\varphi \Rightarrow \varphi$.

Epistemic temporal logic is typically defined on *interpreted systems*, but here define its semantics on Kripke structures, to enable a clean comparison to HyperLTL. We consider the epistemic operator with *perfect recall semantics* and with the *synchronous time assumption*. We introduce a parameter i in the validity relation to indicate the current point of time to the semantics of LTL and define the semantics of epistemic temporal logic as follows:

$$\begin{array}{lll}
t,i \models_K a & \text{iff} & a \in t(i) \\
t,i \models_K \neg\varphi & \text{iff} & t,i \not\models_K \varphi \\
t,i \models_K \varphi_1 \wedge \varphi_2 & \text{iff} & t,i \models_K \varphi_1 \text{ and } t,i \models_K \varphi_2 \\
t,i \models_K \bigcirc\varphi & \text{iff} & t,i+1 \models_K \varphi \\
t,i \models_K \varphi_1\, \mathcal{U}\, \varphi_2 & \text{iff} & \text{there is } k \geq i:\ t,k \models_K \varphi_2 \text{ and} \\
 & & \quad \text{for all } i \leq j < k:\ t,j \models_K \varphi_1 \\
t,i \models_K \mathcal{K}_A\varphi & \text{iff} & \forall t' \in \mathsf{Traces}(K,s_0).\ t[0,i] =_A t'[0,i] \ \Rightarrow\ t',i \models_K \varphi\,,
\end{array}$$

where $t[0,i] =_A t'[0,i]$ denotes the equivalence of t and t' on the atomic propositions in A for all positions in $[0,i]$. A Kripke structure K satisfies a formula φ, denoted $K \models \varphi$, iff for all traces t in $\mathsf{Traces}(K,s_0)$ it holds $t,0 \models_K \varphi$.

Even though knowledge operators may occur inside temporal operators, it only refers to the Kripke structure via its set of traces $\mathsf{Traces}(K,s_0)$. Epistemic temporal logics is thus preserved under trace equivalence, i.e. it is a linear-time logic. In the following, we show that LTL_K can be encoded in HyperLTL, when we also allow for quantification via propositions and we discuss related results that show that this is necessary.

Theorem 7.1.1. *HyperLTL with quantification over propositions subsumes LTL_K.*

Proof. We start by considering the straight forward unification of the logics HyperLTL with quantification over propositions and epistemic temporal logics: $\mathrm{HyperLTL}_K$. The validity relation $\Pi, i \models_K \varphi$ of the logic is defined as follows:

$$
\begin{array}{lll}
\Pi, i \models_K a_\pi & \text{iff} & a \in L\big(\Pi(\pi)(i)\big) \\
\Pi, i \models_K \neg\varphi & \text{iff} & \Pi, i \not\models_K \varphi \\
\Pi, i \models_K \varphi_1 \vee \varphi_2 & \text{iff} & \Pi, i \models_K \varphi_1 \text{ or } \Pi, i \models_K \varphi_2 \\
\Pi, i \models_K \bigcirc\varphi & \text{iff} & \Pi, i+1 \models_K \varphi \\
\Pi, i \models_K \varphi_1 \mathcal{U} \varphi_2 & \text{iff} & \text{for some } k \geq i :\ \Pi, k \models_K \varphi_2 \text{ and} \\
 & & \quad \text{for all } i \leq j < k :\ \Pi, j \models_K \varphi_1 \\
 & & \quad \text{for all } i \leq j < k :\ t, j \models_K \varphi_1 \\
\Pi, i \models_K \mathcal{K}_{A,\pi}\varphi & \text{iff} & \forall t' \in \mathsf{Traces}(K, s_0).\ \Pi(\pi)[0,i] =_A t'[0,i] \ \Rightarrow\ t', i \models_K \varphi \\
\Pi, i \models_K \exists\pi.\varphi & \text{iff} & \text{for some } t \in \mathsf{Traces}(K, s_0) :\ \Pi[\pi \mapsto t], i \models_K \varphi \\
\Pi, i \models_K \exists a.\varphi & \text{iff} & \text{for some } t \in (2^a)^\omega :\ \Pi[a \mapsto t], i \models_K \varphi \\
\Pi, i \models_K a & \text{iff} & a \in \Pi(a)(i),
\end{array}
$$

The knowledge operator $\mathcal{K}_{A,\pi}\varphi$ here received an additional parameter π, to indicate which path the knowledge refers to. Note that the semantics includes an operator $\exists a$ for quantification over propositions and also includes propositions a that are not bound to any path.

The logic is a straight-forward extension of HyperLTL with with extended path quantification and it also subsumes LTL$_K$: given any LTL$_K$ formula ψ, we prepend a universal quantifier $\forall\pi$ and index all atomic propositions and knowledge operators in ψ with π.

In the following, we show how to remove a single knowledge operator from a HyperLTL$_K$ formula. Applying the construction repeatedly, until no knowledge operator is left, provides a HyperLTL formula with quantification over propositions.

Let $\varphi = \mathbf{Q}.\varphi'$ be a HyperLTL formula in NNF that possibly has knowledge operators, let $\mathbf{Q}$ be its quantifier prefix, and let φ' be the quantifier-free part of φ. Let t and u be propositions that φ does not refer to and that are not in the alphabet of the Kripke structure. In case a knowledge operator $\mathcal{K}_{A,\pi}\psi$ occurs in φ' with positive polarity (i.e. non-negated), we translate φ as follows:

$$
\mathbf{Q}.\exists u.\ \forall t.\ \forall\pi'.\ \varphi'|_{\mathcal{K}_{A,\pi}\psi \to u} \ \wedge\ \Big((t\,\mathcal{U}\,(u \wedge t \wedge \bigcirc\Box\neg t)) \ \wedge\ \Box(t \to A_\pi = A_{\pi'}) \ \to \\
\Box(t \wedge \bigcirc\neg t \to \psi|_{\pi\to\pi'}) \Big)
$$

and, if the knowledge operator occurs negated, we translate φ as follows:

$$
\mathbf{Q}.\exists u.\ \forall t.\ \exists\pi'.\ \varphi'|_{\neg\mathcal{K}_{A,\pi}\psi \to u} \ \wedge\ \Big((t\,\mathcal{U}\,(u \wedge t \wedge \bigcirc\Box\neg t)) \ \to \\
\Box(t \to A_\pi = A_{\pi'}) \ \wedge\ \Box(t \wedge \bigcirc\neg t \to \neg\psi|_{\pi\to\pi'}) \Big)
$$

where π' is assumed to be fresh, $\varphi'|_{\mathcal{K}_{A,\pi}\psi\to u}$ denotes that in φ' one occurrence of the knowledge operator $\mathcal{K}_{A,\pi}\psi$ is replaced by proposition u, and $\psi|_{\pi\to\pi'}$ denotes the formula ψ where indices π are replaced by π' (assuming that π is not bound again in ψ).

