SECURITY MODELS IN REWRITING LOGIC FOR CRYPTOGRAPHIC PROTOCOLS AND BROWSERS

BY

RALF SASSE

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2012

Urbana, Illinois

Abstract

This dissertation tackles crucial issues of web browser security. Web browsers are now a central part of the trusted code base of any end-user computer system, as more and more usage shifts to services provided by web sites that are accessed through those browsers. Towards this goal we identify three key aspects of web browser security: (i) the *machine-to-user communication*, (ii) *internal browser security concerns* and (iii) *machine-to-machine communication.*

We address aspects (i) and (ii) by developing a methodology that creates a formal model of a web browser and analyzes that model. We showcase this on the graphical user interface of both Internet Explorer and the Illinois Browser Operating System (IBOS) web browsers. Internal security aspects are addressed in the IBOS browser for the same origin policy.

For aspect (iii) we look at the formal analysis of cryptographic protocols, independent of any particular browser. We focus on the formal analysis of protocols *modulo algebraic properties* of their cryptographic functions, since it is well-known the protocol verification methods that ignore such algebraic properties using a standard Dolev-Yao model can verify as correct protocols that can be in fact broken using the algebraic properties. We adopt a symbolic approach and use the Maude-NPA cryptographic protocol analysis tool, which has extended unification capabilities modulo theories based on the new narrowing strategy we developed. We present case studies showing that appropriate protocols can be analyzed so that either attacks are found, or the absence of attacks can be proven.
Keywords: browser security, visual invariants, same origin policy, semantic unification, variant narrowing, cryptographic protocol analysis, rewriting logic.

ACKNOWLEDGMENT

I am grateful to all the welcoming, friendly and smart people that I have met over the course of my PhD program here at the Department of Computer Science at the University of Illinois at Urbana-Champaign. Among them are the fellow students, the professors and the whole academic and administrative staff of the department as well as members of the community that I was privileged to interact with.

Now I want to single out a few people for special thanks as they made this overall experience possible. First of all, I am extremely grateful to my advisor and friend, José Meseguer, for all his support, both technical and financial, as well as the challenges and opportunities he has provided me with over the years, and for instilling the ideals of scholarly pursuit in me by exemplifying them at all times.

I am also very thankful to my committee members Shuo Chen, Sam King, Catherine Meadows and Grigore Roșu for all their kindness and help throughout this work. I would also like to thank all my collaborators for the privilege of having worked with them, and being able to learn from them during that work.

I want to especially thank my parents, Heidrun Sasse and Fritz Sasse, for all their invaluable support and unceasing motivation and encouragement over the years.

Let me try to recall as best as I can all the people that have influenced this endeavour, and let me apologize in advance for any of the inevitable omissions that will slip through:

José Meseguer, Shuo Chen, Catherine Meadows, Sam King, Grigore Roșu, Mike Katelman, Camilo Rocha, Edgar Pek, Santiago Escobar, Peter Ölveczky, Raúl Gutiérrez, Timo Latvala, Artur Boronat, Joe Hendrix, Musab AlTurki, Kyungmin Bae, Azadeh Farzan, Mark Hills, Traian Șerbănuță, Andrei Ștefănescu, Jeff Green, Francisco Duran, Narciso Martí-Oliet, Rajesh

Kumar Karmani, Chucky Ellison, Patrick Meredith, Dennis Griffith, Felix Schernhammer, Beatriz Alarcón, Sonia Santiago, Alexandre Duchâteau, Jonas Eckhardt, Tobias Mühlbauer, Nana Arizumi, Miguel Palomino, Mark-Oliver Stehr, Steven Lauterburg, Pavithra Prabhakar, Stephen Skeirik, Rustan Leino, Wolfram Schulte, Helen Wang, Yi-Min Wang, Ravinder Shankesi, Shuo Tang, Deepak Kapur, Christopher Lynch, Paliath Narendran, Carl A. Gunter, Andrea Whitesell, Donna Coleman, Holly Bagwell, Kathy Runck, Mary Beth Kelley, Kay Tomlin, Wolfgang Ahrendt, Andreas Roth, Peter H. Schmitt, Heidrun Sasse, Fritz Sasse.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

A current trend in computing is to move applications to the web, i.e., instead of having an application installed on one's computer for a specific task, one goes to a website that offers that functionality as a service. Some popular examples are e-mail (Google Mail and others), banking and tax filing (TurboTax and others), and electronic commerce. These applications are obviously quite security critical as confidentiality of the information is supremely important. With these applications migrating to the web and being run as services, the web browser that is used to access these applications becomes a crucial part of the chain of trust for these services.

In some sense, the browser is becoming the operating system of the future. In the past the operating system was the most security critical component, with any breach of it potentially leading to disaster, but now the web browser has to be included in that critical set of code. However, most browsers have been developed with a limited emphasis on security. Browsers are historically built on top of prior browser versions and thus tend to be monolithic constructions and therefore quite large and complex. The chain of trust for services like those mentioned above can therefore be significantly strengthened by efforts aimed at improving browser security.

Browser security can be considered to have three aspects: (i) the *machine-to-user communication*, (ii) *internal browser security concerns* and (iii) the *machine-to-machine communication*.

In this dissertation we formally model and analyze two web browsers: Internet Explorer (IE) and the Illinois Browser Operating System (IBOS) with respect to security aspects (i) for IE and IBOS and (ii) for IBOS. We also present new contributions to the security aspect (iii) in a way that is browser generic by focusing on the formal analysis of the *cryptographic protocols* used to secure machine-to-machine communication.

For the *machine-to-user communication*, trusted visual cues need to indeed be (proven) trustworthy. Specifically, we analyze the correctness of the address bar with respect to the content of the current page in both Internet Explorer (IE) and the Illinois Browser Operating System (IBOS). In IE we find a number of possible attacks while we can show that IBOS is safe. In IE we go still further and analyze the status bar for static HTML as well. For the status bar of IE we find a number of potential attacks, but we also show how to make IE safe against them.

Address bar correctness is a very important property to help prevent certain kinds of phishing attacks, or at least make phishing more difficult. Phishing is the process in which an attacker tries to steal a user's credentials for any kind of service. To this end, the attacker prepares a website that looks exactly like the website of a real service, e.g., a bank's website. Then the attacker will send out a massive number of e-mails, asking users to log into the service at his copy of the website. If a user falls for this deception, the attacker gains access to that user's credentials, can manipulate his/her account, and potentially steal the user's money. With a correct address bar, the user has some defense, as it is possible to spot that the address bar shows some string that is not the URL of the web page expected. This of course only helps those users that are diligent in checking such indicators. Now, the status bar correctness of IE is relevant as well, as Outlook Express (an e-mail reader app) at the time used the same logic for showing the URL of a target link in an e-mail. So, if both the address bar and status bar can be manipulated, this leads to a perfect spoofing chain for the attacker, going from the e-mail, where checking the mouse-over link produces the right URL, to the website, which looks perfect, as it is a straight copy of the real one, to the address bar, which shows the same URL.

For *internal browser security* concerns we look at the *same origin policy* (SOP) and check it in IBOS. The SOP essentially states that one website cannot get information from another website through the browser, see [115] for more detail. A number of properties, taken together, is sufficient for SOP to hold. Some further properties are also considered.

For the *machine-to-machine communication* the underlying cryptographic protocols need to be secure. Even though the chain of trust for machine-to-machine communication uses cryptographic protocols in a ubiquitous fashion, the formal analysis of the security of cryptographic protocols is mostly based on the Dolev-Yao model [39]. It is a formal model in which legitimate participants are trying to execute a protocol together. There is also an adversary trying to learn all possible secrets to which it should not have access. The adversary is active, meaning it can send and receive messages as if it were a normal participant. However, it also controls the network, i.e., the adversary can read all messages, can drop messages that are inconvenient for its nefarious purposes, and can add messages to the network that look like they are from any one legitimate participant chosen by the adversary. The only limit placed on the adversary is that the underlying cryptography is perfect and thus the model treats it as a black box. The adversary cannot read the contents of encrypted messages or create encrypted messages, unless it knows the required key for that message.

In practice, though, the protocol functions will have some inherent algebraic properties that need to be accounted for. If the algebraic properties are not taken into account in the verification, then even a 'proven-secure' (according to the Dolev-Yao model) protocol can have exploitable vulnerabilities based on those properties [110]. For formal verification to be able to deal with such algebraic properties, the properties need to be explicitly modeled as part of the cryptographic protocol specification. So we are using an extension of the Dolev-Yao model in which some of the algebraic properties are explicitly modeled.

To improve on the state-of-the-art for the above-mentioned protocol security issues, two things are needed: (i) cryptographic protocol analysis methods and tools that can take those algebraic properties into account; and (ii) ways to actually deal with different algebraic properties, which for symbolic analysis purposes boils down to being able to perform unification modulo the relevant algebraic theories.

This dissertation develops a formal specification and verification methodology, based on rewriting logic and implemented in Maude, capable of analyzing and verifying, as well as finding bugs (i.e., counter-examples), in models of browser software and of cryptographic protocols used for machine-to-machine communication. Specifically, it addresses the three different aspects (i), (ii) and (iii) of browser security mentioned above as follows:

- **Browser to user connection by GUI.** We analyze GUI security (correctness!) aspects of both IE and IBOS.

- **Internal browser security.** We prove the same origin policy for

IBOS.

- **Machine-to-machine connection by cryptographic protocols.** We extend the cryptographic protocol analysis tool Maude-NPA with a new generic unification algorithm based on a new version of narrowing and show case studies on cryptographic protocols that are analyzed modulo their algebraic properties using our generic unification algorithm.

Web browser security is highly important as more and more applications are moved from the traditional desktop application framework to on-demand online versions (often called services) which are usually accessed through a web browser. The web browser is thus mutating to a kind of operating system for services. We consider security aspects first of a commercial off-the-shelf web browser (Internet Explorer (IE)) and the effects its graphical user interface (GUI) design can have on security. For IE we exclusively look at GUI properties. Then we turn to analyze the design of a new, state-of-the-art and secure-by-design web browser which is tightly integrated into a (proven) secure operating system microkernel (IBOS [117]). For IBOS we look at both GUI properties and internal browser security considerations.

This methodology consists of two steps: (i) creating a formal model out of given source code and code developer comments[1] and (ii) formal analysis of the model created in step (i).

Overall the analysis is based on browser models which have been extracted from the source code, with developer interaction to get at the intent.[2] This modeling process by itself has found many unknown types of attacks in IE and some bugs directly in IBOS, whenever code reading and developer intent did not match up as they should. The models of the browsers are given in rewriting logic and can thus be executed and analyzed using the Maude tool.

Browser to user connection by GUI. For the GUI analysis work the

[1]This step, by being a passage from an informal description to a formal model, is not itself amenable to formal verification; however, the executable nature of the formal specification so obtained makes it amenable to thorough testing to ensure that the right formal model has been captured.

[2]In the case of Internet Explorer, since the source code is proprietary, we relied on one of our collaborators in that work (Shuo Chen) to first extract pseudocode models from IE's source code, and based our Maude models on such pseudocode. Instead, in the case of the IBOS browser we were able to directly study the source code and to have extensive discussions with the developers.

particular concern was the browser's *address bar*, which should always be guaranteed to truthfully tell the user what website is currently being viewed. The address bar has been analyzed for both IE and IBOS. The *status bar* is also of interest whenever navigation to a new page is considered and should be a good indicator of what the contents of the next web page (and address bar) should be. The status bar has only been analyzed for IE.

Internal browser security. In the analysis of the newly developed IBOS browser we looked at the *same origin policy* (SOP) that says that a website cannot get user information from another website through the browser. This is necessary so one web page cannot steal login information or any other credentials that the user is presenting to a different web page. The SOP can be broken down into a set of properties that taken together imply SOP. We have done that and we show all the components and how they are verified. We conclude that IBOS fulfills SOP. As already stated above, the address bar in IBOS has also been under scrutiny.

Protocol analysis. For the analysis of cryptographic protocols we develop new unification methods. This is to improve the capabilities of a crypt-analysis tool, the Maude-NPA [48], which we apply to case-studies. The analysis of cryptographic protocols we are interested in takes into account underlying algebraic properties of the protocol to be analyzed and is not restricted to just the Dolev-Yao model, for which the whole cryptography is a black box. This requires a modeling framework, as well as tool support, that is capable of handling these algebraic properties of interest. This deeper level of analysis is needed in practice as protocols shown secure under Dolev-Yao have been broken using the algebraic properties of their cryptographic operations [110]. In this dissertation we focus on two aspects of this: to be able to do protocol analysis modulo algebraic properties, we develop a generic algorithm for unification modulo theories which is based on narrowing, and second, we show some case studies on protocols for which our new generic unification algorithm is needed for the analysis, since the protocols in question could not be analyzed properly before. Development of the above-mentioned unification modulo a theory has been integrated into the Maude-NPA cryptographic protocol analysis tool, which is used in the case study carried out subsequently.

1.1 Summary of Contributions

This dissertation contributes to several ongoing research efforts within the areas of formal methods, web browser security, and cryptographic protocol analysis. This list highlights the main contributions of the dissertation:

1. A methodology for web browser formal modeling and analysis which is effective in finding real-life bugs that allow attacks, while increasing confidence in the absence of attacks.

2. In the Internet Explorer browser graphical user interface we find 9 attacks in the form of HTML tree types for status bar spoofing, and 4 actual attack types for address bar spoofing.

3. In the IBOS browser we verify the same origin policy and the consistency of the address bar. We also find a bug in the display memory management leading to browser tabs being unusable.

4. A new generic method for the effective computation of unifiers modulo equational theories, based on narrowing modulo axioms, called *folding variant narrowing*.

5. An automatic method to check whether a given equational theory has the finite variant property.

6. Applications of folding variant narrowing in the cryptographic protocol analysis tool Maude-NPA. This very generic way of obtaining finitary equational unification algorithms by narrowing and folding variant narrowing has further automated deduction applications.

7. A case study for cryptographic protocol analysis modulo exclusive-or and other cryptographic equational properties. Attacks are found in insecure protocols, while secure protocols can be proved to be so by our methods.

1.2 Organization

The dissertation is organized into chapters as follows:

- *Chapter 1.* Identifies problems this thesis is addressing and motivates at a high-level the approach being used.

- *Chapter 2.* Discusses technical preliminaries required for the rest of the dissertation.

- *Chapter 3.* Looks at the Internet Explorer graphical user interface, models the status bar and address bar and analyzes their properties. Finds multiple attacks on both and proposes fixes.

- *Chapter 4.* In this chapter IBOS is motivated, described, and modeled. In a number of case studies security properties of IBOS are proven and bugs are exposed.

- *Chapter 5.* Considers narrowing strategies and optimal variant termination before introducing folding variant narrowing. The finite variant property is explained and automatic checking is considered. This leads to equational unification algorithms based on variants.

- *Chapter 6.* Analyzes protocols modulo the combination of some theories. In particular, introduces specification and analysis of a protocol in the Maude-NPA and the unification algorithm based on the prior chapter. Three protocols are analyzed and attacks in them are found, one of the protocols is then fixed and shown that no further attacks remain.

- *Chapter 7.* This chapter presents the conclusions of the dissertation and discusses future research directions.

Noting that (i), (ii) and (iii) are quite different in nature we are presenting the related work as it is appropriate in each Chapter of this dissertation. See Sections 3.5, 4.4, 5.9, and 6.4.

CHAPTER 2

PRELIMINARIES

In this thesis, we follow the classical notation and terminology from [119] for term rewriting, and from [89] for rewriting logic and order-sorted notions. We assume an order-sorted signature $\mathbf{\Sigma} = (\mathsf{S}, \leq, \Sigma)$ with poset of sorts $(\mathsf{S}, \leq)$ and such that for each sort $\mathsf{s} \in \mathsf{S}$ the connected component of s in $(\mathsf{S}, \leq)$ has a top sort, denoted $[\mathsf{s}]$, and all $f : \mathsf{s_1} \cdots \mathsf{s_n} \to \mathsf{s}$ with $n \geq 1$ have a top sort overloading $f : [\mathsf{s_1}] \cdots [\mathsf{s_n}] \to [\mathsf{s}]$. We also assume an S-sorted family $\mathcal{X} = \{\mathcal{X}_\mathsf{s}\}_{\mathsf{s} \in \mathsf{S}}$ of disjoint variable sets with each $\mathcal{X}_\mathsf{s}$ countably infinite. $\mathcal{T}_\Sigma(\mathcal{X})_\mathsf{s}$ is the set of terms of sort s, and $\mathcal{T}_{\Sigma,\mathsf{s}}$ is the set of ground terms of sort s. We write $\mathcal{T}_\Sigma(\mathcal{X})$ and $\mathcal{T}_\Sigma$ for the corresponding order-sorted term algebras. For a term t, $\mathit{Var}(t)$ denotes the set of all variables in t.

Positions are represented by sequences of natural numbers denoting an access path in the term when viewed as a tree. The top or root position is denoted by the empty sequence Λ. We define the relation $p \leq q$ between positions as $p \leq p$ for any p; and $p \leq p.q$ for any p and q. Given $U \subseteq \Sigma \cup \mathcal{X}$, $\mathit{Pos}_U(t)$ denotes the set of positions of a term t that are rooted by symbols or variables in U. The set of positions of a term t is written $\mathit{Pos}(t)$, and the set of non-variable positions $\mathit{Pos}_\Sigma(t)$. The subterm of t at position p is $t|_p$ and $t[u]_p$ is the term t where $t|_p$ is replaced by u.

A *substitution* $\sigma \in \mathcal{S}\mathit{ubst}(\Sigma, \mathcal{X})$ is a sorted mapping from a finite subset of $\mathcal{X}$ to $\mathcal{T}_\Sigma(\mathcal{X})$. Substitutions are written as $\sigma = \{X_1 \mapsto t_1, \ldots, X_n \mapsto t_n\}$ where the domain of σ is $\mathit{Dom}(\sigma) = \{X_1, \ldots, X_n\}$ and the set of variables introduced by terms $t_1, \ldots, t_n$ is written $\mathit{Ran}(\sigma)$. The identity substitution is *id*. Substitutions are homomorphically extended to $\mathcal{T}_\Sigma(\mathcal{X})$. The application of a substitution σ to a term t is denoted by $t\sigma$. For simplicity, we assume that every substitution is idempotent, i.e., σ satisfies $\mathit{Dom}(\sigma) \cap \mathit{Ran}(\sigma) = \emptyset$. Substitution idempotency ensures $t\sigma = (t\sigma)\sigma$. The restriction of σ to a set of variables V is $\sigma|_V$; sometimes we write $\sigma|_{t_1,\ldots,t_n}$ to denote $\sigma|_V$ where $V = \mathit{Var}(t_1) \cup \cdots \cup \mathit{Var}(t_n)$. Composition of two substitutions is denoted by $\sigma\sigma'$. Combination of two substitutions is denoted by $\sigma \cup \sigma'$. We call an idempotent substitution σ a variable *renaming* if there is another idempotent substitution σ^{-1} such that $(\sigma\sigma^{-1})|_{Dom(\sigma)} = id$.

A Σ-*equation* is an unoriented pair $t = t'$, where $t, t' \in \mathcal{T}_\Sigma(\mathcal{X})_\mathsf{s}$ for some sort $\mathsf{s} \in \mathsf{S}$. Given Σ and a set $\mathcal{E}$ of Σ-equations, order-sorted equational logic induces a congruence relation $=_\mathcal{E}$ on terms $t, t' \in \mathcal{T}_\Sigma(\mathcal{X})$ (see [90]). Throughout this thesis we assume that $\mathcal{T}_{\Sigma,\mathsf{s}} \neq \emptyset$ for every sort s, because this affords a simpler deduction system. An *equational theory* $(\Sigma, \mathcal{E})$ is a pair with Σ an order-sorted signature and $\mathcal{E}$ a set of Σ-equations.

The $\mathcal{E}$-*subsumption* preorder $\sqsubseteq_\mathcal{E}$ (or just $\sqsubseteq$ if $\mathcal{E}$ is understood) holds between $t, t' \in \mathcal{T}_\Sigma(\mathcal{X})$, denoted $t \sqsubseteq_\mathcal{E} t'$ (meaning that t' is *more general* than t modulo $\mathcal{E}$), if there is a substitution σ such that $t =_\mathcal{E} t'\sigma$; such a substitution σ is said to be an $\mathcal{E}$-*match* from t to t'. The $\mathcal{E}$-renaming equivalence $t \approx_\mathcal{E} t'$, holds if there is a variable renaming θ such that $t\theta =_\mathcal{E} t'$. We write $t \sqsubset_\mathcal{E} t'$ if $t \sqsubseteq_\mathcal{E} t'$ and $t \not\approx_\mathcal{E} t'$. Relations $\approx_\mathcal{E}$ and $\sqsubset_\mathcal{E}$ are extended to substitutions in a similar way. For substitutions σ, ρ and a set of variables V we define $\sigma|_V =_\mathcal{E} \rho|_V$ if $x\sigma =_\mathcal{E} x\rho$ for all $x \in V$; $\sigma|_V \sqsubseteq_\mathcal{E} \rho|_V$ if there is a substitution η such that $\sigma|_V =_\mathcal{E} (\rho\eta)|_V$; and $\sigma|_V \approx_\mathcal{E} \rho|_V$ if there is a renaming η such that $(\sigma\eta)|_V =_\mathcal{E} \rho|_V$. We write $\sigma \sqsubset_\mathcal{E} \sigma'$ if $\sigma \sqsubseteq_\mathcal{E} \sigma'$ and $\sigma \not\approx_\mathcal{E} \sigma'$.

An $\mathcal{E}$-*unifier* for a Σ-equation $t = t'$ is a substitution σ such that $t\sigma =_\mathcal{E} t'\sigma$. For $Var(t) \cup Var(t') \subseteq W$, a set of substitutions $CSU^W_\mathcal{E}(t = t')$ is said to be a *complete* set of unifiers for the equation $t = t'$ modulo $\mathcal{E}$ away from W iff: (i) each $\sigma \in CSU^W_\mathcal{E}(t = t')$ is an $\mathcal{E}$-unifier of $t = t'$; (ii) for any $\mathcal{E}$-unifier ρ of $t = t'$ there is a $\sigma \in CSU^W_\mathcal{E}(t = t')$ such that $\rho|_W \sqsubseteq_\mathcal{E} \sigma|_W$; (iii) for all $\sigma \in CSU^W_\mathcal{E}(t = t')$, $Dom(\sigma) \subseteq (Var(t) \cup Var(t'))$ and $Ran(\sigma) \cap W = \emptyset$. If the set of variables W is irrelevant or is understood from the context, we write $CSU_\mathcal{E}(t = t')$ instead of $CSU^W_\mathcal{E}(t = t')$. An $\mathcal{E}$-unification algorithm is *complete* if for any equation $t = t'$ it generates a complete set of $\mathcal{E}$-unifiers. Note that this set needs not be finite. A unification algorithm is said to be *finitary* and complete if it always terminates after generating a finite and complete set of solutions. A unification algorithm is said to be *minimal* if it always provides a maximal (w.r.t. $\sqsubseteq_\mathcal{E}$) set of unifiers, i.e., for any two unifiers $\rho_1, \rho_2 \in CSU^W_\mathcal{E}(t = t')$ such that $\rho_1|_W \neq_\mathcal{E} \rho_2|_W$, we have that $\rho_1|_W \not\sqsubseteq_\mathcal{E} \rho_2|_W$ and $\rho_2|_W \not\sqsubseteq_\mathcal{E} \rho_1|_W$.

A *rewrite rule* is an oriented pair $l \to r$, where $Var(r) \subseteq Var(l)$ and $l, r \in \mathcal{T}_\Sigma(\mathcal{X})_\mathsf{s}$ for some sort $\mathsf{s} \in \mathsf{S}$. An *(unconditional) order-sorted rewrite* *theory* is a triple (Σ, Ax, R) with Σ an order-sorted signature, Ax a set of Σ-equations, and R a set of rewrite rules. The rewriting relation on $\mathcal{T}_\Sigma(\mathcal{X})$, written $t \to_R t'$ or $t \to_{p,R} t'$ holds between t and t' iff there exist $p \in Pos_\Sigma(t)$, $l \to r \in R$ and a substitution σ, such that $t|_p = l\sigma$, and $t' = t[r\sigma]_p$. The subterm $t|_p$ is called a *redex.* The relation $\to_{R/Ax}$ on $\mathcal{T}_\Sigma(\mathcal{X})$ is $=_{Ax}; \to_R; =_{Ax}$. Note that $\to_{R/Ax}$ on $\mathcal{T}_\Sigma(\mathcal{X})$ induces a relation $\to_{R/Ax}$ on the free (Σ, Ax)-algebra $\mathcal{T}_{\Sigma/Ax}(\mathcal{X})$ by $[t]_{Ax} \to_{R/Ax} [t']_{Ax}$ iff $t \to_{R/Ax} t'$. The transitive (resp. transitive and reflexive) closure of $\to_{R/Ax}$ is denoted $\to^+_{R/Ax}$ (resp. $\to^*_{R/Ax}$). We say that a term t is $\to_{R/Ax}$-irreducible (or just R/Ax-irreducible) if there is no term t' such that $t \to_{R/Ax} t'$.

For a rewrite rule $l \to r$, we say that it is *sort-decreasing* if for each substitution σ, we have $r\sigma \in \mathcal{T}_\Sigma(\mathcal{X})_{\mathsf{s}}$ implies $l\sigma \in \mathcal{T}_\Sigma(\mathcal{X})_{\mathsf{s}}$. We say a rewrite theory (Σ, Ax, R) is sort-decreasing if all rules in R are. For a Σ-equation $t = t'$, we say that it is *regular* if $Var(t) = Var(t')$, and it is *sort-preserving* if for each substitution σ, we have $t\sigma \in \mathcal{T}_\Sigma(\mathcal{X})_{\mathsf{s}}$ implies $t'\sigma \in \mathcal{T}_\Sigma(\mathcal{X})_{\mathsf{s}}$ and vice versa. We say an equational theory $(\Sigma, \mathcal{E})$ is regular, resp. sort-preserving, if all equations in $\mathcal{E}$ are.

For substitutions σ, ρ and a set of variables V we define $\sigma|_V \to_{R/Ax} \rho|_V$ if there is $x \in V$ such that $x\sigma \to_{R/Ax} x\rho$ and for all other $y \in V$ we have $y\sigma =_{Ax} y\rho$. A substitution σ is called *R/Ax-normalized* (or normalized) if $x\sigma$ is R/Ax-irreducible for all $x \in V$.

We say that the relation $\to_{R/Ax}$ is *terminating* if there is no infinite sequence $t_1 \to_{R/Ax} t_2 \to_{R/Ax} \cdots t_n \to_{R/Ax} t_{n+1} \cdots$. We say that the relation $\to_{R/Ax}$ is *confluent* if whenever $t \to^*_{R/Ax} t'$ and $t \to^*_{R/Ax} t''$, there exists a term t''' such that $t' \to^*_{R/Ax} t'''$ and $t'' \to^*_{R/Ax} t'''$. An order-sorted rewrite theory (Σ, Ax, R) is confluent (resp. terminating) if the relation $\to_{R/Ax}$ is confluent (resp. terminating). In a confluent, terminating, sort-decreasing, order-sorted rewrite theory, for each term $t \in \mathcal{T}_\Sigma(\mathcal{X})$, there is a unique (up to Ax-equivalence) R/Ax-irreducible term t' obtained from t by rewriting to canonical form, which is denoted by $t \to^!_{R/Ax} t'$, or $t{\downarrow}_{R/Ax}$ when t' is not relevant.

2.1 The Maude Tool

The *Maude* tool [31] is a high-performance implementation of rewriting logic. It allows equational specification in *functional modules*, corresponding to equational theories $(\Sigma, E \cup Ax)$, and full rewrite theories $\mathcal{R} = (\Sigma, E \cup Ax, R)$ can be specified as *system modules.* In functional modules other modules can be included, sorts and subsorts can be declared and operator symbols can be defined, possibly with equational attributes (called axioms) like associativity, commutativity and/or identity. Sorts, subsorts, conditional equations and memberships define the computations that are possible. Reasonable executability requirements are needed to make a module *admissible* (see [31], Sections 4.6 and 6.3), including termination (modulo axioms), ground confluence and sort-decreasingness. Then, Maude can execute the module by equational simplification modulo the axioms, where the equations in E are used as rules from left to right and Maude's built-in matching for the axioms Ax leads for each term t to its canonical form with a least sort. For functional modules this yields an operational semantics, defined by the algebra of canonical forms $Can_{\Sigma/E\cup Ax}$ corresponding to the initial algebra semantics given by $T_{\Sigma/E\cup Ax}$ (see Sections 4.6-4.8 in [31]). Equational simplification modulo axioms is executed by the `reduce` command in Maude.

In order to be admissible, a system module has to, in addition to its equational component being admissible, satisfy the ground coherence requirement of its rules R with respect to equations in E and also needs to ensure that all variables in rules can be instantiated by (incremental) matching. Such a module can be executed in Maude by rewriting with the rules and oriented equations modulo axioms Ax. This is exactly the mathematical semantics of $\mathcal{R}$. Rewrites in a system module are performed in Maude by the `rewrite` command which is position-fair and rule-fair. There is also breadth-first search available using the `search` command. A linear temporal logic (LTL) model checker is built-in for safety and liveness properties.

2.2 Maude-NPA

The Maude-NPA [48] is a cryptographic protocol specification and analysis engine based on rewriting logic and implemented in the Maude tool. It supports cryptographic protocol analysis dealing explicitly with algebraic properties. See Chapters 5 and 6 for more details.

2.3 Internet Explorer

Internet Explorer (IE) is a widely used web browser created by Microsoft. Because of its wide adoption, IE has been a prime target for security attacks, leading to many well-known security violations. The analysis in this thesis has been executed on IE version 6.5 and had a substantial impact on IE version 7. The browser is written in a monolithic style in which just about any component can interact with any other component, and in practice does so, mostly for historical reasons.

2.4 Illinois Browser Operating System

The Illinois Browser Operating System (IBOS) [117] is a modern, security-conscious web browser designed at the University of Illinois and is integrated into a secure operating system. The basic idea is to go away from the monolithic approach and modularize the different processes of the browser. There is only one truly trusted process, the kernel. All other processes, like, e.g., web page instances, network processes, storage, etc., are not trusted. Security of all non-compromised components is desired, even with some compromised components in the mix. For that reason, all communication has to go through the kernel, which will allow or disallow it based on defined policies. See Chapter 4 for more details.

CHAPTER 3

GUI LOGIC ANALYSIS FOR IE

This chapter is based on joint work with Shuo Chen, José Meseguer, Helen Wang and Yi-Min Wang and has been partially published in [23]. To achieve end-to-end security, traditional machine-to-machine security measures are insufficient if the integrity of the human-computer interface is compromised. GUI logic flaws are a category of software vulnerabilities that result from logic bugs in GUI design/implementation. Visual spoofing attacks that exploit these flaws can lure even security conscious users to perform unintended actions. The focus of this chapter is to formulate the problem of GUI logic flaws and to develop a methodology for uncovering them in software implementations. Specifically, based on an in-depth study of key subsets of Internet Explorer (IE) browser source code, we have developed in [23] a formal model for the browser GUI logic and have applied formal reasoning to uncover new spoofing scenarios, including nine for status bar spoofing and four for address bar spoofing. The IE development team has confirmed all these scenarios and has fixed most of them in their latest build. Through this work, we demonstrate that a crucial subset of visual spoofing vulnerabilities originate from GUI logic flaws, which have a well-defined mathematical meaning allowing a systematic analysis.

Today, the trustworthiness of the web relies on the use of machine-to-machine security protocols (e.g., SSL or TLS) to provide authentication over the Internet to ensure that the client software (i.e., the browser) communicates with the intended server. However, such trustworthiness can be easily shattered by the last link between the client machine and its user. Indeed, the user-interface trust should be considered as a part of the trusted path problem in secure communications [45, 59, 128].

The exposure of the weakness between computer and human is not limited to non-technical social engineering attacks where naive users are fooled into clicking on an arbitrary hyperlink and download malicious executables

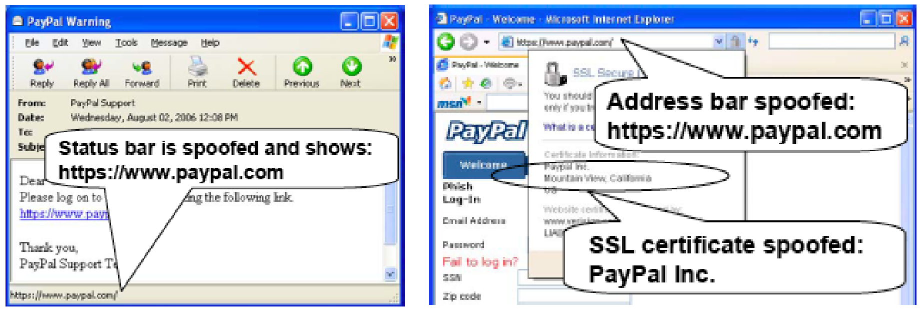

PayPal Warning
Status bar is spoofed and shows: https://www.paypal.com
https://www.paypal.com/
PayPal - Welcome - Microsoft Internet Explorer
Address bar spoofed: https://www.paypal.com
SSL certificate spoofed: PayPal Inc.

Figure 3.1: (a) Status Bar Spoofing - (b) Address Bar Spoofing

without any security awareness.

Even for a technology-savvy and security-conscious user, this last link can be spoofed visually. As shown in Figure 3.1(a), even if a user examines the status bar of the email client before she clicks on a hyperlink, she will not be able to tell that the status bar is spoofed and she will navigate to an unexpected web site, instead of `https://www.paypal.com`. Furthermore, as shown in Figure 3.1(b), even if a user checks the correspondence between the URL displayed in the browser address bar and the top level web page content, she will not realize that the address bar is spoofed and the page comes from a malicious web site. Indeed, the combination of the email status bar spoofing and the browser address bar spoofing can give a rather "authentic" navigation experience to a faked PayPal page. Even SSL is not helpful - as shown in Figure 3.1(b), the spoofed page contains a valid PayPal certificate. Obviously, this can result in many bad consequences, such as identity theft (e.g. phishing), malware installation, and spreading of faked news.

Visual spoofing attack is a generic term referring to any technique using a misleading GUI to gain trust from the user. Design/implementation flaws enabling such attacks are already a reality and have been sporadically discovered in commodity browsers [18, 19, 20], including IE, Firefox, and Netscape Navigator. This chapter focuses on a class of visual spoofing attacks that exploit GUI logic flaws, which are bugs in the GUI's design/implementation that allow the attacker to present incorrect information in parts of the authentic GUI that the user trusts, such as the email client status bar and the browser address bar. Figure 3.1(a) and (b) are just two instances of many such flaws that we discovered using the methodology described in this chapter, which expands the ideas presented in [23].

A second class of visual spoofing attack, which has been extensively discussed in previous research work [37, 59, 126, 128], is to exploit graphical similarities. These attacks exploit picture-in-picture rendering [128] (i.e., a faked browser window drawn inside a real browser window), chromeless window (e.g., a window without the address bar or the status bar [59, 128]), popup window covering the address bar, and symbol similarity (e.g., "1" vs. "l", "vv" vs. "w" [37], and non- English vs. English characters). We do not consider such attacks in this chapter, but in Section 3.4 we briefly discuss how the graphical similarity problems are being addressed by researchers and browser vendors.

Our goal is to formulate the GUI logic problem and to develop a systematic methodology for uncovering logic flaws in GUI implementations. This is analogous to the body of work devoted to catching software implementation flaws, such as buffer overruns, data races, and deadlocks, through the means of static analysis or formal methods. Nevertheless, a unique challenge in finding GUI logic flaws is that these flaws are about what the user sees — user's vision and actions are integral parts of the spoofing attacks. Thus, the modeled system should include not only the GUI logic itself, but also how the user interacts with it.

In a nutshell, our methodology first requires mapping a *visual invariant*, such as "the URL that a user navigates to must be the same as that indicated on the status bar when the mouse hovers over an element in a static HTML page", to a well-defined *program invariant*, which is a Boolean condition about *user state and software state*. This mapping is done based on an in-depth understanding of the source code of the software. Our goal is then to discover all possible inputs to the software which can cause the visual invariant to be violated. In the example of finding status bar spoofing scenarios, we want to discover all HTML document tree structures that can cause the inconsistency between the URL indicated on the status bar and the URL that the browser is navigating to upon a click event; the resulting HTML tree structures can be used to craft instances of status bar spoofing attacks. To systematically derive these scenarios, we employ a formal reasoning tool to reason about the well-defined program invariant.

The methodology is applied to discover two classes of important GUI logic flaws in IE. The first class is the static-HTML-based *status-bar spoofing*. Flaws of this class are critical because static-HTML pages (i.e., pages without scripts) are considered safe to be rendered in email clients (e.g., Outlook[1] and Outlook Express) and to be hosted on blogging sites and social networking sites (e.g., myspace.com), and the status bar is the only trustworthy information source for the user to see the target of a hyperlink. The second class of flaws we studied is *address bar spoofing*, which allows a malicious web site to hide its true URL and pretend to be a benign site. In both case studies, we use the Maude formal reasoning tool [29] to derive these spoofing scenarios, taking as input the browser GUI logic, program invariants, and user actions.

We have discovered nine canonical HTML tree structures leading to status bar spoofing and four scenarios of address bar spoofing. The IE development team has confirmed these scenarios and fixed eleven of them in the latest build, and scheduled to fix the remaining two in the next version. In addition to finding these flaws, we made the interesting observation that many classic programming errors, such as semantic composition errors, atomicity errors and race conditions are also manifested in the context of GUI implementation. More importantly, this chapter demonstrates that GUI logic flaws can be expressed in well-defined Boolean invariants, so that finding these flaws can be done by exhaustively searching for violations of the Boolean invariants.

The rest of the chapter is organized as follows. Section 3.1 gives an overview of our methodology. Sections 3.2 and 3.3 present case studies on status bar spoofing and address bar spoofing with IE. Section 3.4 discusses issues related to GUI security. Related work is discussed in Section 3.5.

3.1 Overview of Our Methodology

Figure 3.2 gives a big picture overview of our approach, based on formal analysis techniques. Existing formal analysis techniques have been successfully applied to reasoning about program invariants, e.g., the impossibility of buffer overrun in a program, guaranteed mutual exclusion in an algorithm, deadlock freedom in a concurrent system, secrecy in a cryptographic proto-

[1]Outlook does not show the target URL on the status bar, but on a small yellow tooltip near the mouse cursor. Because IE, Outlook and Outlook Express use the same HTML engine, most status bar spoofing scenarios can be transformed to email format to spoof Outlook tooltip and Outlook Express status bar.

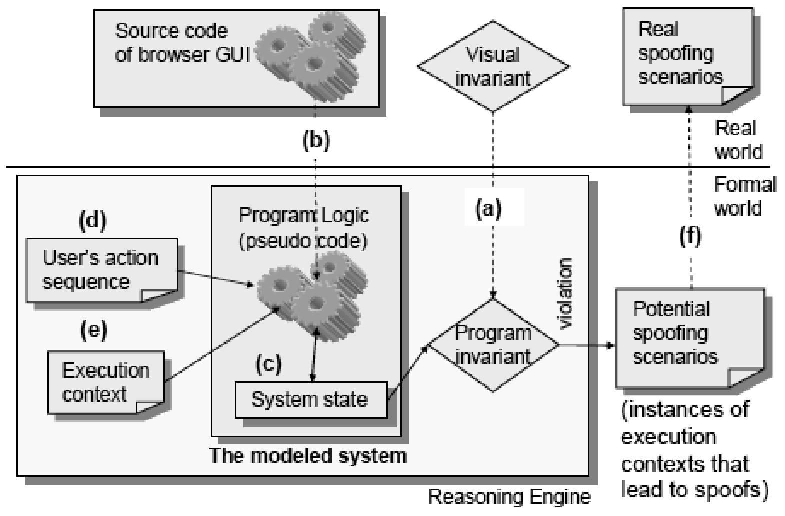


Figure 3.2: Overview of Our Methodology

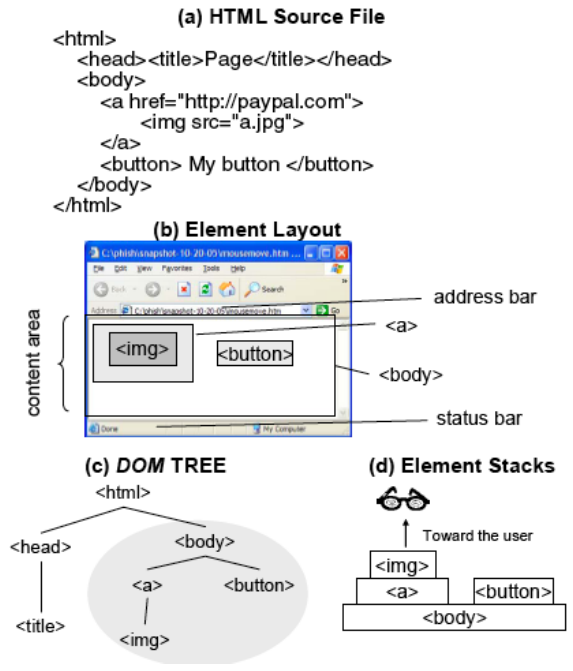
col, and so on. These program invariants have well-defined mathematical meaning. Uncovering GUI logic flaws, on the other hand, requires reasoning about what the user sees. The "invariant" in the user's vision does not have an immediately obvious mathematical meaning. For example, the visual invariant of the status bar is that if the user sees foo.com on the status bar before a mouse click, then the click must navigate to the foo.com page. It is important to map such a visual invariant to a program invariant in order to apply formal reasoning, which is shown as step (a) in Figure 3.2.

The mapping between a visual invariant and a program invariant is determined by the logic of the GUI implementation, e.g., a browser's logic for mouse handling and page loading. An in-depth understanding of the logic is crucial in deriving the program invariant. Towards this goal, we conducted an extensive study of the source code of the IE browser to extract pseudo code to capture the logic (shown as step (b)) which was then specified as a rewrite theory in Maude. In addition, we needed to explicitly specify the "system state" (shown as step (c)), including both the browser's internal state and possibly what the user memorizes. Steps (d) and (e) depict the formalization of the user's action sequence and the execution context as the inputs to the program logic. The user's action sequence is an important component in the GUI logic problem. For example, the user may move and click the mouse, or open a new page. Each action can change the system state. To capture this we need to make an abstraction of the user's capabilities as well as be exhaustive w.r.t. the remaining possible user action combinations. Another input to specify is the execution context of the system, e.g., a web page is an execution context for the mouse handling logic — the same logic and the same user action, when executed on different web pages, can produce different results. Again some abstraction and then exhaustive generation is required.