The sequence of the atomic proposition u indicates all points of time the knowledge operator needs to be true. By requiring that $\varphi'|_{\neg\mathcal{K}_{A,\pi}\psi\rightarrow u}$ holds, we make sure that this sequence of points satisfies φ'. Subsequently, we quantify over proposition t, which marks a *particular* point in time when the knowledge operator needs to be true: when the sequence of t turns from true to false. The existence of this point is guaranteed by the subformula $t\,\mathcal{U}\,(u \wedge t \wedge \bigcirc\Box\neg t)$.

Consider the first case where the knowledge operator occurs positively. In this setup, we require that all paths π' that share the same sequence of propositions A on path π, the subformula ψ has to hold on path π' exactly at the point marked by t. Otherwise, if the knowledge operator occurred negatively, we require that there is a path π' that shares the same sequence of propositions A on path π and on which $\neg\psi$ holds at this point of time. □

Bozzelli et al. [19] recently proved that this claim does not hold for HyperLTL without quantification over propositions. In fact, the expressiveness of HyperLTL without quantification over propositions and LTL_K is incomparable.

Theorem 7.1.2 ([19]). *The LTL_K property $\bigcirc\Diamond\mathcal{K}_\emptyset a$ cannot be expressed in HyperLTL.*

Theorem 7.1.3 ([19]). *The HyperLTL property $\exists\pi.\exists\pi'.a_\pi\mathcal{U}\Big(a_\pi\wedge\neg a_{\pi'}\wedge\bigcirc\Box(a_\pi \leftrightarrow a_{\pi'})\Big)$ cannot be expressed in LTL_K.*

The knowledge operator pinpoints a particular point of time and considers all traces at this point of time. The quantification over paths in HyperLTL cannot do that in general, as the traces are quantified before the time is quantified. When we allow for quantification over propositions, however, we unify the two worlds. In particular, this shows that quantification over propositions adds more to HyperLTL than ω-regularity (over single traces).

7.2 Fixed-point Calculi

In this section we compare the expressiveness of various fixed-point calculi to the expressiveness of HyperLTL and HyperCTL*.

Modal μ-Calculus

Expressions of the *modal μ-calculus* [73] are generated by the following grammar:

$$\varphi \quad ::= \quad a \quad | \quad X \quad | \quad \neg\varphi \quad | \quad \varphi \vee \varphi \quad | \quad \langle\delta\rangle\varphi \quad | \quad \mu X.\varphi(X)$$

where X is chosen from a set N of set variable names and a is chosen from a set of atomic propositions. The expression $\varphi(X)$ denotes a formula in which X occurs only under an even number of negations. We also define the derived operator $[\delta]\varphi \equiv \neg\langle\delta\rangle\neg\varphi$.

Semantics of the modal μ-calculus. The semantics of the modal μ-calculus is defined via assignments of set variables. An *assignment* of set variables is a partial function $\beta : N \to 2^S$, where S is a set of states in a given Kripke structure and N is a set of names for set variables. For a given expression φ in the modal μ-calculus, an assignment β, and a Kripke structure $K = (S, s_0, \delta, \mathsf{AP}, L)$ we define the set $||\varphi||_{\beta,K} \subseteq S$ as follows:

$$\begin{array}{lcl}
||a||_{\beta,K} & := & \left\{ s \in S \mid a \in L(s) \right\} \\
||X||_{\beta,K} & := & \beta(X) \\
||\neg\varphi||_{\beta,K} & := & S \setminus ||\varphi||_{\beta,K} \\
||\varphi_1 \vee \varphi_2||_{\beta,K} & := & ||\varphi_1||_{\beta,K} \cup ||\varphi_2||_{\beta,K} \\
||\langle\delta\rangle\varphi||_{\beta,K} & := & \left\{ s \in S \mid \delta(s) \cap ||\varphi||_{\beta,K} \right\} \\
||\mu X.\varphi(X)||_{\beta,K} & := & \bigcap \left\{ T \subseteq S \mid ||\varphi(X)||_{\beta[X \mapsto T],K} \subseteq T \right\}
\end{array}$$

Validity on states of a Kripke structure K, written $(K, s) \models \varphi$, is then defined as $s \in ||\varphi||_{\beta,K}$. A Kripke structure $K = (S, s_0, \delta, \mathsf{AP}, L)$ satisfies a modal μ-calculus expression φ, denoted with $K \models \varphi$, iff $(K, s_0) \models \varphi$.

Incomparability of HyperLTL/HyperCTL* and the modal μ-calculus. Noninterference can be expressed in HyperLTL and HyperCTL*, but not in the modal μ-calculus [5]. Vice versa, HyperLTL and HyperCTL* cannot express all ω-regular properties on paths, as shown in Theorem 5.5.2.

Polyadic Modal μ-Calculus

Polyadic modal μ-calculus is an extension of the μ-calculus to multiple Kripke structures [7]. It operates on tuples of states (possibly of different Kripke structures) and extends the operator $\langle\delta\rangle$ by an index i denoting to the position in the tuple of states that does a step. Also the atomic propositions receive an index. The grammar of the polyadic modal μ-calculus is thus:

$$\varphi \quad ::= \quad a_i \quad | \quad X \quad | \quad \neg\varphi \quad | \quad \varphi \vee \varphi \quad | \quad \langle\delta\rangle_i \varphi \quad | \quad \mu X.\varphi(X)$$

where $i \in \mathbb{N}$, X is chosen from a set N of set variable names, and a is chosen from a set of atomic propositions. The expression $\varphi(X)$ denotes a formula in which X occurs only under an even number of negations.

Semantics of the polyadic modal μ-calculus. In contrast to the modal μ-calculus, set variables here range over *tuples of states*. An assignment of set variables is thus a partial function $\beta : N \to 2^{S^+}$, where S is a set of states in a given Kripke structure and N is a set of names for set variables. For a given expression φ in the polyadic modal μ-calculus, an assignment β, and a tuple of Kripke structures $\overline{K} = (K_1,\dots,K_n)$ we define the set $\|\varphi\|_{\beta,\overline{K}} \subseteq S$ as follows:

$$\begin{array}{lcl}
\|a_i\|_{\beta,\overline{K}} & := & \Big\{(s_1,\dots,s_n) \in S^n \mid a \in L_i(s_i)\Big\} \\
\|X\|_{\beta,\overline{K}} & := & \beta(X) \\
\|\neg\varphi\|_{\beta,\overline{K}} & := & S^n \setminus \|\varphi\|_{\beta,\overline{K}} \\
\|\varphi_1 \vee \varphi_2\|_{\beta,\overline{K}} & := & \|\varphi_1\|_{\beta,\overline{K}} \cup \|\varphi_2\|_{\beta,\overline{K}} \\
\|\langle\delta\rangle_i\varphi\|_{\beta,\overline{K}} & := & \Big\{(s_1,\dots,s_n) \in S^n \mid \exists s_i' \in \delta_i(s_i) : (s_1,\dots,s_i',\dots,s_n) \in \|\varphi\|_{\beta,\overline{K}}\Big\} \\
\|\mu X.\varphi(X)\|_{\beta,\overline{K}} & := & \bigcap\Big\{T \subseteq S^n \mid \|\varphi(X)\|_{\beta[X\mapsto T],\overline{K}} \subseteq T\Big\}
\end{array}$$

Validity on tuples $\overline{s}$ of states of Kripke structures $\overline{K}$, written $(\overline{K},\overline{s}) \models \varphi$, is then defined as $\overline{s} \in \|\varphi\|_{\beta,\overline{K}}$. A tuple of Kripke structures $\overline{K}$ satisfies a polyadic modal μ-calculus expression φ, denoted with $\overline{K} \models \varphi$, iff $(\overline{K},\overline{s_0}) \models \varphi$, where $\overline{s_0}$ is the tuple of initial states of the Kripke structures $\overline{K}$ (in that order). Note that we adapted the definition of the polyadic modal μ-calculus to Kripke structures. The original definition uses to transition systems.

Similar to HyperLTL and HyperCTL* its model checking problem of the polyadic modal μ-calculus is decidable, while its satisfiability problem is undecidable. Its model checking problem is even not harder than the model checking problem of the modal μ-calculus, i.e. PTIME.

Unlike the semantics of HyperLTL and HyperCTL*, the progression of time in the polyadic modal μ-calculus is not synchronous by default. Instead we always replace one of the states in the tuple by one of its successors. This further allows us to express that two Kripke structures are bisimilar [7].

The expressiveness of HyperLTL/HyperCTL* and the polyadic modal μ-calculus is incomparable too. While the argument that the polyadic modal μ-calculus is not subsumed by HyperLTL or HyperCTL* is the same as for the modal μ-calculus, the other direction is harder, as the interpretation over pairs of states of the same Kripke structure allows us to express noninterference in the polyadic modal μ-calculus.

Theorem 7.2.1. *The polyadic modal μ-calculus does not subsume HyperLTL.*

Proof. HyperLTL can express properties that are EXPSPACE-hard in the size of the Kripke structure. Any property expressible in the polyadic modal μ-calculus can be checked in PTIME. Thus there must be properties expressible in HyperLTL that cannot be expressed in the polyadic modal μ-calculus. □

Higher-dimensional Modal μ-Calculus

The higher-dimensional μ-calculus can be seen as an extension of the polyadic modal μ-calculus by the *variable replacement* operator [81]. The variable replacement operator allows us for example to permute the states in the tuple. Similar to the polyadic modal μ-calculus, the model checking problem of the higher-dimensional modal μ-calculus is in PTIME. This implies that the expressiveness of HyperLTL/HyperCTL* and the higher-dimensional modal μ-calculus is again incomparable.

More interestingly, however, Lange and Lozes prove that the higher-dimensional modal μ-calculus can express any bisimulation invariant property of systems that can be decided in PTIME. That is, the higher-dimensional modal μ-calculus subsumes alternation-free HyperCTL*.

Holistic and Incremental Hyperproperties

Milushev and Clarke proposed the *holistic hyperproperties logic* as a means to specify information-flow properties [94]. Similar to HyperCTL* the holistic hyperproperties logic introduces the quantification over paths to an existing logic, the least fixed point logic, which is a variant of first order logic with fixed point operators. In contrast to the μ-calculi considered before, the holistic hyperproperties logic is unable to refer to the branching structure of the system and can therefore only express linear-time properties. The logic is thus incomparable to HyperCTL* in terms of expressiveness. (The holistic hyperproperty logic also cannot be subsumed by HyperLTL, as we will see below, but the other direction is open.)

The algorithmic problems of the holistic hyperproperties logic are not yet established. Given the generality of the logic it comes at no surprise that its model checking problem is undecidable. We sketch the proof here to suggest that holistic hyperproperties are not suited as a specification logic for algorithmic verification.

Theorem 7.2.2. *The model checking problem of holistic hyperproperties is undecidable.*

Proof. We encode Post's correspondence problem [113] in the model checking problem of incremental hyperproperties. Post's correspondence problem is to decide, given two lists of finite words $a_1,\dots,a_n$ and $b_1,\dots,b_n$, whether there exists a sequence of indices $i_1,\dots,i_k$ such that $a_{i_1}\dots a_{i_k} = b_{i_1}\dots b_{i_k}$.

We consider a class of Kripke structures that represents the two lists of finite words and admits any sequence of the words as a trace. Additionally we require that on the start of each word, the Kripke structure also provides labels to indicate whether the word is from the first or the second list of words and to identify the word index. (Word indices can be represented sequentially, such that a finite number of labels suffices.) It is clear that we can pick a representation such that each Kripke structure in this family has finitely many states. The holistic hyperproperty existentially quantifies over two traces π_a and π_b. We require with co-inductive predicates

- that π_a only contains words a_i,
- that π_b only contains words b_i,
- that the letters of both sequences (except for the word numbers) agree in each position of the pair of traces, and

- that the sequence of word indices agree.