When the user action sequence, the execution context, the program logic, the system state and the program invariant are formally specified on the reasoning engine, formal reasoning is performed to check if the user action sequence applied to the system running in the execution context violates the program invariant. Each discovered violation is output as a potential spoofing scenario, which consists of the user action sequence, the execution context and the inference steps leading to the violation. Finally, we manually map each potential spoofing scenario back to a real-world scenario (shown as step (f)). This involves an effort to construct a malicious web page that sets up the execution context and lures the user to perform the actions. The mappings (a)(b)(f) between the real world and the formal model are currently done manually, some of which require significant effort. In this chapter, our contribution is mainly to formalize the GUI logic problem. Reducing the manual effort is future work.

3.2 Case Study 1: Status Bar Spoofing Based on Static HTML

Many web attacks, such as browser buffer overruns, cross-site scripting attacks, browser crossframe attacks and phishing attacks, require the user to navigate to a malicious URL. Therefore, it is important for the user to know the target URL of a navigation, which is displayed on the status bar before the user clicks the mouse. Status bar spoofing is damaging if it can be constructed using only static HTML (i.e., without any active content such as JavaScript), because: (i) email clients, e.g., Outlook and Outlook Express, render static HTML contents only, and email is an important media to propagate malicious messages; (ii) blogging sites and social networking sites (e.g., *myspace.com*) usually sanitize user-posted contents to remove scripts, but



```
<html>
  <head><title>Page</title></head>
  <body>
    <a href="http://paypal.com">
        <img src="a.jpg">
    </a>
    <button> My button </button>
  </body>
</html>
```

Figure 3.3: DOM Tree and Layout of an HTML Page

allow static HTML contents.[2]

3.2.1 Background: Representation and Layout of an HTML Page

Background knowledge about HTML representation is a prerequisite for this case study. We give a brief tutorial here. An HTML page is represented as a tree structure, namely a *Document Object Model* tree, or *DOM tree.* Figure 3.3 shows an HTML source file, its DOM tree, and the layout of the page. The mapping from the source file (Figure 3.3(a)) to the DOM tree (Figure 3.3(c)) is straightforward — element A enclosing element B is represented by A being the parent of B in the DOM tree. The tree root is a `<html>` element, which has a `<head>` subtree and a `<body>` subtree. The `<body>` subtree is rendered in the browser's content area. Since status bar spoof is caused by user interactions with the content area, we focus on the `<body>` subtree in this case study.

[2]A status bar spoof using a script is not a major security concern — it gets into a chicken-and-egg situation: a well-known site does not run an arbitrary script supplied from an arbitrary source. If the victim user has already been lured to a malicious site, the goal of the spoofing has been achieved.

Figure 3.3(b) shows the layouts of elements from the user's viewpoint. In general, parent elements have larger layouts to contain children elements. Conceptually, these elements are stacked upwards (toward the user), with `<body>` sitting at the bottom (see Figure 3.3(d)). In HTML, `<a>` represents an anchor, and `<img>` represents an image.

3.2.2 Program Logic of Mouse Handling and Status Bar Behavior

Mouse handling logic plays an important role in status bar spoofs. We extracted the logic from the IE source code. It is presented here using pseudo code, which will be formalized into a rewrite theory in Section 3.2.3.

Central Logic

The mouse device can generate several raw messages. When a user moves the mouse onto an element and clicks on it, the sequence of raw messages consists of several MOUSEMOVEs, an LBUTTONDOWN (i.e., left button down), and then a LBUTTONUP (i.e., left button up).

The core functions for mouse handling are called `OnMouseMessage` and `SendMsgToElem`, which dispatch mouse messages to appropriate elements. Every element has its specific virtual functions `HandleMessage`, `DoClick` and `ClickAction` to implement the element's behaviors.

Each raw mouse message invokes an `OnMouseMessage` call (pseudo code shown in Table 3.1). The parameter `element` is the HTML element that is immediately under the mouse cursor. The parameter `message` is the type of the message, which can be either MOUSEMOVE, or LBUTTONDOWN, or LBUTTONUP. An `OnMouseMessage` call can potentially send three messages to HTML elements in the DOM tree: (i) if element is different from `elementLastMouseOver`, which is the element immediately under the mouse in the most recent `OnMouseMessage` call, then a MOUSELEAVE message is sent to `elementLastMouseOver`; (ii) the raw message itself (i.e., `message`) is sent to `element`; (iii) if `element` is different from `elementLastMouseOver`, a MOUSEOVER message is sent to `element`.

In the function `SendMsgToElem()`, `btn` is the closest `Button` ancestor of `element`. If `btn` exists and `message` is LBUTTONUP (i.e., a click),

```
OnMouseMessage(x,y,message) {
 element=HitTestPoint(x,y)
 if (element!= elementLastMouseOver)
    PumpMessage(MOUSELEAVE, elementLastMouseOver)
  PumpMessage(message, element)
  if (element!= elementLastMouseOver)
    PumpMessage(MOUSEOVER, element)
 elementLastMouseOver = element
}


PumpMessage(message,element) {
 if (message != LBUTTONUP)
      element->FireJavaScriptNonClick(message)
 loopElement=element
 repeat
    BubbleCanceled =
      loopElement->HandleMessage(message)
    loopElement = loopElement->parent
 until BubbleCanceled or loopElement is the root
 if (message == LBUTTONUP)
     element->DoClick() //handle mouse single click
}
```

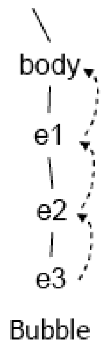


Table 3.1: `OnMouseMessage` and `SendMsgToElem`

then `element` becomes the button `btn`. It essentially means that any click on a descendant of a button is treated as a click on the button. Then, a *message bubbling* loop begins - starting from element, the virtual function `HandleMessage` of every element along the DOM tree path is invoked. Each `HandleMessage` call can cancel or continue the bubble (i.e., break out of or continue the loop) by setting a Boolean `BubbleCanceled`. After the bubbling loop, a mouse click is handled by calling the virtual function `DoClick` of `element`, when `message` is LBUTTONUP.

HTML Element Behaviors

An object class is implemented for each type of HTML element, such as `Anchor`, `Form`, `Button`, `InputField`, `Label`, `Image`, etc. These object classes inherit from the `AbstractElement` base class. The three virtual functions of `AbstractElement`, namely, `HandleMessage`, `DoClick` and `ClickAction`, implement default behaviors of real HTML elements. Function `DoClick` of `AbstractElement`, written `AbstractElement::DoClick`, implements a loop to invoke `ClickAction` of each element along the DOM tree path, similar to the bubbling in `SendMsgToElem`. `HandleMessage` and `ClickAction` of `AbstractElement` are basically "placeholders" - they simply return in order to continue the bubble.

Each HTML element class can override these virtual functions given in `AbstractElement` to implement its specific behaviors. A subset of virtual functions of the `Anchor`, `Label` and `Image` elements is shown in Table 3.2. These examples demonstrate the complexity of the mouse handling logic due to the intrinsic behavioral diversity of individual elements and the possible compositions. For example, when the mouse is over an **anchor**, the target URL of this anchor will be displayed on the status bar by calling `SetStatusBar`, and the bubble continues, as indicated in `Anchor::HandleMessage`. When an anchor is clicked, `FollowHyperlink` is called to jump to the target URL, and the bubble is canceled, as indicated in Anchor::ClickAction. When the mouse is over a label, there is no `SetStatusBar` call, and the bubble is canceled. According to the HTML specification, a **label** can be associated with another element in the page, which is called "`ForElement`". Clicking on the label is equivalent to clicking on `ForElement`, as shown in `Label::ClickAction`. An **image** element can

Table 3.2: Virtual Functions of `Anchor`, `Label` and `Image` Elements

```
Bool Anchor::HandleMessage(message) {
 switch (message)
  case LBUTTONDOWN
    or LBUTTONUP:
     return true; //cancel bubble
  case MOUSEOVER:
    SetStatusBar(targetURL)
    return false; //continue bubble
  Other:
    return false;
}

Bool Anchor::ClickAction() {
    FollowHyperlink(targetURL);
    return true; // cancel bubble
}
```

```
Bool Label::HandleMessage(message) {
 switch (message)
  case MOUSEOVER
    or MOUSELEAVE:
    return true; //cancel bubble
  Other:
    return false;
}

Bool Label::ClickAction() {
 forElement = GetForElement()
 if (forElement != NULL)
     forElement->DoClick();
  return true;
}
```

```
Bool Image::HandleMessage(message) {
  if a map is associated with this image
    MapTarget = GetTargetFromMap();
    switch (message)
      case MOUSEOVER:
        SetStatusBar(MapTarget)
        return true;
}

Bool Image::ClickAction() {
  if a Map is associated with this image
    MapTarget = GetTargetFromMap();
    FollowHyperlink(MapTarget);
  else pAnchor=GetContainingAnchor();
    pAnchor->ClickAction();
  return true;
}
```

be associated with a map, which associates different screen regions on the image with different target URLs. When the mouse is over a region, the URL of the region is set to the status bar, as indicated in `Image::HandleMessage`. When the mouse clicks on the region, a `FollowHyperlink` call is made, as indicated in `Image::ClickAction`. If an image is not associated with a map, then the URL of the containing anchor of the image (i.e., the closest ancestor anchor of the image on the DOM) determines the status bar text and the hyperlink to follow.

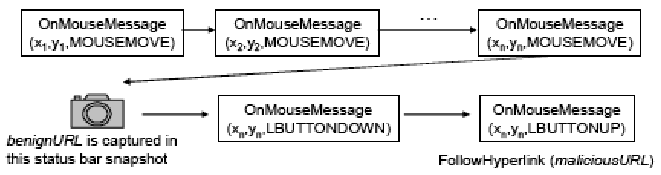

OnMouseMessage (x_1,y_1,MOUSEMOVE)
OnMouseMessage (x_2,y_2,MOUSEMOVE)
...
OnMouseMessage (x_n,y_n,MOUSEMOVE)
benignURL is captured in this status bar snapshot
OnMouseMessage (x_n,y_n,LBUTTONDOWN)
OnMouseMessage (x_n,y_n,LBUTTONUP)
FollowHyperlink (*maliciousURL*)

Figure 3.4: Function Level View of Status Bar Spoof

3.2.3 Formalization of Status Bar Spoofing

The visual invariant of the status bar is intuitively that the target URL of a click must be identical to the URL displayed on the status bar when the user stops the mouse movement. The negation of this invariant defines a spoofing scenario (Figure 3.4): First, MOUSEMOVE messages on elements $O_1, O_2, ..., O_n$ invoke a sequence of `OnMouseMessage` calls. When the mouse stops moving, the user inspects the status bar and memorizes `benignURL`. Then, a LBUTTONDOWN and a LBUTTONUP message are received, resulting in a `FollowHyperlink(maliciousURL)` call, where `maliciousURL` is different from `benignURL`.

We now apply the approach described in Figure 3.2.

(1) Specifying system state and state transitions (Step (c) in Figure 3.2). *System State* includes the browser state `statusBar` and the user state `memorizedURL`. State transitions are triggered by the `SetStatusBar` action and the user's `Inspection` action as below, where `AL` is an arbitrary action list.

```
op Inspection : -> Action .
op SetStatusBar : URL -> Action .
vars AL : ActionList . vars Url Url' : URL .
rl [SetStatusBar(Url) ; AL ] statusBar(Url')
  => [AL] statusBar(Url) .
rl [Inspection ; AL] statusBar(Url) memorizedURL(Url')
  => [AL] statusBar(Url) memorizedURL(Url) .
```

The first rule specifies the semantics of `SetStatusBar(Url)`: if the current action list starts with a `SetStatusBar(Url)` action, and the status bar displays `Url'`, then after this action is completed, it disappears from the action list, and the status bar is updated to `Url`. The second rule specifies the Inspection action: if `statusBar` displays `Url`, the `memorizedURL` is an arbitrary value `Url'`, and the action list starts with `Inspection`, then after

Table 3.3: Rules to specify `HandleMessage` and `ClickAction` of `Anchor`

```
ceq [AnchorHandleMessage(O,M) ; AL]                *** equation 1
  = [cancelBubble ; AL]
    if M == LBUTTONUP or M == LBUTTONDOWN .
crl [AnchorHandleMessage(O,M); AL] <O |targetURL: Url , ...>
  => [SetStatusBar(Url) ; AL] < O | targetURL: Url , ...>
    if M == MOUSEOVER .                            *** rule 2
ceq [AnchorHandleMessage(O,M) ; AL]                *** equation 3
  = [no-op ; AL]
    if M =/= LBUTTONUP, LBUTTONDOWN or MOUSEOVER .
crl [AnchorClickAction(O) ; AL] < O | targetURL: Url , ... >
  => [FollowHyperlink(Url) ; cancelBubble ; AL]
     < O | targetURL: Url , ... > .                *** rule 4
```

the inspection is made, `Inspection` disappears from the action list, and the URL on the status bar is copied to the user's memory, i.e., `memorizedURL`.

(2) Modeling the program logic (Step (b) in Figure 3.2). Modeling the functions shown in Table 3.1 and Table 3.2 is straightforward using Maude, e.g., `HandleMessage` and `ClickAction` of the `Anchor` element are specified in Table 3.3. Other functions are modeled in a similar manner.

It is easy to verify that these rules and equations indeed faithfully specify the behaviors of an anchor shown in Table 3.1: Equation 1 specifies that if an action list starts with an `AnchorHandleMessage(M,O)` action, this action should rewrite to a `cancelBubble`, if `M` is `LBUTTONUP` or `LBUTTONDOWN`. Rule 2 specifies that `AnchorHandleMessage(M,O)` should indeed rewrite to `SetStatusBar(Url)` when handling `MOUSEOVER`, where `Url` is the target URL of the anchor. For any other type of message `M`, `AnchorHandleMessage(M,O)` should rewrite to `no-op` to continue the bubble, which is specified by equation 3. Rule 4 rewrites `AnchorClickAction(O)` to the concatenation of `FollowHyperlink(Url)` and `cancelBubble`, where `Url` is the target URL of the anchor.

(3) Specifying the program invariant (Step (a) in Figure 3.2). A key question is how to define the negation of the program invariant to find status bar spoofs. It is specified as the pattern searched for in the `search` command:

```
ops maliciousUrl benignUrl empty : URL .
vars O1 O2 : Element . var Url : URL . var AL : ActionList .
```

```
search CanonicalActionSeq(O1,O2) ExecutionContext
          statusBar(empty) memorizedUrl(empty)
    =>! [FollowHyperlink(maliciousUrl) ; AL]
          statusBar(Url) memorizedUrl(benignUrl) X:StateMultiSet .
```

The command gives a well-defined mathematical meaning to status bar spoofing scenarios: "the Maude initial term `CanonicalActionSeq(O1,O2) ExecutionContext statusBar(empty) memorizedUrl(empty)` can be rewritten to the term `[FollowHyperlink (maliciousUrl) ; AL] statusBar(Url) memorizedUrl(benignUrl)`, which indicates that the user memorizes `benignURL`, but `FollowHyperlink(maliciousUrl)` is the next action to be performed by the browser".

Before going to the results there are two more things that we need to discuss. The `CanonicalActionSeq(O1,O2)` above is the representation of the generation of the user input, and `ExecutionContext` is the simplified representation of the execution context currently considered. As such, in some reading, both of these are not part of the state space exploration, but rather they generate a (set of) initial states for that exploration.

(4) Specifying the user action sequence and the execution context (Steps (d) and (e) in Figure 3.2). A challenging question is how the spoofing possibilities can be systematically explored, given that the web page can be arbitrarily complex and the user's action sequence can be arbitrarily long. Canonicalization is a common form of abstraction used in formal reasoning practice to handle a complex problem space. For this particular problem, our goal is to map a set of user action sequences to a single *canonical action sequence*, and map a set of web pages to a single *canonical DOM tree*. Because any instance in the original problem space only trivially differs from its canonical form, we only need to explore the canonical state space to find all "representative" instances.

(4.1) Canonicalization of the user action sequence. In general the user action sequence consists of a number of mouse moves, followed by a status bar inspection, followed by a mouse click (button down and up). In a canonical action sequence, the number of mouse moves can be reduced to *two*. This is because, although each MOUSEMOVE can potentially update the status bar, the status bar is a memoryless object, which means: (i) upon every mouse action, how to update the status bar does not depend on any previous update, but only on the DOM tree branch corresponding to the current

mouse coordinates; (ii) the whole sequence of status bar updates is equivalent to the last update. Thus, a canonical action sequence from element O1 to element O2 can be represented by the equation below, where the semicolon denotes sequential composition, and the MOUSEOVER on O1 invokes the last update of the status bar before the mouse arrives at O2 (O1 and O2 can be identical).

```
op CanonicalActionSeq: Element Element -> ActionList .
eq CanonicalActionSeq (O1,O2)
       = [ onMouseMessage(O1,MOUSEMOVE) ;
           onMouseMessage(O2,MOUSEMOVE) ;
           Inspection ;
           onMouseMessage(O2,LBUTTONDOWN);
           onMouseMessage(O2,LBUTTONUP) ] .
```

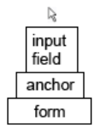
Note here that we use an equation instead of a rule. The difference between these is that an equation specifies a functional computation while a rule specifies a (possibly nondeterministic) state transition.

(4.2) Canonicalization of the execution context (i.e., DOM trees). In general a DOM tree may have arbitrarily many branches, but we can restrict the number of branches of a canonical DOM tree to at most two. This is because the canonical action sequence contains at most two MOUSEMOVEs — the third branch of the DOM tree would be superfluous as it would not receive any mouse message. Each HTML element in the DOM tree is represented as an object with a unique identifier, a class, a parent attribute (specifying the DOM tree structure) and possibly other attributes. We currently model `Anchor`, `Button`, `Form`, `Image`, `InputField` and `Label` element classes, plus a `Body` element always at the root. For example, the term

`< O | class:anchor, parent:O’ >`

represents anchor element `O` whose parent is `O’`. Our analysis is restricted to canonical DOM trees of bounded size but sufficiently rich to uncover useful scenarios.

We have analyzed all one- and two-branch DOM trees with at most six elements. At the level of most elements in this analysis all resulting spoofs were instances of simpler, smaller examples already. Also, keep in mind there are only seven HTML elements being modeled, and we also specify the generation rules so that all canonical DOM trees satisfy the required HTML wellformedness restrictions. E.g., an anchor cannot be embedded in another



```
<form action="http://foo.com/" >
  <a href="http://paypal.com">
    <input type="image" src="faked.jpg">
  </a>
</form>
```

Figure 3.5: Illustration of Scenario 1

anchor, an InputField can only be a leaf node, etc. The generation of all these initial canonical execution contexts is done in the tool as part of the search command shown above. Combined with the generation of canonical user action sequences, one for each possible movement, i.e., one for each pair of HTML elements, this initial state space generation creates the starting point from which the execution of our model runs.

3.2.4 Scenarios Suggested by the Results

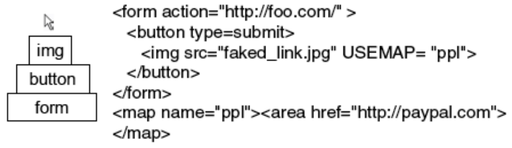
The scenarios are found by running the above-mentioned search command which first generates the execution context and action sequence and then executes the resulting system to completion.

We found nine combinations of canonical DOM trees and user action sequences that resulted in violations of the program invariant. All were due to unintended compositions of multiple HTML elements features. This section presents four representative scenarios in detail.

Shown in Figure 3.5, Scenario 1 has an `InputField` embedded in an `anchor`, and the `anchor` is embedded in a `form`.

When the mouse is over the `InputField`, the `HandleMessage` of each element is called to handle the MOUSEOVER message that bubbles up to the DOM tree root. Only the anchor's `HandleMessage` writes its target URL `paypal.com` to the status bar, but when the `InputField` is clicked, its `ClickAction` method retrieves the target URL from the `form` element, which is `foo.com`. This scenario indicates the flaw in message bubbling — the MOUSEOVER bubbles up to the `anchor`, but the click is directly passed from the `InputField` to the `form`, skipping the `anchor`.

Scenario 2 (Figure 3.6) is very different from Scenario 1: an `img` (i.e., image) associated with a map `ppl` is on top of a button. The target URL of `ppl` is set to `paypal.com`. When `img` gets a MOUSEOVER, it sets the status bar to `paypal.com` and cancels the bubble. When the mouse is clicked



```
<form action="http://foo.com/" >
  <button type=submit>
    <img src="faked_link.jpg" USEMAP= "ppl">
  </button>
</form>
<map name="ppl"><area href="http://paypal.com">
</map>
```

Figure 3.6: Illustration of Scenario 2

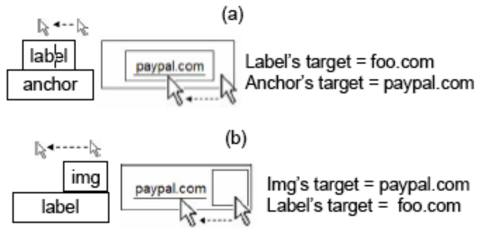


Figure 3.7: Illustration of (a) Scenario 3 and (b) Scenario 4

on `img`, because `img` is a child of `button`, the click is treated as a click on the `button`, according to the implementation of `SendMsgToElem()`. The button click, of course, leads to a navigation to `foo.com`. This scenario indicates a design flaw — an element (e.g., button) can hijack the click from its child, but it does not hijack the MOUSEOVER message, and thus causes the inconsistency.

Scenario 3 contains a label embedded in an anchor (Figure 3.7(a)). When the mouse is moved toward the `label`, it must first pass over the `anchor`, and thus sets `paypal.com` on the status bar. When the `label` is clicked, the page is navigated to *foo.com*, because the `label` is associated with an `anchor` of `foo.com`. An opposite scenario shown as scenario 4 in Figure 3.7(b) seems more surprising, which suggests an outward mouse movement from a child to a parent. Such a movement makes it feasible to spoof the status bar using an `img` sitting on top of a `label`. Note that, because HTML syntax only allows an `img` to be a leaf node, such an outward mouse movement, which is suggested by the Maude analysis, is critical in the spoofing attack.

We also derived several scenarios with two-branch DOM trees. They demonstrate the varieties of DOM trees and layout arrangements that can be utilized in spoofing, e.g., a spoof page places the two leafs side-by-side, another page uses Cascading Style Sheets (CSS) [97] to set element positions, etc.

3.3 Case Study 2: Address Bar Spoofing

Address bar spoofing is another category of spoofing attack. It fools users into trusting the current page when it comes from an untrusted source. The combination of a status bar spoofing and an address bar spoofing gives an end-to-end scenario to hide the identity of the malicious site, and thus is a serious security threat. In this section, we first introduce the background knowledge about the address bar logic, then present the Maude-based analysis technique and real spoofing scenarios uncovered by the analysis.

3.3.1 Background: Address Bar Basics

An IE process can create multiple `browsers`. Each one is implemented as a thread. A `browser`, built on the OLE framework [98], is a container (including the title bar, the address bar, the status bar, etc.) hosting a client document in the content area. Many types of client documents can be hosted in IE, such as HTML, Microsoft Word, Macromedia Flash and PDF. The object used to represent an HTML document is called a `renderer`. A `renderer` can host multiple `frames`, each displaying an HTML page downloaded from a URL.

An HTML page is stored as a `markup` data structure. A `markup` consists of the URL and the DOM tree of the content from the URL. The top level `frame`, i.e., the one associated with the entire content area, is called the `primaryFrame` of the `renderer`. Figure 3.8 shows a browser displaying a page from `http://MySite`. The `renderer` has three `frames` — `PrimaryFrame` from `MySite`, `Frame1` from `PayPal.com` and `Frame2` from `MSN.com`. Each `frame` is associated with a *current markup* and, at the navigation time, a *pending markup*. Upon navigation completion, the pending markup is switched in and becomes the current markup.

Informally, the program invariant of the address bar correctness is that: (1) *the content area is rendered according to the current markup of primaryFrame;* and (2) *the URL on the address bar is the URL of the current markup of primaryFrame.* In the example shown in Figure 3.8, the address bar should display "`http://MySite`".

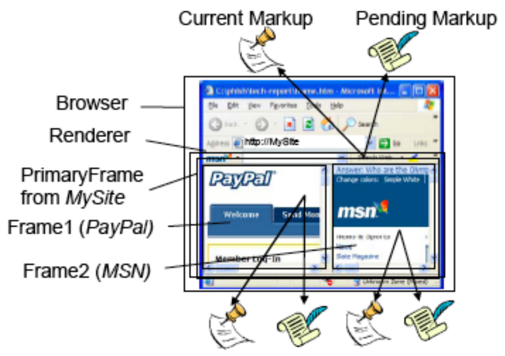


Figure 3.8: *Browser*, *Renderer*, *Frames* and *Markups*

3.3.2 Overview of the HTML Navigation Logic

HTML navigation consists of multiple tasks — loading HTML content, switching markup, completing navigation and rendering the page. A `renderer` has an event queue to schedule these tasks. The event queue is a crucial mechanism for handling events asynchronously, so that the browser is not blocked to wait for the completion of the entire navigation. We studied three types of navigation: (1) loading a page into the current *renderer*; (2) traveling in the history of the current *renderer*; and (3) opening a page in a new renderer. Figure 3.9 only illustrates a small subset of functions involved in the navigations for better readability.

Figure 3.9(a) shows the event sequence of loading a page in the current *renderer*. It is initiated by a `FollowHyperlink`, which posts a `start navigation` event. Function `PostMan` is responsible for downloading the new HTML content to a pending markup. Event *ready* is posted to invoke `SetInteractive`, to make the downloaded contents effective. `SetInteractive` first invokes `SwitchMarkup` to replace the current markup with the pending markup, and calls `NavigationComplete`. If the downloaded markup belongs to *primaryFrame*, function `SetAddressBar` is invoked to update its address bar. An `Ensure` event is posted by `SwitchMarkup`, which invokes `EnsureView` to construct a *View* structure containing element layouts derived from the current markup of *primaryFrame*. The OS periodically posts an *OnPaint* event to paint the content area by calling `RenderView`. Figure 3.9(b) shows the event sequence of a history travel. `History_Back` and `Travel` look up a history log to initialize the navigation. `PostMan`, in this case, loads HTML contents from a persistent storage in the hard disk, rather

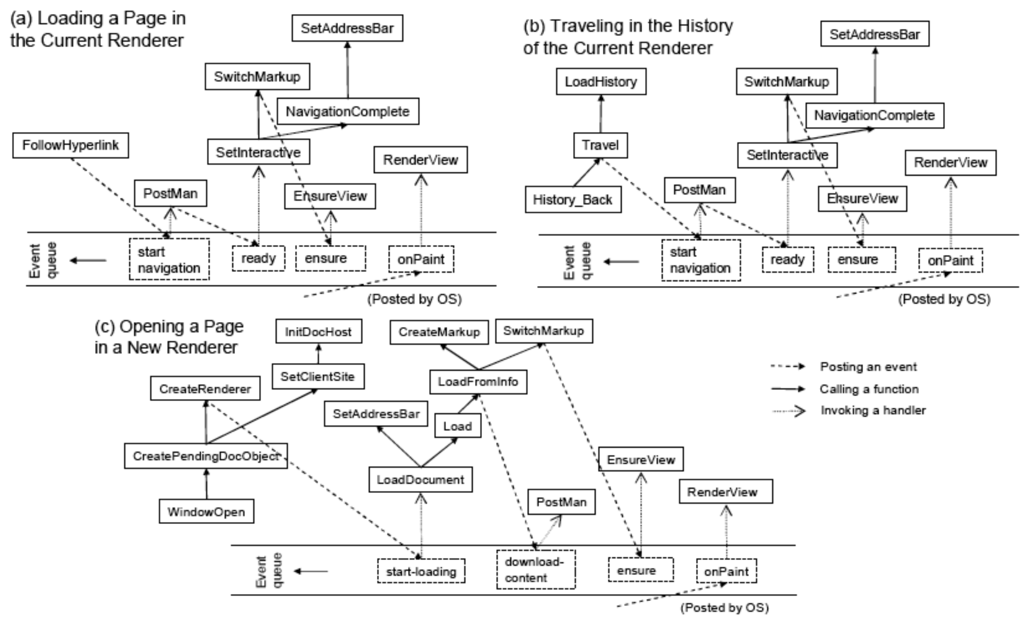


Figure 3.9: Logic of HTML Navigations

than from the Internet. The remaining portion of the sequence is similar to that of Figure 3.9(a).

Figure 3.9(c) shows the event sequence of loading a new page into a new renderer. `WindowOpen` is the starting point. It calls the method `CreatePendingDocObject` to create a new renderer and then calls `SetClientSite`. `SetClientSite` prepares a number of Boolean flags as the properties of the new *renderer*, and calls `InitDocHost` to associate the *renderer* with the browser (i.e., the container). The new *renderer* at this moment is still empty. The `start-loading` event invokes `LoadDocument` which first calls `SetAddressBar` to set the address bar and then calls `Load` which calls `LoadFromInfo`. `CreateMarkup` and `SwitchMarkup` are called from `LoadFromInfo` before posting a `download-content` event to download the actual content for the newly created markup. Function `PostMan` does the downloading as above. The remainder of the sequence is similar to both prior sequences.

3.3.3 Formalization of the Navigations and the Address Bar Behavior

(1) Modeling the system state (Step (c) in Figure 3.2). Because an address bar spoofing is by definition the inconsistency between the address bar and the content area of the same browser, "spoofability" is a property of the logic of a single browser. This does not mean that only one browser is allowed in a spoofing scenario — there can be other browsers that create a hostile execution context to trigger a logic flaw in one particular browser. Nevertheless, we only need to model the system as one browser and prove its logical correctness (or uncover its flaws), and treat the overall effect of other browsers as the context of this browser.

The system state of a browser includes the URL displayed in the address bar, the URL of the *View* in the content area, a travel log and the primary frame. The Maude specification defines a set of *Frames* and a set of *Markups*. For example, if *Markup* `m1` is downloaded from URL `u1`, and it is the *currentMarkup* of *Frame* `f1`, we specify `f1` and `u1` as:

```
<f1 | currentMarkup: m1, pendingMarkup: ...>
<m1 | URL: u1, frame: f1, ...>
```

The system state also includes a function call queue and an event queue. The function call queue is denoted as [$\texttt{call}_1$; $\texttt{call}_2$; ... ; $\texttt{call}_n$], and the event queue is denoted as {$\texttt{event}_1$; $\texttt{event}_n$; ... ; $\texttt{event}_n$}.

(2) *Specifying the user action sequence* (Step (d) in Figure 3.2). In the scenario of an address bar spoofing, the user's only action is to access an untrusted HTML page. The page contains a JavaScript calling navigation functions `FollowHyperlink`, `HistoryBack` and/or `WindowOpen`. The behavior of the JavaScript is modeled by a rule that conditionally appends a navigation call to the function list. As explained in Figure 3.9, each navigation call generates a sequence of events. It is guaranteed that all possible interleavings of event sequences are exhaustively searched, because Maude explores all viable rewrite orders.

(3) *Specifying the execution context* (Step (e) in Figure 3.2). Many Boolean conditions affect the execution path, e.g., conditions to return from a function and conditions to create a new frame. These conditions constitute the execution context of the system. We defined rules to explore both

Table 3.4: Pseudo Code and Rewrite Rule of `SetInteractive`

Pseudo Code

```
MARKUP::SetInteractive() {
  if (BOOLEXP1) return;
  this->frame->SwitchMarkup(this);
  if (BOOLEXP2) NavigationComplete(frame)
}
```

Rewrite Rule to Specify `SetInteractive`

```
var F : Frame . var M : Markup . var FQ : FunctionQueue .
rl [SetInteractive(M) ; FQ] < M | frame : F , ... >
 => [(if BOOLEXP1 != true
         then SwitchMarkup(M,F) else noop fi) ;
     (if BOOLEXP2 == true
         then NavigationComplete(F) else noop fi) ; FQ]
     < M | frame: F , ... > .
```

possible paths depending on the true and false values of these conditions. Therefore the `search` command explores both paths at each branch in the pseudo code. The assignments of the Boolean conditions, combined with the function call sequence, constitute a *potential* spoofing scenario. These may include false positive scenarios, in the sense that such Boolean values cannot at the same time be attained by different variables, and thus, as shown in Figure 3.2, mapping a potential scenario back to the real-world is important. It is a manual effort guided by the formally derived potential scenarios. We discuss this in Section 3.3.4.

(4) *Modeling Function Calls and Events* (Step (b) in Figure 3.2). There are three types of actions shown in Figure 3.9: calling a function, invoking an event handler and posting an event. A function call is implemented as a term substitution in the function call queue. For example, the function call `SetInteractive` is specified by the following rule in Table 3.4, where `F` is the frame of *Markup* `M`, and `SetInteractive(M)` can conditionally rewrite to `SwitchMarkup(M,F)` (if `BOOLEXP1` is false) followed by `NavigationComplete(F)` (if `BOOLEXP2` is true).

Posting of an event happens by appending the event to the event queue, for example, `FollowHyperlink` is specified by removing itself from the function queue and adding a *startNavigation* event to the end of the event queue.

```
var U : Url . var F : Frame .
var FQ : FunctionQueue . var EQ : EventQueue .
rl [FollowHyperlink(U, F) ; FQ] { EQ }
  => [FQ] { EQ ; startNavigation(U, F) } .
```

The third type of action is the invocation of an event handler. Any event can only be invoked when its previous event handler returns. To model this restriction, any rule of an event handler invocation specifies that the first event in the event queue can be dequeued and translated into a function call only when the function queue is empty. Below is the rule to specify the handling of the `ready` event, which invokes the handler `SetInteractive`.

```
var EQ : EventQueue .
rl [empty] { ready(M) ; EQ }
  => [SetInteractive(M)] { EQ } .
```

5) *Specifying the program invariant of address bar correctness* (Step (a) in Figure 3.2). A good state is a state where the URL on the address bar matches the URL of the *View* and is also the URL of the content that is painted on the screen. In addition to that, the URL is the URL of the *currentMarkup* of the *primaryFrame*. Therefore the program invariant is defined by the following `goodState` predicate:

```
var U: URL . var F : Frame .  var M : Markup .
equation goodState (addressBar(U) urlOfView(U)
         urlPaintedOnScreen(U) primaryFrame(F)
         < F | currentMarkup: M , ...> < M | url: U , ...>)
    = true .
```

It is also important to specify the initial state for the search command. In the initial state, both the event queue and the function call queue are empty. The *primaryFrame* is `f1`. The *currentMarkup* of `f1` is `m0`. The *pendingMarkup* of `f1` is uninitialized. `m0` is downloaded from `URL0`. The address bar displays `URL0`, the *View* is derived from `URL0`, and the *View* is painted on the screen. The following equation specifies `initialState`:

```
const f1:Frame m0:Markup url0:URL empty:EventQueue
equation initialState
   = { empty } [ empty ] primaryFrame(f1)
     < f1 | currentMarkup: m0 , pendingMarkup: nil >
     < m0 | url: url0 , frame: f1 > addressBar(url0)
     urlOfView(url0) urlPaintedOnScreen(url0) .
```

3.3.4 Uncovered Spoofing Scenarios

We used Maude's `search` command to find all execution paths in the model that start with the initial state and finish in a bad state (i.e., denoted as "`not goodState`" in Maude). The search was performed on two navigations, i.e., two `FollowHyperlinks`, two `History_Backs`, one `FollowHyperlink` with one `History_Back`, and one `WindowOpen` with one `FollowHyperlink`. The generation of the states to be explored is done by the search command at runtime. Whenever a condition could influence which path to take, we did not model the condition's value explicitly, but rather had two rules, one for each possible value of the condition, and then looking at the execution trace allowed us to identify the rule being used (note all rules are labeled) and thus which value the condition should have.

Each condition shown in Table 3.5 is present in at least one execution context of a potential spoofing scenario uncovered by Maude. Some function names in the Location column were not shown in Figure 3.9, because Figure 3.9 only shows a sketch of the logic of navigation, while the actual model we implemented is more detailed (see Appendix A). The `search` result in Table 3.5 provides a roadmap for a systematic investigation: (1) we have verified that when each of these conditions is manually set to true in the corresponding location using a debugger, the real IE executable will be forced to take an execution path leading to a stable bad state; therefore, our investigation should be focused on these conditions; (2) many other conditions present in the pseudo code are not in Table 3.5, such as those conditions in `SwitchMarkup`, `LoadHistory` and `CreateRenderer`, therefore these functions do not need further investigation.

The versions in our study are IE 6 and IE 7 Beta 1 through Beta 3. In the rest of this section, we will focus on conditions No. 2, 9, 11 and 18, for which we have succeeded in constructing real spoofing scenarios. For the other conditions, we have not found successful scenarios to make them real without the debugger. They may be false positives due to the fact that our model does not include the complete logic of updating and correlating these conditions, but simply assumes that each condition can be true or false at any point during the execution. In this sense, our address bar modeling is not exact (too permissive). Because of the imprecision in modeling these Boolean conditions, we need a considerable amount of effort to understand

Table 3.5: Conditions of Potential Spoofing Scenarios

	Location	Condition
1	FireNavigationComplete	GetHTMLWinUrl() = NULL
2	**FireNavigationComplete**	**GetPFD(bstrUrl) = NULL**
3	FireNavigationComplete	ActivatedView = true
4	NavigationComplete	DontFireEvents = true
5	NavigationComplete	DocInPP = true
6	NavigationComplete	ViewWOC = true
7	NavigationComplete	ObjectTG = true
8	NavigationComplete	CreateDFU = true
9	**SetAddressBar**	**CurrentUrl = NULL**
10	SetClientSite	QIClassID()= OK
11	**LoadHistory**	**HTMLDoc = NULL**
12	CreateMarkup	NewMarkup = NULL
13	SetInteractive	pPWindowPrxy = NULL
14	SetInteractive	IsPassivating = true or IsPassivated = true
15	SetInteractive	HtmCtx() = NULL
16	SetInteractive	HtmCtx()→BindResult = OK
17	EnsureView	IsActive() = false
18	**RenderView**	**RSFC = NULL**

their semantics. Constructing successful scenarios is still a non-trivial "security hacking" task. Nevertheless, Table 3.5 provides a valuable roadmap to narrow down our investigations.

Scenarios based on condition 2 and condition 9 *(silent-return conditions)*. For ease of presentation, we assume there is a malicious site `http://evil` (or `https://evil`) in this section. The function call traces associated with condition 2 (i.e. `GetPFD(url)= NULL` in `FireNavigationComplete`) and condition 9 (i.e. `CurrentURL = NULL` in `SetAddressBar`) indicate similar spoofing scenarios: there are silent-return conditions along the call stack of the address bar update. If any one of these conditions is true, the address bar will remain unchanged, but the content area will be updated. Therefore, if the script first loads `paypal.com` and then loads `http://evil` to trigger such a condition, the user will see "`http://paypal.com`" on the address bar whereas the content area is from `http://evil`.

We found that both condition 2 and condition 9 can be true when the URL of the page has certain special formats. In each case, the function

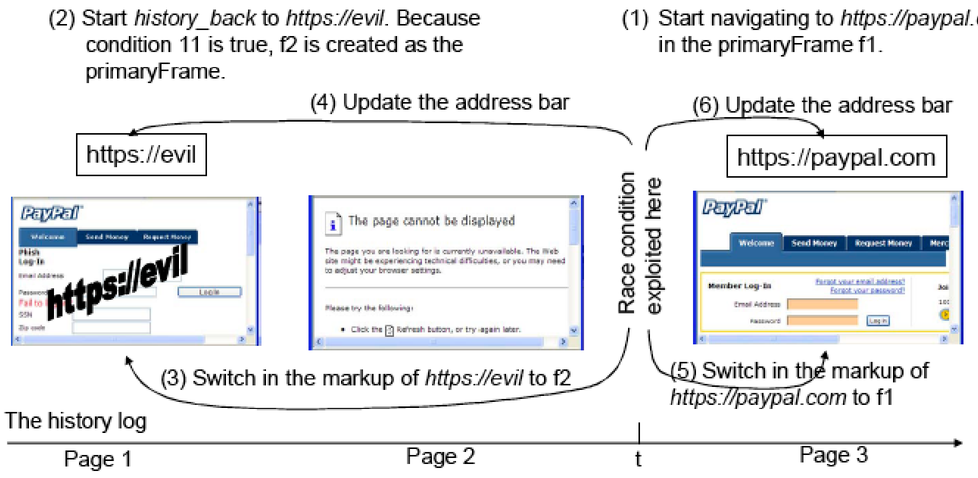

(2) Start history_back to https://evil. Because condition 11 is true, f2 is created as the primaryFrame.
(1) Start navigating to https://paypal.com in the primaryFrame f1.
(4) Update the address bar
(6) Update the address bar
https://evil
https://paypal.com
https://evil
The page cannot be displayed
Race condition exploited here
(3) Switch in the markup of https://evil to f2
(5) Switch in the markup of https://paypal.com to f1
The history log
Page 1
Page 2
t
Page 3

Figure 3.10: Spoofing Scenario Due to a Race Condition

(i.e., `FireNavigationComplete` or `SetAddressBar`) cannot handle the special URL, but instead of asserting the failure condition, the function silently returns when the condition is encountered. For condition 9, we observed that all versions of IE are susceptible; for condition 2, only IE 7 Beta 1 is susceptible, in which case even the SSL certificate of *PayPal* is present with the faked page, because the certificate stays with the address bar. In other versions of IE, although they have exactly the same silent-returning statement, condition 2 cannot be triggered because the special URL has been modified at an earlier stage during the execution before `GetPFD` is called. However, even for these seemingly unaffected versions, having the silent-returning condition is still problematic — IE must guarantee that such a condition can never be true in order to prevent the spoofing.

These two examples demonstrate a new challenge in graphical interface design — *atomicity is important*. In the navigation scenarios, once the pending markup is switched in, the address bar update should be guaranteed to succeed. No "silent return" should be allowed. Even in a situation where atomicity is too difficult to guarantee, the browser should at least raise an exception to halt its execution rather than leave it in an inconsistent state.

Scenario based on condition 11 *(a race condition)*. Condition 11 is associated with a function call trace which indicates a situation where two frames co-exist in a *renderer* and compete to be the primary frame. Figure 3.10 illustrates this scenario.

The malicious script first loads *Page 1* from `https://evil`. Then it intentionally loads an error page (i.e., *Page 2*) in order to make conditional 11 true when `LoadHistory()` is called later. The race condition is exploited at time *t*, when two navigations start at the same time. The following event sequence results in a spoof: (1) the *renderer* starts to navigate to `https://paypal.com`. At this moment, the primary frame is `f1`; (2) the *renderer* starts to travel back in the history log. Because condition 11 is true, i.e., `HTMLDoc = NULL`, a new frame `f2` is created as the primary frame. This behavior is according to the logic of `LoadHistory()`; (3) the markup of `https://evil` in the history is switched into `f2`; (4) the address bar is updated to `https://evil`; (5) the downloading of the `paypal.com` page is completed, so its markup is switched into `f1`. Since `f1` is not the primary frame anymore, it will not be rendered in the content area; (6) the address bar is updated to `https://paypal.com` despite the fact that `f1` is no longer the primary frame. When all these six events occur in such an order, the user sees `http://paypal.com` on the address bar, but the `https://evil` page in the content area. The SSL certificate is also spoofed because it gets updated with the address bar.

This race condition can be exploited on IE 6, IE 7 Beta 1 and Beta 2 with a high probability of success: in our experiments, the race condition could be exploited more than half of the time. The exploit does not succeed in every trial because event (5) and event (6) may occur before event (3) and event (4), in which case the users sees the address "`https://evil`" on the address bar.

It is worth noting that race conditions are likely to exist in the logic supporting the tab-browsing mode as well, in which multiple *renderers* share and compete for a single address bar.

Scenario based on condition 18 *(a hostile environment)*. Condition 2 and condition 9 trigger the failures of address bar updates, while condition 18 (i.e., `RSFC = NULL` in `RenderView`) triggers the failure of the content area update. We found that the condition can be true when a certain type of system resource is exhausted. A malicious script is able to create such an environment by consuming a large amount of the resource and then navigating the browser from `http://evil` to `http://paypal.com`.

When the timing of the navigation is appropriate, the browser will succeed to update the address bar and fail to update the content area, leaving the `http://evil` content and the `paypal.com` URL visible to the user.

Once again, this example demonstrates the importance of atomicity in graphical interface implementations. In addition to the correctness of the internal logic of a browser, this scenario emphasizes the need for resilience against a hostile execution environment.

3.4 Discussions

In order to better put our work into perspective, this section presents higher-level discussions about possible defense techniques, other visual spoofing flaws and various techniques for GUI logic analysis.

3.4.1 How to Defend Against GUI Logic Exploits

The most direct defense against spoofing attacks is bug fixing. All scenarios that we have discovered have been confirmed by the IE development team. In a build after IE 7 Beta 3, all the status bar spoofing bugs and two address bar bugs have been fixed. Two other address bar bugs have been investigated, and their fixes have been proposed.

In situations where the vendor's patches are not yet available, vulnerability-driven filtering can provide fast and easy-to-deploy patch-equivalent protection. In particular, our colleagues have explored the possibility of using *BrowserShield* [104] to foil spoofing attacks. In *BrowserShield*, web pages are intercepted at a browser extension, which injects a script-rewriting library into the pages and sends them to the browser. The rewriting library is executed during page rendering at the browser, and rewrites HTML pages and any embedded scripts into safe equivalents. The equivalent safe pages contain logic for recursively applying run-time checks according to policies that detect and remove known attack patterns that we described earlier. In the proof-of-concept implementation, they authored policies for both status-bar spoofing removal and address-bar spoofing removal. The status bar policy is to inject JavaScript code into static HTML contents to monitor the status bar before the mouse click, and compare it with the URL argument of the `FollowHyperlink` call. One of the address bar policies is to inject JavaScript code to check if a URL can cause a silent failure of the address bar update.

3.4.2 Achieving GUI Integrity is Challenging

The objective of this chapter is to bring the GUI logic problem to the attention of the research community, rather than claiming that the visual spoofing problem as a whole can be solved in the short term. In particular, the following two questions are not addressed by this work.

(1) *Is GUI-logic correctness important to users that are security-unconscious and completely ignore any security indicators?* User-studies have raised the concern that many average users still lack the knowledge or the attention to examine the information provided by security indicators, such as the address bar, the status bar, SSL certificate and security warning dialogs [37, 126]. Many users readily believe the authenticity of whatever is displayed in the content area. We agree that this is the current fact, and argue that a significant effort should be spent on user education about secure browsing. But such an education would be ineffective without the trustworthiness of the security indicators — if their information can be spoofed, even we, as computer science professionals, do not know what to trust. The success of anti-phishing must be achieved by a joint effort between the browser vendors and the end users. It is analogous to automobile-safety: drivers have the responsibility to buckle up, and the automobile manufacturers need to guarantee that the seat-belts are effective.

(2) *How to deal with other types of visual spoofs that are not due to GUI logic flaws?* In the introduction, we listed a few visual spoofing scenarios due to graphical similarities. These issues have little to do with logic problems, so their treatments are very different from the approach presented in this chapter. For example, the current version of IE disallows a script from the Internet zone to open a chromeless window (i.e., a window having only the content area). It is also clearly specified in design that the URL displayed on the address bar should be left-justified after each address bar update, and no pop-up window can stay "always-on-top", etc. *SpoofStick* is designed to interpret any confusing URL on the address bar [116]; *Dynamic Security Skins* [38] and *Passpet* [129] use trusted images to defeat certain spoofing attacks. Ye and Smith proposed several ideas to implement trusted paths for browsers by disallowing the page *content elements* to forge the page *status elements* [128]. Virtual machine techniques have also been used to provide trusted browser GUI elements, e.g., the *Tahoma* window manager provides

a virtual screen abstraction to each browser instance [35]. Nevertheless, when the internal GUI logic is flawed as shown in this chapter, ensuring unforgeable GUI elements is not a remedy. Therefore, GUI logic flaw and graphic similarity can be viewed as two different problems under the same umbrella of visual spoofing.

3.4.3 A Broad Spectrum of Tools Can Be Used for Systematic Exploration

The essence of our approach is that we systematically explore GUI logic. Whether the exploration is done by symbolic formal analysis (such as theorem proving or model checking) or by exhaustive testing is less important. As an example of exhaustive testing, we used the binary instrumentation tool *Detours* [75] to test the status bar logic. The basic idea is that since we know the program invariant and how to generate canonical user action sequences and canonical DOM trees, we can generate actual canonical HTML pages and actual mouse messages to test the actual IE status bar implementation. The advantage of the exhaustive testing approach is that it does not require manual modeling of the behaviors of each HTML element, and therefore can avoid the potential inaccuracies in the logic model. Applying this technique, we were able to find all spoofs derived from our previous modeling.

Nevertheless, there is no fundamental difference as to whether the exploration is done symbolically (e.g., by Maude) or by exhaustive testing (e.g., by Detours), because both techniques are based on the same understanding of the search space and the test case construction. The main effort for the symbolic exploration is to correctly specify the GUI logic in sufficient detail. The exhaustive testing requires much effort to drive the system's internal state transitions. For example, to test the address bar logic, we would need to exhaustively enumerate all event interleaving possibilities in an actual renderer, which is a nontrivial task.

3.5 Related Work

The contributions of our work are: (1) the formulation of GUI logic correctness as a research problem, and (2) the proposal of a systematic approach to uncover GUI logic flaws leading to visual spoofs. There is little existing work related to our first contribution, but a wealth of work is related to the second — formal methods and program analysis techniques have been successful in discovering software reliability and security flaws. We summarize a few techniques below.

The SLAM technique [14] uses theorem proving and model checking tools to statically verify whether or not predefined "API usage rules" are obeyed in large programs. A static driver verifier is built on the SLAM technique, and has been deployed for Windows driver implementation correctness. Model checking techniques are also developed to find file system bugs [127] and security vulnerabilities [22] in large bodies of legacy source code. Much research has been done in formal verification of security protocols [87]. A static analysis technique is used for detecting higher level vulnerabilities such as SQL injections, cross-site scripting, and HTTP splitting attacks [84]. Our work is complementary to the existing research, because we have focused on machine-user link trustworthiness.

Also related are research papers on phishing attacks, e.g., *PwdHash* is a browser plug-in that transparently produces a different password for each site to prevent phishing sites from obtaining usable passwords [106]. Florencio and Herley designed a technique to detect password phishing by monitoring password-reuse patterns between a well-known site and an unfamiliar site [60].

CHAPTER 4

BROWSER SECURITY ANALYSIS FOR IBOS

In this chapter we present work that was in part done together with Sam King, Shuo Tang and José Meseguer.

From Chapter 3 we already know about web browsers and some possible attacks against them, specifically on the graphical user interface (GUI). In this chapter we are going further than the post mortem analysis of GUI security which was described in the previous chapter. We consider the idea of basing the browser design on explicit security requirements to begin with. And we look at bugs our browser analysis can find in the browser, as well as discussing guarantees we can give if no bugs are found.

The notion of a browser that is to be secure by design is exemplified in the work on the Illinois Browser Operating System (IBOS) [117] web browser, which is a newly designed browser that builds upon the earlier work on the OP2 [66] browser. OP2 also aimed at being secure by design and it did use formal modeling and validation in Maude. We use IBOS as the basis of our analysis in this chapter. Our analysis was able to influence the design, led to bug fixes and has increased the overall assurance about the correctness of IBOS.

In this chapter we get a result on the address bar being correct at all times, which is an important property for a browser. We already know that property from Chapter 3. Also, we are able to show that the browser adheres to the *same origin policy.* The purpose of the same origin policy is to remove any possibility of any data leaking from one visited web page to another.

We will explain more about the IBOS browser in Section 4.1, and then explain the modeling methodology used for the browser in Section 4.2. We then show three case studies in Section 4.3: (i) a case study on the display memory in Section 4.3.1, (ii) a case study on the address bar in Section 4.3.2, and (iii) a case study on the same origin policy in Section 4.3.3.

4.1 IBOS

The Illinois Browser Operating System (IBOS) [117] is a web browser developed at the University of Illinois with the goal of increased security. In particular, security considerations are taken into account in the initial design phase and during implementation. The issue with state-of-the-art web browsers is that they are complex, have a huge trusted computing base and are integrated closely into the actual operating system, and thus are a prime avenue for malicious attackers to access a computer. The trusted computing base is the subset of the software in which any exploitable error would lead to the whole system being potentially compromised.

IBOS is a combination web browser and operating system that reduces the trusted computing base. It does so by utilizing a microkernel and exposing browser-level abstractions at the lowest software layer, which allows removal of almost all traditional OS components and services from the trusted computing base by directly mapping those browser abstractions to hardware abstractions. Overall, this approach turns out to be flexible enough to allow browser security policies while supporting traditional applications. Also, the overhead added to the browsing experience is small.

Indeed, web-based applications (web apps) and the browser itself have become quite popular targets for attacks on computer systems. The vulnerabilities in web apps are ever increasing, so isolation of the web apps is highly desirable. For example, the formerly most common security vulnerability, the buffer overflow, has been overtaken by cross-site scripting, which essentially is a form of script injection into web apps [76]. Vulnerabilities in the actual web browsers are not as common as web app vulnerabilities, but occur often enough to be troubling. In 2009, Internet Explorer, Chrome, Safari and Firefox had 349 new security vulnerabilities [77], which get commonly exploited by attackers [125, 96, 103, 77]. Further vulnerabilities are possible in the operating system, its services or libraries.

Not all attacks are created equal, of course, and attacks at the top of the software stack, e.g., using cross-site scripting to attack web apps, will only give the attacker access to the browsers current vulnerable web app. Further down the stack, attacks on the browser would give the attacker access to all web apps, their data, and system resources the browser can access. At the bottom of that stack, attacks on the operating system itself can be the most devastating, as the attacker can gain full control of the system. Vulnerabilities (and attacks) higher in the stack turn out to be more common, but are less damaging. Attacks lower in the stack have a much higher threat potential, and that is what IBOS is trying to address.

There are other alternative browser projects with similar goals, but they all share the caveat of being built on a legacy operating system and include complex libraries and shared system services inside their trusted computing base. IBOS consists of an operating system and browser that are co-designed to minimize the trusted computing base at the web browser level. IBOS achieves this by moving device drivers, network protocol implementation, the storage stack, and window management software, among other system services, outside of the trusted computing base. These components then run on top of the trusted kernel of IBOS, which can enforce security policies. The contrast with current state-of-the-art browsers is that they add one layer on top of another, particularly the fact that the browser is running on top of the general-purpose operating system. For all the details on IBOS, see [117].

Let us note one important additional consideration. Even though IBOS is built on top of the L4Ka::Pistachio microkernel [92], that is itself not especially more trustworthy than other microkernel operating systems, there is another variant of that microkernel which has been formally verified. That microkernel, seL4 [80], which uses a very similar set of function calls, could have been used instead of L4Ka::Pistachio. In that case, all the good properties of the underlying microkernel would have been inherited by IBOS. We are doing our analysis in this chapter *under the assumption of a correct underlying microkernel.* As seL4 was not publicly available at the initial development time of IBOS, it was not used for IBOS. Also, seL4 is completely single-threaded, which L4Ka::Pistachio is not, so some additional performance loss would be unavoidable.

Now, IBOS is designed to compartmentalize all the different processes as much as possible, and all communication is being forcibly routed through the trusted kernel, which can then implement its policies. The IBOS kernel decides, based on the policies, which communication between processes is allowed, and thus possible. As we will see in the next section, the communication between different web page instance, network processes, the network card, the display memory and the central kernel is modeled. We will analyze the adherence of IBOS to the *Same Origin Policy*, as well as check the ad-

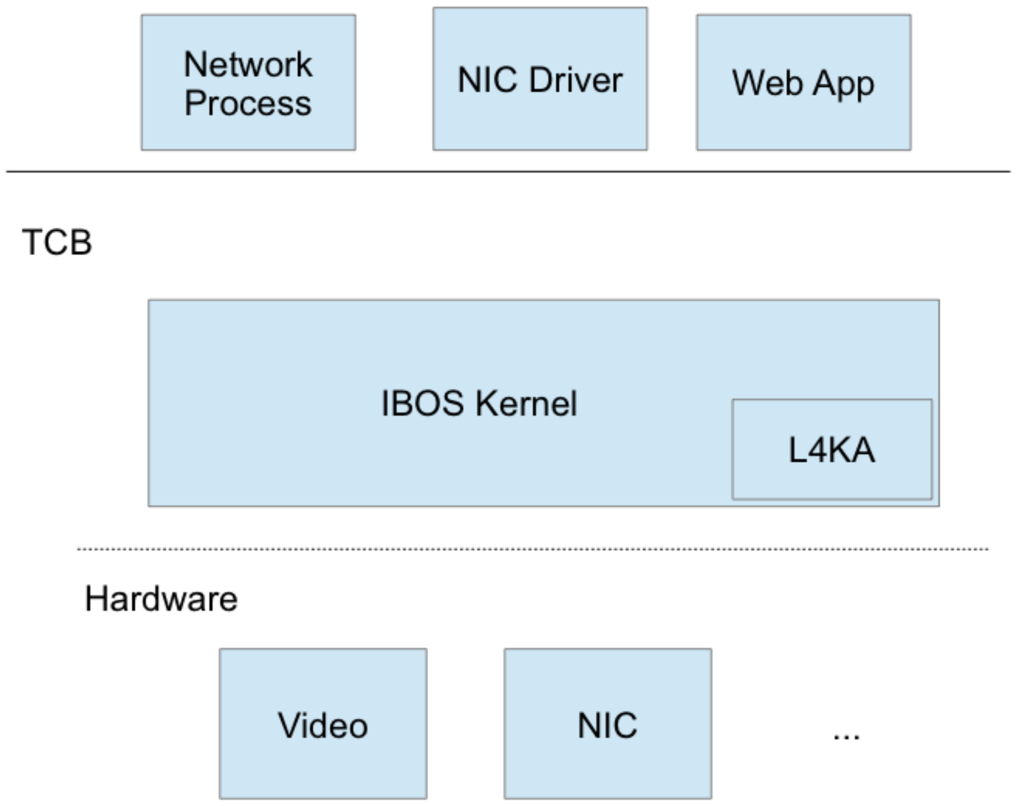


Figure 4.1: IBOS Architecture

dress bar correctness like we did for Internet Explorer in Section 3.3, and we also look at the display memory, where we did find a bug.

4.1.1 IBOS Architecture

In Figure 4.1 we show a simplified presentation of the architecture of IBOS. For all details, please see [117, Section 2]. As shown in the figure, the hardware is at the bottom of the stack, the IBOS kernel is on top of that, and part of the trusted computing base as well. Everything on top of the kernel is not part of the TCB. Specifically, all web apps, network processes and the NIC driver do not need to be trusted. Also, the figure does not show that all other traditional applications work on top of a UNIX layer, outside the TCB, on top of the IBOS kernel as well.

Some of the key goals of IBOS are the following, see [117] for all the goals and more detail:

- Security decisions happen at the lowest possible level: small TCB.
- Enough browser states and events exposed, so as to allow for security policy checking; this makes IBOS flexible to allow new browser security policies.

A key property of the IBOS browser is that *all communication*, i.e., all messages sent or received, *get transmitted through the* IBOS *kernel.* This is because the message passing is implemented as system calls, which of course go the the microkernel operating system, which is tightly integrated with the IBOS kernel. The components of the IBOS architecture which we want to highlight are the following three:

- **The IBOS kernel.** The IBOS kernel builds upon the L4Ka microkernel and is the central component of the IBOS web browser. It takes care of traditional OS tasks, e.g., process creation and application memory management. Message passing is based on the L4Ka::Pistachio message passing implementation, forcing all messages through the kernel, and specifically allows the checking of the security policies. The case studies in Section 4.3 will show some of those policies.
- **Network process.** The network process is responsible for HTTP requests. It transforms HTTP data into a TCP stream and in turn into a series of Ethernet frames which are passed to the NIC driver.
- **Web apps.** A new web app is created for each individual page visit of the user; specifically, whenever a link is clicked or a new URL is entered into the address bar. A web app sends out the HTTP request to the network process, parses HTML and runs JavaScript and renders web content to a tab. Each web app is labeled with the origin of the HTTP request used at creation.

We will look into the modeling of parts of these IBOS browser elements in detail in Section 4.2.1.

4.2 Formal Modeling Methodology

The basis of our formal modeling for IBOS is the source code, explained by one of the developers, who clarified the design ideas when there were any questions. Any disagreement between the stated design intent and the source code were brought up for clarification with the developer. To be perfectly clear, in the end the intended design as stated by the developers took precedence over the actual source code reading we did, with discrepancies reported to the developer to be fixed.

The underlying operating system microkernel is not part of this modeling. As mentioned in Section 4.1, the microkernel could be replaced by the fully verified seL4 microkernel, if that is desired, and thus we assume the microkernel to be working without error. Also, the underlying hardware is not taken into account. Naturally, some level of abstraction between the source code and our model is of course unavoidable. For example, no actual memory addresses are used in the model, but just different pointers.

What is modeled is the *architecture* of IBOS, which includes: (i) the kernel; (ii) general message passing; (iii) web apps; (iv) network processes; and (v) network interface card access; to mention some of the most important pieces. Looking at the central piece, in the kernel we have the policy checking mechanism for messages, an address bar, the content currently displayed on the screen, etc. Indeed, the UI is also abstracted away into the kernel.

All messages are forced to go through the kernel and they are thus subjected to the policies it wants to enforce. This is already a design decision in IBOS, which the browser enforces, and it is reflected in our model in the way messages are passed. Each process can only directly send messages to the kernel, and the message will include the actual final destination in some way; but only the kernel is able to send messages to any of the processes. In our model, we ensure this by having two one-way pipes for messages for each process and the kernel, i.e., one incoming and one outgoing pipe. No process can access the pipes of another process, which forces all communication to go through the kernel. Thus, the kernel is the only connecting point and the policy checking is easily centralized.

Note that our formal modeling process, similar to our approach in Chapter 3, is done completely by hand. This of course creates two issues: one being that attacks found in the model might not actually be attacks in the real browser. That is easily checkable though, and we have no actual false positive attacks. The other issue is that the model is an abstraction of the actual browser, as well as a (possibly) imperfect translation of the code. So, all security guarantees given are based on our model are of course always with regards to the design and cannot guarantee the total absence of programmer introduced bugs in the browser implementation that are not covered in the design.

Keeping these caveats in mind, we would argue that the success in finding spoofing attacks on Internet Explorer, see Chapter 3, is a good indication that a formal modeling approach, as we are taking it here, can in the absence of attacks in the model give good assurance on the design. Indeed, this is the foundation upon which the whole browser rests. In comparison, having machine-based checking of the source code requires a specification to be included in the source code, and then that still needs to be grounded somehow. We believe the right grounding is a design that has been checked already, as shown in this chapter. Indeed we view our work here as complementary of, and a natural preliminary to, a future formal verification of IBOS at the code level, since design verification should precede code verification.

This chapter is different from Chapter 3 in that here we are not exclusively looking at visual invariants, even though we do look at the address bar spoofability in Section 4.3.2. We look also at connectivity properties inside the browser, specifically regarding connection between content from different URLs in the form of the same origin policy as in Section 4.3.3. The display memory is analyzed first, in Section 4.3.1.

4.2.1 IBOS Architecture Modeling

For the full model with explanations see Appendix B. In this section we point out key properties and give a general flavor of the model. At the top level, our state space is made up of objects with an object identifier, a type, and a set of attributes. Each network process, web app, and the kernel is modeled as a single object. To illustrate this, we show Figure 4.2. In that figure all objects outside the kernel are shown as rectangles. Note that pipes are a special kind of object that connects the objects at its left and right end. Other than that, arrows show connectivity. The ellipses inside the kernel contain relevant pieces of the kernel, that are not objects themselves. There will of course be multiple copies of most objects, except for the NIC, display and web app manager.

Let us start looking at the kernel, and particularly the message pass-

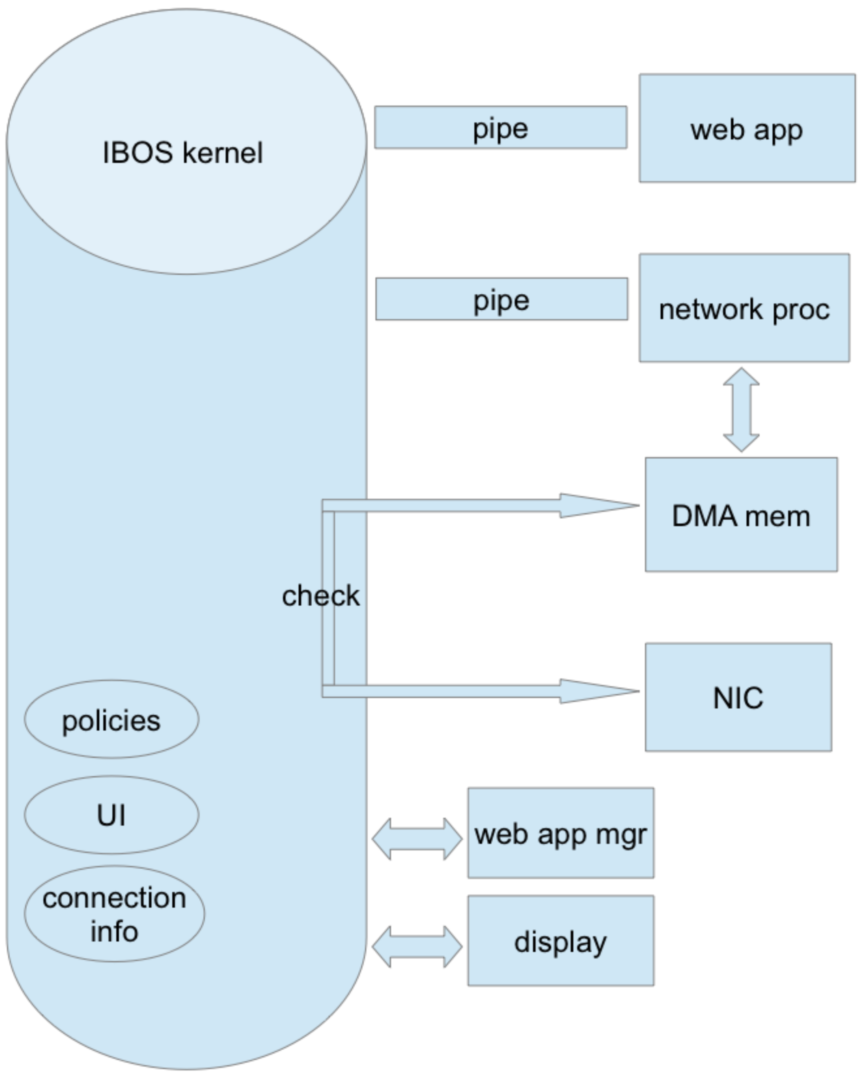


Figure 4.2: IBOS Model State

ing mechanism. First, we present more information on the messages. All messages are passed as system calls, where the browser-specific part of the message is encapsulated in the system call. First, the message part specific to the browser has the following format, which we call the `payload` of the encapsulating system call:

```
op payload : Oid Oid MsgType MsgVal
             String typed untyped -> Payload [ctor] .
```

The arguments of payload are the sender (as `Oid`), the receiver (as `Oid`), the message type (as `MsgType`), some auxiliary message info (as `MsgVal`), an argument commonly containing the URL that is requested or sent (as `String`) and two more arguments (`typed` and `untyped`) that could transport more data, and which we are going to ignore here. The sort `Oid` is that of object or process identifiers. Each web app, network process, etc., has an `Oid`. Note that the correct sender `Oid` is enforced by the kernel, as it knows which process sent the system call encapsulating this `payload`.

The actual message is then built using the `payload` and system call type:

```
op msg : SyscallType Payload -> Message [ctor] .
op OPOS-SYSCALL-FD-SEND-MESSAGE : -> SyscallType .
```

where `OPOS-SYSCALL-FD-SEND-MESSAGE` is the most commonly used type of system call for sending browser messages.

To model the fact that the kernel knows which process actually sent a message (as a system call) and to make sure that in the model no two processes can send messages directly to each other, but are forced to send messages via the kernel, the model defines one pipe object per process (using the same `Oid` as the associated process), which contains two one-way pipes, going to the kernel from the process and going to the process from the kernel:

```
op pipe : -> Cid [ctor] .

op fromKernel : MessageList -> Attribute [ctor] .
op toKernel : MessageList -> Attribute [ctor] .
```

Let us show an example pipe object for the process with `1050` as `Oid` which currently holds no message going either way:

```
< 1050 : pipe | fromKernel(mt), toKernel(mt) >
```

Suppose this process wants to send for example the message:

```
msg(OPOS-SYSCALL-FD-SEND-MESSAGE,
    payload(1050, 256, MSG-FETCH-URL, 0,
            l(http,dom("test"),port(81)),
            mtTyped, mtUntyped))
```

This message comes from web app `1050` and goes to (presumably) network process `256`, sending the message to fetch a URL (`MSG-FETCH-URL`) from the (fictional) domain `http://test:81`. This message would then be appended to the list of messages in `toKernel` in the pipe object. The kernel enforces correct sender `Oid` based on the pipe's id by simply changing the given sender `Oid`, if necessary.

As part of the policy checking when a network process and a web app communicate, their connection is checked. This means that both of them need to be linked to the same domain. This is modeled by the equation:

```
eq < kernel-id : kernel |
     handledCurrently(checkConnection(Num:Nat, Num':Nat, M)) ,
     weblabels(pi(Num':Nat, L:Label),
               WPIS:WebappProcInfoSet) ,
     networklabels(pi(Num:Nat, L:Label, L':Label),
                   NPIS:NetworkProcInfoSet) ,
     Att >
 =  < kernel-id : kernel |
      handledCurrently(M) ,
      weblabels(pi(Num':Nat, L:Label),
                WPIS:WebappProcInfoSet) ,
      networklabels(pi(Num:Nat, L:Label, L':Label),
      NPIS:NetworkProcInfoSet) ,
      Att > .
```

The property being checked here is that the receiving web app with id `Num':Nat` is associated to a URL `L:Label` in the kernel storage for web app connections `weblabels`, and that the sending network process with id `Num:Nat` is associated with the same URL `L:Label` in the network process connection storage `networklabels`. Then the message is simply being passed on, by dropping the `checkConnection` wrapper around the message `M`. The kernel is only handling one thing at a time, which is stored in `handledCurrently`. Once the current instruction has been dealt with, any of the currently incoming messages can become the next message to be executed. This is modeled by the rule:

```
rl [kernelReceivesOPMessage] :
  < kernel-id : kernel |
      handledCurrently(mt) ,
      msgPolicy(MP), Att >
  < ID : pipe |
       toKernel(
         msg(ST:SyscallType,
             payload(N, N', M:MsgType, V:MsgVal, S:String,
                        T:typed, U:untyped)), ML) ,
                Att2 >
  =>
  < kernel-id : kernel |
        handledCurrently(policyAllows(
           msg(ST:SyscallType,
                payload(ID, N', M:MsgType, V:MsgVal, S:String,
                        T:typed, U:untyped)), MP)) ,
        msgPolicy(MP), Att >
  < ID : pipe | toKernel(ML) , Att2 > .
```

Note that the kernel does not take the message to be dealt with directly, but wraps the actual message inside the `policyAllows` operator together with the set of message policies `MP` as an extra argument, which is an attribute of the kernel wrapped in `msgPolicy`. Also, in the message the sender id `N` which was given by the sender is forcibly changed to the actual sender id `ID`, which is the process id of the pipe (and thus the associated process).

For the network process we are using (as does IBOS) the process id 256 through 1023. The attributes of a network process are:

```
op returnTo : ProcId -> Attribute [ctor] .
op in : LabelList -> Attribute [ctor] .
op out : LabelList -> Attribute [ctor] .
```

The `returnTo` attribute stores the process id of the web app that this network process will return data to, while the attributes `in` and `out` hold the lists of labels (representing URLs) that the network process will ask data from and has received data from already. The simplification we use here is to not use the HTML code from a given URL, but just use a URL as representing the data from that URL.

For web apps we are using the process id 1024 through 1055. Their attributes are:

```
op rendered : Label -> Attribute [ctor] .
op URL : Label -> Attribute [ctor] .
op loading : Nat -> Attribute [ctor] .
```

The label inside `rendered` is the URL for which the web app has put the data on the screen, provided it is the active web app. The label inside `URL` is the location where this web app wants to load data from. `loading` is just a binary flag indicating whether the web app has already sent a request to load data. Initially, the `rendered` field for a new web app will be empty, and `loading` is `0`, meaning that it has not yet started to load. This equation sends the message to start loading:

```
eq < N : proc | rendered(L) , URL(L') , loading(0) , Att >
   < N : pipe | toKernel(ML) , Att2 >
 = < N : proc | rendered(L) , URL(L') , loading(1) , Att >
   < N : pipe | toKernel(ML,
        msg(OPOS-SYSCALL-FD-SEND-MESSAGE,
            payload(N, network-id, MSG-FETCH-URL, 0, L',
                    mtTyped, mtUntyped))) ,
                  Att2 > .
```

The message is sent to fetch the data from URL `L'` and the `loading` attribute changes to `1`. On return of the requested data, `rendered` will change to `L'`.

The hardware pieces of Figure 4.1, video card, NIC, etc., are not modeled in any detail. Only the NIC is modeled, and it receives target URLs from the memory set aside for this purpose through the kernel, and then, after a potential delay, returns the representation of the resulting data.

For the display memory case study in Section 4.3.1, the model has been extended with the required notions of memory, page table and page faults. This extended model exposed a bug in IBOS.

4.3 Case Studies

Our analysis of the IBOS web browser includes three different case studies. In the first study, we analyze the display memory and find a bug that has been fixed for IBOS. For the second study we analyze the address bar correctness. In the third study, we look at the *Same Origin Policy* (SOP), which comes in the form of a number of sub-properties that are required for SOP to hold that were proposed in [117]. The second and third case studies show that the browser design, as modeled, using appropriate reductions in our proofs, is secure and so these good properties are true of the actual browser.

4.3.1 Case Study 1: Display Memory

In IBOS, only the currently active web app is able to write to the display memory, i.e., change what the user sees on the screen in the content area. This is a security feature that prevents other web apps from manipulating the output of the current web app and makes them unable to eavesdrop content that is being displayed. In broad terms, the way this is handled is that the display memory is completely flushed whenever the active web app changes, the old active web app loses its access to the display memory and the new active web app gains access to that display memory. The IBOS developers knew that under some (at that point unknown) circumstances it could happen that the browser's content area simply was empty and further did not update upon switching from one tab to another. Note that this is not a security concern. No data can be leaked and no mis-match between the content area and the displayed URL is possible. This is rather a usability concern as that tab became unusable.

We modeled the interaction between the web apps, the kernel (holding the page table) and the memory abstraction we use. The cleansing of the display memory is enforced by the memory page table, which resides in the kernel. To illustrate this, let us first describe how the bug that we ultimately uncovered in the model works. The bug is reproducible in the actual browser and a fix also has been proposed.

1. Assuming a single web app, `A`, that is currently active, we note that the `display-mem` pointer of `A` (used for accessing the video memory) goes to the location `A-VID` in the page table, and that maps to the actual video memory, and displays the content of the web page `A` is associated with.

2. Adding a second web app `B` by creating, e.g., a new tab (this makes `B` the active web app) has the following effects:

    - The page table in the kernel maps `A-VID` to `NULL` instead of the video memory.
    - The `display-mem` pointer of `B` points to `B-VID` in the page table, which in turn points to the video memory, and the content that `B` is associated with gets displayed.

3. A redraw request for `A`, while it is not the active web app, is the crucial piece of the puzzle for this bug. When that happens, the page fault for `A-VID` is dealt with by making it point to some memory we call `dummy` memory. The changed content for `A` is put there, but that memory is of course not presented on the screen.

4. When the user then switches back to the tab containing `A` the behavior is similar to before when the new tab was created:

   - The page table in the kernel maps `B-VID` to `NULL`.
   - But, as `A-VID` already points to `DUMMY`, there is no page fault when that memory gets updated. Without a page fault, `A-VID` is *not* mapped to the video memory. Therefore, the display stays blank.

The key lesson here is that an update to a background web app will lead to the content area of the screen not properly updating when the user switches back to that web app. Usually, such updates will happen to the active web app only, but if it does happen in the background, this problem appears. The reason for this to happen is that page faults are used to (re-)assign the pointers in the page table. The simple fix is to force the pointer `A-VID` from above to change appropriately.

Knowing how to fix the design allowed us to propose a fix for the actual implementation as well. The model allowed us to extract the required order of tab switching and data loading that lead to the browser exhibiting this error. We explain how we find it, in the rest of this section.

First, let us explain a necessary part of the search exploration, which is the command `explore-space`:

```
op explore-space :  -> Configuration .
eq explore-space =  < testMsg : testMsg | cmd( explore ) > .
```

where `testMsg` is a wrapping process, which allows this to be put at the top level of our multi-set of processes, and `cmd` is a wrapper allowing this to follow the usual way of storing information in process attributes. The key is the `explore` command inside:

```
op explore : -> Cmd .
op explore : Nat -> Cmd .
```

```
rl explore => explore(3) .
rl explore(0) => mtCmdList .
rl explore(s(N:Nat)) => new-tab , explore(N:Nat) .
rl explore(s(N:Nat)) => update , explore(N:Nat) .
rl explore(s(N:Nat)) => tab-switch , explore(N:Nat) .
```

where the number 3 can be replaced by any desired number. The `explore` command will then unroll, step by step, and at each step will create either a `new-tab` command, an `update` command or a `tab-switch` command. These simulate user input. As `explore` is defined by rules, the search we use this command in will explore all combinations of all orders and repetitions of these commands. We do not show how each of those commands will additionally get assigned to it one of a number of possible URLs, as well as one of the existing web-apps (for `update` only), as needed. Again, all of the possible combinations will be explored. The `explore` command indeed includes creation of new tabs, loading data to any existing web app, and switching between web apps.

We give the following search command, that in `initial-test` starts with one active web app, and in `explore-space` contains the exploration of different commands as we have just described.

```
search in MEMORY : initial-test explore-space
  =>! X:Configuration
      < kernel-id : kernel |
           Att:AttributeSet, activeWebapp(N:Nat),
           pg-table(pg:PGTESet,
                      pg-table-entry(N:Nat, otherMemory)),
           vidMem(about-blank) > .
```

The goal state we are looking for in this search command is one in which there is an active web app, but the page table entry for that web apps memory is pointing at the afore-mentioned dummy memory, noted as `otherMemory` here, while the content of the actual `vidMem` is empty, shown by the `about-blank` inside.

The search does lead to a number of states that are similar modulo some renaming, and removal of unneeded instructions. By looking at those result states, and at the trace leading to them, we can see what happens, and find the order of actions, which we have described above already, that leads to the display memory becoming empty and unchanging for a particular web app. Now we can distill the commands from that exploration to a list of three commands, which we call `bug-trigger`:

```
eq bug-trigger =  < testMsg : testMsg |
       cmd( new-tab(Url2) , update(1050, Url3) ,
            tab-switch(1050)) > .
```

where the `testMsg` process wraps the `cmd` wrapper which includes the actual set of commands that triggers the bug. Note that `Url2` and `Url3` are just any URLs and `1050` is the process id of the initially existing web app process from `initial-test`.

In the search above we can replace `explore-space` by `bug-trigger` and then we get a single resulting goal state, showing this bug. To note again, this is not a security issue but it certainly is a usability issue that the modeling has been able to uncover.

4.3.2 Case Study 2: Address Bar

Another important property for a web browser is the trustworthiness of user interface elements, in their capacity to counter spoofing attacks. Particularly, the address bar needs to be trustworthy, so that the user always knows which site is actually being visited right now. It is truly important to know whether the currently visited site is really his/her banking web site, where entering credentials is fine, or if it is instead a phishing web site, where if the user enters his/her account information monetary loss is imminent. We all know that it is possible, even simple, for malicious attackers to create phishing web sites that are indistinguishable on the surface from the real web sites. A careful user should be able to trust the address bar, to prevent such phishing from succeeding. Also see Section 3.3 about the address bar spoofing possibilities we found in our Internet Explorer analysis.

As IBOS is designed with security in mind, our goal in this section is not only to find flaws that could be abused by attackers, if they exist, but also, more importantly, in the case of the absence of such flaws we will be able to gain a higher level of assurance that no such spoofing attacks are possible. The concern about the address bar is a security concern, so it is more important than the usability concern we have looked at in Section 4.3.1.

The important property for the address bar is that the content of the displayed page is always from the address which is displayed in the address bar. In our model, the kernel keeps track of the address bar by means of the data stored in the `displayedTopBar`. The source of the content being displayed is stored in the display process abstraction, which has the `displayedContent` field to store the information. At all times, the content of both these fields needs to be the same, the exception being when there currently is no content, which is modeled by the `about-blank` URL. If one of the two fields is empty, in that sense, the other one can have any value.

We start the search for potential attacks, in the form of a mismatch of these two fields, from an initialized kernel, together with the driver `inspect-space`. We are looking for any configuration in which there is a mismatch between the value of `displayedTopBar` and `displayedContent`. If no solution to this search is found, then there is no attack.

```
search init-simp-kernel
inspect-space
=>*
  X:Configuration
  < kernel-id : kernel | Att:AttributeSet ,
          displayedTopBar(URL:Label) >
  < display-id : proc |
    displayedContent(URL':Label),
    Att2:AttributeSet >
such that URL:Label =/= URL':Label
     and  URL:Label =/= about-blank
     and  URL':Label =/= about-blank .
```

First let us note that `inspect-space` is similar to `explore-space` from Section 4.3.1:

```
op inspect-space :  -> Configuration .
eq inspect-space = < testMsg : testMsg | cmd( inspect ) > .
```

with the same `testMsg` wrapping process and `cmd` the wrapper for the actual sequence of commands to be tested. The key here is the `inspect` command. We will call the rules for `inspect` the *trigger rules*, and write them as R_T. All other rules we have presented here, and in Appendix B, belong to the *internal rules* of the model, written as R_I. We are working modulo the equations E, which are all the equations given here and in the appendix. So we are actually rewriting with $\rightarrow_{R_{(I \cup T)/E}}$, which can be split into $\rightarrow_{R_{I/E}}$ and $\rightarrow_{R_{T/E}}$. We will use the short-hands $\rightarrow_I$ and $\rightarrow_{I/E}$ (resp. $\rightarrow_T$ and $\rightarrow_{T/E}$) to represent $\rightarrow_{R_{I/E}}$ (resp. $\rightarrow_{R_{T/E}}$).

```
op inspect : -> Cmd .
op inspect : Nat -> Cmd .
rl inspect => inspect(3) .
rl inspect(0) => mtCmdList .
rl inspect(s(N:Nat)) => new-url , inspect(N:Nat) .
rl inspect(s(N:Nat)) => switch-tab , inspect(N:Nat) .
```

This shows that `inspect` is unrolled step by step. The number `3` can of course be changed, but that number is picked in particular so that two web apps can be created and the tab can then be switched as well. This is enough to show the property of our choice here, as we explain below. At each step either a `switch-tab` or `new-url` will be generated. This simulates user input again. Compared to the prior section there is no explicit `update` here as the current active web app can update the content area at any time and we do not have to explicitly force that. As `inspect` is defined by rules, the `search` command will create all possible combinations. Not shown here is how `new-url` gets assigned a new URL and how `switch-tab` picks any of the web apps to be the new active web app. Here, we do work on a model without the display memory bug that has been exposed in Section 4.3.1. Indeed, when we run the above search command we get the result that there are no solutions:

```
No solution.
states: 247743  rewrites: 3663864 in 247886ms cpu
                (248055ms real) (14780 rewrites/second)
```

We now have to discuss what this really tells us about the browser and the security of the address bar. In Section 4.2 we have already discussed the limitations and conditions of our approach in general, but now we can look into this specific case study. First, note that the two objects we care about, the address bar and the content as stored in the display process, are both stateless objects. That is, they have no memory what was stored in them before, but only know what is there right now.

Both the address bar and the display content are only changed due to the current web app interacting with the kernel when created or when the tab is switched to it. To create a mis-match between the two, two different URLs are all that is needed, which can be provided by just two web apps. This allows us to make the reduction that only the last two web apps that are on the screen need to be taken into account. The rest of the browser model state and the length of the run of the browser model is irrelevant and thus abstracted away.

Assume we needed to consider a third web app, then that would only be the case if that web app made a change to either of the two objects in question; but then one of the other two does not make a change (or does a duplicate one), so then that other web app becomes irrelevant and we are back to the case of two web apps. If there was a way for more than two web apps to create such a mis-match, then the deciding last step (we would stop at such a mis-matching point) must be either a new web app being added or the tab being switched. But then, that whole trace of actions and number of web apps can be simplified to just the state before that last action, with only the old active web app and the new active web app taken into account to create the exact same mis-match. Now we can focus on the interaction of only two web apps, which requires search up to depth three, due to the need of also allowing a tab switch.