Model checking holistic hyperproperties is thus undecidable. □

To obtain a logic with a decidable model checking problem, Milushev and Clarke propose the fragment named *incremental hyperproperties* and give a reduction of a subset of incremental hyperproperties to the polyadic modal μ-calculus.

Chapter 8

Conclusions

We propose using temporal logics as specification languages for information-flow control to overcome the limited flexibility of the available enforcement approaches. However, the commonly considered temporal logics, like LTL and CTL*, cannot express information-flow properties, as they lack the ability to relate multiple traces. We extend linear-time temporal logics and branching-time temporal logics in a simple way to overcome this limitation. We demonstrate that the resulting temporal logics, HyperLTL and HyperCTL*, enable us to express various information-flow properties. It turns out that HyperLTL and HyperCTL* are also suited to express properties from other areas that have not been considered before. We study the connection between HyperLTL and HyperCTL* and related extensions of temporal logics. This also yields the result that HyperLTL extended by the quantification over propositions subsumes epistemic temporal logics. We study the model checking problems of HyperLTL and HyperCTL*. We give an automata-theoretic algorithm, analyze its complexity, and provide matching lower bounds, both in the formula size and in the size of the system. Finally, we consider the symbolic model checking of alternation-free formulas for hardware designs. We propose a circuit construction that enables us to reuse existing model checking technology in an effective way. Along several case studies we demonstrate the flexibility of the approach for information-flow control and applications beyond security.

To summarize, the temporal logic approach to information-flow control provides a simple formal basis for the analysis of systems for violations of confidentiality and integrity and an effective and flexible enforcement approach for hardware security.

Outlook

The temporal logic approach to information-flow control opens pressing questions for further research. Besides the practical problem of improving the applicability of the approach, there are more fundamental question about the limits of expressiveness while maintaining verifiability.

Information-flow Control for Software. The promise of information-flow control to provide comprehensive information-security for *software* is not yet fulfilled. To prevent the next Heartbleed or Shellshock with the property-oriented approach of information-flow control, we still need to devise flexible enforcement methods for software that scale to (many) thousands of lines of code. Recent advances in symbolic software model checking raise hope for *scalable* software model checking for HyperLTL and HyperCTL*.

On a more technical level, the analysis of software requires to reason about the synchronization of execution traces as typical system models of software are oblivious of execution time. A first example of an appropriate property is the encoding of Goguen and Meseguers noninterference in Chapter 3. In other approaches the synchronization has been included in the verification approach itself [71].

Software can be also be represented with system models with infinite state spaces. For instance, it is an open problem what hyperproperties can be verified for Petri nets.

Practical Aspects of Specification Languages and Modeling. While temporal logics emphasize abstraction and simplicity, there are other aspects to consider for the usability of specification languages. A big challenge will be to enable non-expert users to write specifications that truthfully represent their intent and solutions are likely domain specific (cf. [53]). HyperLTL and HyperCTL* may serve here as a backbone logic to ensure the verifiability of specifications. First practical languages in language based information-flow control frame specification languages as a part of the programming language [145].

Modeling the system and the security threat is a very related issue [14]. The property oriented approach of information-flow control reduces the dependence of information security on implementation details. However, The approach still depends on the truthful representation of an attacker's ability to influence and observe about the system. A holistic security analysis should therefore start already with the choice of the system and attacker model.

Cryptography. Information-flow control assumes that whenever an attacker obtains some information about the secret, confidentiality is lost. Cryptography offers a refined view that takes into account that the attacker has only limited computing resources. Even though the encrypted communication between two parties (typically) contains the full information about the secret, it is practically impossible to reconstruct the secret. Küsters and Truderung demonstrated that with the assumption that cryptographic primitives cannot be circumvented, we can abstract from cryptography in a way that enables us to use information-flow control to ensure the correctness of the rest of the program [78].

Beyond Security. The ability to express properties outside information-flow control raises the question what other applications HyperLTL and HyperCTL* could be useful for. Memory models [141] and fault detection [18] seem to profit from properties that relate multiple executions and their encodings in HyperLTL and HyperCTL* may lead to effective verification approaches.

Quantitative Properties Clarkson and Schneider observed that also quantitative properties, such as the mean response time, quantitative noninterference, and probabilistic properties, are hyperproperties [31]. A first approach may be to combine quantifiers over named paths, as introduced in HyperLTL and HyperCTL*, and *quantitative quantification* such as the probabilistic operator in PCTL [59].

Fixed-point Calculi. The quantification over execution traces in fixed-point logics easily leads to undecidability, as discussed in Chapter 7. It will be interesting to test the limits of expressiveness, while preserving the decidability of the model checking problem.

Algorithmics. The alternation-free fragments of HyperLTL and HyperCTL* provided effective verification approaches. To scale the verification to large systems we should explore abstractions that make use of the fact that we analyze multiple copies of the same system.

Some interesting properties, such as trace equivalence, strictly require quantifier alternations. Focussing on certain classes of systems or combinations of classes of systems and fragments of the logics may offer a way around the exponential explosion in the model checking complexity.

Bibliography

[1] Common weakness enumeration - a community-developed dictionary of software weakness types. http://cwe.mitre.org. Accessed on June 29, 2015.

[2] Property specification language reference manual. http://www.eda.org/vfv/docs/PSL-v1.1.pdf. Accessed on July 7, 2015. Version 1.1.

[3] David Albright, Paul Brannan, and Christina Walrond. Did Stuxnet take out 1,000 centrifuges at the Natanz enrichment plant? Institute for Science and International Security, http://media.washingtonpost.com/wp-srv/world/documents/stuxnet_update_15Feb2011.pdf, 2010. Accessed Feb 27, 2015.

[4] Bowen Alpern and Fred B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181 – 185, 1985.

[5] Rajeev Alur, Pavol Černý, and Steve Zdancewic. Preserving secrecy under refinement. In *Proceedings of ICALP*, pages 107–118, 2006.

[6] Torben Amtoft and Anindya Banerjee. Information flow analysis in logical form. In *Proceedings of SAS*, pages 100–115, 2004.

[7] Henrik Reif Andersen. A polyadic modal μ-calculus, 1994. Technical Report.

[8] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. The MIT Press, 2008.

[9] Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K Rajamani. Automatic predicate abstraction of C programs. In *ACM SIGPLAN Notices*, volume 36, pages 203–213. ACM, 2001.