Now, due to the reduction we can conclude that, since there are no mis-matches for the limited number of web apps and steps, there will not be any such mis-match at all. Now that we have sufficiently motivated the property, let us make it precise and formal by presenting first a necessary lemma and then our key theorem.

Internal Normalization Between Trigger Rules

Let us first note that this is a general observation, that will also be helpful in Section 4.3.3. There is no interference between internal rules I and trigger rules T, i.e., we can re-order them in any way we please. In particular, we like to normalize with the internal rules after each execution of a trigger rule. That means, for execution using both internal rules and trigger rules, $\rightarrow^{*}_{(T\cup I)}$, we will rearrange that to $\rightarrow_T\rightarrow^{!}_{I} \ldots \rightarrow_T\rightarrow^{!}_{I} \ldots \rightarrow_T\rightarrow^{*}_{I}$. The last set of internal rules does not have to be carried all the way to normalization, to take into account the fact that the combination of trigger and internal rules might not normalize either. Let us phrase this claim formally as a lemma, noting that by $\rightarrow^{i}_{T/E}$ we mean the i-th use of a rule from T/E:

Lemma 1 *Given terms s_1 and s_2, for any chain of rewrites of the form $s_1 \rightarrow^*_{(T\cup I)/E} s_2$, with n uses of trigger rules, we can rearrange that sequence, using the same rewrites, to $s_1 \rightarrow^1_{T/E}\rightarrow^!_{I/E} \dots \rightarrow^i_{T/E}\rightarrow^!_{I/E} \dots \rightarrow^n_{T/E}\rightarrow^*_{I/E} s_2$.*

Proof. *Given any state t for which both an internal rule I_0 and a trigger rule T_0 are available for rewriting, we claim that the application of these rules commutes. That is, there are t' and t'' so that $t \rightarrow_{I_0} t' \rightarrow_{T_0} s$ as well as $t \rightarrow_{T_0} t'' \rightarrow_{I_0} s$. Now, let us prove this claim.*

- *Using a rule I_0 to go from t to t' will still leave the rule T_0 enabled, because all rules in T use only `inspect` on the left hand side, while none of the rules in I use `inspect` at all.*
- *Using a rule T_0 to go from t to t'' also leaves rule I_0 enabled as any of the rules in T transforms `inspect` (which is unusable by any rule in I anyway) but leaves the rest of the state intact.*

The resulting term s is indeed the same, for either order of internal and trigger rule execution, if both are enabled. Now, based on the proof of any one step commuting, we can actually rearrange the whole rewrite sequence so that we always take all possible steps using internal rules first. Only upon hitting a normal form by the internal rule steps will we take a single step with a trigger rule. We then repeat this. This is illustrated in Figure 4.3, where I_i is an internal rule, T_i is a trigger rule, and ! means that we go for normalization using rules from I.

We can now consider the effect of each trigger rule on the state by itself. We let the model do all internal computations until finished before using another trigger.

General Notes on Trigger Rules

Let us discuss the results of each trigger in a rewriting sequence first, which will be relevant in Section 4.3.3 as well. Note that we consider not just the trigger rule application, but the following normalization by the internal rules, which is associated to this trigger. There are two kinds of triggers, `switch-tab` and `new-url`. Let us look at them one at a time:

`switch-tab`: The `switch-tab` trigger makes the kernel switch the active tab, as if a user interaction to switch had happened. In particular, in

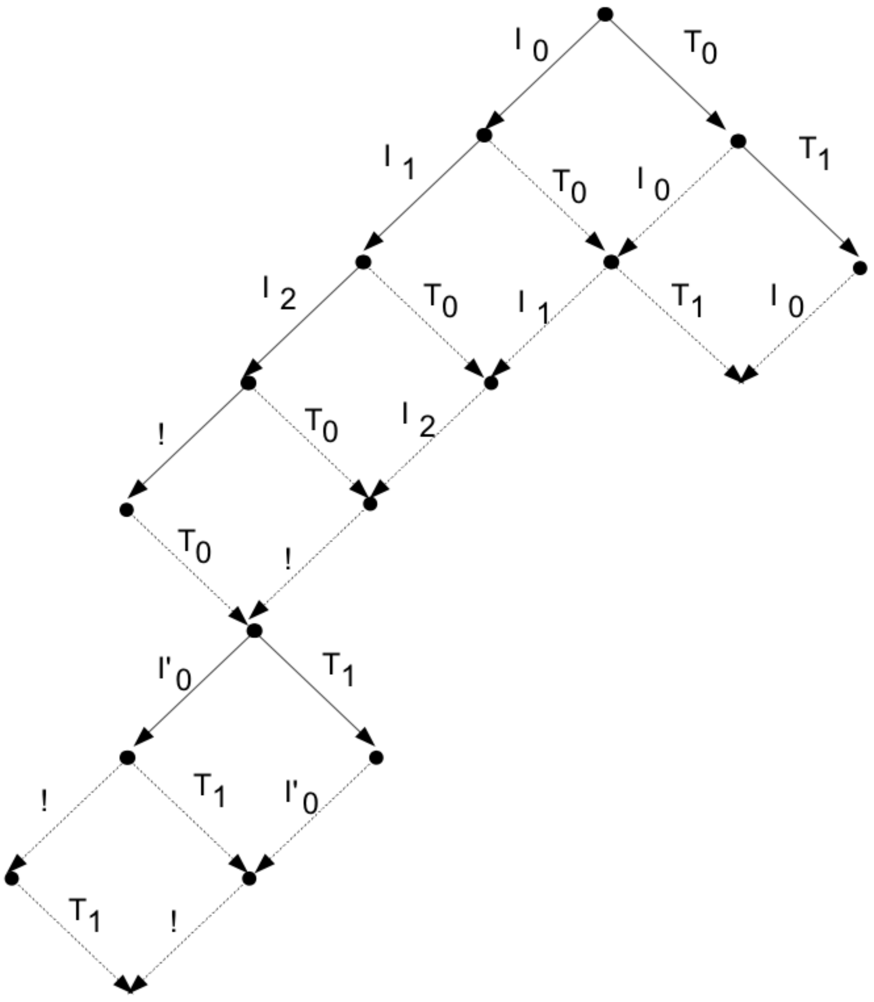


Figure 4.3: Commuting Diagram for Internal and Trigger Rules

the model it can impact the `activeWebapp`, the `displayedContent` and the `displayedTopbar`. Keep in mind that all of these data fields are without history, only the current value is retained.

`new-url:` The `new-url` trigger models the user giving a URL. It will lead to the creation of a new web app, find or create an appropriate network process, and transfer data from target URL (request and response) via the NIC. It will also extend the mapping of web apps and network processes to URLs, and thus each other. Also, the active tab is switched, as described above in the summary of `switch-tab`. Each `new-url` is independent of all prior triggers of the type.

Note that in both cases, if no violation is found at the trigger step (including the following normalization by internal transitions), then the trigger potentially increases the state space size. But, as all URLs and process ids are generic in the model (i.e., rules are blind with regards to the exact id or URL) this trigger could be ignored anyway, unless it adds the process that is active at the time of a violation being found, or if it adds the specific URL needed for the violation.

Also, note that the kernel mappings of processes to URLs, for both web apps and network processes, never has any element modified, but is only added to.

Internal Rules Termination

Let us now consider the internal rules and make sure that they actually do terminate, as the above reordering of trigger rules and internal rules does rely on this fact. Essentially, the internal rules deal with the passing of messages, and those messages get consumed ultimately. New messages are only created as responses to the consumption of existing messages, but those messages have less potential for spawning further messages down the line. There is a clear order on the rules that does not include any loops. That is, an initial message will get passed around and transformed into different messages, but will ultimately disolve once its travels are completed.

Additional messages can of course be added to the system by trigger rules. Initially (not taking trigger rules into account) a system will contain a fixed number of messages, potentially in different points of this descending chain of possible rule applications. Each of this set number of messages will be consumed in the end and the rewrite system using the internal rules, without the trigger rules, will thus terminate.

We look at a single generic *data transfer* and explain how its messages will go from one rule to the other but will descend in the number of potential further rule applications. The internal rules do terminate, and the details for this with the detailed order of the internal rules can be found in Appendix B.2.

Address Bar Correctness Proof

Due to the previously discussed Lemma 1 we only need to consider sequences of the form $\rightarrow_T \rightarrow_I^! \rightarrow_T \rightarrow_I^! \ \ldots \ \rightarrow_T \rightarrow_I^*$. As we have shown, our bounded model checking has analyzed all sequences with at most 3 trigger rules being used, and found no possible violation. So, the address bar is correct for all sequences with at most 3 trigger rules. We now state and prove a theorem, showing that this correctness extends to sequences with *any* number of trigger rules being used.

Theorem 1 *The property of address bar correctness holds for any rewrite sequence, using any number of trigger rule steps.*

Proof. *The base case of at most* 3 *trigger rule steps is proven by the above model-checking analysis. The reduction given in Lemma 2 then completes the proof by reducing any violation of greater length to use at most* 3 *trigger rule steps, so no violation can exist.*

We now give the reduction lemma that assures us that all longer sequences can be reduced to shorter ones:

Lemma 2 *Any sequence of trigger rule steps that leads to a violation of the address bar correctness and uses 4 or more trigger rule steps can be reduced by a step. This yields that all the possible trigger rule sequences leading to a violation must be of length 3 or less.*

Proof. *First note that for all rewrite sequences with at most* 3 *trigger rule steps used, our model-checking analysis has proved that there are no violations. Now let us assume that our claim is incorrect, that is, that there is at least one sequence of rewrites for which a violation of the property is found in* 4 *or more steps. Let us pick any such sequence with the smallest possible number of trigger rule steps used, and let that number be N, obviously with $N \geq 4$.*

Looking at the search command above, which establishes correctness for up to 3 triggers, we see that in a violation there need to be two different URLs, so there must have been at least two trigger rule steps of the type `new-url(U)` *that have been used. Looking at the violation description (in the* `search` *command above), we find that for one of them* `U` $=$ `URL:Label` *and for the other* `U` $=$ `URL':Label`*. Naturally, there will also be the last trigger rule step used, after which the violation is exhibited. Let us denote that last trigger by n, while we denote the two* `new-url` *cases by l_1 (with* `URL:Label`*) and l_2 (with* `URL':Label`*). Also, note that if one of those triggers appears* *more than once, we can always pick its last appearance for l_1 respectively l_2. Now, whenever n is not the same as l_1 and not the same as l_2, then there are two cases, one where l_1 happens before l_2 (written $l_1 < l_2 < n$) and one where l_2 happens before l_1 (written $l_2 < l_1 < n$). In all cases, n is the last trigger happening. We will consider the special cases of $n = l_1$ and $n = l_2$ later. We will now look at sequences leading to a violation and show that we can construct a sequence using fewer triggers, in contradiction to having picked the sequence with smallest N.*

Let us first look at the case where $l_1 < l_2 < n$, with a violation occurring during the internal steps after n (or at n itself). As we know that $N \geq 4$, there will be at least one further trigger rule step used, let us call it k. That leads to the case distinction of such an additional rule step being before the three other triggers, between l_1 and l_2 or between l_2 and n. If there is more than one such trigger rule step happening, we will pick one which is before l_1 if possible, between l_1 and l_2 if there is one there and no step before l_1, and only pick a step after l_2 if there are no other steps earlier.

Case 1: $k < l_1 < l_2 < n$. There can be multiple additional trigger rule steps before l_1, we actually pick the immediate predecessor of l_1 for k.

- *Trigger rule step k is a `switch-tab`: This changes the active web app immediately before introducing the URL in l_1 which appears in the violation. Upon the execution of the trigger in l_1 and normalization by the internal rules, all changes made by this `switch-tab` are completely undone, and as there was no violation here (or we would have a shorter sequence already) we can drop this `switch-tab` trigger from our list of triggers to be executed, and will reach the same violation at the end of the sequence, which is now one step shorter.*

- *Trigger rule step k is a `new-url(U)`: Using this trigger rule step does increase the state space size by adding all the pieces mentioned before in the description of what such a trigger does. Now, if `U` $\neq$ `URL:Label` and `U` $\neq$ `URL':Label`, then all the additions are part of the state, but no violation occurs here. Whatever k added is just data being carried in the state that has no effect on the violation that will be found later as it is for a different URL. Thus, we can shorten the sequence by dropping this trigger k and will get the same violation at the end, but with a shorter sequence.*

In case `U` $=$ `URL:Label`*, then the following step* l_1 *will duplicate that trigger step, in which case we only need the last one of the two, and can again drop the trigger* k *and have a sequence that is one shorter with the same resulting violation.*

In case `U` $=$ `URL':Label`*, then the step* l_2 *which happens later will duplicate it, and again we only need the last one, so we can drop this trigger and the resulting sequence will give rise to the same violation but be one shorter.*

Case 2: $l_1 < k < l_2 < n$*. There can be multiple additional trigger rule steps between* l_1 *and* l_2*, we actually pick the immediate predecessor of* l_2 *for* k*. This works quite similar to case 1.*

- *Trigger rule step* k *is a* `switch-tab`*: This changes the active web app immediately before introducing the URL in* l_2 *which appears in the violation. Upon the execution of the trigger in* l_2 *and normalization by the internal rules, all changes made by this* `switch-tab` *are completely undone, and as there was no violation here (or we would have a shorter sequence already) we can drop this* `switch-tab` *trigger from our list of triggers to be executed, and will reach the same violation at the end of the sequence, which is now one step shorter.*

- *Trigger rule step* k *is a* `new-url(U)`*: Using this trigger rule step does increase the state space size by adding all the pieces mentioned before in the description of what such a trigger does. Now, if* `U` $\neq$ `URL:Label` *and* `U` $\neq$ `URL':Label`*, then all the additions are part of the state, but no violation occurs here. Whatever* k *added is just data being carried in the state that has no effect on the violation that will be found later as it is for a different URL. Thus, we can shorten the sequence by dropping this trigger step* k *and will get the same violation at the end, but with a shorter sequence.*

  In case `U` $=$ `URL:Label`*, then this is duplicating the prior step* l_1*. We stated earlier that we pick* l_1 *to be the last appearance of the trigger rule step* `new-url(URL:Label)`*, so this case is moot.*

  In case `U` $=$ `URL':Label`*, then the following step* l_2 *will duplicate that trigger step, in which case we only need the last one of the two, and can* *again drop the trigger step k and have a sequence that is one shorter with the same resulting violation.*

Case 3: $l_1 < l_2 < k < n$. There can be multiple additional trigger rule steps between l_2 and n. If there is any `new-url(U)` type trigger, we pick the last of them as k. If there are only `switch-tab` triggers, we pick the immediate predecessor of n for k.

- *Trigger rule step k is a `new-url(U)`: Using this trigger rule step does increase the state space size by adding all the pieces mentioned before in the description of what such a trigger does. Now, if `U` $\neq$ `URL:Label` and `U` $\neq$ `URL':Label`, then all the additions are part of the state, but no violation occurs here. Whatever k added is just data being carried in the state that has no effect on the violation that will be found as it is for a different URL. Thus, we can shorten the sequence by dropping this trigger step k and will get the same violation at the end, but with a shorter sequence.*

  In case `U` $=$ `URL:Label`, then this is duplicating the prior step l_1. We stated earlier that we pick l_1 to be the last appearance of the trigger rule step `new-url(URL:Label)`, so this case is moot.

  In case `U` $=$ `URL':Label`, then this is duplicating the prior step l_2. We stated earlier that we pick l_2 to be the last appearance of the trigger rule step `new-url(URL':Label)`, so this case is also moot.

- *Trigger rule step k is a `switch-tab`: This changes the active web app immediately before the step n which creates the violation. Let us first emphasize that there are no `new-url(U)` triggers in the sequence except for l_1 and l_2. All the triggers between l_2 and n are of the `switch-tab` form.*

  If there are multiple such triggers, then in addition to k directly before n there is another one, call it m. Now, trigger m and all its changes will be completely undone by trigger k. So, trigger m is not needed, and removing trigger m will still yield the exact same violation.

  If there is only one such trigger, then we have $l_1; l_2; k; n$ as the sequence giving a violation. But, then k can only switch the active tab to either the web app associated to l_1 or l_2. In the case of k making the web app *of l_1 the active one, we get the same result by moving l_1 to the spot of k in the order and dropping k, that is, $l_2; l_1; n$ will yield the violation and be shorter. On the other hand, if k makes the web app of l_2 active, that is not needed, as it already is the active one at that point. So $l_1; l_2; n$ will similarly yield the violation and it is shorter.*

Now we need to look at the case when $l_2 < l_1 < n$, but it works just like the case of $l_1 < l_2 < n$, so we do not repeat all the arguments.

This leaves us with the two cases where either $l_1 = n$ or $l_2 = n$.

First, we look at $l_1 = n$. This means we have $l_2 < n$, so any additional trigger k can either be before or after l_2, giving rise to two cases.

Case 1: $k < l_2 < n$.

- *Trigger rule step k is a `switch-tab`: This changes the active web app immediately before introducing the URL in l_2 which appears in the violation. Upon the execution of the trigger in l_2 and normalization by the internal rules, all changes made by this `switch-tab` are completely undone, and as there was no violation here (or we would have a shorter sequence already) we can drop this `switch-tab` trigger from our list of triggers to be executed, and will reach the same violation at the end of the sequence, which is now one step shorter.*

- *Trigger rule step k is a `new-url(U)`: Using this trigger rule step does increase the state space size by adding all the pieces mentioned before in the description of what such a trigger does. Now, if `U` $\neq$ `URL:Label` and `U` $\neq$ `URL':Label`, then all the additions are part of the state, but no violation occurs here. Whatever k added is just data being carried in the state that has no effect on the violation that will be found later as it is for a different URL. Thus, we can shorten the sequence by dropping this trigger k and will get the same violation at the end, but with a shorter sequence.*

  In case `U` $=$ `URL:Label`, then this is duplicating the later step n, in which case we only need the last one of the two, and can again drop the trigger k and have a sequence that is one shorter with the same resulting violation.

  In case `U` $=$ `URL':Label`, then the following step l_2 will duplicate that trigger, in which case we only need the last one of the two, and can *again drop the trigger k and have a sequence that is one shorter with the same resulting violation.*

Case 2: $l_2 < k < n$. There can be multiple additional trigger rule steps between l_2 and n. If there is any `new-url(U)` type trigger, we pick the last of them as k. If there are only `switch-tab` triggers, we pick the immediate predecessor of n for k.

- *Trigger rule step k is a `new-url(U)`: Using this trigger rule step does increase the state space size by adding all the pieces mentioned before in the description of what such a trigger does. Now, if `U` $\neq$ `URL:Label` and `U` $\neq$ `URL':Label`, then all the additions are part of the state, but no violation occurs here. Whatever k added is just data being carried in the state that has no effect on the violation that will be found as it is for a different URL. Thus, we can shorten the sequence by dropping this trigger k and will get the same violation at the end, but with a shorter sequence.*

  In case `U` $=$ `URL:Label`, then this is duplicating the later step n, in which case we only need the last one of the two, and can again drop the trigger k and have a sequence that is one shorter with the same resulting violation.

  In case `U` $=$ `URL':Label`, then this is duplicating the prior step l_2. We stated earlier that we pick l_2 to be the last appearance of the trigger rule step `new-url(URL':Label)`, so this case is moot.

- *Trigger rule step k is a `switch-tab`: This changes the active web app immediately before the step n which creates the violation. Let us first emphasize that there are no `new-url(U)` triggers in the sequence except for l_2. All the triggers between l_2 and n are of the `switch-tab` form.*

  If there are multiple such triggers, then in addition to k directly before n there is another one, call it m. Now, trigger m and all its changes will be completely undone by trigger k. So, trigger m is not needed, and removing trigger m will still yield the exact same violation.

  If there is only one such trigger, then we have $l_2; k; n$ as the sequence giving a violation. k can change the active web app but there is only one (created by l_2) and it is already active, so we can drop k and have a *shorter sequence leading to the violation. (This sequence is only of length three anyway, and thus, by model-checking, has already been proven to have no violation.)*

Then, we consider $l_2 = n$, so we have $l_1 < n$ and additional triggers k are either before l_1 or after l_1, yielding two cases. But, this works just like the case of $l_1 = n$ so we do not repeat all the arguments.

So, in each case we have shown that there is a sequence one shorter, which still violates the property. This is in contradiction to our assumption that we have picked the sequence with the smallest number of triggers, N. □

The lemma that allows reduction of all sequences to 3 or less trigger rule steps, proved just above, together with the model-checking analysis of the base case of up to 3 trigger rule steps thus prove that the address bar does behave correctly at all times. This holds for *any* number of trigger rule steps used in a sequence.

4.3.3 Case Study 3: Same Origin Policy and More

The *same origin policy* (SOP) is the primary security policy that all modern browsers implement. We present a very short summary here. A much more complete discussion of this policy is available in [115].

In essence, SOP is a non-interference policy. It is designed to isolate web pages, including all the associated information regarding them, by their source. The labels being used for this are the domains of their origin in the form of an URL. If the browser has opened two web pages from domain `A` and domain `B`, a correctly implemented SOP will enforce these two web pages to be isolated. It turns out that commodity browsers do not do a very good job of doing this correctly [24], due to the fact that the required checks are scattered through their large code base.

Each web page is a frame, containing a HTML document and any material linked from that HTML document. It can include references to network objects, e.g., images, JavaScript, etc. When downloading these elements the browser will label them with the frame level URL. In some sense, the original web page is responsible for all elements it loads. Say, the top level URL is `A`, but a JavaScript from `B` is downloaded. That JavaScript then runs with the permissions of `A`, not those of `B`. This means that the script can access all the information currently loaded from `A` in this frame, but not any information from `B`, even if there is a separate frame from `B`. For linked HTML elements SOP is more restrictive in that it requires those objects to have the same source as the frame level.

To motivate how crucial the SOP is, let us look at a standard browsing experience: you have two web pages open, where one of those is your bank's web site. If the other currently open web page turns out to be malicious (say, you clicked on some random link by accident), we want to ensure that it will not be able to get information about your bank account, or worse, make any changes to your bank account or start a transaction. As the two pages will be from different origins (otherwise your bank has been hacked already) there should be no way for them to communicate.

In [117] a number of security considerations for IBOS are presented, and a subset of those turns out to be the SOP. In this chapter we will look more closely at those security requirements which result in the browser implementing SOP. As it turns out, our verification in the Maude model shows that these properties are true, and thus that IBOS indeed implements the SOP.

To model check this property in our browser model, we use the model of the internal logic of the browser which we have mentioned already in Section 4.2; it includes the policies being enforced by the kernel. We already noted that all messages go through the kernel and thus are subject to being checked with respect to the policies. We then also have to create canonical messages that different components can try to send to each other. That is, we need a small set of messages that is *generic*, so that the instances of these generic messages can cover all messages. Then the model checking analysis can in fact verify that none of those messages can reach disallowed destinations.

Theorem 2 *The Same-Origin Policy holds for any rewrite sequence, using any number of trigger rule steps.*

Proof. *The proof consists of proving the properties* (1)–(7) *below. This is done in the remainder of this section and in particular in Theorems 3–7.* □

Now that we have completed the high level overview of the SOP, we will look at each of the properties that altogether make up SOP in the context of IBOS one at a time. These properties are based on [117]:

1. The kernel must route network requests from web page instances to the proper network process.
2. The kernel must route Ethernet frames from the network interface card (NIC) to the proper network process.
3. Ethernet frames from network processes to the NIC must have an IP address and TCP port that matches the origin of the network process.
4. HTTP data from network processes to web page instances must adhere to the SOP.
5. Network processes for different web page instances must remain isolated.
6. The browser chrome (UI elements) and web page content displays are isolated.
7. Only the current tab can access the screen, mouse, and keyboard.
8. All components can only perform their designated functions.
9. The URL of the current tab is displayed to the user.

The SOP is given by properties (1)–(7). Property (8) is another good property for IBOS, while property (9) aids in verifying property (7).

Let us look at the first property which is part of SOP:

- (1) *The kernel must route network requests from web page instances to the proper network process.*

Simply said, each web page instance and each network process have an associated URL which identify them to the kernel, in addition to their actual process id. This URL is the URL they are allowed to communicate with. Now, whenever a web page instance tries to communicate with a network process, the kernel checks the process id and associated URL for both. For this purpose, the kernel stores a mapping of process id to URLs. If no appropriate network process exists, a new one will be created by the kernel at this point. In practice, the kernel (and its representation in our model) enforces that only matching processes communicate. For checking property (1) we look at each message that is received by any network process and compare the URLs of sender and receiver using the kernel's mapping. Note that sender and receiver names cannot be forged as these are their process ids and enforced by the kernel based on the underlying guarantees of the operating system.

Indeed, the execution for property (1) does not make use of a history of what happened before, but only of the current assignment of each process to URL. We abstract away from a long sequence of network requests to simply one single network request. As the state is generic and the correctness of the property only depends on one network request, if we can show the absence of errors for this one network request, we know that any arbitrary number of them still will not exhibit any errors. Otherwise, we could take just that network request which triggers the error and use it to get the error by itself, contradicting the fact that we show that no single message creates an error.

Checking property (1) then boils down to checking executions (up to some depth of input), from canonical starting points, to see whether there is a mis-match between URLs in the resulting configuration for any message. If there is no mis-match for all starting points, then all communications have been legal and property (1) is actually proved. We can limit the depth of execution, i.e., the number of messages being considered, and still be complete. Each message is generic and representative of a set of messages. The reason we can limit the depth is that if the property would turn out to be possibly violated at an arbitrary number of messages, then that final message triggering the failure will only have one source process and one destination process. That violation can then be boiled down to the triggering network request, and the setup for those involved two processes, which would be a total depth of three actions.

The following search, presented in simplified fashion, returns no solution, meaning that no illegal (according to SOP) communication happened. Note that `L1, L1', L2, L2'` are the URLs and `N` is the process id of a web app and `N'` is the process id of a network process.

```
search init =>*
X:Configuration
< N : pipe | incoming(msg(from(N'), to(N), L1), ...) , ... >
< kernel-id : kernel | ... ,
        weblabels(pi(N',L1'), ...) ,
        networklabels(pi(N, L2', L2), ...) >
such that  L1 =/= L2 or L1' =/= L2' .
```

For this property, let us also show the actual search command with all the detail, to give the reader a better idea of what this looks like.

```
search init-simp-kernel inspect-space =>*
X:Configuration
< N:Nat : pipe |
   toKernel(ML:MessageList) ,
   fromKernel(msg(OPOS-SYSCALL-FD-SEND-MESSAGE,
         payload(Num:Nat, N:Nat, MSG-FETCH-URL,
                 V:MsgVal, L1:Label, T:typed, U:untyped)),
         ML':MessageList) , Att:AttributeSet >
< kernel-id : kernel | Att2:AttributeSet ,
     weblabels(pi(Num:Nat,L1':Label),
               WAPIS:WebappProcInfoSet) ,
     networklabels(pi(N:Nat, L2':Label, L2:Label),
                   NPIS:NetworkProcInfoSet)  ,
     displayedTopBar(URL:Label) >
such that  L1:Label =/= L2:Label or L1':Label =/= L2':Label .
```

As the above search did not return any solution, no illegal messages were passed. All network requests indeed end up going to the proper network processes.

Let us note here that `weblabels` and `networklabels` are the data structures which store the connection between URLs and web apps, and, respectively, network processes. Indeed, `pi` contains the relation of one web app or network process and the URL it is assigned to. In the case of the network process there are two URLs, the first, which will exactly match that of the associated web app, and the second, which is used for communication to the outside. That URL may be the same as the first, or more specific.

After the above motivation, let us remind you of Lemma 1, so that we only need to consider sequences of the form $\rightarrow_T\rightarrow_I^! \ldots \rightarrow_T\rightarrow_I^! \ldots \rightarrow_T\rightarrow_I^*$. Looking at the `search` command for property (1) we see that there is one web app with id `Num:Nat` and one network process with id `N:Nat` that impact a violation. The web app `Num:Nat` is sending a message to the network process `N:Nat`. For this, both the web app and the network process could have been created by a single trigger rule step `new-url(U)` or by two separate such trigger rule steps of form `new-url(U)`. Of course there is again the last trigger rule step that causes the violation, which can be of the form `new-url(U)` and can create none, either one, or both the web app and network process in question. Let us now give the formal theorem for the property (1):

Theorem 3 *Property* (1) *holds for any rewrite sequence, using any number of trigger rule steps.*

Proof. *The base case of at most* 3 *trigger rule steps is proven by the above model-checking analysis. The reduction given in Lemma 3 then completes the proof by reducing any violation of greater length to use at most 3 trigger rule steps, so no violation can exist.*

We now give the reduction lemma that assures as that all longer sequences can be reduced to shorter ones:

Lemma 3 *Any sequence of trigger rule steps that leads to a violation of property* (1) *and uses* 4 *or more trigger rule steps can be reduced by a step. This yields that all the possible trigger rule sequences leading to a violation must be of length* 3 *or less.*

Proof. *For rewrite sequences with at most* 3 *trigger rule steps, the model-checking via* `search` *has already proven that there are no violations. We now assume our claim is incorrect. Then there must exist some sequence leading to a violation. Let us pick any such sequence with the smallest possible number of trigger rule steps being used, and let that number be L. Now note that we are only considering $L \geq 4$. We will now consider all the possible cases, based on what the last trigger step can do.*

First, let us assume that the last trigger step, called l, is of type `new-url(U)` *and creates both the web app with id* `Num:Nat` *and the network process with id* `N:Nat`*. This means that there will be (at least) three further trigger steps before l, due to $L \geq 4$. Let us call these three steps k_1, k_2, k_3 with $k_1; k_2; k_3; l$ being the end of the sequence of trigger steps that leads to the violation.*

Any `switch-tab` *only changes the active web app, so if k_1 (or k_2) is a* `switch-tab`*, then the result of that trigger rule step will be overwritten by the following trigger rule step, so we can drop k_1 (or k_2) from the sequence and the resulting sequence still leads to the same violation, but is one trigger rule step shorter, which contradicts L being minimal.*

So we know that k_1 must be of form `new-url(U)` *(otherwise we are back in the prior case). Clearly, it can create neither of the two processes relevant for the violation, as they are added at l. So whatever k_1 added to the state is just data that is being carried along and that has no effect on the violation that will be found as it is not connected to the two processes in question. Thus, we can shorten the sequence by dropping k_1 and will still get the same violation at the end, but with a shorter sequence, contradicting the minimality of L.*

Let us now assume that the last trigger rule step l is of type `new-url(U)` *and creates the web app* `Num:Nat` *but not the network process* `N:Nat`*. Then there must be another trigger rule step* `new-url(U)` *that creates the network process* `N:Nat`*, which we call l_1.*

Now $l_1 < l$. Let us mark this part of the proof as $(*)$ as it will be reused. As there are at least 4 trigger rule steps there need to be at least 2 more.

Case 1: There is a trigger rule step before l_1, call the immediate predecessor k. Then $k < l_1 < l$. We now reason by cases:

- *Trigger rule k is a* `switch-tab`*: This changes the active web app but upon the execution of the trigger l_1 and normalization by internal rules all changes by this* `switch-tab` *are completely undone. So we can remove this trigger, and the shorter sequence still leads to the same violation, against the minimality of L.*

- *Trigger rule k is a* `new-url(U)`*: Then k can create neither of the two processes relevant for the violation, as they are added at l_1 and l. So whatever k adds to the state is just data that is being carried along and that has no effect on the violation that will be found as it is not connected to the two processes in question. Thus, we can shorten the sequence by dropping k and will still get the same violation at the end, but with a shorter sequence, against the minimality of L.*

Case 2: All other triggers are between l_1 and l: $l_1 < k_1 < k_2 < l$.

- *Any* `switch-tab` *only changes the active web app, so if k_1 is of type* `switch-tab`*, then the result of that trigger rule step will be overwritten by the following trigger rule step, so we can drop k_1 from the sequence and the resulting sequence still leads to the same violation, but is one trigger rule step shorter, which contradicts L being minimal.*

- *Trigger rule k_1 has to be a* **`new-url(U)`** *(otherwise we are in the prior case): Then k_1 can create neither of the two processes relevant for the violation, as they are added at l_1 and l. So whatever k adds to the state is just data that is being carried along and that has no effect on the violation that will be found as it is not connected to the two processes in question. Thus, we can shorten the sequence by dropping k and will still get the same violation at the end, but with a shorter sequence, contradicting L minimal.*

Let us assume the last trigger rule step l is of type **`new-url(U)`** *and creates the network process* **`N:Nat`**, *but not the web app* **`Num:Nat`**. *Then the same reasoning as for the prior analysis where l is creating the web app and not the network process holds, as none of the proof steps needed that distinction. Therefore we do not repeat all those arguments here.*

In the last set of cases we assume that the last trigger step l creates neither the web app **`Num:Nat`**, *nor the network process* **`N:Nat`**. *So, l can be either a* **`new-url(U)`** *or a* **`switch-tab`**. *There also needs to be either one trigger step l_0 that creates both processes (web app and network process), or separate trigger steps creating one each, l_1 for the web app* **`Num:Nat`** *and l_2 for the network process* **`N:Nat`**. *All of them are of the* **`new-url(U)`** *type.*

First let us look at the case with a single trigger step creating both processes. Then $l_0 < l$. The case analysis we get is just like the one at $(*)$ above, so we do not repeat all of those arguments here.

In the case of two trigger rule steps creating the two processes we have two possibilities, $l_1 < l_2 < l$ and $l_2 < l_1 < l$. We will look at $l_1 < l_2 < l$ and note that the other case works similarly, so we will omit it from this presentation. There has to be at least one additional trigger step. If one exists before l_1 we pick the direct predecessor. Otherwise we pick a trigger step between l_1 and l_2 (predecessor of l_2) if such a one exists. Otherwise we pick the immediate predecessor of l. We call this pick k.

Case 1: $k < l_1 < l_2 < l$

- *Trigger rule k is a* **`switch-tab`**: *This changes the active web app but upon the execution of the trigger l_1 and normalization by internal rules all changes by this* **`switch-tab`** *are completely undone. So we can remove this trigger, and the shorter sequence still leads to the same violation, contradicting L minimal.*

- *Trigger rule k is a* ***new-url(U)****: Then k can create neither of the two processes relevant for the violation, as they are added at l_1 and l_2. So whatever k adds to the state is just data that is being carried along and that has no effect on the violation that will be found as it is not connected to the two processes in question. Thus, we can shorten the sequence by dropping k and will still get the same violation at the end, but with a shorter sequence, contradicting L minimal.*

Case 2: $l_1 < k < l_2 < l$

- *Trigger rule k is a* ***switch-tab****: This changes the active web app but upon the execution of the trigger l_2 and normalization by internal rules all changes by this* ***switch-tab*** *are completely undone. So we can remove this trigger, and the shorter sequence still leads to the same violation, contradicting L minimal.*

- *Trigger rule k is a* ***new-url(U)****: Then k can create neither of the two processes relevant for the violation, as they are added at l_1 and l_2. So whatever k adds to the state is just data that is being carried along and that has no effect on the violation that will be found as it is not connected to the two processes in question. Thus, we can shorten the sequence by dropping k and will still get the same violation at the end, but with a shorter sequence, contradicting L minimal.*

Case 3: $l_1 < l_2 < k < l$

- *Trigger rule k is a* ***new-url(U)****. Pick k to be this if there is any such* ***new-url(U)*** *between l_2 and l. Then k can create neither of the two processes relevant for the violation, as they are added at l_1 and l_2. So whatever k adds to the state is just data that is being carried along and that has no effect on the violation that will be found as it is not connected to the two processes in question. Thus, we can shorten the sequence by dropping k and will still get the same violation at the end, but with a shorter sequence, a contradiction.*

- *Trigger rule k is a* ***switch-tab****. Now note that there are only the two web apps created by l_1 and l_2 as in case there is another* ***new-url(U)*** *trigger step we are in the case before. If there is more than one trigger of type* ***switch-tab*** *here, all but the last can be dropped as they are* *overwritten right away, and we would have a shorter sequence leading to the same violation, a contradiction.*

This means that there is only one ***switch-tab*** *between l_2 and l, at k. It either makes the web app of l_1 or the web app of l_2 the active one. If it makes l_2 active, then k can be dropped as it is superfluous and makes active the process which is already active, and we have the shorter sequence $l_1; l_2; l$ leading to the violation, a contradiction.*

If k makes l_1 active then we can just reorder the sequence to be $l_2; l_1; l$ while dropping k, as it would then again make the active web app active, which is not needed, so we get a contradiction.

So, in each case we have shown that there is a sequence one shorter, which still violates the property. This is in contradiction to our assumption that we have picked the sequence with the smallest number of triggers, L. □

With the reduction explained in the proof, we have extended the model-checking proof for sequences with at most three triggers, to sequences with *any* number of triggers.

The next SOP property is:

- (2) *The kernel must route Ethernet frames from the network interface card (NIC) to the proper network process.*

Similarly to (1), the kernel knows which URL a network process is allowed to communicate with. The following search is designed to check that only acceptably sourced data from the NIC gets transmitted to the network process.

```
search init-simp-kernel inspect-space =>*
X:Configuration
< N:Nat : mem | in(L1:Label, Ll:LabelList),
                Att:AttributeSet >
< kernel-id : kernel |
   networklabels(pi(N:Nat, L':Label, L2:Label),
                 NPIS:NetworkProcInfoSet) ,
     Att2:AttributeSet >
such that L1:Label =/= L2:Label .
```

Here, `L2:Label` is the URL that the network process `N` is allowed to communicate with, and `N:Nat : mem` represents the network process `N` memory, used for receipt of Ethernet frames. With no mismatch between the URL it is allowed to receive and the source of the data, we know that property (2) is maintained. The search started that way indeed returns no solution.

Based on this motivation let us give our formal theorem for the property (2):

Theorem 4 *Property* (2) *holds for any rewrite sequence, using any number of trigger rule steps.*

Proof. *The base case of at most* 3 *trigger rule steps is proven by the above model-checking analysis. The reduction given in Lemma 4 then completes the proof by reducing any violation of greater length to use at most 3 trigger rule steps, so no violation can exist.*

We now give the reduction lemma that assures as that all longer sequences can be reduced to shorter ones:

Lemma 4 *Any sequence of trigger rule steps that leads to a violation of property* (2) *and uses* 4 *or more trigger rule steps can be reduced by a step. This yields that all the possible trigger rule sequences leading to a violation must be of length* 3 *or less.*

Proof. *For rewrite sequences with at most* 3 *trigger rule steps, the model-checking via* `search` *has already proven that there are no violations. We now assume our claim is incorrect. Then there must be some sequence leading to a violation. Let us pick any such sequence with the smallest possible number of trigger rule steps being used, and let that number be* L*. Now note that we must have* $L \geq 4$*. We will now consider all the possible cases, based on what the last trigger step can do.*

For a violation to possibly occur there need to be two different URLs available, both of which get generated by different trigger rule steps of type `new-url(U)`*. Now the last trigger step, called* l*, can be either of the two* `new-url(L1:Label)` *and* `new-url(L2:Label)`*, or, alternatively, does not add either of the two URLs. In that case it can be of type* `new-url(U)` *with* `U` *neither of those URLs, or it can be of type* `switch-tab`*.*

First, let us deal with the case where l *is* `new-url(L1:Label)`*. The case where* l *is* `new-url(L2:Label)` *works just the same, so we do not spell that case out. In this case there needs to be one other trigger rule step* *`new-url(L2:Label)`, whose last appearance we call l_1. Of course we know $l_1 < l$. As there are at least 4 trigger rule steps, there need to be at least 2 more.*

Case 1: $k < l_1 < l$, meaning there is a trigger rule step before l_1, call the immediate predecessor k.

- *Trigger rule k is a `switch-tab`. As there is no violation during execution of k and the following normalization, all traces of k will be eliminated during the following execution (and normalization) of l_1. So we can remove this trigger and the shorter sequence still leads to the same violation, a contradiction.*

- *Trigger rule k is a `new-url(U)`: Using this trigger rule step does increase the state space size by adding all the pieces mentioned before in the description of what such a trigger does. Now, if `U` $\neq$ `L1:Label` and `U` $\neq$ `L2:Label`, then all additions are part of the state, but no violation occurs here. Whatever k added is just data being carried along in the state that has no effect on the violation that will be found as it is for a different URL. Thus, we can shorten the sequence by dropping this trigger k and will get the same violation at the end, but with a shorter sequence, a contradiction.*

  In case that `U` $=$ `L1:Label`, then this is duplicating the later step l, in which case we only need the last one of the two appearances, and can drop the trigger k and have a sequence one shorter with the same resulting violation, a contradiction.

  In case that `U` $=$ `L2:Label`, then this is duplicating the immediately following step l_1, in which case we also only need the last of the two appearances, and can thus drop the trigger k to get a sequence that is one shorter, which has the same resulting violation, a contradiction.

Case 2: $l_1 < k < l$. There are no triggers before l_1, but there can be multiple additional trigger rule steps between l_1 and l. If there is any `new-url(U)` type trigger, we pick the last of them as k. If there are only `switch-tab` triggers, we pick the immediate predecessor of l for k.

- *Trigger rule step k is `new-url(U)`: Using this trigger rule step does increase the state space size by adding all the pieces mentioned before* *in the description of what such a trigger does. Now, if* ***U*** $\neq$ ***L1:Label*** *and* ***U*** $\neq$ ***L2:Label****, then all the additions are part of the state, but no violation occurs here. Whatever k added is just data being carried along in the state that has no effect on the violation that will be found as it is for a different URL. Thus, we can shorten the sequence by dropping this trigger k and will get the same violation at the end, but with a shorter sequence, a contradiction.*

In case ***U*** $=$ ***L1:Label****, then this is duplicating the next step l, in which case we only need the last of the two, and can again drop trigger k to get a sequence that is one shorter but leads to the same violation, a contradiction.*

In case ***U*** $=$ ***L2:Label****, then this would be duplicating the prior step l_1. But, we had picked l_1 to be the last such trigger rule step, so we have a contradiction and this case is moot.*

- *Trigger rule k is a* ***switch-tab****: This changes the active web app immediately before the step l which creates the violation. Let us first emphasize that there are no* ***new-url(U)*** *triggers in the sequence except for l_1. All the triggers between l_1 and l are of the* ***switch-tab*** *form, otherwise we are in the prior case.*

  If there are multiple such triggers, then in addition to k directly before l, there is another one, call it m. Now, trigger m and all its changes will be completely undone by trigger k. So, trigger m is not needed, and removing trigger m will still yield the exact same violation, a contradiction.

  If there is only one such trigger, then we have $l_1; k; l$ as the sequence giving the violation. k can change the active web app but there is only one (created by l_1) and it is already active, so we can drop k and have a shorter sequence leading to the violation, a contradiction.

Now we look at the case where l does not add either of the two URLs in question. In this case there need to be two other trigger rule steps in the sequence. We call l_1 the last appearance of ***new-url(L1:Label)****, and we call l_2 the last appearance of* ***new-url(L2:Label)****. Now there are two cases that work just the same, depending on $l_1 < l_2$ or $l_2 < l_1$. We will give in detail the case $l_1 < l_2$. We will look at additional trigger rule steps k, first if there* *are any before l_1, then if there are none before l_1 but some between l_1 and l_2, and otherwise with them between l_2 and l. As usual we pick the immediate predecessors of other steps for k.*

Case 1: $k < l_1 < l_2 < l$. Given multiple additional trigger rule steps before l_1, we pick k to be the immediate predecessor of l_1.

- *Trigger rule step k is a `switch-tab`: This changes the active web app immediately before introducing the URL in l_1 which appears in the violation. Upon the execution of the trigger in l_1 and normalization by the internal rules, all changes made by this `switch-tab` are completely undone, and as there was no violation here (or we would have a shorter sequence already) we can drop this `switch-tab` trigger from our list of triggers to be executed, and will reach the same violation at the end of the sequence which is now one step shorter, a contradiction.*

- *Trigger rule step k is a `new-url(U)`: Using this trigger rule step does increase the state space size by adding all the pieces mentioned before in the description of what such a trigger does. Now, if `U` $\neq$ `L1:Label` and `U` $\neq$ `L2:Label`, then all the additions are part of the state, but no violation occurs here. Whatever k added is just data being carried along in the state that has no effect on the violation that will be found later as it is for a different URL. Thus, we can shorten the sequence by dropping this trigger k and will get the same violation at the end, but with a shorter sequence, a contradiction.*

  In case `U` $=$ `L1:Label`, then the following step l_1 will duplicate that trigger step, in which case we only need the last one of the two, and can again drop the trigger k and have a sequence that is one shorter with the same resulting violation, a contradiction.

  In case `U` $=$ `L2:Label`, then the step l_2 which happens later will duplicate it, and again we only need the last one, so we can drop this trigger and the resulting sequence will give rise to the same violation but be one shorter, a contradiction.

Case 2: $l_1 < k < l_2 < l$. We pick the immediate predecessor of l_2 for k, in case there are multiple additional trigger rule steps between l_1 and l_2. This works similar to case 1.

- *Trigger rule step k is a* ***switch-tab****: This changes the active web app immediately before introducing the URL in l_2 which appears in the violation. Upon the execution of the trigger in l_2 and normalization by the internal rules, all changes made by this* ***switch-tab*** *are completely undone, and as there was no violation here (or we would have a shorter sequence already) we can drop this* ***switch-tab*** *trigger from our list of triggers to be executed, and will reach the same violation at the end of the sequence which is now one step shorter, a contradiction.*

- *Trigger rule step k is a* ***new-url(U)****: Using this trigger rule step does increase the state space size by adding all the pieces mentioned before in the description of what such a trigger does. Now, if* ***U*** $\neq$ ***L1:Label*** *and* ***U*** $\neq$ ***L2:Label****, then all the additions are part of the state, but no violation occurs here. Whatever k added is just data being carried in the state that has no effect on the violation that will be found later as it is for a different URL. Thus, we can shorten the sequence by dropping this trigger k and will get the same violation at the end, but with a shorter sequence, a contradiction.*

  In case ***U*** $=$ ***L1:Label****, then this is duplicating the earlier step l_1. But, we picked l_1 to be the last such trigger rule step, so this is a contradiction and this case is moot.*

  In case ***U*** $=$ ***L2:Label****, then the following step l_2 will duplicate that trigger step, in which case we only need the last one of the two, and can again drop the trigger k and have a sequence that is one shorter with the same resulting violation, a contradiction.*

Case 3: $l_1 < l_2 < k < l$. There can be multiple additional trigger rule steps between l_2 and l. If there is any ***new-url(U)*** *type trigger, we pick the last one of them as k. If there are only* ***switch-tab*** *triggers, we pick the immediate predecessor of l for k.*

- *Trigger rule step k is a* ***new-url(U)****: Using this trigger rule step does increase the state space size by adding all the pieces mentioned before in the description of what such a trigger does. Now, if* ***U*** $\neq$ ***L1:Label*** *and* ***U*** $\neq$ ***L2:Label****, then all the additions are part of the state, but no violation occurs here. Whatever k added is just data being carried in the state that has no effect on the violation that will be found as it is* *for a different URL. Thus, we can shorten the sequence by dropping this trigger step k and will get the same violation at the end, but with a shorter sequence, a contradiction.*

In case `U` $=$ `L1:Label`*, then this is duplicating the prior step l_1. We stated earlier that we pick l_1 to be the last appearance of trigger rule step* `new-url(L1:Label)`*, so this case is moot.*

In case `U` $=$ `L2:Label`*, then this is duplicating the prior step l_2. We stated earlier that we pick l_2 to be the last appearance of trigger rule step* `new-url(L2:Label)`*, so this case is also moot.*

- *Trigger rule step k is a* `switch-tab`*: This changes the active web app immediately before the step l which creates the violation. Let us first emphasize that there are no* `new-url(U)` *triggers in the sequence except for l_1 and l_2. All the triggers between l_2 and l are of the* `switch-tab` *form.*

  If there are multiple such triggers, then in addition to k directly before l there is another one, call it m. Now, trigger m and all its changes will be completely undone by trigger k. So, trigger m is not needed, and removing trigger m will still yield the exact same violation, a contradiction.

  If there is only one such trigger, then we have $l_1; l_2; k; n$ as the sequence giving a violation. But, then k can only switch the active tab to either the web app associated to l_1 or l_2. In the case of k making the web app of l_1 the active one, we get the same result by moving l_1 to the spot of k in the order and dropping k, that is, $l_2; l_1; n$ will yield the violation and be shorter, giving us a contradiction. On the other hand, if k makes the web app of l_2 active, that is not needed, as it already is the active one at that point. So $l_1; l_2; n$ will similarly yield the violation and it is shorter, another contradiction.

So, in each case we have shown that there is a sequence one shorter, which still violates the property. This is in contradiction to our assumption that we have picked the sequence with the smallest number of triggers, L. □

With the reduction explained in the proof, we have extended the model-checking proof for sequences with at most three triggers, to sequences with *any* number of triggers.

Another SOP property is:

- (3) *Ethernet frames from network processes to the NIC must have an IP address and TCP port that matches the origin of the network process.*

Outgoing Ethernet frames are created by the network process, but are then checked by the kernel for a match between the included URL and the URL associated to the network process. In the search command below, the check is on the outgoing memory of the network process, before it is being sent out. It indeed looks at the URL the Ethernet frame will be sent to, `L1:Label` noted in `out`, and checks that against the network process associated URL, `L2:Label`.

```
search init-simp-kernel inspect-space =>*
X:Configuration
< N:Nat : mem | out(L1:Label) , Att:AttributeSet >
< kernel-id : kernel |
  networklabels(pi(N:Nat, L':Label, L2:Label),
                NPIS:NetworkProcInfoSet) ,
     Att2:AttributeSet >
such that L1:Label =/= L2:Label .
```

The search does not find a state, which means that there is no mis-match. Again, there is no history to consider here, so a single such outgoing message is either correct, or not. Based on this motivation let us give our formal theorem for the property (3):

Theorem 5 *Property* (3) *holds for any rewrite sequence, using any number of trigger rule steps.*

Proof. *The base case of at most* 3 *trigger rule steps is proven by the above model-checking analysis. The reduction given in Lemma 5 then completes the proof by reducing any violation of greater length to use at most 3 trigger rule steps, so no violation can exist.*

We now give the reduction lemma that assures as that all longer sequences can be reduced to shorter ones:

Lemma 5 *Any sequence of trigger rule steps that leads to a violation of property* (3) *and uses* 4 *or more trigger rule steps can be reduced by a step. This yields that all the possible trigger rule sequences leading to a violation must be of length* 3 *or less.*

Proof. *The proof is identical to the one given for property* (2)*, i.e., for Lemma 4. The difference between property* (2) *and property* (3) *is whether the Ethernet frame is incoming or outgoing, but that difference was not needed in the proof of property* (2)*.* □

The last two theorems have proved that all Ethernet frame handling is according to what SOP allows.

Consider the SOP property:

- (4) *HTTP data from network processes to web page instances must adhere to the SOP.*

By this we mean that the HTTP data that is transmitted has to be from allowable sources for both the sending network process and the receiving web app. In this case we check that the return message from a network process to a web page instance only contains data from an appropriate source, i.e., the labeling for the web app, network process and the data match.

```
search init-simp-kernel inspect-space
=>*
X:Configuration
< N:Nat : proc | rendered(L11:Label) , URL(L'':Label) ,
                 loading(1) , Att:AttributeSet >
< N:Nat : pipe | fromKernel(
       msg(OPOS-SYSCALL-FD-SEND-MESSAGE,
           payload(N':Nat, N:Nat, MSG-RETURN-URL, V:MsgVal,
                   L2:Label, T:typed, U:untyped)),
         ML:MessageList) , Att2:AttributeSet >
< kernel-id : kernel |
  weblabels(pi(N:Nat, L':Label), WAPIS:WebappProcInfoSet) ,
  networklabels(pi(N':Nat, L':Label, L1:Label),
                NPIS:NetworkProcInfoSet) ,
      Att3:AttributeSet >
such that L1:Label =/= L2:Label .
```

This model-checking `search` does not find any result, so we know that property (4) holds. No history is required for this and the important URLs here are `L1:Label` and `L2:Label`. If they disagree that would have been a violation. Based on this motivation let us give our formal theorem for the property (4):

Theorem 6 *Property* (4) *holds for any rewrite sequence, using any number of trigger rule steps.*

Proof. *The base case of at most* 3 *trigger rule steps is proven by the above model-checking analysis. The reduction given in Lemma 6 then completes the proof by reducing any violation of greater length to use at most 3 trigger rule steps, so no violation can exist.*

We now give the reduction lemma that assures as that all longer sequences can be reduced to shorter ones:

Lemma 6 *Any sequence of trigger rule steps that leads to a violation of property* (4) *and uses* 4 *or more trigger rule steps can be reduced by a step. This yields that all the possible trigger rule sequences leading to a violation must be of length* 3 *or less.*

Proof. *For rewrite sequences with at most 3 trigger rule steps, the model-checking via* `search` *has already proven that there are no violations. We now assume our claim is incorrect, then there must be some sequences leading to a violation. Let us pick any such sequence with the smallest possible number of trigger rule steps being used, and let that number be* L. *Now note that we must have* $L \geq 4$. *We will now consider all the possible cases, based on what the last trigger step can do.*

For a violation to possibly occur there need to be two different URLs available, both of which get generated by different trigger rule steps of type `new-url(U)`. *Now the last trigger step, called* l, *can be either of the two* `new-url(L1:Label)` *and* `new-url(L2:Label)`, *or alternatively does not add either of the two URLs. In that case it can be of type* `new-url(U)` *with* `U` *neither of those URLs, or it can be of type* `switch-tab`. *This proof is very similar to that of Theorem 4. The fact that this time we are checking the return data from a network process to a web app instead of the actual Ethernet frame correctness mostly disappears at this point, as these are just two different steps in the same chain of data transmission anyway.*

First, we look at the case where l *does not add either of the two URLs in question. In this case there need to be two other trigger rule steps in the sequence. We call* l_1 *the last appearance of* `new-url(L1:Label)`, *and we call* l_2 *the last appearance of* `new-url(L2:Label)`. *Now there are two cases that work just the same, depending on* $l_1 < l_2$ *or* $l_2 < l_1$. *We treat in detail the* *case $l_1 < l_2$. We will look at additional trigger rule steps k, first if there are any before l_1, then if there are none before l_1 but some between l_1 and l_2, and otherwise with them between l_2 and l. As usual we pick the immediate predecessors of other steps for k.*

Case 1: $k < l_1 < l_2 < l$. Given multiple additional trigger rule steps before l_1, we pick k to be the immediate predecessor of l_1.

- *Trigger rule step k is a `switch-tab`: This changes the active web app immediately before introducing the URL in l_1 which appears in the violation. Upon the execution of the trigger in l_1 and normalization by the internal rules, all changes made by this `switch-tab` are completely undone, and as there was no violation here (or we would have a shorter sequence already) we can drop this `switch-tab` trigger from our list of triggers to be executed, and will reach the same violation at the end of the sequence which is now one step shorter, a contradiction.*

- *Trigger rule step k is a `new-url(U)`: Using this trigger rule step does increase the state space size by adding all the pieces mentioned before in the description of what such a trigger does. Now, if `U` $\neq$ `L1:Label` and `U` $\neq$ `L2:Label`, then all the additions are part of the state, but no violation occurs here. Whatever k added is just data being carried along in the state that has no effect on the violation that will be found later as it is for a different URL. Thus, we can shorten the sequence by dropping this trigger k and we will get the same violation at the end, but with a shorter sequence, a contradiction.*

  In case `U` $=$ `L1:Label`, then the following step l_1 will duplicate that trigger step, so that we only need the last one of the two, and can again drop the trigger k and have a sequence that is one shorter with the same resulting violation, a contradiction.

  In case `U` $=$ `L2:Label`, then the step l_2 which happens later will duplicate it, and again we only need the last one, so we can drop this trigger and the resulting sequence will give rise to the same violation but be one shorter, a contradiction.

Case 2: $l_1 < k < l_2 < l$. We pick the immediate predecessor of l_2 for k, in case there are multiple additional trigger rule steps between l_1 and l_2. This works similar to case 1.

- *Trigger rule step k is a* ***`switch-tab`****: This changes the active web app immediately before introducing the URL in l_2 which appears in the violation. Upon the execution of the trigger in l_2 and normalization by the internal rules, all changes made by this* ***`switch-tab`*** *are completely undone, and as there was no violation here (or we would have a shorter sequence already) we can drop this* ***`switch-tab`*** *trigger from our list of triggers to be executed, and will reach the same violation at the end of the sequence which is now one step shorter, a contradiction.*

- *Trigger rule step k is a* ***`new-url(U)`****: Using this trigger rule step does increase the state space size by adding all the pieces mentioned before in the description of what such a trigger does. Now, if* ***`U`*** $\neq$ ***`L1:Label`*** *and* ***`U`*** $\neq$ ***`L2:Label`****, then all the additions are part of the state, but no violation occurs here. Whatever k added is just data being carried along in the state that has no effect on the violation that will be found later as it is for a different URL. Thus, we can shorten the sequence by dropping this trigger k and will get the same violation at the end, but with a shorter sequence, a contradiction.*

  In case ***`U`*** $=$ ***`L1:Label`****, then this is duplicating the earlier step l_1. But, we picked l_1 to be the last such trigger rule step, so this is a contradiction and this case is moot.*

  In case ***`U`*** $=$ ***`L2:Label`****, then the following step l_2 will duplicate that trigger step, in which case we only need the last one of the two, and can again drop the trigger k and have a sequence that is one shorter with the same resulting violation, a contradiction.*

Case 3: $l_1 < l_2 < k < l$. There can be multiple additional trigger rule steps between l_2 and l. If there is any ***`new-url(U)`*** *type trigger, we pick the last one of them as k. If there are only* ***`switch-tab`*** *triggers, we pick the immediate predecessor of l for k.*

- *Trigger rule step k is a* ***`new-url(U)`****: Using this trigger rule step does increase the state space size by adding all the pieces mentioned before in the description of what such a trigger does. Now, if* ***`U`*** $\neq$ ***`L1:Label`*** *and* ***`U`*** $\neq$ ***`L2:Label`****, then all the additions are part of the state, but no violation occurs here. Whatever k added is just data being carried along in the state that has no effect on the violation that will be found as it* *is for a different URL. Thus, we can shorten the sequence by dropping this trigger step k and will get the same violation at the end, but with a shorter sequence, a contradiction.*

In case `U` = `L1:Label`, then this is duplicating the prior step l_1. We stated earlier that we pick l_1 to be the last appearance of the trigger rule step `new-url(L1:Label)`, so this case is moot.

In case `U` = `L2:Label`, then this is duplicating the prior step l_2. We stated earlier that we pick l_2 to be the last appearance of the trigger rule step `new-url(L2:Label)`, so this case is also moot.

- *Trigger rule step k is a `switch-tab`: This changes the active web app immediately before the step l which creates the violation. Let us first emphasize that there are no `new-url(U)` triggers in the sequence except for l_1 and l_2. All the triggers between l_2 and l are of the `switch-tab` form.*

  If there are multiple such triggers, then in addition to k directly before l there is another one, call it m. Now, trigger m and all its changes will be completely undone by trigger k. So, trigger m is not needed, and removing trigger m will still yield the exact same violation, a contradiction.

  If there is only one such trigger, then we have $l_1; l_2; k; n$ as the sequence giving a violation. But, then k can only switch the active tab to either the web app associated to l_1 or l_2. In the case of k making the web app of l_1 the active one, we get the same result by moving l_1 to the spot of k in the order and dropping k, that is, $l_2; l_1; n$ will yield the violation and be shorter. On the other hand, if k makes the web app of l_2 active, that is not needed, as it already is the active one at that point. So $l_1; l_2; n$ will similarly yield the violation and it is shorter, a contradiction.

Now, let us deal with the case where l is `new-url(L1:Label)`. The case where l is `new-url(L2:Label)` works just the same, so we do not spell that case out. In this case there needs to be one other trigger rule step `new-url(L2:Label)`, whose last appearance we call l_1. Of course we know $l_1 < l$. As there are at least 4 trigger rule steps, there need to be at least 2 more.

Case 1: $k < l_1 < l$, meaning that there is a trigger rule step before l_1, call the immediate predecessor k.

- *Trigger rule k is a `switch-tab`. As there is no violation during execution of k and the following normalization, all traces of k will be eliminated during the following execution (and normalization) of l_1. So we can remove this trigger and the shorter sequence still leads to the same violation, a contradiction.*
- *Trigger rule k is a `new-url(U)`: Using this trigger rule step does increase the state space size by adding all the pieces mentioned before in the description of what such a trigger does. Now, if `U` $\neq$ `L1:Label` and `U` $\neq$ `L2:Label`, then all additions are part of the state, but no violation occurs here. Whatever k added is just data being carried along in the state that has no effect on the violation that will be found as it is for a different URL. Thus, we can shorten the sequence by dropping this trigger k and will get the same violation at the end, but with a shorter sequence, a contradiction.*

  In case `U` $=$ `L1:Label`, then this is duplicating the later step l, in which case we only need the last one of the two appearances, and can drop the trigger k and have a sequence one shorter with the same resulting violation, a contradiction.

  In case `U` $=$ `L2:Label`, then this is duplicating the immediately following step l_1, in which case we also only need the last of the two appearances, and can thus drop the trigger k to get a sequence that is one shorter, which has the same resulting violation, a contradiction.

Case 2: $l_1 < k < l$. There are no triggers before l_1, but there can be multiple additional trigger rule steps between l_1 and l. If there is any `new-url(U)` type trigger, we pick the last of them as k. If there are only `switch-tab` triggers, we pick the immediate predecessor of l for k.

- *Trigger rule step k is `new-url(U)`: Using this trigger rule step does increase the state space size by adding all the pieces mentioned before in the description of what such a trigger does. Now, if `U` $\neq$ `L1:Label` and `U` $\neq$ `L2:Label`, then all the additions are part of the state, but no violation occurs here. Whatever k added is just data being carried along* *in the state that has no effect on the violation that will be found as it is for a different URL. Thus, we can shorten the sequence by dropping this trigger k and will get the same violation at the end, but with a shorter sequence, a contradiction.*

In case `U= L1:Label`*, then this is duplicating the next step l, in which case we only need the last of the two, and can again drop trigger k to get a sequence that is one shorter but leads to the same violation, a contradiction.*

In case `U= L2:Label`*, then this would be duplicating the prior step l_1. But, we had picked l_1 to be the last such trigger rule step, so we have a contradiction and this case is moot.*

- *Trigger rule k is a* `switch-tab`*: This changes the active web app immediately before the step l which creates the violation. Let us first emphasize that there are no* `new-url(U)` *triggers in the sequence except for l_1. All the triggers between l_1 and l are of the* `switch-tab` *form, otherwise we are in the prior case.*

  If there are multiple such triggers, then in addition to k directly before l, there is another one, call it m. Now, trigger m and all its changes will be completely undone by trigger k. So, trigger m is not needed, and removing trigger m will still yield the exact same violation, a contradiction.

  If there is only one such trigger, then we have $l_1; k; l$ as the sequence giving the violation. k can change the active web app but there is only one (created by l_1) and it is already active, so we can drop k and have a shorter sequence leading to the violation, another contradiction.

So, in each case we have shown that there is a sequence one shorter, which still violates the property. This is in contradiction to our assumption that we have picked the sequence with the smallest number of triggers, L. □

Regarding the SOP property:

- (5) *Network processes for different web page instances must remain isolated.*

By virtue of the aforementioned labeling we have for all web apps and network processes, each network process can only communicate with the right web page instances. Simply by construction of our model (all messages going through the kernel) there is no way for network processes to communicate with each other directly. Therefore the isolation of network processes holds.

Consider now the SOP property:

- (6) *The browser chrome (UI elements) and web page content displays are isolated.*

This property easily holds in the model, due to its construction. The web page content is represented in `displayedContent(...)` of the process representing the display `< display-id :  ...  >`, while the UI elements are part of the kernel, in particular `displayedTopBar` for the address bar, see Section 4.3.2.

Last SOP property:

- (7) *Only the current tab can access the screen, mouse, and keyboard.*

For this property, let us note that all input is given to the kernel, which in turn passes it to the active web app. We have not explicitly modeled a mouse or keyboard. With regards to the screen, we ask you to look below at property (9) and realize that property (7) is a corollary of property (9) and the address bar correctness.

General good property:

- (8) All components can only perform their designated functions.

This property holds in the model, as that is the way it was built. The design of each component as captured in the model is exactly that set of designated functions, only.

Last property:

- (9) The URL of the current tab is displayed to the user.

To relate this to (7), let us note that the current tab is representative of the active web app, which has control of the screen, and that we take this property to actually mean the stronger property that the screen contents are also of the currently active web app whenever the screen is not `about-blank`.

```
search init-simp-kernel inspect-space =>*
  X:Configuration
  < kernel-id : kernel | Att:AttributeSet ,
          displayedTopBar(URL:Label) >
  < display-id : proc |
    activeWebapp(W:ProcId),
    Att2:AttributeSet >
  < W:ProcId : proc |
    URL(URL':Label),
    Att3:AttributeSet >
such that URL:Label =/= URL':Label  .
```

Indeed, the URL associated to the active web app is being presented to the user in the address bar (`displayedTopBar`).

Based on this motivation let us give our formal theorem for the property (9):

Theorem 7 *Property* (9) *holds for any rewrite sequence, using any number of trigger rule steps.*

Proof. *The base case of at most* 3 *trigger rule steps is proven by the above model-checking analysis. The reduction given in Lemma 7 then completes the proof by reducing any violation of greater length to use at most 3 trigger rule steps, so no violation can exist.*

We now give the reduction lemma that assures as that all longer sequences can be reduced to shorter ones:

Lemma 7 *Any sequence of trigger rule steps that leads to a violation of property* (9) *and uses* 4 *or more trigger rule steps can be reduced by a step. This yields that all the possible trigger rule sequences leading to a violation must be of length* 3 *or less.*

Proof. *For rewrite sequences with at most 3 trigger rule steps, the model-checking via* ***search*** *has already proven that there are no violations. We now assume our claim is incorrect. Then there must be some sequences leading to a violation. Let us pick any such sequence with the smallest possible number of trigger rule steps being used, and let that number be L. Now note that we must have $L \geq 4$. We will now consider all the possible cases, based on what the last trigger step can do.*

To make a violation possible there need to be two different URLs, created by two different ***new-url(U)*** *trigger rule uses. One appears in the address bar, while the other is the source URL for the web app that is currently active. The last trigger rule l can thus either be a* ***switch-tab****, a* *`new-url(URL:Label)`, a `new-url(URL':Label)` (the active web app's URL) or a different trigger rule of type `new-url(U)`. There will then be one or two further `new-url(U)` type trigger rule steps.*

First, we consider l to be `switch-tab`, which means there are two other trigger rule steps. We call l_1 the last step `new-url(URL:Label)` and we call l_2 the last step `new-url(URL':Label)`. Now either $l_1 < l_2$ or $l_2 < l_1$. We only spell out the case $l_1 < l_2$, as the other works just the same. So, we have $l_1 < l_2 < l$.

Case 1: $k < l_1 < l_2 < l$. Given multiple additional trigger rule steps before l_1, we pick k to be the immediate predecessor of l_1.

- *Trigger rule step k is a `switch-tab`: This changes the active web app immediately before introducing the URL in l_1 which appears in the violation. Upon the execution of the trigger in l_1 and normalization by the internal rules, all changes made by this `switch-tab` are completely undone, and as there was no violation here (or we would have a shorter sequence already) we can drop this `switch-tab` trigger from our list of triggers to be executed, and will reach the same violation at the end of the sequence which is now one step shorter, a contradiction.*

- *Trigger rule step k is a `new-url(U)`: Using this trigger rule step does increase the state space size by adding all the pieces mentioned before in the description of what such a trigger does. Now, if `U` $\neq$ `URL:Label` and `U` $\neq$ `URL':Label`, then all the additions are part of the state, but no violation occurs here. Whatever k added is just data being carried along in the state that has no effect on the violation that will be found later as it is for a different URL. Thus, we can shorten the sequence by dropping this trigger k and we will get the same violation at the end, but with a shorter sequence, a contradiction.*

  In case `U` $=$ `URL:Label`, then the following step l_1 will duplicate that trigger step, in which case we only need the last one of the two, and can again drop the trigger k and have a sequence that is one shorter with the same resulting violation, a contradiction.

  In case `U` $=$ `URL':Label`, then the step l_2 which happens later will duplicate it, and again we only need the last one, so we can drop this trigger and the resulting sequence will give rise to the same violation *but be one shorter, a contradiction.*

Case 2: $l_1 < k < l_2 < l$. We pick the immediate predecessor of l_2 for k, in case there are multiple additional trigger rule steps between l_1 and l_2. This works similar to case 1.

- *Trigger rule step k is a `switch-tab`: This changes the active web app immediately before introducing the URL in l_2 which appears in the violation. Upon the execution of the trigger in l_2 and normalization by the internal rules, all changes made by this `switch-tab` are completely undone, and as there was no violation here (or we would have a shorter sequence already) we can drop this `switch-tab` trigger from our list of triggers to be executed, and will reach the same violation at the end of the sequence which is now one step shorter, a contradiction.*

- *Trigger rule step k is a `new-url(U)`: Using this trigger rule step does increase the state space size by adding all the pieces mentioned before in the description of what such a trigger does. Now, if `U` $\neq$ `URL:Label` and `U` $\neq$ `URL':Label`, then all the additions are part of the state, but no violation occurs here. Whatever k added is just data being carried along in the state that has no effect on the violation that will be found later as it is for a different URL. Thus, we can shorten the sequence by dropping this trigger k and will get the same violation at the end, but with a shorter sequence, a contradiction.*

  In case `U` $=$ `URL:Label`, then this is duplicating the earlier step l_1. But, we picked l_1 to be the last such trigger rule step, so this is a contradiction and this case is moot.

  In case `U` $=$ `URL':Label`, then the following step l_2 will duplicate that trigger step, in which case we only need the last one of the two, and can again drop the trigger k and have a sequence that is one shorter with the same resulting violation, a contradiction.

Case 3: $l_1 < l_2 < k < l$. There can be multiple additional trigger rule steps between l_2 and l. If there is any `new-url(U)` type trigger, we pick the last one of them as k. If there are only `switch-tab` triggers, we pick the immediate predecessor of l for k.

- *Trigger rule step k is a* ***`new-url(U)`****: Using this trigger rule step does increase the state space size by adding all the pieces mentioned before in the description of what such a trigger does. Now, if* **`U`** $\neq$ **`L1:Label`** *and* **`U`** $\neq$ **`L2:Label`***, then all the additions are part of the state, but no violation occurs here. Whatever k added is just data being carried along in the state that has no effect on the violation that will be found as it is for a different URL. Thus, we can shorten the sequence by dropping this trigger step k and will get the same violation at the end, but with a shorter sequence, a contradiction.*

  In case **`U`** $=$ **`L1:Label`***, then this is duplicating the prior step l_1. We stated earlier that we pick l_1 to be the last appearance of the trigger rule step* **`new-url(L1:Label)`***, so this case is moot.*

  In case **`U`** $=$ **`L2:Label`***, then this is duplicating the prior step l_2. We stated earlier that we pick l_2 to be the last appearance of the trigger rule step* **`new-url(L2:Label)`***, so this case is also moot.*

- *Trigger rule step k is a* **`switch-tab`***: This changes the active web app immediately before the step l which creates the violation. Let us first emphasize that there are no* **`new-url(U)`** *triggers in the sequence except for l_1 and l_2. All the triggers between l_2 and l are of the* **`switch-tab`** *form.*

  If there are multiple such triggers, then in addition to k directly before l there is another one, call it m. Now, trigger m and all its changes will be completely undone by trigger k. So, trigger m is not needed, and removing trigger m will still yield the exact same violation, a contradiction.

  If there is only one such trigger, then we have $l_1; l_2; k; l$ as the sequence giving a violation. But, then k can only switch the active tab to either the web app associated to l_1 or l_2. In the case of k making the web app of l_1 the active one, we get the same result by moving l_1 to the spot of k in the order and dropping k, that is, $l_2; l_1; n$ will yield the violation and be shorter, yielding a contradiction. On the other hand, if k makes the web app of l_2 active, that is not needed, as it already is the active one at that point. So $l_1; l_2; n$ will similarly yield the violation and it is shorter, another contradiction.

Now we consider l to be `new-url(URL:Label)`, then there will be another trigger rule step l_1 of the form `new-url(URL':Label)`. Other trigger rule steps can happen before or after l_1. If there are any before l_1 we pick from there for k, only otherwise do we look after l_1.

Case 1: $k < l_1 < l$. We pick the immediate predecessor of l_1 for k in case there are multiple trigger rules before l_1.

- *Trigger rule k is a `switch-tab`. As there is no violation during execution of k and the following normalization, all traces of k will be eliminated during the following execution (and normalization) of l_1. So we can remove this trigger and the shorter sequence still leads to the same violation, a contradiction.*

- *Trigger rule k is a `new-url(U)`: Using this trigger rule step does increase the state space size by adding all the pieces mentioned before in the description of what such a trigger does. Now, if `U` $\neq$ `URL:Label` and `U` $\neq$ `URL':Label`, then all additions are part of the state, but no violation occurs here. Whatever k added is just data being carried along in the state that has no effect on the violation that will be found as it is for a different URL. Thus, we can shorten the sequence by dropping this trigger k and will get the same violation at the end, but with a shorter sequence, a contradiction.*

  In case `U`$=$ `URL:Label`, then this is duplicating the later step l, in which case we only need the last one of the two appearances, and can drop the trigger k and have a sequence one shorter with the same resulting violation, a contradiction.

  In case `U` $=$ `URL':Label`, then this is duplicating the immediately following step l_1, in which case we also only need the last of the two appearances, and can thus drop the trigger k to get a sequence that is one shorter, which has the same resulting violation, a contradiction.

Case 2: $l_1 < k < l$. There are no triggers before l_1, but there can be multiple additional trigger rule steps between l_1 and l. If there is any `new-url(U)` type trigger, we pick the last of them as k. If there are only `switch-tab` triggers, we pick the immediate predecessor of l for k.

- *Trigger rule step k is `new-url(U)`: Using this trigger rule step does increase the state space size by adding all the pieces mentioned before* *in the description of what such a trigger does. Now, if* `U` $\neq$ `URL:Label` *and* `U` $\neq$ `URL':Label`*, then all the additions are part of the state, but no violation occurs here. Whatever k added is just data being carried along in the state that has no effect on the violation that will be found as it is for a different URL. Thus, we can shorten the sequence by dropping this trigger k and will get the same violation at the end, but with a shorter sequence, a contradiction.*

In case `U` $=$ `URL:Label`*, then this is duplicating the next step l, in which case we only need the last of the two, and can again drop trigger k to get a sequence that is one shorter but leads to the same violation, a contradiction.*

In case `U` $=$ `URL':Label`*, then this would be duplicating the prior step l_1. But, we had picked l_1 to be the last such trigger rule step, so we have a contradiction and this case is moot.*

- *Trigger rule k is a* `switch-tab`*: This changes the active web app immediately before the step l which creates the violation. Let us first emphasize that there are no* `new-url(U)` *triggers in the sequence except for l_1. All the triggers between l_1 and l are of the* `switch-tab` *form, otherwise we are in the prior case.*

  If there are multiple such triggers, then in addition to k directly before l, there is another one, call it m. Now, trigger m and all its changes will be completely undone by trigger k. So, trigger m is not needed, and removing trigger m will still yield the exact same violation, a contradiction.

  If there is only one such trigger, then we have $l_1; k; l$ as the sequence giving the violation. k can change the active web app but there is only one (created by l_1) and it is already active, so we can drop k and have a shorter sequence leading to the violation, a contradiction.

Then we consider l to be `new-url(URL':Label)`*, this case works just the same as the prior case of l being* `new-url(URL:Label)`*.*

So, in each case we have shown that there is a sequence one shorter, which still violates the property. This is in contradiction to our assumption that we have picked the sequence with the smallest number of triggers, L. □

With all of these properties verified we know that the SOP holds for the model of IBOS. The same caveats, as explained earlier in this chapter, of course do apply. That is, there could be a coding error in the actual implementation that the model does not capture or the model simply translates the code imperfectly.

As you see in this section, we always limited the amount of steps we are running the model for to check our properties of interest. This reduced number of steps still provides a complete analysis, because all of the properties we look at are independent of any history. Each of them only requires the current state. Building all possible canonical current states can be done in few steps. We have given reduction-based proofs for all of the theorems.

4.4 Related Work

In 2009, Internet Explorer, Chrome, Safari and Firefox had 349 new security vulnerabilities [77], which get commonly exploited by attackers [125, 96, 103, 77]. This shows the need for work on secure browsers. Some work on a web browser that uses formal modeling as part of its design has been done before, for the OP2 [66] browser. For Internet Explorer, the graphical user interface security has been addressed in [23].

A fully verified microkernel operating system is now available in the form of seL4 [80], which uses a very similar set of function calls as the L4Ka::Pistachio microkernel [92]. In the Illinois Browser Operating System (IBOS) [117] web browser the underlying microkernel is L4Ka due to seL4 not being available at the time. IBOS is based on some of the ideas of OP2 but takes them further.

The *same origin policy* (SOP) is discussed in [115] and it turns out that commodity browsers do not do a good job of implementing SOP correctly [24], due to the fact that the required checks are scattered through their large code base.

CHAPTER 5

FOLDING VARIANT NARROWING AND OPTIMAL VARIANT TERMINATION

This chapter is based on [57], which is joint work with Santiago Escobar and José Meseguer. Automated reasoning modulo an equational theory $\mathcal{E}$ is a fundamental technique in many applications. If $\mathcal{E}$ can be split as a disjoint union $E \cup Ax$ in such a way that E is confluent, terminating, sort-decreasing, and coherent modulo a set of equational axioms Ax, narrowing with E modulo Ax provides a complete $\mathcal{E}$-unification algorithm [78]. However, except for the hopelessly inefficient case of full narrowing, little seems to be known about effective narrowing strategies in the general modulo case beyond the quite depressing observation that basic narrowing is *incomplete* modulo AC. Narrowing with equations E modulo axioms Ax can be turned into a practical automated reasoning technique by systematically exploiting the notion of E, Ax*-variants* of a term. After reviewing such a notion, originally proposed by Comon-Lundh and Delaune, and giving various necessary and/or sufficient conditions for it, we explain how narrowing strategies can be used to obtain narrowing algorithms modulo axioms that are: (i) *variant-complete* (generate a complete set of variants for any input term), (ii) *minimal* (such a set does not have redundant variants), and (iii) are *optimally variant-terminating* (the strategy will terminate for an input term t iff t has a finite complete set of variants). We define a strategy called *folding variant narrowing* that satisfies above properties (i)–(iii); in particular, when $E \cup Ax$ has the *finite variant property*, that is, when any term t has a finite complete set of variants, this strategy terminates on any input term and provides a *finitary* $E \cup Ax$-unification algorithm. We also explain how folding variant narrowing has a number of interesting applications in areas such as unification theory, cryptographic protocol verification, and proofs of termination, confluence and coherence of a set of rewrite rules R modulo an equational theory E.

Narrowing is a fundamental rewriting technique useful for many purposes, including equational unification and equational theorem proving [74], combinations of functional and logic programming [65, 69, 95], partial evaluation [4], symbolic reachability analysis of rewrite theories understood as transition systems [91], and symbolic model checking [51].

Narrowing with confluent and terminating equations E enjoys key completeness results, including the generation of a complete set of E-unifiers and the covering of all rewrite sequences starting at an instance of term t by a normalized substitution (see [74]). However, full narrowing (i.e., narrowing at all non-variable term positions) can be quite inefficient both in space and time. Therefore, much work has been devoted to *narrowing strategies* that, while remaining complete, can have a much smaller search space. For instance, the *basic narrowing* strategy [74] was shown to be complete w.r.t. a complete set of E-unifiers for confluent and terminating equations E.

Termination aspects are another important potential benefit of narrowing strategies, since they can sometimes *terminate*, generating a finite search tree when narrowing an input term t, while full narrowing may generate an infinite search tree on the same input term. For example, works such as [74, 7] investigate conditions under which basic narrowing, one of the most fully studied strategies for termination purposes, terminates. Similarly, so-called lazy narrowing strategies also seek to both reduce the search space and to increase the chances of termination. However, the extensive literature on lazy narrowing strategies [105, 10, 55] is mainly focused on efficient evaluation strategies (efficient in the number of narrowing steps or the generality of computed substitutions to reach a term that cannot be narrowed any more) whereas we are interested in narrowing strategies that are terminating and complete for variant generation. The topic of efficient evaluation strategies is outside the scope of this chapter and can be seen as complementary to the narrowing strategies for variant generation developed here. See [9, 71] for references on lazy narrowing strategies. On the other hand, lazy narrowing strategies are demand-driven, and we are not aware of demand-driven strategies for the modulo case, or even of a notion of needed (or demanded) evaluation for the modulo case.

By decomposing an equational theory $\mathcal{E}$ into a set of rules E and a set of equational axioms Ax for which a finite and complete Ax-unification algorithm exists, and imposing natural requirements such as confluence, termination and coherence of the rules E *modulo* Ax, narrowing can be generalized to narrowing *modulo* axioms Ax. As known since the original study [78], the good completeness properties of standard narrowing extend naturally to similar completeness properties for narrowing modulo Ax. This generalization of narrowing to the modulo case has many applications. It is, to begin with, a key component of theorem proving systems that often reason modulo axioms such as associativity-commutativity, and greatly improves the efficiency of general paramodulation. It is, furthermore, very important for adding functional-logical features to algebraic functional languages supporting rewriting modulo combinations of equational axioms. Yet another recent area with many applications is cryptographic protocol analysis, where there is strong interest in analyzing protocol security *modulo* the algebraic theory $\mathcal{E}$ of a protocol's cryptographic functions. That is because protocols deemed to be secure under the standard Dolev-Yao model, which treats the underlying cryptography as a black box, can sometimes be broken by clever use of algebraic properties, e.g., [110].

However, very little is known at present about effective narrowing strategies in the modulo case, and some of the known anomalies ring a cautionary note, to the effect that the naive extensions of standard narrowing strategies can fail rather badly in the modulo case. Indeed, except for [78, 123], we are not aware of any studies about narrowing strategies in the modulo case. Furthermore, as work in [32, 123] shows, narrowing modulo axioms such as associativity-commutativity (AC) can very easily lead to non-terminating behavior and, what is worse, as shown in the Example 1 below, due to Comon-Lundh and Delaune, basic narrowing modulo AC is *not* complete.

Example 1 *[32] Consider the equational theory $(\Sigma, E \cup Ax)$ where E contains the following equations and Ax contains associativity*[1] *and commutativity (AC) for +:*

$$\begin{array}{rclrclrcl} a+a &=& 0\,(5.1) & a+a+X &=& X\ (5.3) & 0+X &=& X\,(5.5) \\ b+b &=& 0\,(5.2) & b+b+X &=& X\ (5.4) & & & \end{array}$$

The set E is terminating, AC-confluent, and AC-coherent. Consider now the unification problem $X_1+X_2 \overset{?}{=} 0$ and one of the possible solutions $\sigma = \{X_1 \mapsto$

[1]We use AC operators many times in the chapter and we often write terms using AC symbols in its varyadic form, e.g., given an AC symbol $+$, we write $a+a+X$ or $+(a,a,X)$ instead of $a+(a+X)$, $+(a,+(a,X))$, $(a+X)+a$, or $+(+(a,X),a)$.

$a + b; X_2 \mapsto a + b\}$, which is a normalized solution. It is well-known that in the free case (when $Ax = \emptyset$) basic narrowing is complete for unification in the sense of lifting all innermost rewriting sequences into basic narrowing sequences (see [94]). That is, given a term t and a (normalized) substitution σ, every innermost rewriting sequence starting from $t\sigma$ can be lifted *to a basic narrowing sequence from t computing a substitution more general than σ. This completeness property fails for basic narrowing modulo AC as shown by the above example when we consider the term $t = X_1+X_2$ instantiated with σ and the following* innermost rewriting sequence modulo *AC from $t\sigma$:* $(a+b)+(a+b) \rightarrow_{E,AC} b+b \rightarrow_{E,AC} 0$. *As further explained in Example 6 below, basic narrowing modulo AC, i.e., the extension of basic narrowing to AC where we just replace syntactic unification by AC-unification, cannot lift the above innermost sequence for $t\sigma$ into a more general basic narrowing sequence, because it is necessary to narrow inside the term generated by instantiation. Therefore, basic narrowing modulo AC is incomplete in the sense of* not *providing a complete $E \cup AC$-unification algorithm, even though E may be confluent, terminating, and coherent modulo AC.*

It seems clear that full narrowing, although complete, is hopelessly inefficient in the free case, and even more so modulo a set Ax of axioms. The above example shows that known efficient strategies like basic narrowing can totally fail to enjoy the desired completeness properties modulo axioms. What can be done? For equational theories of the form $E \cup Ax$, where E is confluent, terminating, and coherent modulo Ax, and such that $E \cup Ax$ has the *finite variant property* (FV) in the sense of [32], we proposed in [54] a narrowing strategy that is complete in the sense of generating a complete set of most general $E \cup Ax$-unifiers, and *terminates* for any input term computing its complete set of variants. And in [53] we gave a method that can be used to check if $E \cup Ax$ is FV. However, FV is a quite strong restriction. What can be done for *any* confluent, terminating and coherent theory modulo axioms Ax?