[10] Musard Balliu, Mads Dam, and Gurvan Le Guernic. Epistemic temporal logic for information flow security. In *Proceedings of PLAS*, 2011.

[11] Gilles Barthe, Juan Manuel Crespo, and César Kunz. Relational verification using product programs. In *Proceedings of FM*, pages 200–214. Springer, 2011.

[12] Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. Secure information flow by self-composition. In *Proceedings of CSFW*, pages 100–114. IEEE Computer Society Press, 2004.

[13] Gilles Barthe, Pedro R D'Argenio, and Tamara Rezk. Secure information flow by self-composition. *Mathematical Structures in Computer Science*, 21(06):1207–1252, 2011.

[14] Jason Bau and John C. Mitchell. Security modeling and analysis. *IEEE Security & Privacy*, 9(3):18–25, 2011.

[15] D. Elliot Bell and Leonard J. LaPadula. Secure computer systems: Mathematical foundations. Technical Report 2547, Volume I, MITRE Corporation, Mar 1973.

[16] Armin Biere, Edmund M. Clarke, Richard Raimi, and Yunshan Zhu. Verifiying safety properties of a power PC microprocessor using symbolic model checking without BDDs. In *Proceedings of CAV*, volume 1633 of *LNCS*, pages 60–71. Springer, 1999.

[17] Armin Biere, Keijo Heljanko, and Siert Wieringa. AIGER 1.9 and beyond. `http://fmv.jku.at/hwmcc11/beyond1.pdf`, 2011. Accessed Feb 6, 2015. Via website: `http://fmv.jku.at/aiger/`.

[18] Marco Bozzano, Alessandro Cimatti, Marco Gario, and Stefano Tonetta. Formal design of fault detection and identification components using temporal epistemic logic. In *Proceedings of TACAS*, pages 326–340, 2014.

[19] Laura Bozzelli, Bastien Maubert, and Sophie Pinchinat. Unifying hyper and epistemic temporal logics. In *Proceedings of FoSSaCS*, pages 167–182, 2015.

[20] Aaron R. Bradley. SAT-based model checking without unrolling. In *Proceedings of VMCAI*, volume 6538 of *LNCS*, pages 70–87. Springer, 2011.

[21] Robert K. Brayton and Alan Mishchenko. ABC: an academic industrial-strength verification tool. In *Proceedings of CAV*, volume 6174 of *LNCS*, pages 24–40. Springer, 2010.

[22] David Brumley and Dan Boneh. Remote timing attacks are practical. *Computer Networks*, 48(5):701–716, 2005.

[23] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, and David L. Dill. Sequential circuit verification using symbolic model checking. In *Proceedings of DAC*, pages 46–51. IEEE CS Press, 1990.

[24] Rohit Chadha, Stéphanie Delaune, and Steve Kremer. Epistemic Logic for the Applied Pi Calculus. In *Proceedings of FMOODS/FORTE*, pages 182–197. Springer, 2009.

[25] Jason Cipriani. Heartbleed bug: Check which sites have been patched. `http://www.cnet.com/how-to/which-sites-have-patched-the-heartbleed-bug/`, 2014. Accessed June 4, 2015.

[26] Koen Claessen, Niklas Eén, and Baruch Sterin. A circuit approach to LTL model checking. In *Proceedings of FMCAD*, pages 53–60, 2013.

[27] David Clark, Sebastian Hunt, and Pasquale Malacaria. Quantitative analysis of the leakage of confidential data. In *Proceedings of QAPL*, volume 59 of *ENTCS*, pages 238–251. ENTCS, 2002.

[28] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In Kurt Jensen and Andreas Podelski, editors, *Proceedings of TACAS*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.

[29] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs*, pages 52–71. Springer, 1982.

[30] Michael R. Clarkson, Bernd Finkbeiner, Masoud Koleini, Kristopher K. Micinski, Markus N. Rabe, and César Sánchez. Temporal logics for hyperproperties. In *Proceedings of POST*, pages 265–284, 2014.

[31] Michael R. Clarkson and Fred B. Schneider. Hyperproperties. *J. of Computer Security*, 18(6):1157–1210, 2010.

[32] Inc. Community Health Systems. Form 8-K, Commission File Number 001-15925. `http://phx.corporate-ir.net/phoenix.zhtml?c=120730&p=irol-SECText&TEXT=aHR0cDovL2FwaS50ZW5rd2l6YXJkLmNvbS9maWxpbmcueG1sP2lwYWdlPTk3NjE4NTImRFNFUT0wJlNFUT0wJlNRREVTQz1TRUNUSU9OX0VOVElSRSZzdWJzaWQ9NTc%3d`, 2014. Accessed June 4, 2015.

[33] Byron Cook, Heidy Khlaaf, and Nir Piterman. Faster temporal reasoning for infinite-state programs. In *Proceedings of FMCAD*, pages 75–82, 2014.

[34] Byron Cook, Heidy Khlaaf, and Nir Piterman. Fairness for infinite-state systems. In *Proceedings of TACAS*, pages 384–398. Springer, 2015.

[35] Dorothy E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, May 1976.

[36] Dorothy E Denning and Peter J Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, 1977.

[37] Dominique Devriese and Frank Piessens. Noninterference through secure multi-execution. In *Proceedings of S&P*, pages 109–124. IEEE, 2010.

[38] Rayna Dimitrova, Bernd Finkbeiner, Máté Kovács, Markus N. Rabe, and Helmut Seidl. Model checking information flow in reactive systems. In *Proceedings of VMCAI*, pages 169–185, 2012.

[39] Klaus Dräger, Andrey Kupriyanov, Bernd Finkbeiner, and Heike Wehrheim. Slab: A certifying model checker for infinite-state concurrent systems. In *Proceedings of TACAS*, Lecture Notes in Computer Science. Springer, 2010.

[40] Niklas Eén, Alan Mishchenko, and Robert K. Brayton. Efficient implementation of property directed reachability. In *Proceedings of FMCAD*, pages 125–134, 2011.

[41] E. Allen Emerson and Joseph Y. Halpern. "Sometimes" and "not never" revisited: on branching versus linear time temporal logic. *JACM*, 33:151–178, 1986.

[42] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of OSDI*, pages 1–6, Berkeley, CA, USA, 2010. USENIX Association.

[43] Yves Eudes. Poodle, une faille de sécurité pour "espionner local". http://www.lemonde.fr/pixels/article/2014/10/17/poodle-une-faille-de-securite-pour-espionner-local_4508285_4408996.html, 2014. Accessed May 31, 2015.

[44] Yves Eudes. Shellshock, la faille de sécurité majeure découverte "presque par hasard" par un Francais. http://www.lemonde.fr/pixels/article/2014/10/02/shellshock-la-faille-de-securite-majeure-decouverte-presque-par-hasard-par-un-francais_4498904_4408996.html, 2014. Accessed May 31, 2015.

[45] Ronald Fagin, Yoram Moses, Joseph Y. Halpern, and Moshe Y. Vardi. *Reasoning About Knowledge*. MIT Press, 1995.

[46] Jonathan Fildes. Stuxnet worm 'targeted high-value Iranian assets'. http://www.bbc.com/news/technology-11388018, 2010. Accessed May 31, 2015.

[47] Bernd Finkbeiner and Markus N. Rabe. The linear-hyper-branching spectrum of temporal logics. *it - Information Technology*, 56:273–279, November 2014.

[48] Bernd Finkbeiner, Markus N. Rabe, and César Sánchez. Algorithms for model checking HyperLTL and HyperCTL*. In *Proceedings of CAV*, volume 9206 of *LNCS*, pages 30–48. Springer, 2015.

[49] National Institute for Standards and Technology. Heartbleed, CVE-2014-0160. https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-0160, 2014. Accessed Dec 30, 2014.

[50] National Institute for Standards and Technology. The poodle attack, CVE-2014-3566. https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-3566, 2014. Accessed Jan 4, 2015.

[51] National Institute for Standards and Technology. Shellshock, CVE-2014-6271. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-6271, 2014. Accessed Dec 30, 2014.

[52] Tim French. Decidability of quantifed propositional branching time logics. In *Proceedings of AI*, pages 165–176. Springer, 2001.

[53] Daniel Giffin, Stefan Heule, Amit Levy, David Mazières, John Mitchell, Alejandro Russo, Amy Shen, Deian Stefan, David Terei, and Edward Z. Yang. Security and the average programmer. In *Proceedings of POST*, April 2014.

[54] Rob J. Glabbeek. The linear time - branching time spectrum. In J.C.M. Baeten and J.W. Klop, editors, *Proceedings of CONCUR*, volume 458 of *Lecture Notes in Computer Science*, pages 278–297. Springer, 1990.

[55] J. A. Goguen and J. Meseguer. Security policies and security models. In *Proceedings of S&P*, pages 11–20, 1982.

[56] Marcel J. E. Golay. Notes on digital coding. *Proceedings IRE*, 37:657, 1949.

[57] Lawrence A Gordon and Martin P Loeb. The economics of information security investment. *ACM Transactions on Information and System Security (TISSEC)*, 5(4):438–457, 2002.

[58] Christian Hammer. *Information Flow Control for Java - A Comprehensive Approach based on Path Conditions in Dependence Graphs*. PhD thesis, Universität Karlsruhe (TH), Fak. f. Informatik, July 2009. ISBN 978-3-86644-398-3.

[59] Hans Hansson and Bengt Jonsson. A logic for reasoning about time and reliability. *Formal aspects of computing*, 6(5):512–535, 1994.

[60] Stephen Henson. Add heartbeat extension bounds check. http://git.openssl.org/gitweb/?p=openssl.git;a=commitdiff;h=96db902. Accessed on June 5, 2015. Commit 96db902.

[61] Matthew Hicks, Cynthia Sturton, Samuel T King, and Jonathan M Smith. Specs: A lightweight runtime mechanism for protecting software from security-critical processor bugs. In *Proceedings of ASPLOS*, pages 517–529. ACM, 2015.

[62] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[63] Marieke Huisman, Pratik Worah, and Kim Sunesen. A temporal logic characterisation of observational determinism. In *Proceedings of CSFW*. IEEE Computer Society, 2006.

[64] James W. Gray III. Toward a mathematical foundation for information flow security. In *Proceedings of S&P*, pages 210–34, may 1991.

[65] Johan Anthony Wilem Kamp. *Tense logic and the theory of linear order*. PhD thesis, University of California, Los Angeles, 1968.

[66] Sudeep Kanav, Peter Lammich, and Andrei Popescu. A conference management system with verified document confidentiality. In *Proceedings of CAV*, volume 8559, pages 167–183. Springer International Publishing, 2014.

[67] Sudeep Kanav, Peter Lammich, and Andrei Popescu. A conference management system with verified document confidentiality. In *Proceedings of CAV*, pages 167–183, 2014.

[68] Heather Kelly. The 'Heartbleed' security flaw that affects most of the internet. `http://www.cnn.com/2014/04/08/tech/web/heartbleed-openssl/`, 2014. Accessed May 31, 2015.

[69] Yonit Kesten, Amir Pnueli, and Li-on Raviv. Algorithmic verification of linear temporal logic specifications. In *Proceedings of ICALP*, pages 1–16, 1998.

[70] B. Köpf and D. Basin. An information-theoretic model for adaptive side-channel attacks. In *Proceedings of CCS*, pages 286–296. ACM, 2007.

[71] Máté Kovács. *Information Flow Security in Tree-Manipulating Processes*. PhD thesis, Institut für Informatik, Technische Universität München, March 2014.

[72] Máté Kovács, Helmut Seidl, and Bernd Finkbeiner. Relational abstract interpretation for the verification of 2-Hypersafety Properties. In *Proceedings of CCS*, pages 211–222, November 2013.

[73] Dexter Kozen. Results on the propositional μ-calculus. *Theor. Comput. Sci.*, 27:333–354, 1983.

[74] Orna Kupferman. Augmenting branching temporal logics with existential quantification over atomic propositions. In *Proceedings of CAV*, pages 325–338. Springer, 1995.

[75] Orna Kupferman and Moshe Y. Vardi. Weak alternating automata are not that weak. *ACM TOCL*, 2(3):408–429, 2001.

[76] Orna Kupferman, Moshe Y. Vardi, and Pierre Wolper. An automata-theoretic approach to branching-time model checking. *JACM*, 47(2):312–360, 2000.

[77] James A Kupsch and Barton P Miller. Manual vs. automated vulnerability assessment: A case study. In *Proceedings of MIST*, pages 83–97, 2009.