To the best of our knowledge, except for the hopelessly inefficient case of full narrowing, nothing is known at present about a *general* narrowing strategy that is effective and complete in an adequate sense, including being complete for computing $E \cup Ax$-unifiers, for any theory $E \cup Ax$ under the minimum requirements that E is confluent, terminating, sort-decreasing and coherent modulo Ax, and under minimal requirements on Ax, such as having a finitary Ax-unification algorithm. It turns out that the general notion of *variant*, which makes sense for any such theory $E \cup Ax$ and does not depend on FV, provides the key to obtaining a strategy meeting these requirements, and sheds considerable light on the very process of computing $E \cup Ax$-unifiers by narrowing. In [56] we proposed such a general and effective strategy, called *folding variant narrowing*, which can be applied to any theory $E \cup Ax$, with E confluent, terminating, sort-decreasing, and coherent modulo Ax, and showed that it is both *complete* – both in the sense of computing a complete set of $E \cup Ax$-unifiers, and of computing a minimal and complete set of variants for any input term t – and *optimally variant-terminating* – in the sense that it will terminate for an input term t if and only if t has a finite, complete set of variants. To the best of our knowledge, folding variant narrowing is the only practical, yet complete, general narrowing strategy modulo a set of axioms Ax; in particular the only such one for the AC case. Furthermore, we showed in [56] that there is no other such complete strategy that can terminate on an input term when folding variant narrowing does not. It transforms the, up to now theoretically possible but practically hopeless, mechanism of narrowing modulo axioms Ax into a practically usable automated deduction method, which has already been exploited in a wide range of applications as explained in Section 5.8.

This chapter extends and unifies within a common theoretical framework our earlier contributions in [54, 53, 56], and is already published in [57]. Our goal is to provide the most complete and accessible reference to this general body of ideas by developing in detail its mathematical foundations and its fundamental algorithms. The plan of the chapter, and its main contributions, can be summarized as follows:

1. Comon-Lundh and Delaune's notion of *variant* [32] is the fundamental notion underlying the entire approach. After some preliminaries in Section 5.1, in Section 5.2 we further refine this notion by formalizing the E, Ax-variants of a term t as *pairs* (t', θ), with θ a substitution and t' an E, Ax-canonical form for $t\theta$, and making explicit the preorder relation of generalization that holds between such pairs and the corresponding notion of most general variants in such a preorder.

2. We then give, in Section 5.3, general notions of narrowing strategy

and precise definitions of what it means for a strategy to be: (i) *variant complete*, i.e., it computes a complete set of variants (and possibly also *minimal*, in the sense of the preorder relation of generalization explained above), and (ii) *optimally variant-terminating*, i.e., it will terminate iff there is a finite complete set of variants. Note that we are not interested in efficient narrowing evaluation strategies (as widely studied in the literature of narrowing) and not even on the standard completeness results for narrowing strategies, so we define variant completeness and variant termination notions. These are the essential requirements that will guide us in the search for the desired strategy. To illustrate how tight these essential requirements are, so that none of the known strategies satisfy them, we show that basic narrowing, *both* in the free case ($Ax = \emptyset$) and in the AC case, fails to satisfy properties (i) and/or (ii).

3. A key contribution is the *parametric* notion of *folding narrowing* of Section 5.4. The essential idea is to associate to any narrowing strategy $\mathcal{S}$ a corresponding "folding" version of it. That is, $\mathcal{S}$ is a local strategy, i.e., in the sense of which narrowing steps are allowed from a term, whereas $\mathcal{S}^{\circlearrowleft}$ is a global strategy, i.e., in the sense of tracking variants and avoiding repeated generation of variants. We prove that for any complete strategy $\mathcal{S}$, its folding version $\mathcal{S}^{\circlearrowleft}$ is always variant complete, which is property (i) in (2) above. The presentation of folding narrowing in [56] has been improved in this chapter.

4. What about minimality, and about the termination property (ii) in (2)? Another key contribution is the *variant narrowing strategy* (VN), which takes into account properties of confluence, termination and coherence of the rules E modulo the axioms Ax to restrict the narrowing steps from each term. We prove that VN is variant complete. However, although VN is not variant-terminating, we show that its folding version $VN^{\circlearrowleft}$ is variant complete and optimally variant-terminating, thus variant minimal. The variant narrowing of [54] has been completely redesigned in this chapter.

5. Although all the above results hold for any theory $E \cup Ax$ with E confluent, terminating, sort-decreasing, and coherent modulo Ax, the

case when $E \cup Ax$ has the *finite variant property* (FV) in the sense of [32], that is, when any term t has a finite, complete set of variants, is of particular interest, since then the folding variant narrowing strategy is guaranteed to terminate and to compute a complete and minimal set of variants for *any* input term t. This case is studied in detail in Section 5.5. In particular, we study a number of sufficient and/or necessary conditions for $E \cup Ax$ to enjoy FV.

6. A related practical question is: given $E \cup Ax$, how can we check whether it has the finite variant property? Under appropriate assumptions on $E \cup Ax$, we give an algorithm in Section 5.6 that can be used to check FV. The key idea is to view FV as a generalized termination property. Our algorithm extends and adapts to the variant generation case ideas from the dependency pairs method [12], which is a well-known technique for proving termination of rewriting (modulo axioms). Note that we do not really extend the dependency pairs technique to narrowing: we simply reuse the dependency pairs technique to approximate that there are no infinite variant-preserving narrowing sequences. The same methods can also be used for *disproving* FV for a given theory $E \cup Ax$. The algorithm of [53] has been improved in this chapter, since we were computing bounds for the depth of the narrowing tree in [53] that are not necessary in our improved presentation.

7. Section 5.7 studies in detail one key application of folding variant narrowing, namely, to provide a *finitary* unification algorithm when $E \cup Ax$ enjoys FV. This is very useful for many applications, for example in the analysis of cryptographic protocols. Also, in practice, if $E \cup Ax$ and $E' \cup Ax'$ both enjoy FV, their union $E \cup E' \cup Ax \cup Ax'$ is often FV, either because of disjointness, or because it is quite easy to show it by checking the required conditions. That is, variant-based unification is a quite modular approach, although we do not discuss modularity issues in this chapter.

8. Section 5.8 discusses a number of applications of folding variant narrowing and of variant-based unification, including: (i) cryptographic protocol verification modulo equational properties; (ii) proof techniques for termination of rewriting modulo axioms; and (iii) proof techniques

for proving confluence and coherence of rewrite rules modulo axioms.

5.1 Preliminaries: R, Ax-rewriting

Since Ax-congruence classes can be infinite, $\rightarrow_{R/Ax}$-reducibility is undecidable in general. Therefore, R/Ax-rewriting is usually implemented [78] by R, Ax-rewriting. We assume the following properties on R and Ax:

1. Ax is regular and sort-preserving; furthermore, for each equation $t = t'$ in Ax, all variables in $Var(t)$ have a top sort.
2. Ax has a finitary and complete unification algorithm.
3. The rewrite rules R are sort-decreasing, confluent, and terminating.

Definition 1 (Rewriting modulo) [124] *Let* (Σ, Ax, R) *be an order-sorted rewrite theory satisfying properties* (1)–(3)*. We define the relation* $\rightarrow_{R,Ax}$ *on* $\mathcal{T}_\Sigma(\mathcal{X})$ *by* $t \rightarrow_{p,R,Ax} t'$ *(or just* $t \rightarrow_{R,Ax} t'$*) iff there is a non-variable position* $p \in Pos_\Sigma(t)$*, a rule* $l \rightarrow r$ *in* R*, and a substitution* σ *such that* $t|_p =_{Ax} l\sigma$ *and* $t' = t[r\sigma]_p$*.*

Note that, since Ax-matching is decidable, $\rightarrow_{R,Ax}$ is decidable. Notions such as confluence, termination, irreducible terms, and normalized substitution, are defined in a straightforward manner for $\rightarrow_{R,Ax}$. Note that since R is sort-decreasing, confluent, and terminating, i.e., the relation $\rightarrow_{R/Ax}$ is confluent and terminating, and $\rightarrow_{R,Ax} \subseteq \rightarrow_{R/Ax}$, the relation $\rightarrow^!_{R,Ax}$ is decidable, i.e., it terminates and produces a unique term (up to Ax-equivalence) for each initial term t, denoted by $t{\downarrow}_{R,Ax}$. Of course $t \rightarrow_{R,Ax} t'$ implies $t \rightarrow_{R/Ax} t'$, but the converse does not need to hold in general. To prove completeness of $\rightarrow_{R,Ax}$ w.r.t. $\rightarrow_{R/Ax}$ we need the following additional *coherence* assumption; we refer the reader to [62, 124, 79] for coherence completion algorithms.

4. $\rightarrow_{R,Ax}$ is Ax*-coherent* [78], i.e., $\forall t_1, t_2, t_3$ we have $t_1 \rightarrow_{R,Ax} t_2$ and $t_1 =_{Ax} t_3$ implies $\exists t_4, t_5$ such that $t_2 \rightarrow^*_{R,Ax} t_4$, $t_3 \rightarrow^+_{R,Ax} t_5$, and $t_4 =_{Ax} t_5$. See Figure 5.1 for a graphical illustration.

Let us explain in detail the practical meaning of Ax-coherence, at least for the common associative-commutative (AC) case. The best way to illustrate

$$\begin{tikzcd}
t_1 \arrow[r, "{R,Ax}"'] \arrow[d, Leftrightarrow, "Ax"] & t_2 \arrow[dd, dotted, "{*}_{R,Ax}"] \\
t_3 \arrow[d, dotted, "{+}_{R,Ax}"] & \\
t_5 \arrow[r, dotted, leftrightarrow, "Ax"'] & t_4
\end{tikzcd}$$

Figure 5.1: Ax-coherence

it is by its *absence*. Consider Example 1 where symbol $\_+\_$ is declared AC. Now consider the equation $b + b = 0$. This equation, if not completed by another equation, is *not* coherent modulo AC. What this means is that there will be term *contexts* in which the equation *should* be applied, but it cannot be applied. Consider, for example, the term $b + (a + b)$. Intuitively, we should be able to apply to it the above equation to simplify it to the term $a + 0$ in one step. However, since we are using the weaker rewrite relation $\rightarrow_{E,Ax}$ instead of the stronger but much harder to implement relation $\rightarrow_{E/Ax}$, we cannot! The problem is that the equation cannot be applied (even if we match modulo AC) to either the top term $b+(a+b)$ or the subterm $a+b$. We can however make our equation *coherent* modulo *AC* by adding the extra equation $b + b + Y = 0 + Y$, which, using also the equation $X + 0 = X$, we can slightly simplify to the equation $b + b + Y = Y$. This extended version of our equation will now apply to the term $b + (a + b)$, giving the simplification $b+(a+b) \longrightarrow_{E,Ax} a$. Technically, what coherence means is that the weaker relation $\rightarrow_{E,Ax}$ becomes semantically equivalent to the stronger relation $\rightarrow_{E/Ax}$.

Coherence can be handled implicitly or explicitly, i.e., either the matching mechanism is modified to take care of this issue or the rules are explicitly extended, which is the option shown above; see [122] for a comparison between implicit and explicit extensions. For rewriting, implicit extensions are sufficient in many cases, as the implicit Ax-coherence completion provided by the Maude tool [29] for any combination of associativity (A), commutativity (C), and identity (U) axioms. For narrowing, implicit extension is more complicated and it is sufficient in common cases such as combinations of C, AC, and ACU axioms to consider explicit single-variable extensions, i.e., given an equation $s = t$ one considers $s + x = t + x$ where x is a new variable. The method is as follows for AC. For any symbol f which is AC, and for any equation of the form $f(u, v) = w$ in E, we add also the equation $f(f(u, v), X) = f(w, X)$, where X is a new variable not appearing in u, v, w. In an order-sorted setting, we should give to X *the biggest sort possible*, so that it will apply in all generality. As an additional optimization, note that some equations may already be coherent modulo AC, so that we need not add the extra equation. For example, if the variable X has the biggest possible sort it could have, then the equation $X + 0 = X$ of Example 1 is already coherent, since X will match "the rest of the +-expression," regardless of how big or complex that expression might be, and of where in the expression a constant 0 occurs.

The following theorem in [78, Proposition 1] that generalizes ideas in [102] and has an easy extension to order-sorted theories, links $\rightarrow_{R/Ax}$ with $\rightarrow_{R,Ax}$.

Theorem 8 (Correspondence) [102, 78] *Let* (Σ, Ax, R) *be an order-sorted rewrite theory satisfying properties* (1)–(4). *Then* $t_1 \rightarrow^!_{R/Ax} t_2$ *iff* $t_1 \rightarrow^!_{R,Ax} t_3$, *where* $t_2 =_{Ax} t_3$.

Finally, we provide the notion of decomposition of an equational theory into rules and axioms.

Definition 2 (Decomposition) [54] *Let* $(\Sigma, \mathcal{E})$ *be an order-sorted equational theory. We call* (Σ, Ax, E) *a* decomposition *of* $(\Sigma, \mathcal{E})$ *if* $\mathcal{E} = E \cup Ax$ *and* (Σ, Ax, E) *is an order-sorted rewrite theory satisfying properties* (1)–(4) *above.*

Note that we abuse notation and call (Σ, Ax, E) a decomposition of an order-sorted equational theory $(\Sigma, \mathcal{E})$ even if $\mathcal{E} \neq E \cup Ax$ but E is the explicitly extended Ax-coherent version of a set E' such that $\mathcal{E} = E' \cup Ax$.

5.2 Variants

Given an equational theory $\mathcal{E}$, *the* $\mathcal{E}$*-variants* of a term t are pairs (t', θ) such that $t\theta =_{\mathcal{E}} t'$. This notion can be very useful for reasoning about t modulo $\mathcal{E}$, e.g., unification modulo $\mathcal{E}$ of two terms t and t' can be understood as an appropriate intersection of sets of $\mathcal{E}$-variants for t and t' (as shown in Section 5.7).

Definition 3 (Variants) [32] *Given a term t and an order-sorted equational theory $(\Sigma, \mathcal{E})$, we say that (t', θ) is an $\mathcal{E}$-*variant *of t if $t\theta =_{\mathcal{E}} t'$, where $Dom(\theta) \subseteq Var(t)$ and $Ran(\theta) \cap Var(t) = \emptyset$.*

Example 2 *Let us consider the following equational theory for both the exclusive-or operator and the cancellation equations for public encryption and decryption. The exclusive-or symbol is $\oplus$ and the symbols pk and sk are used for public and private key encryption, respectively. This equational theory is useful for protocol verification (see [91]) and it is relevant here because there are no unification procedures available in the literature which are directly applicable to it, e.g., unification algorithms for exclusive-or such as [8] do not directly apply when extra equations are added.*

$$\begin{array}{rcl@{\qquad}rcl}
X \oplus Y & = & Y \oplus X & pk(K, sk(K, M)) & = & M \\
X \oplus (Y \oplus Z) & = & (X \oplus Y) \oplus Z & sk(K, pk(K, M)) & = & M \\
X \oplus 0 & = & X & & & \\
X \oplus X & = & 0 & & &
\end{array}$$

Given the term $M \oplus M$, we have that: (i) $(0, id)$, (ii) $(0, \{M \mapsto pk(K, sk(K, M'))\})$, and (iii) $(0, \{M \mapsto M' \oplus M' \oplus M''\})$ are some of its variants. Given the term $X \oplus Y$, we have that: (i) $(X \oplus Y, id)$, (ii) $(0, \{X \mapsto U, Y \mapsto U\})$, (iii) $(Z, \{X \mapsto 0, Y \mapsto Z\})$, and (iv) $(Z, \{X \mapsto Z, Y \mapsto 0\})$ are some of its variants.

Suppose that a rewrite theory (Σ, Ax, E) is a decomposition of $(\Sigma, \mathcal{E})$. Given a term t, we can obtain a tighter notion of variant of t (also called an *E,Ax-variant* of t) as a pair (t', θ) with t' an E,Ax-canonical form of the term $t\theta$. That is, the variants of a term now give us all the irreducible *patterns* that instances of t can reduce to.

Definition 4 (Complete set of variants) [32] *Let (Σ, Ax, E) be a decomposition of an order-sorted equational theory $(\Sigma, \mathcal{E})$.* A complete set of E, Ax-variants *(up to renaming) of a term t is a subset V of $\mathcal{E}$-variants of t such that, for each substitution σ, there is a variant $(t', \theta) \in V$ and a substitution ρ such that: (i) t' is E, Ax-irreducible, (ii) $(t\sigma){\downarrow_{E,Ax}} =_{Ax} t'\rho$, and (iii) $(\sigma{\downarrow_{E,Ax}})|_{Var(t)} =_{Ax} (\theta\rho)|_{Var(t)}$.*

Example 3 *The equational theory $(\Sigma, \mathcal{E})$ of Example 2 has a decomposition into E consisting of the oriented equations below, and Ax the associativity and commutativity (AC) axioms for $\oplus$:*

$$
\begin{aligned}
X \oplus 0 &= X && (5.6) & \qquad pk(K, sk(K, M)) &= M && (5.9)\\
X \oplus X &= 0 && (5.7) & \qquad sk(K, pk(K, M)) &= M && (5.10)\\
X \oplus X \oplus Y &= Y && (5.8)
\end{aligned}
$$

Note that equations (5.6)–(5.7) *are not AC-coherent, but adding equation* (5.8) *is sufficient to recover that property (see [124, 43]). For term $t = M \oplus M$, the set $\{(0, id)\}$ provides a complete set of E, Ax-variants, since any possible variant of t is an instance of $(0, id)$.*

The following characterization of variants in terms of a variant semantics for decompositions is useful in various applications discussed later in the chapter.

Definition 5 (Variant Semantics) *Let (Σ, Ax, E) be a decomposition of an equational theory $(\Sigma, \mathcal{E})$ and t be a Σ-term. We define the set of (normalized) E, Ax-variants of t as*

$$[\![t]\!]^{\star}_{E,Ax} = \{(t', \theta) \mid \theta \in \mathcal{S}ubst(\Sigma, \mathcal{X}), t\theta \rightarrow^{!}_{E,Ax} t'', \text{ and } t'' =_{Ax} t'\}.$$

Of course, some variants are *more general* than others, that is, there is a natural preorder $(t', \theta') \sqsubseteq_{E,Ax} (t'', \theta'')$ defining when variant (t'', θ'') is *more general* than variant (t', θ'). This is important, because even though the set of E,Ax-variants of a term t may be infinite, the set of *most general variants* (that is maximal elements in the generalization preorder up to Ax-equivalence and variable renaming) may be finite. Our notion of being more general takes into account not only the instantiation relation between the two substitutions θ_1 and θ_2 and the two normal forms t_1 and t_2 of a term t, but also whether θ_2 is already an E,Ax-normalized substitution, since, for a substitution θ, the less E,Ax rewrite steps, the better.

Definition 6 (Variant Preordering) *Let (Σ, Ax, E) be a decomposition of an equational theory $(\Sigma, \mathcal{E})$ and t be a Σ-term. Given two variants (t_1, θ_1), $(t_2, \theta_2) \in [\![t]\!]^{\star}_{E,Ax}$, we write $(t_1, \theta_1) \sqsubseteq_{E,Ax} (t_2, \theta_2)$, meaning (t_2, θ_2) is* more general *than* (t_1, θ_1)*, iff there is a substitution* ρ *such that* $t_1 =_{Ax} t_2\rho$ *and* $(\theta_1{\downarrow_{E,Ax}})|_{Var(t)} =_{Ax} (\theta_2\rho)|_{Var(t)}$*. We write* $(t_1, \theta_1) \sqsubset_{E,Ax} (t_2, \theta_2)$ *iff* $(t_1, \theta_1) \sqsubseteq_{E,Ax} (t_2, \theta_2)$ *and for every substitution* ρ *such that* $t_1 =_{Ax} t_2\rho$ *and* $(\theta_1{\downarrow_{E,Ax}})|_{Var(t)} =_{Ax} (\theta_2\rho)|_{Var(t)}$*,* ρ *is not a renaming.*

We are, indeed, interested in equivalence classes for variant semantics to provide a notion of semantic equality, written $\simeq_{E,Ax}$, based on $\sqsubseteq_{E,Ax}$.

Definition 7 (Variant Equality) *Let* (Σ, Ax, E) *be a decomposition of an equational theory* $(\Sigma, \mathcal{E})$ *and* t *be a* Σ*-term. For* $S_1, S_2 \subseteq [\![t]\!]^{\star}_{E,Ax}$*, we write* $S_1 \sqsubseteq_{E,Ax} S_2$ *iff for each* $(t_1, \theta_1) \in S_1$*, there exists* $(t_2, \theta_2) \in S_2$ *s.t.* $(t_1, \theta_1) \sqsubseteq_{E,Ax} (t_2, \theta_2)$*. We write* $S_1 \simeq_{E,Ax} S_2$ *iff* $S_1 \sqsubseteq_{E,Ax} S_2$ *and* $S_2 \sqsubseteq_{E,Ax} S_1$*.*

Despite the previous semantic notion of equivalence, we write $(t_1, \theta_1) =_{Ax} (t_2, \theta_2)$ to denote that $t_1 =_{Ax} t_2$ and $\theta_1 =_{Ax} \theta_2$, and we provide a notion of equality of variants up to renaming. Both relations $=_{Ax}$ and $\approx_{Ax}$ will be useful.

Definition 8 (Ax-Equality) *Let* (Σ, Ax, E) *be a decomposition of an equational theory* $(\Sigma, \mathcal{E})$ *and* t *be a* Σ*-term. For* $(t_1, \theta_1), (t_2, \theta_2) \in [\![t]\!]^{\star}_{E,Ax}$*, we write* $(t_1, \theta_1) \approx_{Ax} (t_2, \theta_2)$ *if there is a renaming* ρ *such that* $t_1\rho =_{Ax} t_2\rho$ *and* $(\theta_1\rho)|_{Var(t)} =_{Ax} (\theta_2\rho)|_{Var(t)}$*. For* $S_1, S_2 \subseteq [\![t]\!]^{\star}_{E,Ax}$*, we write* $S_1 \approx_{Ax} S_2$ *if for each* $(t_1, \theta_1) \in S_1$*, there exists* $(t_2, \theta_2) \in S_2$ *s.t.* $(t_1, \theta_1) \approx_{Ax} (t_2, \theta_2)$*, and for each* $(t_2, \theta_2) \in S_2$*, there exists* $(t_1, \theta_1) \in S_1$ *s.t.* $(t_2, \theta_2) \approx_{Ax} (t_1, \theta_1)$*.*

The preorder of Definition 6 allows us to define a most general and complete set of variants that encompasses (modulo Ax and modulo renaming) all the variants for a term t.

Definition 9 (Most General and Complete Variant Semantics) *Let* (Σ, Ax, E) *be a decomposition of an equational theory* $(\Sigma, \mathcal{E})$ *and* t *be a* Σ*-term. A* most general and complete variant semantics *of* t*, denoted* $[\![t]\!]_{E,Ax}$*, is a subset* $[\![t]\!]_{E,Ax} \subseteq [\![t]\!]^{\star}_{E,Ax}$ *such that: (i)* $[\![t]\!]^{\star}_{E,Ax} \sqsubseteq_{E,Ax} [\![t]\!]_{E,Ax}$*, and (ii) for each* $(t_1, \theta_1) \in [\![t]\!]_{E,Ax}$*, there is no* $(t_2, \theta_2) \in [\![t]\!]_{E,Ax} \setminus \{(t_1, \theta_1)\}$ *s.t.* $(t_1, \theta_1) \sqsubseteq_{E,Ax} (t_2, \theta_2)$*.*

For any term t, $[\![t]\!]_{E,Ax}$ characterizes the set of *maximal elements* of the preorder $([\![t]\!]^{\star}_{E,Ax}, \sqsubseteq_{E,Ax})$. The set $[\![t]\!]_{E,Ax}$ is unique up to $\approx_{Ax}$-equivalence. By definition, $[\![t]\!]_{E,Ax} \subset [\![t]\!]^{\star}_{E,Ax}$ and all the substitutions in $[\![t]\!]_{E,Ax}$ are E,Ax-normalized.

Example 4 *In the equational theory of Example 3, for terms* $t = M \oplus sk(K, pk(K, M))$ *and* $s = X \oplus sk(K, pk(K, Y))$*, we have that* $[\![t]\!]_{E,Ax} = \{(0, id)\}$ *and*

$$
\begin{array}{lll}
[\![s]\!]_{E,Ax} = \{ & (X \oplus Y, id), & (0, \{X \mapsto U, Y \mapsto U\}), \\
& (Z, \{X \mapsto U, Y \mapsto Z \oplus U\}), & (Z, \{X \mapsto 0, Y \mapsto Z\}), \\
& (Z, \{X \mapsto Z \oplus U, Y \mapsto U\}), & (Z, \{X \mapsto Z, Y \mapsto 0\}), \\
& (Z_1 \oplus Z_2, & \\
& \ \{X \mapsto U \oplus Z_1, Y \mapsto U \oplus Z_2\})\} &
\end{array}
$$

These two sets are the most general ones w.r.t. $\sqsubseteq_{E,Ax}$.

In the next section, we study how to compute the variants of a term.

5.3 Narrowing Strategies and Optimal Variant Termination

In this section, we introduce narrowing, narrowing strategies and their use for variant generation. As already mentioned, we are not interested in optimal evaluation narrowing strategies [9, 71], which is an extensive topic in the literature on functional logic programming, and not even on the standard completeness results for narrowing strategies. We are interested in narrowing strategies that are terminating and complete for computing variants. A comparison of the *folding variant narrowing strategy*, defined in this chapter, with the related literature on optimal evaluation narrowing strategies is outside the scope of this chapter.

Narrowing generalizes rewriting by performing unification at non-variable positions instead of the usual matching. The essential idea behind narrowing is to *symbolically* represent the rewriting relation between terms as a narrowing relation between more general terms with variables.

Definition 10 (Narrowing modulo) [78, 91] *Let* $\mathcal{R} = (\Sigma, Ax, R)$ *be an order-sorted rewrite theory. Let* $CSU_{Ax}(u = u')$ *be a finite and complete set of Ax-unifiers for any pair of terms* u, u' *with the same top sort. Let* t *be a* Σ*-term and* W *be a set of variables such that* $Var(t) \subseteq W$*. The* R, Ax*-narrowing relation on* $\mathcal{T}_\Sigma(\mathcal{X})$ *is defined as* $t \rightsquigarrow_{p,\sigma,R,Ax} t'$ *(*$\rightsquigarrow_{\sigma,R,Ax}$ *if* p *is understood,* $\rightsquigarrow_\sigma$ *if* R, Ax *are also understood, and* $\rightsquigarrow$ *if* σ *is also understood)* *if there is a non-variable position $p \in Pos_{\Sigma}(t)$, a rule $l \to r \in R$ properly renamed s.t. $Var(l) \cap W = \emptyset$, and a unifier $\sigma \in CSU^{W'}_{Ax}(t|_p = l)$ for $W' = W \cup Var(l)$, such that $t' = (t[r]_p)\sigma$.*

For convenience, in each narrowing step $t \leadsto_{\sigma} t'$ we only specify the part of σ that binds variables of t. The transitive (resp. transitive and reflexive) closure of $\leadsto$ is denoted by $\leadsto^{+}$ (resp. $\leadsto^{*}$). We may write $t \leadsto^{k}_{\sigma} t'$ if there are $u_1, \ldots, u_{k-1}$ and substitutions $\rho_1, \ldots, \rho_k$ such that $t \leadsto_{\rho_1} u_1 \cdots u_{k-1} \leadsto_{\rho_k} t'$, $k \geq 0$, and $\sigma = \rho_1 \cdots \rho_k$.

Example 5 *Consider Example 3. Given the term $t = X \oplus Y$, there are several narrowing steps that can be performed*

$X \oplus Y \leadsto_{\phi_1,E,Ax} Z$ *using* $\phi_1 = \{X \mapsto 0, Y \mapsto Z\}$ *and Equation* (5.6)
$X \oplus Y \leadsto_{\phi_2,E,Ax} Z$ *using* $\phi_2 = \{X \mapsto Z, Y \mapsto 0\}$ *and Equation* (5.6)
$X \oplus Y \leadsto_{\phi_3,E,Ax} Z$ *using* $\phi_3 = \{X \mapsto Z \oplus U, Y \mapsto U\}$ *and Equation* (5.8)
$X \oplus Y \leadsto_{\phi_4,E,Ax} Z$ *using* $\phi_4 = \{X \mapsto U, Y \mapsto Z \oplus U\}$ *and Equation* (5.8)
$X \oplus Y \leadsto_{\phi_5,E,Ax} 0$ *using* $\phi_5 = \{X \mapsto U, Y \mapsto U\}$ *and Equation* (5.7)
$X \oplus Y \leadsto_{\phi_6,E,Ax} Z_1 \oplus Z_2$ *using* $\phi_6 = \{X \mapsto U \oplus Z_1, Y \mapsto U \oplus Z_2\}$ *and Equation* (5.8)

And some redundant narrowing steps with non-normalized substitutions due to the prolific AC-unification such as

$X \oplus Y \leadsto_{\phi_7,E,Ax} Z_1 \oplus Z_2$ *using* $\phi_7 = \{X \mapsto Z_1 \oplus 0, Y \mapsto Z_2\}$ *and Equation* (5.6)
$X \oplus Y \leadsto_{\phi_8,E,Ax} Z_1 \oplus Z_2$ *using* $\phi_8 = \{X \mapsto Z_1, Y \mapsto 0 \oplus Z_2\}$ *and Equation* (5.6)
$X \oplus Y \leadsto_{\phi_9,E,Ax} Z$ *using* $\phi_9 = \{X \mapsto U \oplus U, Y \mapsto Z\}$ *and Equation* (5.8)
$X \oplus Y \leadsto_{\phi_{10},E,Ax} Z$ *using* $\phi_{10} = \{X \mapsto Z, Y \mapsto U \oplus U\}$ *and Equation* (5.8)

$$\begin{array}{ll} X \oplus Y \rightsquigarrow_{\phi_{11},E,Ax} Z_1 \oplus Z_2 & \textit{using } \phi_{11} = \{X \mapsto U \oplus U \oplus Z_1, Y \mapsto Z_2\} \\ & \textit{and Equation } (5.8) \\ X \oplus Y \rightsquigarrow_{\phi_{12},E,Ax} Z_1 \oplus Z_2 & \textit{using } \phi_{12} = \{X \mapsto Z_1, Y \mapsto U \oplus U \oplus Z_2\} \\ & \textit{and Equation } (5.8) \end{array}$$

Indeed, the narrowing **`search`** *command of Maude [30] computes* 124 *different narrowing steps from term t. When we consider narrowing sequences instead of single steps, we can easily get a combinatorial explosion, since after any of the narrowing steps: $X \oplus Y \rightsquigarrow_{\phi_6,E,Ax} Z_1 \oplus Z_2$, $X \oplus Y \rightsquigarrow_{\phi_8,E,Ax} Z_1 \oplus Z_2$, or $X \oplus Y \rightsquigarrow_{\phi_{11},E,Ax} Z_1 \oplus Z_2$, we have another* 124 *different narrowing steps. Also, there are clearly many infinite narrowing sequences, such as the one repeating substitution ϕ_6 again and again: $X \oplus Y \rightsquigarrow_{\phi_6,E,Ax} Z_1 \oplus Z_2 \rightsquigarrow_{\phi'_6,E,Ax} Z'_1 \oplus Z'_2 \rightsquigarrow_{\phi''_6,E,Ax} Z''_1 \oplus Z''_2 \rightsquigarrow_{E,Ax} \cdots$ where $\phi'_6 = \{Z_1 \mapsto U' \oplus Z'_1, Z_2 \mapsto U' \oplus Z'_2\}$ and $\phi''_6 = \{Z'_1 \mapsto U'' \oplus Z''_1, Z'_2 \mapsto U'' \oplus Z''_2\}$. Clearly, strategies that dramatically reduce this search space, yet are complete, are surely needed.*

5.3.1 Completeness of Narrowing w.r.t. Rewriting

Several notions of completeness of narrowing w.r.t. rewriting have been given in the literature (e.g., [74, 78, 91]).

Theorem 9 (Completeness of Full Narrowing Modulo) [78] *Let (Σ, Ax, E) be a decomposition of an equational theory $(\Sigma, \mathcal{E})$. Let t_1 be a Σ-term and σ be an E,Ax-normalized substitution. If $t_1\sigma \to_{E,Ax} t_2 \to_{E,Ax} \cdots \to_{E,Ax} t_n$ such that $t_n = (t_1\sigma){\downarrow_{E,Ax}}$, then there exist terms $t'_2, \ldots, t'_n$ and E,Ax-normalized substitutions $\theta_1, \ldots, \theta_n$ and ρ s.t. $t_1 \rightsquigarrow_{\theta_1,E,Ax} t'_2 \rightsquigarrow_{\theta_2,E,Ax} \cdots \rightsquigarrow_{\theta_n,E,Ax} t'_n$, $\sigma|_{Var(t_1)} =_{Ax} (\theta_1 \cdots \theta_n\rho)|_{Var(t_1)}$, and $t_i =_{Ax} t'_i\rho$ for $1 \leq i \leq n$.*

We can easily extend the previous result to allow non-normalized substitutions.

Lemma 8 (Completeness) *Let (Σ, Ax, E) be a decomposition of an equational theory $(\Sigma, \mathcal{E})$. Let t_1 be a Σ-term and θ be any substitution. If $t_1\theta \to^!_{E,Ax} t_2$, then there exists a term t'_2 and two E, Ax-normalized substitutions σ and ρ s.t. $t_1 \rightsquigarrow^*_{\sigma,E,Ax} t'_2$, $(\theta{\downarrow_{E,Ax}})|_{Var(t_1)} =_{Ax} (\sigma\rho)|_{Var(t_1)}$, and $t_2 =_{Ax} t'_2\rho$.*

Proof. *Let $\bar{\theta} = \theta{\downarrow_{E,Ax}}$. By coherence, confluence and termination of $\rightarrow_{E,Ax}$, $t_1\theta \rightarrow^!_{E,Ax} t_2$ implies $\exists t_3 : t_1\bar{\theta} \rightarrow^!_{E,Ax} t_3$ and $t_3 =_{Ax} t_2$. By Theorem 9, there exists a term t'_3 and two E, Ax-normalized substitutions σ and ρ s.t. $t_1 \leadsto^*_{\sigma,R,E} t'_3$, $\bar{\theta}|_{Var(t_1)} =_{Ax} (\sigma\rho)|_{Var(t_1)}$, and $t_3 =_{Ax} t'_3\rho$.* □

As a direct consequence of Lemma 8 we obtain the following result.

Corollary 1 (Complete Variant Semantics by Full Narrowing) *Let (Σ, Ax, E) be a decomposition of an equational theory $(\Sigma, \mathcal{E})$. Then for each term t, the set*

$$[\![t]\!]^{Full}_{E,Ax} = \{(t', \theta) \mid t \leadsto^*_{\theta,E,Ax} t' \wedge t' = t'{\downarrow_{E,Ax}}\}$$

is a complete set of variants, i.e., $[\![t]\!]^{\star}_{E,Ax} \sqsubseteq_{E,Ax} [\![t]\!]^{Full}_{E,Ax}$.

Note that, although $[\![t]\!]^{\star}_{E,Ax} \sqsubseteq_{E,Ax} [\![t]\!]^{Full}_{E,Ax}$, not all $(t', \theta) \in [\![t]\!]^{Full}_{E,Ax}$ need to be most general, i.e., $[\![t]\!]^{Full}_{E,Ax}$ is not necessarily a most general complete set of variants as shown by Example 5. Therefore, full narrowing gives us a way of computing a *complete variant semantics*, $[\![t]\!]^{Full}_{E,Ax}$, from which we would like to obtain a subset $S \subseteq [\![t]\!]^{Full}_{E,Ax}$ such that S is a *most general and complete variant semantics*, i.e., $S = [\![t]\!]_{E,Ax}$. The key question, then, is:

> *Can we compute the set $[\![t]\!]_{E,Ax}$ of most general $\mathcal{E}$-variants of a term t effectively?*

This is not entirely obvious. Full (i.e., unrestricted) E,Ax-narrowing may never terminate and the set $[\![t]\!]^{Full}_{E,Ax}$ can easily be infinite, even though a finite set of most general elements for it exists. The solution, of course, is that we should look for adequate narrowing *strategies* that have better properties than full E,Ax-narrowing so that if $[\![t]\!]_{E,Ax}$ is *finite*, then the narrowing strategy will *terminate* and will compute $[\![t]\!]_{E,Ax}$.

5.3.2 Narrowing Strategies and Their Properties

In order to obtain an appropriate narrowing strategy that enjoys better properties than full E,Ax-narrowing and allows to compute $[\![t]\!]_{E,Ax}$, we need to characterize what a narrowing strategy is and which properties it must satisfy. E.g., the notion of variant-completeness rather than the standard full narrowing completeness becomes essential.

First, we define the notion of a narrowing strategy and several useful properties. Given a narrowing sequence $\alpha : (t_0 \rightsquigarrow_{p_0,\sigma_0,R,Ax} t_1 \cdots \rightsquigarrow_{p_{n-1},\sigma_{n-1},R,Ax} t_n)$, we denote by α_i the narrowing sequence $\alpha_i : (t_0 \rightsquigarrow_{p_0,\sigma_0,R,Ax} t_1 \cdots \rightsquigarrow_{p_{i-1},\sigma_{i-1},R,Ax} t_i)$ which is a prefix of α. Given an order-sorted rewrite theory $\mathcal{R}$, we denote by $\mathit{Full}_{\mathcal{R}}(t)$ the (possibly infinite) set of all narrowing sequences starting at term t.

Definition 11 (Narrowing Strategy) *A* narrowing strategy $\mathcal{S}$ *is a function of two arguments, namely, a rewrite theory $\mathcal{R} = (\Sigma, Ax, R)$ and a term $t \in \mathcal{T}_{\Sigma}(\mathcal{X})$, which we denote by $\mathcal{S}_{\mathcal{R}}(t)$, such that $\mathcal{S}_{\mathcal{R}}(t) \subseteq \mathit{Full}_{\mathcal{R}}(t)$. We require $\mathcal{S}_{\mathcal{R}}(t)$ to be prefix closed, i.e., for each narrowing sequence $\alpha \in \mathcal{S}_{\mathcal{R}}(t)$ of length n, and each $i \in \{1, \ldots, n\}$, we also have $\alpha_i \in \mathcal{S}_{\mathcal{R}}(t)$.*

Note that this definition of a narrowing strategy is very general and does not consider any aspect about efficient narrowing strategies at all, see [9] for efficient narrowing strategies.

Each narrowing strategy is trivially sound w.r.t. rewriting. We say that a narrowing strategy $\mathcal{S}$ is *complete* w.r.t. rewriting if it satisfies Theorem 9 above, concretized as follows.

Definition 12 (Completeness of a Narrowing Strategy) *Let $\mathcal{R} = (\Sigma, Ax, E)$ be a decomposition of an equational theory $(\Sigma, \mathcal{E})$. A narrowing strategy $\mathcal{S}_{\mathcal{R}}$ is called* complete *iff for each pair of terms t_1 and t_2 and each E,Ax-normalized substitution θ such that $t_1\theta \rightarrow^{!}_{E,Ax} t_2$, there exists a term t'_2 and two E,Ax-normalized substitutions σ and ρ s.t. $(t_1 \rightsquigarrow^{*}_{\sigma,E,Ax} t'_2) \in \mathcal{S}_{\mathcal{R}}(t)$, $\theta|_{\mathit{Var}(t_1)} =_{Ax} (\sigma\rho)|_{\mathit{Var}(t_1)}$, and $t_2 =_{Ax} t'_2\rho$.*

In this chapter we are interested in a notion of completeness of a narrowing strategy slightly different than previous notions, which we call *variant-completeness.* First, we extend the variant semantics to narrowing strategies and consider only narrowing sequences to normalized terms.

Definition 13 (Narrowing Variant Semantics) *Let $\mathcal{R} = (\Sigma, Ax, E)$ be a decomposition of an equational theory $(\Sigma, \mathcal{E})$ and $\mathcal{S}_{\mathcal{R}}$ be a narrowing strategy. We define the set of* narrowing variants *of a term t w.r.t. $\mathcal{S}_{\mathcal{R}}$ as $[\![t]\!]^{\mathcal{S}_{\mathcal{R}}}_{E,Ax} = \{(t', \theta) \mid (t \rightsquigarrow^{*}_{\theta,E,Ax} t') \in \mathcal{S}_{\mathcal{R}}(t) \text{ and } t' = t'{\downarrow_{E,Ax}}\}$.*

Now, we can define our notion of variant-completeness.

Definition 14 (Variant Completeness and Minimality) *Let $\mathcal{R} = (\Sigma, Ax, E)$ be a decomposition of an equational theory $(\Sigma, \mathcal{E})$. A narrowing strategy $\mathcal{S}_\mathcal{R}$ is called* E, Ax-variant-complete *(or just variant-complete) iff for any Σ-term t we have that $[\![t]\!]_{E,Ax} \simeq_{E,Ax} [\![t]\!]^{\mathcal{S}_\mathcal{R}}_{E,Ax}$. The narrowing strategy $\mathcal{S}_\mathcal{R}$ is called* E, Ax-variant-minimal *(or just variant-minimal) iff, in addition, for any Σ-term t we have that $[\![t]\!]_{E,Ax} \approx_{Ax} [\![t]\!]^{\mathcal{S}_\mathcal{R}}_{E,Ax}$ and for each pair of variants $(t_1, \theta_1), (t_2, \theta_2) \in [\![t]\!]^{\mathcal{S}_\mathcal{R}}_{E,Ax}$ such that $(t_1, \theta_1) \neq_{Ax} (t_2, \theta_2)$, we have that $(t_1, \theta_1) \not\approx_{Ax} (t_2, \theta_2)$.*

In practice, the set $\mathcal{S}_\mathcal{R}(t)$ of narrowing sequences from a term t will be generated by an *algorithm* $\mathcal{A}_{\mathcal{S}_\mathcal{R}}$. That is, $\mathcal{A}_{\mathcal{S}_\mathcal{R}}$ is a computable function such that, given a pair $(\mathcal{R}, t)$, it enumerates the set $\mathcal{S}_\mathcal{R}(t)$. Even when $\mathcal{R} = (\Sigma, Ax, E)$ is a decomposition of an equational theory, the strategy $\mathcal{S}_\mathcal{R}$ is variant-complete, and $[\![t]\!]_{E,Ax}$ is finite on an input term t, it may happen that $[\![t]\!]^{\mathcal{S}_\mathcal{R}}_{E,Ax}$ is not finite. Furthermore, even if $[\![t]\!]^{\mathcal{S}_\mathcal{R}}_{E,Ax}$ is finite, its enumeration using the algorithm $\mathcal{A}_{\mathcal{S}_\mathcal{R}}$ may not terminate. We are of course interested in variant-complete narrowing strategies that will *always* terminate on an input term t whenever $[\![t]\!]_{E,Ax}$ is finite. This leads to the following notion of variant termination for an algorithm $\mathcal{A}_\mathcal{S}$, further restricting the class of algorithms we are interested in.