[78] Ralf Küsters, Tomasz Truderung, and Jürgen Graf. A Framework for the Cryptographic Verification of Java-like Programs. In *Proceedings of CSF*, pages 198–212. IEEE Computer Society, 2012.

[79] Richard E. Ladner. Application of model theoretic games to discrete linear orders and finite automata. *Information and Control*, 33(4):281 – 303, 1977.

[80] Leslie Lamport. A new solution of Dijkstra's concurrent programming problem. *Commun. ACM*, 17(8):453–455, August 1974.

[81] Martin Lange and Étienne Lozes. Model-checking the higher-dimensional modal mu-calculus. In *FICS*, pages 39–46, 2012.

[82] K Rustan M Leino and Rajeev Joshi. A semantic approach to secure information flow. *LNCS*, 1422:254–271, 1998.

[83] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive Systems: Specification*. The Temporal Logic of Reactive and Concurrent Systems. Springer, 1992.

[84] Zohar Manna and Amir Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer, 1995.

[85] Heiko Mantel. Possibilistic definitions of security - an assembly kit. In *Proceedings of CSFW*, pages 185–199, 2000.

[86] Heiko Mantel. The framework of selective interleaving functions and the modular assembly kit. In *Proceedings of FMSE*, pages 53–62, New York, NY, USA, 2005. ACM.

[87] Daryl McCullough. Noninterference and the composability of security properties. In *Proceedings of S&P*, pages 177–186, April 1988.

[88] John McLean. Proving noninterference and functional correctness using traces. *J. of Computer Security*, 1:37–58, 1992.

[89] John McLean. A general theory of composition for trace sets closed under selective interleaving functions. In *Proceedings of S&P*, pages 79–93, May 1994.

[90] Kenneth L McMillan. *Symbolic model checking*. Springer, 1993.

[91] Kenneth L. McMillan. Craig interpolation and reachability analysis. In *Proceedings of SAS*, volume 2694 of *LNCS*, page 336. Springer, 2003.

[92] Robin Milner. What is a process?, September 2009. http://www.cs.rice.edu/~vardi/papers/milner09.pdf.

[93] Dimiter Milushev. *Reasoning about Hyperproperties*. PhD thesis, Katholieke Universiteit Leuven, Faculty of Engineering, Celestijnenlaan 200A, box 2402, B3001 Heverlee, Belgium, 6 2013.

[94] Dimiter Milushev and Dave Clarke. Towards incrementalization of holistic hyperproperties. In *Proceedings of POST*, pages 329–348. Springer, 2012.

[95] Satoru Miyano and Takeshi Hayashi. Alternating finite automata on omega-words. *Theor. Comput. Sci.*, 32:321–330, 1984.

[96] Bodo Möller, Thai Duong, and Krzysztof Kotowicz. This poodle bites: Exploiting the ssl 3.0 fallback. https://www.openssl.org/~bodo/ssl-poodle.pdf, 2014. Accessed Jan 4, 2015.

[97] David E. Muller, Ahmed Saoudi, and Paul E. Schupp. Weak alternating automata give a simple explanation of why most temporal and dynamic logics are decidable in exponential time. In *Proceedings of LICS*, pages 422 –427, 0-0 1988.

[98] Randall Munroe. How the Heartbleed bug works. http://xkcd.com/1354/, 2014. Accessed June 1, 2015.

[99] Toby Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, Timothy Bourke, Sean Seefried, Carmen Lewis, Xin Gao, and Gerwin Klein. sel4: from general purpose to a proof of information flow enforcement. In *Proceedings of S&P*, pages 415–429. IEEE, 2013.

[100] Andrew C Myers. Jflow: Practical mostly-static information flow control. In *Proceedings of POPL*, pages 228–241. ACM, 1999.

[101] NA. Le virus Stuxnet viserait le nucléaire iranien. http://www.lemonde.fr/technologies/article/2010/11/17/le-virus-stuxnet-viserait-le-nucleaire-iranien_1441212_651865.html, 2010. Accessed May 31, 2015.

[102] NA. Poodle bug less bite than Heartbleed, say experts. http://www.bbc.com/news/technology-29627887, 2014. Accessed May 31, 2015.

[103] Sumit Nain and Moshe Y. Vardi. Branching vs. linear time: Semantical perspective. In *Proceedings of ATVA*, volume 4762 of *Lecture Notes in Computer Science*, pages 19–34. 2007.

[104] Zhazira Oskenbayeva. Monitoring SecLTL for android applications. Master's thesis, Saarland University, 2013.

[105] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of aes. In *Topics in Cryptology–CT-RSA 2006*, pages 1–20. Springer, 2006.

[106] Jose Pagliery. Poodles are attacking the internet. http://money.cnn.com/2014/10/15/technology/security/poodle-bug/, 2014. Accessed May 31, 2015.

[107] Donn B Parker. *Fighting computer crime: A new framework for protecting information*. John Wiley & Sons, Inc., 1998.

[108] Anindya C. Patthak, Indrajit Bhattacharya, Anirban Dasgupta, Pallab Dasgupta, and P. P. Chakrabarti. Quantified Computation Tree Logic. *Information Processing Letters*, 82:123–129, 2002.

[109] Nicole Perlroth. Security experts expect shellshock software bug in bash to be significant. http://www.nytimes.com/2014/09/26/technology/security-experts-expect-shellshock-software-bug-to-be-significant.html. Accessed on June 27, 2015.

[110] Amir Pnueli. The temporal logic of programs. In *Proceedings of FOCS*, pages 46–57, 1977.

[111] Andreas Podelski and Andrey Rybalchenko. Transition invariants. In *Proceedings of LICS*, pages 32–41. IEEE, 2004.

[112] Edward C. Posner, Lawrence L. Rauch, and Boyd D. Madsen. Voyager mission telecommunication firsts. *Communications Magazine, IEEE*, 28(9):22 –27, sept. 1990.

[113] Emil L Post. A variant of a recursively unsolvable problem. *Bulletin of the American Mathematical Society*, 52(4):264–268, 1946.

[114] Markus N. Rabe. MCHyper: A model checker for hyperproperties. http://www.react.uni-saarland.de/tools/mchyper/, 2015. Accessed Feb 6, 2015.

[115] Markus N. Rabe, Peter Lammich, and Andrei Popescu. A shallow embedding of HyperCTL*. *Archive of Formal Proofs*, April 2014. http://afp.sf.net/entries/HyperCTL.shtml, Formal proof development.

[116] Frank Rieger. Der digitale Erstschlag ist erfolgt. http://www.faz.net/aktuell/feuilleton/debatten/digitales-denken/trojaner-stuxnet-der-digitale-erstschlag-ist-erfolgt-1578889.html, 2010. Accessed May 31, 2015.

[117] A. William Roscoe. CSP and determinism in security modelling. In *Proceedings of S&P*, pages 114–127. IEEE Computer Society Press, 1995.

[118] Roni Rosner. *Modular synthesis of reactive systems*. PhD thesis, Weizmann Institute of Science, 1992.

[119] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *Selected Areas in Communications, IEEE Journal on*, 21(1):5–19, Jan 2003.

[120] Andrei Sabelfeld and David Sands. Dimensions and principles of declassification. In *Proceedings of CSFW*, pages 255–269. IEEE Computer Society, 2005.

[121] Jerome H Saltzer and Michael D Schroeder. The protection of information in computer systems. *Proceedings of the Symposium on Operating System Principles*, 63(9):1278–1308, 1975.

[122] Stefan Schulz. Der Herzfehler. `http://www.faz.net/aktuell/feuilleton/debatten/ueberwachung/die-sicherheitsluecke-heartbleed-zeigt-wir-brauchen-mehr-internetsicherheit-12893695.html`, 2014. Accessed May 31, 2015.

[123] Atika Shubert. Cyber warfare: A different way to attack Iran's reactors. `http://www.cnn.com/2011/11/08/tech/iran-stuxnet/`, 2010. Accessed May 31, 2015.

[124] A. Prasad Sistla and Edmund M. Clarke. The complexity of propositional linear temporal logics. *J. ACM*, 32(3):733–749, 1985.

[125] A. Prasad Sistla, Moshe Y. Vardi, and Pierre Wolper. The complementation problem for Büchi automata with appplications to temporal logic. *TCS*, 49:217–237, 1987.

[126] Geoffrey Smith. On the foundations of quantitative information flow. In *Proceedings of FOSSACS*, pages 288–302. Springer, 2009.

[127] Larry Stockmeyer. *The complexity of decision problems in automata theory and logic*. PhD thesis, MIT, 1974.

[128] Michaël Szadkowski. Faille Heartbleed: les sites pour lesquels il est conseillé de changer son mot de passe. `http://www.lemonde.fr/technologies/article/2014/04/11/faille-heartbleed-les-sites-pour-lesquels-il-est-conseille-de-changer-son-mot-de-passe_4399564_651865.html`, 2014. Accessed May 31, 2015.

[129] Tachio Terauchi and Alex Aiken. Secure information flow as a safety problem. In *Proceedings SAS*, pages 352–367, 2005.

[130] Wolfgang Thielke. Code geknackt. `http://www.focus.de/finanzen/news/krankenkassen-code-geknackt_aid_148829.html`, 1994. Accessed Feb 6, 2015.

[131] Wolfgang Thomas. Star-free regular sets of ω-sequences. *Information and Control*, 42(2):148 – 156, 1979.

[132] Wolfgang Thomas. Safety- and liveness-properties in propositional temporal logic: characterizations and decidability. *Banach Center Publications*, 21(1):403–417, 0 1988.

[133] TrustedSec. Chs hacked via Heartbleed vulnerability. `https://www.trustedsec.com/august-2014/chs-hacked-heartbleed-exclusive-trustedsec/`, 2014. Accessed June 4, 2015.

[134] Alan Mathison Turing. On computable numbers, with an application to the entscheidungsproblem. *J. of Math*, 58(5):230–265, 1936.

[135] Ron van der Meyden. Axioms for knowledge and time in distributed systems with perfect recall. In *Proceedings of LICS*, pages 448–457, 1993.

[136] Ron van der Meyden and Thomas Wilke. Preservation of epistemic properties in security protocol implementations. In *Proceedings of TARK*, pages 212–221. ACM, 2007.

[137] Moshe Y. Vardi. Alternating automata and program verification. In Jan Leeuwen, editor, *Computer Science Today*, volume 1000 of *Lecture Notes in Computer Science*, pages 471–485. Springer, 1995.

[138] Moshe Y Vardi. An automata-theoretic approach to linear temporal logic. In *Logics for concurrency*, pages 238–266. Springer, 1996.

[139] Moshe Y. Vardi and Pierre Wolper. Reasoning about infinite computations. *Inf. Comput.*, 115(1):1–37, 1994.

[140] Dennis Volpano, Geoffrey Smith, and Cynthia E. Irvine. A sound type system for secure flow analysis. 1996.

[141] Klaus von Gleissenthall and Andrey Rybalchenko. An epistemic perspective on consistency of concurrent computations. In *CONCUR 2013–Concurrency Theory*, pages 212–226. Springer, 2013.

[142] Jane Wakefield. Heartbleed bug: What you need to know. `http://www.bbc.com/news/technology-26969629`, 2014. Accessed May 31, 2015.

[143] David A. Wheeler. How to prevent the next Heartbleed. `http://www.dwheeler.com/essays/heartbleed.html`, 2014 (updated 2015). Accessed June 1, 2015.

[144] P. Wolper, M. Y. Vardi, and A. P. Sistla. Reasoning about infinite computation paths. In *Proceedings of FOCS*, pages 185–194. IEEE Computer Society, 1983.

[145] Jean Yang, Kuat Yessenov, and Armando Solar-Lezama. A language for automatically enforcing privacy policies. In *ACM SIGPLAN Notices*, volume 47, pages 85–96. ACM, 2012.

[146] H. Yasuoka and T. Terauchi. On bounding problems of quantitative information flow. In *Proceedings of ESORICS*, pages 357–372. Springer, 2010.

[147] Aris Zakinthinos and E. Stewart Lee. A general theory of security properties. In *Proceedings of S&P*, pages 94–102. IEEE Computer Society Press, 1997.

[148] Steve Zdancewic and Andrew C. Myers. Observational determinism for concurrent program security. In *Proceedings of CSFW*, pages 29–43, 2003.

[149] Danfeng Zhang, Aslan Askarov, and Andrew C. Myers. Language-based control and mitigation of timing channels. In *Proceedings of PLDI*, pages 99–110, 2012.