Definition 15 (Optimal Variant Termination) *Let $\mathcal{R} = (\Sigma, Ax, E)$ be a decomposition of an equational theory $(\Sigma, \mathcal{E})$ and $\mathcal{S}_\mathcal{R}$ be an E, Ax-variant-complete narrowing strategy. An algorithm $\mathcal{A}_{\mathcal{S}_\mathcal{R}}$ for computing $\mathcal{S}_\mathcal{R}$ is* variant-terminating *iff $\mathcal{A}_{\mathcal{S}_\mathcal{R}}(t)$ terminates on input $(\mathcal{R}, t)$ iff $[\![t]\!]^{\mathcal{S}_\mathcal{R}}_{E,Ax}$ is finite. An algorithm $\mathcal{A}_{\mathcal{S}_\mathcal{R}}$ is* optimally variant-terminating *iff both $\mathcal{A}_{\mathcal{S}_\mathcal{R}}$ is variant-terminating and $[\![t]\!]^{\mathcal{S}_\mathcal{R}}_{E,Ax}$ is variant-minimal for every Σ-term t.*

By abuse of language, we say that a narrowing strategy $\mathcal{S}$ is variant-terminating (resp. optimally variant-terminating) whenever $\mathcal{A}_\mathcal{S}$ is. The term "optimally variant-terminating" is justified as follows.

Proposition 1 *Let $\mathcal{R} = (\Sigma, Ax, E)$ be a decomposition of an equational theory $(\Sigma, \mathcal{E})$. Let $\mathcal{S}_\mathcal{R}$ be an E, Ax-variant-complete narrowing strategy and $\mathcal{S}'_\mathcal{R}$ be an optimally variant-terminating narrowing strategy. Then, for each Σ-term t such that $\mathcal{S}_\mathcal{R}(t)$ terminates, then $\mathcal{S}'_\mathcal{R}(t)$ also terminates.*

Proof. *If $\mathcal{S}_{\mathcal{R}}(t)$ terminates, then $[\![t]\!]^{\mathcal{S}_{\mathcal{R}}}_{E,Ax}$ is necessarily finite. Therefore, $[\![t]\!]^{\mathcal{S}'_{\mathcal{R}}}_{E,Ax}$ is also necessarily finite, since $\mathcal{S}'_{\mathcal{R}}$ is variant-minimal. Therefore, $\mathcal{S}'_{\mathcal{R}}(t)$ also terminates.* $\square$

Therefore, if a variant-complete narrowing strategy $\mathcal{S}_{\mathcal{R}}$ is optimally variant-terminating, then whenever any other narrowing strategy $\mathcal{S}'_{\mathcal{R}}$ enjoying the same variant-completeness property terminates on a term t, $\mathcal{S}_{\mathcal{R}}$ is guaranteed to terminate on t as well. Such an optimally variant-terminating strategy would be a powerful tool, improving over many narrowing strategies defined previously in the literature, as shown in the next section. Later, in Sections 5.4 and 5.5 below, we introduce a narrowing strategy that is optimally variant-terminating under some conditions.

5.3.3 Basic Narrowing (Modulo) is neither Variant-Complete nor Optimally Variant-Terminating

In this section we show that basic narrowing modulo AC is not variant-complete. Furthermore, we show that even basic narrowing without axioms is not optimally variant-terminating, thus showing that there is room for improvement even in the free case. We extend the standard definition of basic narrowing given in [73] to the modulo case.

Definition 16 (Basic Narrowing modulo Ax) *Let (Σ, Ax, R) be an order-sorted rewrite theory. Given a term $t \in \mathcal{T}_{\Sigma}(\mathcal{X})$, a substitution ρ, and a set W of variables such that $Var(t) \subseteq W$ and $Var(\rho) \subseteq W$, a basic narrowing step modulo Ax for $\langle t, \rho\rangle$ is defined by $\langle t, \rho\rangle \overset{b}{\leadsto}_{p,\theta,R,Ax} \langle t', \rho'\rangle$ iff there is $p \in Pos_{\Sigma}(t)$, a rule $l \to r \in R$ properly renamed s.t. $Var(l) \cap W = \emptyset$, and $\theta \in CSU^{W'}_{Ax}(t|_p\rho = l)$ for $W' = W \cup Var(l)$ such that $t' = t[r]_p$, and $\rho' = \rho\theta$.*

Basic narrowing modulo AC is incomplete w.r.t. innermost rewriting modulo AC [123] despite its completeness in the free case [94], i.e., there are innermost rewriting sequences modulo AC that are not lifted to basic narrowing sequences modulo Ax. In particular, basic narrowing modulo AC is not variant-complete.

Example 6 *The following full narrowing sequence relevant for the unification problem* $X_1 + X_2 \overset{?}{=} 0$ *of Example 1:*

$$
\begin{array}{l}
X_1 + X_2 \rightsquigarrow_{\rho_1,E,Ax} X' + X'' \\
\textit{using } \rho_1 = \{X_1 \mapsto a + X', X_2 \mapsto a + X''\} \textit{ and rule } (5.3) \\
X' + X'' \rightsquigarrow_{\rho_2,E,Ax} 0 \\
\quad \textit{using } \rho_2 = \{X' \mapsto b, X'' \mapsto b\} \textit{ and rule } (5.2)
\end{array}
$$

is not a basic narrowing sequence modulo AC, since after the first step it results in a variable X *and no further basic narrowing step modulo AC is possible:*

$$
\begin{array}{l}
\langle X_1 + X_2, id\rangle \overset{b}{\rightsquigarrow}_{\tau_1,E,Ax} \langle X, \tau_1\rangle \\
\textit{using } \tau_1 = \{X_1 \mapsto a + X', X_2 \mapsto a + X'', X \mapsto X' + X''\} \textit{ and rule } (5.3)
\end{array}
$$

Since the pair $(0, \rho_1\rho_2)$ *is a variant of* $X_1 + X_2$ *not subsumed by any basic narrowing sequence generated from* $X_1 + X_2$*, basic narrowing modulo AC is* not variant-complete*.*

Moreover, basic narrowing in the free case is actually not optimally variant-terminating, as shown by the following example.

Example 7 *Consider the rewrite theory* $\mathcal{R} = (\Sigma, \emptyset, E)$ *where* E *is the set of confluent and terminating rules* $E = \{f(x) \to x, f(f(x)) \to f(x)\}$ *and* Σ *contains only the unary symbol* f *and a constant* a*. The term* $t = f(x)$ *has only one variant:* $[\![f(x)]\!]_{E,Ax} = \{(x, id)\}$*. Indeed, the theory has the finite variant property (see Example 15 in Section 5.5, or also [53]). Basic narrowing performs the following two narrowing steps:*

(i) $\langle f(x), id\rangle \overset{b}{\rightsquigarrow}_{\{x\mapsto x'\},E} \langle x', \{x \mapsto x'\}\rangle$ *and*

(ii) $\langle f(x), id\rangle \overset{b}{\rightsquigarrow}_{\{x\mapsto f(x')\},E} \langle f(x'), \{x \mapsto f(x')\}\rangle$.

However, the second narrowing step leads to the following non-terminating basic narrowing sequence:

$$
\begin{array}{ll}
\langle f(x), id\rangle & \overset{b}{\rightsquigarrow}_{\{x\mapsto f(x')\},E} \langle f(x'), \{x \mapsto f(x')\}\rangle \\
& \overset{b}{\rightsquigarrow}_{\{x'\mapsto f(x'')\},E} \langle f(x''), \{x \mapsto f(f(x'')), x' \mapsto f(x'')\}\rangle \\
& \ldots
\end{array}
$$

and basic narrowing is unable to terminate and provide the finite number of variants associated to the term t.

In the next section we define a variant-complete narrowing strategy.

5.4 Folding Variant Narrowing

In order to compute the variants of a term, we can simply keep track of all the variants generated so far by narrowing, since we know that for any decomposition there is a (possibly infinite) set of most general variants (modulo axioms and modulo renaming) and sooner or later full narrowing will generate those most general variants, thanks to Corollary 1. In this section, we define a narrowing strategy called *folding narrowing*, which works in this way and achieves variant-completeness. Note that the folding narrowing strategy is parametric on another complete narrowing strategy, which will allow us later to define more concise narrowing strategies for obtaining the variants. Also note that only when a term has a finite number of most general variants, a narrowing strategy can be optimally variant-terminating for that term; this is studied in detail in Section 5.5 below.

First, we need to introduce the notion of variant preordering with normalization, which is very close to Definition 6, in order to capture when a newly generated variant is subsumed by a previously generated one.

Definition 17 (Normalized Variant Preordering) *Let (Σ, Ax, E) be a decomposition of an equational theory $(\Sigma, \mathcal{E})$ and t be a Σ-term. Given two variants $(t_1, \theta_1), (t_2, \theta_2) \in [\![t]\!]^{\star}_{E,Ax}$, we write $(t_1, \theta_1) \sqsubseteq^{!}_{E,Ax} (t_2, \theta_2)$, meaning (t_2, θ_2) is a more general variant of t than (t_1, θ_1), iff $(t_1{\downarrow}_{E,Ax}, \theta_1) \sqsubseteq_{E,Ax} (t_2, \theta_2)$.*

We define in Definition 18 below the folding narrowing strategy, which is based on the different levels of reachable states, denoted as $\mathit{Frontier}_{\sqsubseteq^{!}_{E,Ax}}(I)_i$, and the relation $\sqsubseteq^{!}_{E,Ax}$ for identifying variants subsumed by previously generated ones. We are presenting a specialized version of the folding reachable transition system of [51] rolled together with our folding narrowing strategy. Given a decomposition $\mathcal{R} = (\Sigma, Ax, E)$ of an equational theory $(\Sigma, \mathcal{E})$ and a narrowing strategy $\mathcal{S}_{\mathcal{R}}$, we extend $\mathcal{S}_{\mathcal{R}}$ to variants as follows: given a term

t and a substitution ρ, $\mathcal{S}_{\mathcal{R}}((t,\rho)) = \{(t,\rho) \rightsquigarrow^*_{\sigma,E,Ax}(t',\rho\sigma) \mid (t \rightsquigarrow^*_{\sigma,E,Ax} t') \in \mathcal{S}_{\mathcal{R}}(t)\}$.

Definition 18 (Folding Narrowing Strategy) *Let $\mathcal{R} = (\Sigma, Ax, E)$ be a decomposition of an equational theory $(\Sigma, \mathcal{E})$ and $\mathcal{S}_{\mathcal{R}}$ a narrowing strategy. Let t be a Σ-term. The frontier from $I = (t, id)$ with folding $\sqsubseteq^!_{E,Ax}$ is defined as*

$$\begin{aligned}
&\mathit{Frontier}_{\sqsubseteq^!_{E,Ax}}(I)_0 = I,\\
&\mathit{Frontier}_{\sqsubseteq^!_{E,Ax}}(I)_{n+1} =\\
&\quad\{(y,\rho\sigma) \mid (\exists (z,\rho) \in \mathit{Frontier}_{\sqsubseteq^!_{E,Ax}}(I)_n : (z,\rho) \rightsquigarrow_{\sigma,E,Ax}(y,\rho\sigma)) \wedge\\
&\qquad\qquad (\nexists k \leq n, (w,\tau) \in \mathit{Frontier}_{\sqsubseteq^!_{E,Ax}}(I)_k : (y,\rho\sigma) \ \sqsubseteq^!_{E,Ax} \ (w,\tau))\}
\end{aligned}$$

The folding $\mathcal{S}_{\mathcal{R}}$-narrowing strategy, denoted by $\mathcal{S}^{\circlearrowleft}_{\mathcal{R}}(t)$, is defined as

$$\begin{aligned}
\mathcal{S}^{\circlearrowleft}_{\mathcal{R}}(t) = \{t \rightsquigarrow^k_{\sigma,E,Ax} t' \mid ((t, id) \rightsquigarrow^k_{\sigma,E,Ax}(t',\sigma)) \in \mathcal{S}_{\mathcal{R}}(t) \wedge\\
(t',\sigma) \in \mathit{Frontier}_{\sqsubseteq^!_{E,Ax}}(I)_k\}
\end{aligned}$$

We write $\mathit{Full}^{\circlearrowleft}_{\mathcal{R}}$ to denote the folding version of the full narrowing strategy $\mathit{Full}_{\mathcal{R}}$. The following example shows the advantages of folding full narrowing for computing variants, for instance w.r.t. basic narrowing modulo AC.

Example 8 *Considering Example 7. Using the $\mathit{Full}^{\circlearrowleft}_{\mathcal{R}}$ strategy, we only get step (i), since step (ii) is subsumed by step (i). That is, $(f(x'), \{x \mapsto f(x')\}) \sqsubseteq^!_{E,\emptyset} (x', \{x \mapsto x'\})$, since $f(x')\downarrow_{E,Ax} = x'$. So even though basic narrowing does not terminate for this equational theory, $\mathit{Full}^{\circlearrowleft}_{\mathcal{R}}$ does.*

The following example shows what steps are performed by $\mathit{Full}^{\circlearrowleft}_{\mathcal{R}}$ and its termination on our running example.

Example 9 *Using the theory from Example 3, for $t = X \oplus Y$ we get the following $\mathit{Full}^{\circlearrowleft}_{\mathcal{R}}$ steps. First, we show the narrowing steps with normalized substitutions.*

(i) $(X \oplus Y, id) \rightsquigarrow_{\phi_1}(Z, \phi_1)$, *using Equation* (5.6) *and substitution* $\phi_1 = \{X \mapsto 0, Y \mapsto Z\}$,

(ii) $(X \oplus Y, id) \rightsquigarrow_{\phi_2}(Z, \phi_2)$, *using Equation* (5.6) *and substitution* $\phi_2 = \{X \mapsto Z, Y \mapsto 0\}$,

(iii) $(X \oplus Y, id) \rightsquigarrow_{\phi_3} (Z, \phi_3)$, *using Equation* (5.8) *and substitution* $\phi_3 = \{X \mapsto Z \oplus U, Y \mapsto U\}$,

(iv) $(X \oplus Y, id) \rightsquigarrow_{\phi_4} (Z, \phi_4)$, *using Equation* (5.8) *and substitution* $\phi_4 = \{X \mapsto U, Y \mapsto Z \oplus U\}$,

(v) $(X \oplus Y, id) \rightsquigarrow_{\phi_5} (0, \phi_5)$, *using Equation* (5.7) *and substitution* $\phi_5 = \{X \mapsto U, Y \mapsto U\}$,

(vi) $(X \oplus Y, id) \rightsquigarrow_{\phi_6} (Z_1 \oplus Z_2, \phi_6)$, *using Equation* (5.8) *and* $\phi_6 = \{X \mapsto U \oplus Z_1, Y \mapsto U \oplus Z_2\}$.

Non-normalized narrowing steps such as

$(X \oplus Y, id) \rightsquigarrow_{\phi_6} (Z, \phi_7)$, *using Equation* (5.8) *and* $\phi_7 = \{X \mapsto U \oplus U, Y \mapsto Z\}$

are also computed by $Full^{\circlearrowleft}_{\mathcal{R}}$ *but all are finally subsumed by a variant with the normalized version of the same substitution, e.g.,* $(Z, \phi_7) \sqsubseteq_{E,Ax} (Z, \phi_1)$. *Note that* $Full^{\circlearrowleft}_{\mathcal{R}}$ *terminates after generating all narrowing steps above:*

1. *There are no further steps possible from (i)-(iv), since any instantiation of Z for which a narrowing step is possible would mean that the computed substitution is not normalized.*

2. *There is no further step possible from (v), since* 0 *is a normal form.*

3. *There are no further steps possible from (vi), since we are back at the beginning, i.e,* $(Z_1 \oplus Z_2, \phi_6) \sqsubseteq^{!}_{E,Ax} (t, id)$, *and can repeat all of the steps possible from* (t, id), *but all of the results are subsumed by the same step we already have from* (t, id).

Note that by the use of the folding definition we get only the shortest paths to each possible term (depending on the substitution), since longer paths are simply subsumed by shorter ones using $\sqsubseteq_{E,Ax}$.

Any folding narrowing strategy is sound as it is a further restriction of the narrowing strategy. We prove that any folding narrowing strategy $\mathcal{S}^{\circlearrowleft}$ is *variant-complete* provided the given narrowing strategy $\mathcal{S}$ that is restricted by folding is *complete* according to Definition 12. First, we provide two auxiliary definitions and an auxiliary result.

Definition 19 *Given a decomposition* (Σ, Ax, E)*, a term* t*, and two narrowing sequences* $\alpha_1 : t \leadsto^*_{\sigma_1,E,Ax} t_1$ *and* $\alpha_2 : t \leadsto^*_{\sigma_2,E,Ax} t_2$*, we write* $\alpha_1 \sqsubseteq_{E,Ax} \alpha_2$ *if there is a substitution* θ *such that* $(\sigma_1{\downarrow_{E,Ax}})|_{Var(t)} =_{Ax} (\sigma_2\theta)|_{Var(t)}$ *and* $t_1 =_{Ax} t_2\theta$*. We write* $\alpha_1 \approx_{Ax} \alpha_2$ *if there is a renaming substitution* ρ *such that* $\sigma_1|_{Var(t)} =_{Ax} (\sigma_2\rho)|_{Var(t)}$ *and* $t_1 =_{Ax} t_2\rho$*.*

Definition 20 (Most General Narrowing Sequence) *Given a decomposition* (Σ, Ax, E)*, a narrowing sequence* $\alpha : t \leadsto^*_{\theta,E,Ax} (t\theta){\downarrow_{E,Ax}}$ *is called a* most general *narrowing sequence if for any narrowing sequence* $\alpha' : t \leadsto^*_{\theta',E,Ax} (t\theta'){\downarrow_{E,Ax}}$ *such that* $\alpha \sqsubseteq_{E,Ax} \alpha'$*, then* $\alpha \approx_{Ax} \alpha'$*.*

Lemma 9 *Let* $\mathcal{R} = (\Sigma, Ax, E)$ *be a decomposition of an equational theory* $(\Sigma, \mathcal{E})$*. Let* $\mathcal{S}_\mathcal{R}$ *be a complete narrowing strategy. If* $\alpha : t \leadsto^*_{\sigma,E,Ax} (t\sigma){\downarrow_{E,Ax}}$ *and* α *is most general, then there is a narrowing sequence* $\alpha' : t \leadsto^*_{\sigma',E,Ax} (t\sigma'){\downarrow_{E,Ax}}$ *such that* $\alpha' \in \mathcal{S}^{\circlearrowleft}_\mathcal{R}(t)$ *and* $\alpha \approx_{Ax} \alpha'$*.*

Proof. *By contradiction. Let* $\alpha : t \leadsto_{\sigma_1,E,Ax} t_1 \cdots t_{k-1} \leadsto_{\sigma_k,E,Ax} t_k = (t\sigma){\downarrow_{E,Ax}}$*. Since there is no narrowing sequence* $\alpha' : t \leadsto^*_{\sigma',E,Ax} (t\sigma'){\downarrow_{E,Ax}}$ *such that* $\alpha' \in \mathcal{S}^{\circlearrowleft}_\mathcal{R}(t)$ *and* $\alpha' \approx_{Ax} \alpha$*, by completeness of* $\mathcal{S}_\mathcal{R}$ *there is an alternative narrowing sequence* $\beta : t \leadsto_{\theta_1,E,Ax} u_1 \cdots u_{n-1} \leadsto_{\theta_n,E,Ax} u_n = (t\theta){\downarrow_{E,Ax}}$ *in* $\mathcal{S}^{\circlearrowleft}_\mathcal{R}(t)$ *with* $\theta = \theta_1 \cdots \theta_n$ *and* $n \leq k$ *such that* $(t_n, \sigma_1 \cdots \sigma_n) \sqsubseteq^!_{E,Ax} (u_n, \theta_1 \cdots \theta_n)$*, i.e., there is a substitution* ρ *such that* $t_n{\downarrow_{E,Ax}} =_{Ax} u_n\rho$ *and* $((\sigma_1 \cdots \sigma_n){\downarrow_{E,Ax}})|_{Var(t)} =_{Ax} (\theta_1 \cdots \theta_n\rho)|_{Var(t)}$*. Note that* ρ *cannot be a renaming, since* ρ *being a renaming implies* $\beta \approx_{Ax} \alpha$*. Then, by confluence, there is a rewriting sequence starting from* u_n *that reaches* $t\sigma{\downarrow_{E,Ax}}$*, i.e.,* $(u_n\rho\sigma_{n+1} \cdots \sigma_k) \rightarrow^*_{E,Ax} (t\sigma){\downarrow_{E,Ax}}$*. But this rewriting sequence can be lifted to a narrowing sequence, i.e., by completeness of* $\mathcal{S}_\mathcal{R}$ *there is a narrowing sequence* $\beta' : u_n \leadsto^*_{\tau,E,Ax} t''$ *and a substitution* ρ' *such that* $(\sigma_{n+1} \cdots \sigma_k){\downarrow_{E,Ax}}|_{Var(u_n)} =_{Ax} (\tau\rho')|_{Var(u_n)}$ *and* $(t\sigma){\downarrow_{E,Ax}} =_{Ax} t''\rho'$*. Then, we can concatenate both narrowing sequences* $\beta;\beta' : t \leadsto^*_{\theta,E,Ax} u_n \leadsto^*_{\tau,E,Ax} t''$ *such that* $(\sigma_1 \cdots \sigma_n\sigma_{n+1} \cdots \sigma_k){\downarrow_{E,Ax}}|_{Var(t)} =_{Ax} (\theta_1 \cdots \theta_n\rho\tau\rho')|_{Var(t)}$ *and* $(t\theta){\downarrow_{E,Ax}} =_{Ax} t''\rho'\rho$*. Since* ρ *is not a renaming, the narrowing sequence* $\beta;\beta'$ *is more general than* α*. But this contradicts that* α *is a most general narrowing sequence and, thus, the conclusion follows.* □

Theorem 10 (Variant Completeness of Folding Narrowing) *Let* $\mathcal{R} = (\Sigma, Ax, E)$ *be a decomposition of an equational theory* $(\Sigma, \mathcal{E})$*. Let* t_1 *be a* Σ*-term and* θ *be an* E*,*Ax*-normalized substitution. Let* $\mathcal{S}_\mathcal{R}$ *be a complete narrowing strategy. If* $t_1\theta \rightarrow^!_{E,Ax} t_2$ *then there exist a term* t_2' *and* *two E,Ax-normalized substitutions σ and ρ s.t. $(t_1 \rightsquigarrow^*_{\sigma,E,Ax} t_2') \in \mathcal{S}^{\circlearrowleft}_{\mathcal{R}}(t_1)$, $\theta|_{Var(t_1)} =_{Ax} (\sigma\rho)|_{Var(t_1)}$, and $t_2 =_{Ax} t_2'\rho$.*

Proof. *Given $t_1\theta \rightarrow^!_{E,Ax} t_2$, by completeness of narrowing (Theorem 9), there exist a term t_2' and two E,Ax-normalized substitutions σ and ρ such that $(\alpha : t_1 \rightsquigarrow^*_{\sigma,E,Ax} t_2') \in \mathcal{S}_{\mathcal{R}}(t_1)$, $\theta|_{Var(t_1)} =_{Ax} (\sigma\rho)|_{Var(t_1)}$, and $t_2 =_{Ax} t_2'\rho$. Let us assume that α is most general, since there is always at least one most general narrowing sequence. Then, by Lemma 9, there exists $(\beta : t_1 \rightsquigarrow^*_{\phi,E,Ax} u) \in \mathcal{S}_{\mathcal{R}}(t_1)$ such that $\alpha \approx_{Ax} \beta$ and the conclusion follows.* $\square$

We can effectively compute a complete set of variants by folding narrowing in the following way.

Corollary 2 (Computing the Variants) *Let $\mathcal{R} = (\Sigma, Ax, E)$ be a decomposition of an equational theory $(\Sigma, \mathcal{E})$. Let t be a Σ-term. Let $\mathcal{S}_{\mathcal{R}}$ be a complete narrowing strategy. If $(t', \sigma) \in [\![t]\!]_{E,Ax}$, then there are t'', σ', and ρ such that $(t \rightsquigarrow^*_{\sigma',E,Ax} t'') \in \mathcal{S}^{\circlearrowleft}_{\mathcal{R}}(t)$, t'' is $\rightarrow_{E,Ax}$-irreducible, σ' is $\rightarrow_{E,Ax}$-normalized, ρ is a renaming, $t' =_{Ax} t''\rho$, and $\sigma|_{Var(t)} =_{Ax} (\sigma'\rho)|_{Var(t)}$.*

We can conclude that the folding full-narrowing strategy is a variant-complete narrowing strategy.

Corollary 3 *Let $\mathcal{R} = (\Sigma, Ax, E)$ be a decomposition of an equational theory $(\Sigma, \mathcal{E})$. The folding full-narrowing strategy $Full^{\circlearrowleft}_{\mathcal{R}}$ is* variant-complete*, i.e., for each Σ-term t, $[\![t]\!]_{E,Ax} \simeq_{E,Ax} [\![t]\!]^{Full^{\circlearrowleft}_{\mathcal{R}}}_{E,Ax}$.*

Note that folding full-narrowing is not variant-minimal (and thus not optimally variant-terminating).

Example 10 *Consider the following decomposition without axioms*

$$f(s(X)) = g(X) \qquad g(s(X)) = 0 \qquad f(s(s(0))) = 0.$$

For term $f(X)$, we have that $\{(f(X), id), (g(X'), \{X \mapsto s(X')\}), (0, \{X \mapsto s(s(X''))\})\}$ is the set of most general variants. However, folding full-narrowing will generate those three variants plus $(0, \{X \mapsto s(s(0))\})$, which is subsumed by variant $(0, \{X \mapsto s(s(X''))\})\}$:

1. *The variant $(f(X), id)$ without any narrowing step.*

2. *Variants with one narrowing step:* $(g(X'), \{X \mapsto s(X')\})$ *and* $(0, \{X \mapsto s(s(0))\})$*, i.e.,* $(f(X) \leadsto_{\{X \mapsto s(X')\},E,Ax} g(X')) \in Full^{\circlearrowleft}_{\mathcal{R}}$ *and* $(f(X) \leadsto_{\{X \mapsto s(s(0))\},E,Ax} 0) \in Full^{\circlearrowleft}_{\mathcal{R}}$.

3. *The variant* $(0, \{X \mapsto s(s(X''))\})$ *with two narrowing steps:*

$$(f(X) \leadsto_{\{X \mapsto s(X')\},E,Ax} g(X') \leadsto_{\{X' \mapsto s(X'')\},E,Ax} 0) \in Full^{\circlearrowleft}_{\mathcal{R}}$$

In the next section, we refine the folding narrowing strategies and improve over the folding full-narrowing strategy for computing variants.

5.4.1 Variant Narrowing Strategy

We have shown that the folding full-narrowing strategy $Full^{\circlearrowleft}_{\mathcal{R}}$ is variant-complete. However, there is another interesting aspect about narrowing strategies:

> *Are there strategies more effective than full-narrowing which can be extended to folding narrowing in order to compute variants?*

We answered this question in the positive in our paper [54] with the notion of *variant narrowing strategy*, but we improve the presentation here.

Let us first motivate with two ideas why a narrowing strategy which is an alternative to full narrowing can be very useful for a decomposition. First, the completeness of a narrowing strategy w.r.t. a decomposition is restricted to normalized substitutions. Therefore, we are interested in narrowing strategies that provide only narrowing sequences with normalized substitutions. Basic narrowing was an attempt at this but, as we show in Example 6, it is incomplete for the modulo case as well as (possibly) non-terminating for computing variants, as shown in Example 7. Here we present a narrowing strategy that computes *only* normalized substitutions without losing completeness. Second, applying narrowing $\leadsto_{E,Ax}$ to perform $(E \cup Ax)$-unification without any restriction, as done in $Full_{\mathcal{R}}$, is very wasteful, because as soon as a rewrite step $\rightarrow_{E,Ax}$ is enabled in a term that has also narrowing steps $\leadsto_{E,Ax}$, such a rewrite step should always be taken before any further narrowing steps are applied, thanks to confluence and coherence modulo Ax. This idea is consistent with the implementation of rewriting logic [124] and, therefore, the relation $\rightarrow^{!}_{E,Ax}; \rightsquigarrow_{E,Ax}$ makes sense as an optimization of $\rightsquigarrow_{E,Ax}$ (see [70] for discussion about this idea in a context without axioms). However, this is still a naive approach, since a rewrite step and a narrowing step satisfy a more general property, which is the reason for being able to take the rewrite step and avoiding the narrowing step. Namely, for a decomposition $\mathcal{R} = (\Sigma, Ax, E)$, if two narrowing steps $t \rightsquigarrow_{\sigma_1,E,Ax} t_1$ and $t \rightsquigarrow_{\sigma_2,E,Ax} t_2$ are possible and we have that $\sigma_1 \sqsubseteq_{Ax} \sigma_2$ (i.e., σ_2 is more general than σ_1), then it is enough to take only the narrowing step using σ_2. These improvements are formalized as follows. First, we introduce a partial order between narrowing steps, defining when a narrowing step is more general than another narrowing step.

Definition 21 (Preorder and equivalence of narrowing steps) *Let $\mathcal{R} = (\Sigma, Ax, E)$ be a decomposition of $(\Sigma, \mathcal{E})$. Let us consider two narrowing steps $\alpha_1 : t \rightsquigarrow_{\sigma_1,E,Ax} s_1$ and $\alpha_2 : t \rightsquigarrow_{\sigma_2,E,Ax} s_2$. We write $\alpha_1 \preceq_{Ax} \alpha_2$ if $\sigma_1|_{Var(t)} \sqsubseteq_{Ax} \sigma_2|_{Var(t)}$ and $\alpha_1 \prec_{Ax} \alpha_2$ if $\sigma_1|_{Var(t)} \sqsubset_{Ax} \sigma_2|_{Var(t)}$ (i.e., σ_2 is strictly more general than σ_1). We write $\alpha_1 \simeq_{Ax} \alpha_2$ if $\sigma_1|_{Var(t)} \simeq_{Ax} \sigma_2|_{Var(t)}$. The relation $\alpha_1 \simeq_{Ax} \alpha_2$ between two narrowing steps from t defines a set of equivalence classes between such narrowing steps. In what follows we will be interested in choosing a unique representative $\underline{\alpha} \in [\alpha]_{\simeq_{Ax}}$ in each equivalence class of narrowing steps from t. Therefore, $\underline{\alpha}$ will always denote a chosen unique representative $\underline{\alpha} \in [\alpha]_{\simeq_{Ax}}$.*

The relation $\preceq_{Ax}$ provides an improvement on narrowing executions, since narrowing steps with more general computed substitutions will always be selected instead of narrowing steps with more instantiated computed substitutions. Also, this relation ensures that, when both a rewriting step and a narrowing step are available, the rewriting step will always be chosen. Finally, the relation $\simeq_{Ax}$ provides another improvement, since only one narrowing (or rewriting) step is chosen in each equivalence class, reducing the width of the narrowing tree even more. The very last improvement is to restrict to normalized computed substitutions, as motivated at the beginning of this section.

Definition 22 (Variant Narrowing) *Let $\mathcal{R} = (\Sigma, Ax, E)$ be a decomposition of $(\Sigma, \mathcal{E})$. Given a Σ-term t, we define the variant narrowing strategy $VN_{\mathcal{R}}(t) = \{t \rightsquigarrow^{*}_{\sigma,\underline{E},Ax} s\}$, where: (i) $\sigma|_{Var(t)}$ is E, Ax-normalized and (ii) each* *narrowing step $u \rightsquigarrow_{\rho,\underline{E},Ax} v$ is defined as the narrowing step $\underline{\alpha} : u \rightsquigarrow_{\rho,E,Ax} v$ such that $\underline{\alpha}$ is maximal w.r.t. the order $\preceq_{Ax}$, and $\underline{\alpha}$ is the chosen unique representative of its $\simeq_{Ax}$-equivalence class.*

Example 11 *Consider Example 3. For the term $t = X \oplus Y \oplus X \oplus Y$, there are nearly 150 full narrowing steps, since subterm $X \oplus Y$ had 124 narrowing steps as explained in Example 5 and there are even more combinations. However, variant narrowing recognizes that this term is not yet normalized, i.e., $X \oplus Y \oplus X \oplus Y \to 0$, and such a rewriting step is more general than any narrowing step. Thus, variant narrowing performs only a rewriting step and avoids such an exceptionally large number of narrowing steps. Note that there are two other rewrite steps $X \oplus Y \oplus X \oplus Y \to Y \oplus Y$ and $X \oplus Y \oplus X \oplus Y \to X \oplus X$ and variant narrowing will choose one of these three as the unique representative of the $\simeq_{AC}$-equivalence class of rewrite steps.*

We denote the extended folding version of variant narrowing, i.e., *folding variant narrowing*, by $VN^{\circlearrowleft}_{\mathcal{R}}$. The condition in Definition 22 that $\sigma|_{Var(t)}$ is E, Ax-normalized (in contrast to σ being E, Ax-normalized) is essential for a correct behavior of the strategy, as shown below.

Example 12 *Consider the following decomposition $(\Sigma, \emptyset, E)$ where E contains $f(a, b, X) \to f(a, b)$, symbol f is AC, and X is a variable. Consider the term $t = f(a, a, a, b, b, b)$, whose normal form is $f(a, b)$, i.e., $f(a, a, a, b, b, b) \to_{E,Ax} f(a, b)$. Any rewriting sequence leading to its normal form does not consider a normalized substitution, i.e., the first rewriting step of any rewriting sequence will use substitution $\{X \mapsto f(a, a, b, b)\}$. Therefore, we cannot restrict ourselves to normalized substitution w.r.t. rewriting steps.*

On the other hand, consider now the term $s = f(Y_1, Y_2)$ and the narrowing step $f(Y_1, Y_2) \rightsquigarrow_{\rho_2,E,Ax} f(a, b)$ with $\rho_2 = \{Y_1 \mapsto f(a, b, Y_3), X \mapsto f(Y_2, Y_3)\}$. The unifier ρ_2 is not normalized, since $f(a, b, Y_3){\downarrow_{E,Ax}} = f(a, b)$. Note that we cannot normalize the substitution, since it would not correspond to any narrowing step and we simply discard this narrowing step because there is another more general narrowing step (i.e., $(f(a, b), \rho_2{\downarrow_{E,Ax}}) \sqsubseteq_{Ax} (f(a, b), \rho_1)$). Note that the ability to discard narrowing steps in confluent, terminating, and coherent systems whose computed substitution is not normalized is a key *point for achieving termination for variant generation. The set of most general unifiers computed by all the narrowing steps is as follows:*

$$
\begin{aligned}
\rho_1 &= \{Y_1 \mapsto f(a,b), X \mapsto Y_2\}\\
\rho_2 &= \{Y_1 \mapsto f(a,b,Y_3), X \mapsto f(Y_2,Y_3)\}\\
\rho_3 &= \{Y_2 \mapsto f(a,b), X \mapsto Y_1\}\\
\rho_4 &= \{Y_2 \mapsto f(a,b,Y_3), X \mapsto f(Y_1,Y_3)\}\\
\rho_5 &= \{Y_1 \mapsto a, Y_2 \mapsto b\}\\
\rho_6 &= \{Y_1 \mapsto a, Y_2 \mapsto f(b,Y_3), X \mapsto Y_3\}\\
\rho_7 &= \{Y_1 \mapsto b, Y_2 \mapsto a\}\\
\rho_8 &= \{Y_1 \mapsto b, Y_2 \mapsto f(a,Y_3), X \mapsto Y_3\}\\
\rho_9 &= \{Y_1 \mapsto f(a,Y_3), Y_2 \mapsto b, X \mapsto Y_3\}\\
\rho_{10} &= \{Y_1 \mapsto f(a,Y_3), Y_2 \mapsto f(b,Y_4), X \mapsto f(Y_3,Y_4)\}\\
\rho_{11} &= \{Y_1 \mapsto f(b,Y_3), Y_2 \mapsto a, X \mapsto Y_3\}\\
\rho_{12} &= \{Y_1 \mapsto f(b,Y_3), Y_2 \mapsto f(a,Y_4), X \mapsto f(Y_3,Y_4)\}
\end{aligned}
$$

Note that the relation $\rightarrow^!_{E,Ax}; \rightsquigarrow_{E,Ax}$ is (appropriately) simulated by $\rightsquigarrow^+_{\underline{E},Ax}$, since in the relation $\rightsquigarrow^+_{\underline{E},Ax}$ rewriting steps are always given priority over narrowing steps.

Lemma 10 (Normalization of Variant Narrowing) *Let $\mathcal{R} = (\Sigma, Ax, E)$ be a decomposition of $(\Sigma, \mathcal{E})$. Let t be a Σ-term. If t is not E, Ax-irreducible, then, relative to the unique choice of $\underline{\alpha} \in [\alpha]_{\simeq_{Ax}}$ in Definition 21 , there is a unique $\rightsquigarrow_{\underline{E},Ax}$-narrowing sequence from t performing only rewriting steps.*

Proof. *Immediate, since t is not E, Ax-irreducible and the theory is confluent and sort-decreasing.* □

The following result ensures that variant narrowing is complete.

Theorem 11 (Completeness of Variant Narrowing) *Let $\mathcal{R} = (\Sigma, Ax, E)$ be a decomposition of $(\Sigma, \mathcal{E})$. If $\alpha : t \rightsquigarrow^*_{\sigma,E,Ax} (t\sigma){\downarrow_{E,Ax}}$ such that $\sigma|_{Var(t)}$ is E, Ax-normalized and α is a most general narrowing sequence, then there exists σ' such that $t \rightsquigarrow^*_{\sigma',\underline{E},Ax} (t\sigma'){\downarrow_{E,Ax}}$, and $\sigma|_{Var(t)} \approx_{Ax} \sigma'|_{Var(t)}$.*

Proof. *If $\alpha : t \rightsquigarrow^*_{\sigma,E,Ax} (t\sigma){\downarrow_{E,Ax}}$ such that $\sigma|_{Var(t)}$ is E, Ax-normalized and α is a most general narrowing sequence, then it is sufficient to show that the computed substitution at each step in α is maximal w.r.t. $\sqsubseteq_{Ax}$.*

We prove this by contradiction. Let us consider a narrowing step $i \in \{1, \ldots, n\}$ in α, i.e. $t_i \rightsquigarrow_{\sigma_i,E,Ax} t_{i+1}$, such that σ_i is not maximal w.r.t. $\sqsubseteq_{Ax}$. That is, there is an alternative narrowing step from t_i, i.e., $t_i \rightsquigarrow_{\tau,E,Ax} w$, with a strictly more general substitution τ, i.e., there is a substitution τ' s.t. $\sigma_i|_{Var(t_i)} =_{Ax} (\tau\tau')|_{Var(t_i)}$ and τ' is not a renaming. Note that, since α is most general, there is no narrowing sequence $w \rightsquigarrow^*_{\phi,E,Ax} t_n$ and substitution ϕ' such that $\sigma|_{Var(t)} =_{Ax} (\sigma_1 \cdots \sigma_{i-1}\tau\phi\phi')|_{Var(t)}$. Then, we have that $t_i\sigma_i \rightarrow_{E,Ax} t_{i+1}$ and that there is a term w' such that $t_i\sigma_i \rightarrow_{E,Ax} w'$ and $w' =_{Ax} w\tau'$. By confluence, there is a term u such that $t_{i+1} \rightarrow^*_{E,Ax} u$ and $w' \rightarrow^*_{E,Ax} u$. But then, for any narrowing sequence $u \rightsquigarrow^*_{\mu,E,Ax} u'$ such that $\mu|_{Var(t_{i+1})} =_{Ax} (\sigma_{i+1} \cdots \sigma_n)|_{Var(t_{i+1})}$, there is a whole narrowing sequence $t \rightsquigarrow^*_{\sigma',E,Ax} (t\sigma')\downarrow_{E,Ax}$ such that $\sigma'|_{Var(t)} = (\sigma_1 \cdots \sigma_{i-1}\tau\mu)|_{Var(t)}$. This implies that $\sigma \sqsubset_{Ax} \sigma'$, since $(\sigma_i \cdots \sigma_n)|_{Var(t_i)} =_{Ax} (\tau\mu\tau')|_{Var(t_i)}$. Therefore, we have a contradiction because σ' is strictly more general than σ. $\square$

Note that the previous theorem is only valid when E is confluent[2] modulo Ax, and not just *ground confluent* [119] modulo Ax, as shown by the following example.

Example 13 *Let us consider the following rewrite theory without axioms, which is terminating and ground confluent but not confluent:*

$$f(X) = 0 \qquad f(X) = g(X) \qquad g(0) = 0 \qquad g(s(X)) = g(X)$$

If we consider the term $f(X)$ and the narrowing step taking the first equation, then we compute the most general substitution, i.e. $f(X) \rightsquigarrow_{id,E,Ax} 0$. However, if we consider $f(X)$ and the narrowing step that takes the second equation, i.e., $f(X) \rightsquigarrow_{id,E,Ax} g(X)$, we will compute an infinite number of substitutions, i.e., $\forall n \geq 0 : g(X) \rightsquigarrow^*_{\{X \mapsto s^n(0)\},E,Ax} 0$, and none of them is more general than the identity substitution computed with the first equation.

The following interesting result holds for folding variant narrowing, but not for folding full-narrowing.

Theorem 12 (Minimality of Folding Variant Narrowing) *Let $\mathcal{R} = (\Sigma, Ax, E)$ be a decomposition of $(\Sigma, \mathcal{E})$. If $\alpha : t \rightsquigarrow^*_{\sigma,E,Ax} (t\sigma)\downarrow_{E,Ax}$ with*

[2]Note that a decomposition already requires confluence instead of ground confluence.

$\sigma|_{Var(t)}$ being E, Ax-normalized and $\alpha' : t \leadsto^*_{\sigma',E,Ax} (t\sigma')\downarrow_{E,Ax}$ with $\sigma'|_{Var(t)}$ being E, Ax-normalized such that $\sigma|_{Var(t)} \sqsubset_{Ax} \sigma'|_{Var(t)}$, and α' is a most general narrowing sequence, then there is a narrowing sequence $\beta : t \leadsto^*_{\theta,E,Ax} (t\theta)\downarrow_{E,Ax}$ in $VN^{\circlearrowleft}_{\mathcal{R}}$ such that $\alpha' \approx_{Ax} \beta$ but there is no narrowing sequence $\beta' : t \leadsto^*_{\theta',E,Ax} (t\theta')\downarrow_{E,Ax}$ in $VN^{\circlearrowleft}_{\mathcal{R}}$ such that $\alpha \approx_{Ax} \beta'$.

Proof. *The first statement is proved by the most generality of α' and Theorem 11, i.e., there is $\beta : t \leadsto^*_{\theta,E,Ax} (t\theta)\downarrow_{E,Ax}$ in $VN^{\circlearrowleft}_{\mathcal{R}}$ such that $\alpha' \approx_{Ax} \beta$. The second statement is proved by contradiction, i.e., we asume that there is $\beta' : t \leadsto^*_{\theta',E,Ax} (t\theta')\downarrow_{E,Ax}$ in $VN^{\circlearrowleft}_{\mathcal{R}}$ such that $\alpha \approx_{Ax} \beta'$. For simplicity, we assume that $\alpha' \in VN^{\circlearrowleft}_{\mathcal{R}}$ and use α' instead of β in the rest of the proof. Let α and α' be as follows:*

$$\alpha' : t \leadsto_{\sigma'_1,E,Ax} t'_1 \leadsto_{\sigma'_2,E,Ax} t'_2 \cdots t'_{m-1} \leadsto_{\sigma'_m,E,Ax} t'_m = (t\sigma')\downarrow_{E,Ax}$$

and

$$\alpha : t \leadsto_{\sigma_1,E,Ax} t_1 \leadsto_{\sigma_2,E,Ax} t_2 \cdots t_{n-1} \leadsto_{\sigma_n,E,Ax} t_n = (t\sigma)\downarrow_{E,Ax}$$

Let us consider the first narrowing step $i \in \{1, \ldots, n\}$ in α, i.e. $t_{i-1} \leadsto_{\sigma_i,E,Ax} t_i$, where there is a substitution τ such that $\sigma_i|_{Var(t_i)} =_{Ax} (\sigma'_i\tau)|_{Var(t_i)}$ and τ is not a renaming. Since $(\sigma_1 \cdots \sigma_{i-1})|_{Var(t)} \approx_{Ax} (\sigma'_1 \cdots \sigma'_{i-1})|_{Var(t)}$, by coherence and confluence, there are two terms w and w' such that $t\sigma_1 \cdots \sigma_{i-1} \rightarrow^*_{E,Ax} w$, $t\sigma'_1 \cdots \sigma'_{i-1} \rightarrow^*_{E,Ax} w'$, and $w \approx_{Ax} w'$. Let ρ be such that $(\sigma_1 \cdots \sigma_{i-1})|_{Var(t)} =_{Ax} (\sigma'_1 \cdots \sigma'_{i-1}\rho)|_{Var(t)}$ and $w =_{Ax} w'\rho$. We can add substitution σ'_i to have rewrite sequences $t\sigma_1 \cdots \sigma_{i-1}\sigma'_i \rightarrow^*_{E,Ax} w\sigma'_i$ and $t\sigma'_1 \cdots \sigma'_{i-1}\rho\sigma'_i \rightarrow^*_{E,Ax} w\sigma'_i$. By completeness of narrowing, there exist substitutions ϕ and ϕ' and a most general narrowing sequence $\alpha'' : t_{i-1} \leadsto^*_{\phi,E,Ax} u$ such that $\sigma'_i|_{Var(t_{i-1})} =_{Ax} (\phi\phi')|_{Var(t_{i-1})}$, and $w\sigma'_i =_{Ax} u\phi'$. But then there are two narrowing steps from term t_{i-1}, $t_{i-1} \leadsto_{\sigma_i,E,Ax} t_i$ and the first step of α'' s.t. the first step of α'' has a substitution more general than σ_i. But the $VN_{\mathcal{R}}$ strategy would have chosen the first step of α'' instead of the narrowing step $t_{i-1} \leadsto_{\sigma_i,E,Ax} t_i$ and this contradicts that there is $\beta' : t \leadsto^*_{\theta',E,Ax} (t\theta')\downarrow_{E,Ax}$ in $VN^{\circlearrowleft}_{\mathcal{R}}$ such that $\alpha \approx_{Ax} \beta'$. □

Now, we know that $VN^{\circlearrowleft}_{\mathcal{R}}$ is an efficient variant-complete and variant-minimal strategy, so we can use it to effectively compute variants.

Corollary 4 *Let $\mathcal{R} = (\Sigma, Ax, E)$ be a decomposition of an equational theory $(\Sigma, \mathcal{E})$. The folding variant narrowing strategy $VN^{\circlearrowleft}_{\mathcal{R}}$ is variant-complete and variant-minimal, i.e., for any Σ-term t, $[\![t]\!]_{E,Ax} \approx_{Ax} [\![t]\!]^{VN^{\circlearrowleft}_{\mathcal{R}}}_{E,Ax}$.*

Finally, we return to our running example for the $VN^{\circlearrowleft}_{\mathcal{R}}$ strategy.

Example 14 *Consider Example 9. For $t = X \oplus Y$ we get the following $VN^{\circlearrowleft}_{\mathcal{R}}$ steps with normalized substitutions:*

(i) $(X \oplus Y, id) \rightsquigarrow_{\phi_1} (Z, \phi_1)$, *using Equation* (5.6) *and substitution* $\phi_1 = \{X \mapsto 0, Y \mapsto Z\}$,

(ii) $(X \oplus Y, id) \rightsquigarrow_{\phi_2} (Z, \phi_2)$, *using Equation* (5.6) *and substitution* $\phi_2 = \{X \mapsto Z, Y \mapsto 0\}$,

(iii) $(X \oplus Y, id) \rightsquigarrow_{\phi_3} (Z, \phi_3)$, *using Equation* (5.8) *and substitution* $\phi_3 = \{X \mapsto Z \oplus U, Y \mapsto U\}$,

(iv) $(X \oplus Y, id) \rightsquigarrow_{\phi_4} (Z, \phi_4)$, *using Equation* (5.8) *and substitution* $\phi_4 = \{X \mapsto U, Y \mapsto Z \oplus U\}$,

(v) $(X \oplus Y, id) \rightsquigarrow_{\phi_5} (0, \phi_5)$, *using Equation* (5.7) *and substitution* $\phi_5 = \{X \mapsto U, Y \mapsto U\}$,

(vi) $(X \oplus Y, id) \rightsquigarrow_{\phi_6} (Z_1 \oplus Z_2, \phi_6)$, *using Equation* (5.8) *and* $\phi_6 = \{X \mapsto U \oplus Z_1, Y \mapsto U \oplus Z_2\}$.

Note that $VN^{\circlearrowleft}_{\mathcal{R}}$ terminates (as $Full^{\circlearrowleft}_{\mathcal{R}}$ does) after generating all these narrowing steps.

In the following, we study under which conditions the folding variant narrowing strategy is optimally variant-terminating, providing the best narrowing strategy for computing variants in the modulo case but also in the free theory, improving beyond basic narrowing.

5.5 The Finite Variant Property

An interesting case is when we know a priori that any Σ-term has a finite number of most general variants.

Definition 23 (Finite variant property) [32] *Let $\mathcal{R} = (\Sigma, Ax, E)$ be a decomposition of an equational theory $(\Sigma, \mathcal{E})$. Then $(\Sigma, \mathcal{E})$, and thus $\mathcal{R}$, has the* finite variant property (FV) *iff for each Σ-term t, the set $[\![t]\!]_{E,Ax}$ is finite. We call $\mathcal{R}$ a* finite variant decomposition *of $(\Sigma, \mathcal{E})$ iff $\mathcal{R}$ has the finite variant property.*

The following corollary is immediate for finite variant decompositions.

Corollary 5 *Let $\mathcal{R} = (\Sigma, Ax, E)$ be a decomposition of an equational theory $(\Sigma, \mathcal{E})$ and $\mathcal{S}_{\mathcal{R}}$ be an E, Ax-variant-complete narrowing strategy. $\mathcal{S}_{\mathcal{R}}^{\circlearrowleft}$ is variant-terminating iff $\mathcal{R}$ is a finite variant decomposition of $(\Sigma, \mathcal{E})$.*

Proof. *Given a Σ-term t, for each $(t', \sigma) \in [\![t]\!]_{E,Ax}$, by Corollary 2, there are t'', σ', and ρ such that $(t \rightsquigarrow^{*}_{\sigma',E,Ax} t'') \in \mathcal{S}_{\mathcal{R}}^{\circlearrowleft}(t)$, t'' is $\rightarrow_{E,Ax}$-irreducible, $\sigma'|_{Var(t)}$ is $\rightarrow_{E,Ax}$-normalized, ρ is a renaming, $t' =_{Ax} t''\rho$, and $\sigma|_{Var(t)} =_{Ax} (\sigma'\rho)|_{Var(t)}$. Since $[\![t]\!]_{E,Ax}$ is finite and it contains the most general variants w.r.t. $\sqsubseteq_{E,Ax}$, for each possible variant $(u, \phi) \in [\![t]\!]^{\star}_{E,Ax}$, there is a node (u', ϕ') in the narrowing tree such that $(u, \phi) \sqsubseteq_{E,Ax} (u', \phi')$ and, thus, the narrowing tree generated by $\mathcal{S}_{\mathcal{R}}^{\circlearrowleft}(t)$ has a bounded depth.* $\square$

The folding variant narrowing $VN_{\mathcal{R}}^{\circlearrowleft}$ is variant-minimal and the following corollary holds for finite variant decompositions.

Corollary 6 *If $\mathcal{R} = (\Sigma, Ax, E)$ is a finite variant decomposition of $(\Sigma, \mathcal{E})$, then $VN_{\mathcal{R}}^{\circlearrowleft}$ is optimally variant-terminating.*

Proof. *By Corollary 4, $VN_{\mathcal{R}}^{\circlearrowleft}$ is variant-minimal and, thus, the narrowing tree generated by $VN_{\mathcal{R}}^{\circlearrowleft}$ contains all and only all the variants of the set $[\![t]\!]_{E,Ax}$ for a given Σ-term t. Therefore, the narrowing tree is always the shortest tree possible for generating the set of most general variants $[\![t]\!]_{E,Ax}$ and we conclude that $VN_{\mathcal{R}}^{\circlearrowleft}$ is optimally variant-terminating.* $\square$

5.5.1 Computing Variants for Theories with the Finite Variant Property

Comon and Delaune characterize the finite variant property in terms of the following boundedness property, which is *equivalent* to FV.

Lemma 11 [32] *Let $\mathcal{R} = (\Sigma, Ax, E)$ be a decomposition of an equational theory $(\Sigma, \mathcal{E})$. $\mathcal{R}$ has the finite variant property if and only if for every term t, there is a finite set $\Theta(t)$ of substitutions such that*

$$\forall\sigma, \exists\theta \in \Theta(t), \exists\tau : (\sigma{\downarrow_{E,Ax}})|_{Var(t)} =_{Ax} (\theta\tau)|_{Var(t)} \wedge (t\sigma){\downarrow_{E,Ax}} =_{Ax} ((t\theta){\downarrow_{E,Ax}})\tau$$

Definition 24 (Boundedness property) [32] *Let $\mathcal{R} = (\Sigma, Ax, E)$ be a decomposition of an equational theory $(\Sigma, \mathcal{E})$. $\mathcal{R}$ has the* boundedness property *(BP) iff for every term t there exists an integer n, denoted by $\#_{E,Ax}(t)$, such that for every E, Ax-normalized substitution σ the normal form of $t\sigma$ is reachable by an E, Ax-rewriting sequence whose length can be bounded by n (thus independently of σ), i.e.,*

$$\forall t, \exists n, \forall\sigma, t(\sigma{\downarrow_{E,Ax}}) \xrightarrow{\leq n}_{E,Ax} (t\sigma){\downarrow_{E,Ax}}.$$

Lemma 11 and Definition 24 allow the following result.

Theorem 13 [32] *Let $\mathcal{R} = (\Sigma, Ax, E)$ be a decomposition of an equational theory $(\Sigma, \mathcal{E})$. Then, $\mathcal{R}$ satisfies the boundedness property if and only if $\mathcal{R}$ is a finite variant decomposition of $(\Sigma, \mathcal{E})$.*

Obviously, if for a term t, the minimal length of a rewrite sequence to the canonical form of an instance $t\sigma$, with σ normalized, cannot be bounded, the theory does not have the finite variant property. It is easy to see that for the addition equations

$$0 + Y = Y \qquad\qquad s(X) + Y = s(X + Y)$$

the term $t = X + Y$, and the family of substitutions $\sigma_n = \{X \mapsto s^n(0)\}$, $n \in \mathbb{N}$, this is the case, and therefore, since $FV \Leftrightarrow BP$, the addition theory lacks the finite variant property.

Example 15 *Consider again Example 7 consisting of the rewrite theory $\mathcal{R} = (\Sigma, \emptyset, E)$ where E is the set of confluent and terminating rules $E = \{f(x) \to x, f(f(x)) \to f(x)\}$ and Σ contains only the unary symbol f and a constant a. The theory has the finite variant property as it does have the boundedness property, since for any term t and a normalized substitution θ, a bound for t is given by the number of f symbols in the term.*

Proposition 2 (Computing the Finite Variants) [54] *Let $\mathcal{R} = (\Sigma, Ax, E)$ be a finite variant decomposition of an order-sorted equational theory $(\Sigma, \mathcal{E})$. Let t be a Σ-term and $\#_{E,Ax}(t) = n$. Then, $(s, \sigma) \in [\![t]\!]_{E,Ax}$ if and only if there is a narrowing sequence $t \rightsquigarrow^{\leq n}_{\sigma,E,Ax} s$ such that s is $\rightarrow_{E,Ax}$-irreducible and σ is $\rightarrow_{E,Ax}$-normalized.*

Example 16 *Consider again Example 3. For this theory, narrowing clearly does not terminate because $Z_1 \oplus Z_2 \rightsquigarrow_{\{Z_1 \mapsto X_1 \oplus Z_1',\ Z_2 \mapsto X_1 \oplus Z_2'\},E,Ax} Z_1' \oplus Z_2'$ and this can be repeated infinitely often. This equational theory has the boundedness property, as it is shown to have FV in Example 26 below. A bound for this theory is the number of $\oplus$ symbols in the term, so that the narrowing tree can be restricted to depth 1 for the term $t = Z_1 \oplus Z_2$. Let us explain in detail why the bound is the number of $\oplus$ symbols. Given the narrowing sequence*

$$Z_1 \oplus Z_2 \rightsquigarrow_{\{Z_1 \mapsto X_1 \oplus Z_1', Z_2 \mapsto X_1 \oplus Z_2'\},E,Ax} Z_1' \oplus Z_2' \rightsquigarrow_{\{Z_1' \mapsto X_1' \oplus Z_1'', Z_2' \mapsto X_1' \oplus Z_2''\},E,Ax} Z_1'' \oplus Z_2'' \tag{5.11}$$

we have the variant $(Z_1'' \oplus Z_2'', \rho)$ with $\rho = \{Z_1 \mapsto X_1 \oplus X_1' \oplus Z_1'', Z_2 \mapsto X_1 \oplus X_1' \oplus Z_2'', Z_1' \mapsto X_1' \oplus Z_1'', Z_2' \mapsto X_1' \oplus Z_2''\}$. Also, the normalization sequence corresponding to $t\rho$ that mimics the narrowing sequence (5.11) *is*

$$X_1 \oplus X_1' \oplus Z_1'' \oplus X_1 \oplus X_1' \oplus Z_2'' \rightarrow_{E,Ax} X_1' \oplus Z_1'' \oplus X_1' \oplus Z_2'' \rightarrow_{E,Ax} Z_1'' \oplus Z_2'' \tag{5.12}$$

However, we can also reduce $t\rho$ to the same normal form of (5.12) *using only one application of* (5.8) *and the following normalized substitution $\rho = \{X \mapsto X_1 \oplus X_1', Y \mapsto Z_1'' \oplus Z_2''\}$:*

$$X_1 \oplus X_1' \oplus Z_1'' \oplus X_1 \oplus X_1' \oplus Z_2'' \rightarrow_{E,Ax} Z_1'' \oplus Z_2'' \tag{5.13}$$

The trick is that rule (5.8) *allows combining all pairs of canceling terms and thus gets rid of all of them at once. That is why the theory has the finite variant property.*

At this point, we have three different ways of computing variants that we would like to discuss with some examples:

1. Computing the narrowing tree associated to a term t up to the bound $\#_{E,Ax}(t)$ and extracting the variants from the narrowing tree.

2. Computing the narrowing tree using $Full^{\circlearrowleft}_{\mathcal{R}}$ and extracting the variants from the narrowing tree.

3. Computing the narrowing tree using $VN^{\circlearrowleft}_{\mathcal{R}}$ and extracting the variants from the narrowing tree.

$VN^{\circlearrowleft}_{\mathcal{R}}$ is the best approach, since the other two approaches are cruder and can be massively inefficient. This can be illustrated as follows.

Example 17 *Consider again Example 3 and the term* $u = X \oplus Y \oplus X \oplus Y$*, whose most general variant is* $(0, id)$*. As explained in Example 11, this term can be normalized in one rewriting step. However, the approaches* (1)–(3) *work very differently.*

1. *Since we showed that the narrowing bound is the number of* $\oplus$ *symbols, we have* $\#_{E,Ax}(u) = 3$*. The full narrowing tree up to bound* 3 *is huge and we do not include it here (see Examples 5, 9, and 11).*

2. $Full^{\circlearrowleft}_{\mathcal{R}}$ *will behave a little better by producing only narrowing sequences of length* 1*, since it will compute the rewriting step to the term* 0 *among the* 150 *narrowing steps, but all these extra narrowing steps are unnecessary. Again, we are not including here the* $Full^{\circlearrowleft}_{\mathcal{R}}$ *narrowing tree (see Examples 5, 9, and 11).*

3. *Only* $VN^{\circlearrowleft}_{\mathcal{R}}$ *performs just one rewriting step to the normal form, being optimal in both length and number of sequences (see Example 11).*

In the following section, we study conditions for checking whether a theory has the finite variant property or not.

5.5.2 Necessary and Sufficient Conditions for FV

Deciding whether an equational theory has the finite variant property is a nontrivial task, since we have to decide whether we can stop generating normalized substitution instances by narrowing for each term. We present here an algorithm for checking whether a decomposition of an equational theory has the finite variant property (FV) which is based on two notions: (i) a new notion, called *variant-preservingness* (VP), that ensures that an intuitive bottom-up generation of variants is complete; and (ii) the property that there are no infinite sequences when we restrict ourselves to such intuitive bottom-up generation of variants (FVNS). In what follows, we show that $(VP \wedge FVNS) \Rightarrow FV$. Note that the folding variant narrowing $VN^{\circlearrowleft}_{\mathcal{R}}$ will be used for effectively computing the variants but a different narrowing strategy will be used for a bottom-up generation of variants in the procedure of detecting whether a theory has the finite variant property (FV).

Variant-preservingness (VP) ensures that we can perform an intuitive bottom-up generation of variants. The following notion is useful for the definition of VP.

Definition 25 (Variant-pattern) *Let $\mathcal{R} = (\Sigma, Ax, E)$ be a decomposition of $(\Sigma, \mathcal{E})$. We call a term $f(t_1, \ldots, t_n)$ a* variant-pattern *if all subterms $t_1, \ldots, t_n$ are $\rightarrow_{E,Ax}$-irreducible. We say that a term t* has *a variant-pattern if there is a variant-pattern t' s.t. $t' =_{Ax} t$.*

It is worth pointing out that whether a term has a variant-pattern is decidable, assuming a finitary and complete Ax-matching procedure: given a term t, t has a variant-pattern t' iff there is a symbol $f \in \Sigma$ with arity k and variables $X_1, \ldots, X_k$ of the appropriate top sorts and there is a substitution θ such that $t =_{Ax} f(X_1, \ldots, X_k)\theta$ and θ is E, Ax-normalized, where $t' = f(X_1, \ldots, X_k)\theta$. We can simplify this procedure when term t is rooted by an AC symbol to say that we only have to consider the same AC symbol at the root of t, instead of every symbol. And we can simplify this procedure even more when term t is rooted by a free function symbol (i.e., such a symbol does not satisfy any axiom of Ax) to say that t has a variant-pattern if it is already a variant-pattern, i.e., every argument of the root symbol must be E, Ax-irreducible.

Variant-preservingness induces a bottom-up variant generation process; note that bottom-up variant generation is not the same as innermost narrowing.

Definition 26 (Variant-preserving) *Let $\mathcal{R} = (\Sigma, Ax, E)$ be a decomposition of $(\Sigma, \mathcal{E})$. We say that $\mathcal{R}$ is* variant-preserving *(VP) if for any variant-pattern t, either t is $\rightarrow_{E,Ax}$-irreducible or there is a $\rightarrow_{E,Ax}$ step at the top position with a $\rightarrow_{E,Ax}$-normalized substitution.*

Note that a theory can have the finite variant property even if it is not variant-preserving.

Example 18 *Consider the decomposition of Example 12. This theory is not variant-preserving, e.g., given the term $t = f(X,Y)$ and any normalized substitution $\theta \in \{X \mapsto f(a^n), Y \mapsto f(b^n, Z)\}$ for $n \geq 2$, there is no normalized reduction for $t\theta$. However, the theory does have the boundedness property, and therefore FV, since for any term rooted by f (which is the only non-constant symbol), its normal form can be obtained in at most one step.*

The following example motivates why narrowing sequences have to be restricted for a bottom-up variant generation.

Example 19 *Consider the decomposition $f(f(X)) = X$ without axioms. This theory is well-known to be non-terminating for narrowing, e.g.,*

$$c(f(X),X) \rightsquigarrow_{\{X \mapsto f(X')\},E,Ax} c(X', f(X')) \rightsquigarrow_{\{X' \mapsto f(X'')\},E,Ax} c(f(X''), X'') \cdots$$

Although the theory is non-terminating for narrowing, it is FV. When we consider all possible instances of the term $c(f(X),X)$ for normalized substitutions, we obtain the term $c(f(X),X)$ itself and the sequence $c(f(X),X) \rightsquigarrow_{\{X \mapsto f(X')\},E,Ax} c(X', f(X'))$. The theory does have the boundedness property, and therefore FV, since for any term t and a normalized substitution θ, a bound for t is the number of f symbols in the term.

Therefore, for a bottom-up generation of variants in a finite decomposition, not all the narrowing sequences are relevant, as shown in the previous example, and thus we must identify the relevant ones associated to the notion of variant pattern.

Definition 27 (Shortest Rewrite Sequence) *Given a decomposition (Σ, Ax, E), a rewrite sequence $t_0 \rightarrow_{p_1,E,Ax} t_1 \cdots \rightarrow_{p_n,E,Ax} t_n$ is called* shortest *if there is no sequence $t_0 \rightarrow^m_{E,Ax} t'_m$ such that $m < n$ and $t_n =_{Ax} t'_m$.*

Definition 28 (Variant-preserving sequences) *Let $\mathcal{R} = (\Sigma, Ax, E)$ be a decomposition of $(\Sigma, \mathcal{E})$. A rewrite sequence $\alpha : t_0 \rightarrow_{p_1,E,Ax} t_1 \cdots \rightarrow_{p_n,E,Ax} t_n$ is called* variant-preserving *if, for $i \in \{1, \ldots, n\}$, $t_{i-1}|_{p_i}$ has a variant-pattern and α is a shortest rewrite sequence. A narrowing sequence $t_0 \rightsquigarrow_{p_1,\sigma_1,E,Ax} t_1 \cdots \rightsquigarrow_{p_n,\sigma_n,E,Ax} t_n$, $\sigma = \sigma_1 \cdots \sigma_n$, is called* variant-preserving *if σ is E, Ax-normalized and $t_0\sigma \rightarrow_{p_1,E,Ax} t_1\sigma \cdots \rightarrow_{p_n,E,Ax} t_n$ is variant-preserving.*

The set of variant-preserving sequences is not computable in general. However, we provide sufficient conditions in Section 5.6. Note that we are not going to use variant-preserving narrowing sequences for computing variants but only for deciding whether a theory has the finite variant property.

Example 20 *The infinite narrowing sequence of Example 19 is not variant-preserving, since for any finite prefix of length greater than* 1 *the computed substitution is non-normalized. The only variant-preserving sequences for the term $c(f(X), X)$ are the term itself and the one-step sequence with substitution $\{X \mapsto f(X')\}$.*

Example 21 *For Example 3, the narrowing sequence*

$$Z_1 \oplus Z_2 \rightsquigarrow_{\{Z_1 \mapsto X_1 \oplus Z'_1, Z_2 \mapsto X_1 \oplus Z'_2\}, E, Ax} Z'_1 \oplus Z'_2 \rightsquigarrow_{\{Z'_1 \mapsto X'_1 \oplus Z''_1, Z'_2 \mapsto X'_1 \oplus Z''_2\}, E, Ax} Z''_1 \oplus Z''_2$$

is not a variant-preserving sequence, since the alternative rewrite sequence $X_1 \oplus X'_1 \oplus Z''_1 \oplus X_1 \oplus X'_1 \oplus Z''_2 \to_{E,Ax} Z''_1 \oplus Z''_2$ is shorter.

The following result provides sufficient conditions for the finite variant property.

Theorem 14 (Sufficient conditions for FV) *Let $\mathcal{R} = (\Sigma, E, R)$ be a decomposition of $(\Sigma, \mathcal{E})$. If (i) $\mathcal{R}$ is variant-preserving (VP), and (ii) there is no infinite variant-preserving narrowing sequence (FVNS), then $\mathcal{R}$ satisfies the finite variant property.*

Proof. *Since we assume that the Ax unification algorithm is finitary, and therefore the narrowing tree is finitely branching, by König's Lemma the tree of variant-preserving narrowing sequences is finite. Given a term t, we denote by $\#(t)$ the length of the longest variant-preserving narrowing sequence from t. We prove that, for any substitution σ, $t(\sigma{\downarrow_{E,Ax}}) \to^{\leq n}_{E,Ax} (t\sigma){\downarrow_{E,Ax}}$ by induction on $n = \#(t)$.*

- *($n = 0$) Then t is irreducible and, for any substitution σ, $t(\sigma{\downarrow_{E,Ax}})$ is also irreducible.*
- *($n > 0$) Let $t = f(t_1, \ldots, t_k)$ and σ be a substitution. Let us assume that $t\sigma$ is eventually reduced at the top in every variant-preserving rewrite sequence. Otherwise, we can prove by structural induction*

and the boundedness property that the bound for t is the sum of the bounds for the arguments $t_1, \ldots, t_k$. We have $\#(t_i) < \#(t)$. By induction hypothesis, for any substitution σ, $t_i(\sigma{\downarrow_{E,Ax}})$ is bounded by $\#(t_i)$ for $i \in \{1, \ldots, k\}$. Let us pick any variant (t'_i, ρ_i) for each t_i, $i \in \{1, \ldots, k\}$ such that $\sigma \sqsubseteq_{Ax} (\rho_1 \cdots \rho_k)$. Let $t' = f(t'_1, \ldots, t'_k)$. By variant-preservingness, there is a rule $l \to r \in E$ and a normalized substitution θ such that $t' =_{Ax} l\theta$. Since $\#(r) < \#(t)$, we can apply the induction hypothesis and, for any substitution σ', $r(\sigma'{\downarrow_{E,Ax}})$ is bounded by $\#(r)$. Since θ is normalized, $r\theta$ is also bounded by $\#(r)$. Note that $\#(t_1) + \cdots + \#(t_k) + \#(t_r) < \#(t)$. Thus, for any substitution σ, $t\sigma$ is bounded by $\#(t)$. □

Note that variant-preservingness is not a *necessary* condition for FV, as shown in Example 18. However, there are many theories where lack of variant preservingness causes loss of FV, as illustrated below.

Example 22 *Consider again Example 3, which as we show in Example 26 below is an FV decomposition, but let us assume now that some variables in rules* (5.7) *and* (5.8) *of that example are restricted to a subsort* Element*, so that they cannot match any term rooted by $\oplus$. That is, we have two sorts* Xor *and* Element *such that $\_\oplus\_ :$ Xor Xor $\to$ Xor and all other symbols a, b, 0, $pk(\_,\_)$, and $sk(\_,\_)$ are defined on sort* Element *and not on sort* Xor*. The new equations are as follows:*

$$\begin{aligned} X{:}\mathsf{Xor} \oplus 0 &= X{:}\mathsf{Xor} \\ X{:}\mathsf{Element} \oplus X{:}\mathsf{Element} &= 0 \qquad (5.14) \\ X{:}\mathsf{Element} \oplus X{:}\mathsf{Element} \oplus Y{:}\mathsf{Xor} &= Y{:}\mathsf{Xor} \qquad (5.15) \end{aligned}$$

Let us consider the term $t = a \oplus (b \oplus (a \oplus b))$. Rule (5.14) *cannot be applied at any position, and only rule* (5.15) *can be applied at the top. However, there is no possible application with a normalized substitution and thus term t cannot be reduced to its normal form in one step, i.e., $a \oplus (b \oplus (a \oplus b)) \to_{E,Ax} b \oplus b \to_{E,Ax} 0$. Indeed, note that given a term $s = X{:}\mathsf{Xor} \oplus Y{:}\mathsf{Xor}$ and any normalized substitution σ, the number of reduction steps for $s\sigma$ to reach its normal form clearly depends on the number of $\oplus$ symbols introduced by σ, and therefore this modified example fails to satisfy FV.*

Although VP is not a necessary condition, the absence of infinite variant-preserving narrowing sequences is a *necessary* condition for FV.

Theorem 15 (Necessary condition for FV) *Let $\mathcal{R} = (\Sigma, E, R)$ be a decomposition of $(\Sigma, \mathcal{E})$. If there is an infinite variant-preserving narrowing sequence, then $\mathcal{R}$ does not have the finite variant property.*

Proof. *Let us consider an infinite variant-preserving narrowing sequence. We can take any finite prefix $t \rightsquigarrow^*_{\sigma,E,Ax} s$ and build a variant-preserving rewrite sequence $t\sigma \rightarrow^*_{E,Ax} (t\sigma){\downarrow_{E,Ax}}$. Note that $\sigma|_{Var(t)}$ is E, Ax-normalized by definition. Thus, we obtain an infinite number of rewrite sequences with increasing length. Since the theory is terminating for rewriting and the computed substitutions are normalized, the rewrite sequences are increasing in length because the computed substitutions are increasing in depth. Since these rewrite sequences are the shortest ones, this contradicts the boundedness property.* □

5.6 Checking the Finite Variant Property

In the following we show that the property of being variant-preserving is clearly checkable, but the absence of infinite variant-preserving narrowing sequences is not computable in general. In Section 5.6.2, we approximate the absence of infinite variant-preserving narrowing sequences by a checkable condition using the dependency pairs technique of [62] for the modulo case.

5.6.1 Checking Variant-Preservingness

The following class of equational theories is relevant. The notion of *Ax-descendants* is a straightforward extension of the standard notion of descendant for rules.

Definition 29 (Descendants) [119] *Let $A : t \xrightarrow{p}_{l \to r} s$ and $q \in Pos(t)$. The set $q \backslash\!\backslash A$ of descendants of q in s w.r.t. A is defined as follows:*

$$q \backslash\!\backslash A = \begin{cases} \{q\} & \textit{if } q < p \textit{ or } q \parallel p \textit{ (i.e., } q \not\leq p \textit{ and } p \not\leq q\textit{)}, \\ \{p.p_3.p_2 \mid r|_{p_3} = l|_{p_1}\} & \textit{if } q = p.p_1.p_2 \textit{ with } p_1 \in Pos_{\mathcal{X}}(l), \\ & \textit{i.e., } p_1 \textit{ is a variable position} \\ \emptyset & \textit{otherwise.} \end{cases}$$

$$\begin{tikzcd}
t_3 \arrow[d, Leftrightarrow, "Ax\ (p\backslash\backslash_{t_3}=\emptyset)"] & \\
t_1 \arrow[r, "p", "{E,Ax}"'] \arrow[d, dotted, "p'\ (p'\leq p, p'\backslash\backslash_{t_3}=\emptyset)", "{E,Ax}"'] & t_2 \arrow[dd, dotted, "{*}_{E,Ax}"] \\
t'_3 \arrow[d, dotted, "{*}_{E,Ax}"] & \\
t_5 \arrow[r, leftrightarrow, dotted, "Ax"'] & t_4
\end{tikzcd}$$

Figure 5.2: Upper-Ax-coherence

$$\begin{tikzcd}
t_1 \arrow[r, "{E,Ax}"'] \arrow[d, Leftrightarrow, "Ax"] & t_2 \arrow[dd, dotted, "{*}_{E,Ax}"] \\
t_3 \arrow[d, dotted, "{+}_{E,Ax}"] & \\
t_5 \arrow[r, leftrightarrow, dotted, "Ax"'] & t_4
\end{tikzcd}$$

Figure 5.3: Ax-coherence

If $Q \subseteq Pos(t)$ then $Q\backslash\backslash A$ denotes the set $\bigcup_{q\in Q} q\backslash\backslash A$. The notion of descendant extends to rewrite sequences in the obvious way. If Q is a set of pairwise disjoint positions in t and $A : t \to^* s$, then the positions in $Q\backslash\backslash A$ are pairwise disjoint. The notion of descendant is extended to an equational theory Ax as follows.

Definition 30 (Ax-descendants) *Let Ax be a set of regular and sort-preserving Σ-equations. Let $\overset{\leftrightarrow}{Ax} = \{u \to v \mid u = v \text{ or } v = u \in Ax\}$. Given two terms $t =_{Ax} s$, i.e., $A : t \to^*_{\overset{\leftrightarrow}{Ax}} s$, and a set Q of pairwise disjoint positions in t, the Ax-*descendants *of Q in s are $Q\backslash\backslash_s = Q\backslash\backslash A$.*

Now we can introduce the relevant notion of upper-Ax-coherence, depicted in Figure 5.2. Note that dotted arrows imply they are involved in an existential quantifier.

Definition 31 (Upper-Ax-coherence) *Let $\mathcal{R} = (\Sigma, Ax, E)$ be a decomposition of $(\Sigma, \mathcal{E})$. We say $\mathcal{R}$ is* upper-Ax-coherent *iff for all t_1, t_2, t_3, $t_1 \xrightarrow{p}_{E,Ax} t_2$, $t_1 =_{Ax} t_3$, $p > \Lambda$, and $p\backslash\backslash_{t_3} = \emptyset$ imply that for all $p' \leq p$ such that $p'\backslash\backslash_{t_3} = \emptyset$, there exist t'_3, t_4, t_5 such that $t_1 \xrightarrow{p'}_{E,Ax} t'_3$, $t_2 \to^*_{E,Ax} t_4$, $t'_3 \to^*_{E,Ax} t_5$, and $t_4 =_{Ax} t_5$.*

Assuming Ax-coherence (defined by Condition (4) in Section 5.1 and depicted in Figures 5.1 and 5.3, both identical but using R,Ax or E,Ax labels), checking upper-Ax-coherence consists in considering each term t in each equation $t = t' \in Ax$ (or its reverse), finding a position $p \in Pos(t)$ s.t. $p > \Lambda$ and a substitution σ s.t. $t\sigma|_p$ is $\to_{E,Ax}$-reducible and then, if $p = p_1.\cdots.p_k$, then, for $i \in \{1, \ldots, k-1\}$, $t\sigma|_{p_i}$ must be $\to_{E,Ax}$-reducible. In general, upper-Ax-coherence is much more demanding than Ax-coherence, as shown below.

Example 23 *Let us consider the equational theory $E = \{g(f(X)) \to d,\ a \to c\}$ and $Ax = \{g(f(f(a))) = g(b)\}$. For the term $t = g(f(f(a)))$, subterm a is reducible, $t =_{Ax} g(b)$, but subterms $f(f(a))$ and $f(a)$ are not reducible and thus the theory is not upper-Ax-coherent. However, the theory is trivially Ax-coherent because of the use of symbol g at the top of both sides of the equation in Ax.*

Note that upper-AC-coherence and AC-coherence coincide, since the axioms of associativity and commutativity can never satisfy $t_1 =_{AC} t_3$, $p > \Lambda$, and $p\backslash\!\!\backslash_{t_3} = \emptyset$. We can now provide an algorithm for checking variant-preservingness.

Theorem 16 (Checking Variant-preservingness) *Let $\mathcal{R} = (\Sigma, Ax, E)$ be a decomposition of $(\Sigma, \mathcal{E})$ that is upper-Ax-coherent. $\mathcal{R}$ has the variant-preserving property iff for all $l \to r, l' \to r' \in E$ (possibly renamed s.t. $Var(l) \cap Var(l') = \emptyset$) and for each $X \in Var(l)$, the term $t = l\theta$, where $\theta = \{X \mapsto l'\}$ is an order-sorted substitution, satisfies that either: (i) t does not have a variant-pattern, or (ii) otherwise there is a normalized reduction on t.*

Proof. *The only if part is immediate by definition. For the if part, we consider a term $t = f(t_1, \ldots, t_k)$ such that $t_1, \ldots, t_k$ are $\to_{E,Ax}$-irreducible terms. If t is $\to_{E,Ax}$-irreducible, we are done. Otherwise, there is a rule $l \to r \in E$ and a substitution θ such that $t = l\theta$. If θ is $\to_{E,Ax}$-normalized, we are done. Otherwise, we prove below that there is a rule $l' \to r' \in E$ and a substitution θ' such that $t = l'\theta'$ and θ' is $\to_{E,Ax}$-normalized.*

Let $l \to r \in E$ and θ be such that θ has the maximum number of redexes possible for t. Let n be such a maximum number. We prove the fact by induction on n.

$(n = 0)$ *This means that θ is $\to_{E,Ax}$-normalized and we are done.*

$(n > 0)$ *Let $X \mapsto u$ be one of the non-normalized bindings in θ. Let p be one of the topmost positions in u with an actual redex, i.e., there is* *a rule $\hat{l} \to \hat{r} \in E$ and a substitution σ such that $u|_p =_{Ax} \hat{l}\sigma$. We can take the maximum prefix $\hat{u}$ of u with no redexes and build a substitution $\hat{\theta} = \{X \mapsto \hat{u}[\hat{l}]_p\}$. Let us assume that $\hat{u}[\hat{l}]_p$ is properly renamed so that $Var(\hat{u}[\hat{l}]_p) \cap Var(l) = \emptyset$. There is a substitution ρ such that $\theta =_{Ax} \hat{\theta}\rho$. Since the terms $t_1, \ldots, t_k$ are irreducible, $\hat{l}$ is not a subterm of any of them and there is a context $C[\ ]$ of t and another context $\hat{C}[\ ]$ of $l\hat{\theta}$ such that $C[\ ] =_{Ax} \hat{C}[\ ]$ and $\hat{l}$ must overlap with $\hat{C}[\ ]$. Then, $p = \Lambda$, because of coherence, i.e., if $u|_p$ is a redex, then u must also be a redex. Just note that a coherence completion algorithm adds rules of the form $C[l\sigma] \to C[r\sigma]$ for any rule $l \to r$ where $C[\ ]$ and σ are determined by the equational theory Ax. Now, by the condition given in the Theorem, there is a normalized substitution on $l\hat{\theta}$, i.e., there is a rule $l' \to r'$ and a substitution τ such that $l\hat{\theta} =_{Ax} l'\tau$ and τ is $\to_{E,Ax}$-normalized. Finally, when we consider the term $l'\tau\rho$, we can apply the induction hypothesis because ρ contains less redexes than θ and obtain that there is a rule $l'' \to r''$ and a substitution τ' such that $t =_{Ax} l'\tau\rho =_{Ax} l''\tau'$ and τ' is $\to_{E,Ax}$-normalized.* □

The upper-Ax-coherence condition is necessary, as shown below.

Example 24 *The theory of Example 23 satisfies the conditions of Theorem 16 except upper Ax-coherence. That is, when the left-hand sides $g(f(X))$ and a are used to build the term $g(f(a))$, this term does not have a variant-pattern, as required by Theorem 16. Similarly, when the properly renamed left-hand sides $g(f(X))$ and $g(f(X'))$ are used to build the term $g(f(g(f(X'))))$, this term does not have a variant-pattern either. However, according to Definition 26, we have to test also the variant-pattern $g(b)$. Although this term is reducible, it is not $\to_{E,Ax}$-reducible with a normalized substitution. Thus the equational theory is not variant-preserving.*

Let us first show an example of a theory that is not variant-preserving.

Example 25 *Let us consider again Example 12. Let us check this rewrite theory with the condition from Theorem 16. Using the rule given with the renamed version $f(a, b, X') \to f(a, b)$ we get $l\theta = f(a, b, a, b, X')$, which has a variant-pattern, namely $f(f(a, a, X'), f(b, b))$ where the extra appearances of f inside are to show which are the irreducible subterms. Also, there is no* *reduction with a normalized substitution, since the only reduction possible is by using the given rule, with X renamed to V and the substitution $\sigma = \{V \mapsto f(a, b, X')\}$ which is not normalized. So this theory is not variant-preserving.*

Let us prove that the exclusive or theory has the variant-preservingness property.

Example 26 *Let $\mathcal{R} = (\Sigma, E, R)$ be the exclusive or theory from Example 3 (without pk, sk), i.e., with only* (5.6)–(5.8) *used as rules. Using Theorem 16 we find that this theory is variant-preserving. All the combinations of rules not involving* (5.8) *as the first rule do not have a variant-pattern, let us just show one of the combinations of rule* (5.8) *with itself where $l = X \oplus X \oplus Y$ and $l' = X' \oplus X' \oplus Y'$. We get two terms, one for each of the substitutions $\theta_1 = \{X \mapsto l'\}$ and $\theta_2 = \{Y \mapsto l'\}$. We get $l\theta_1 = X' \oplus X' \oplus Y' \oplus X' \oplus X' \oplus Y' \oplus Y$, which does not have a variant-pattern. On the other hand, $l\theta_2 = X \oplus X \oplus X' \oplus X' \oplus Y'$ does have a variant-pattern, but has also a normalized reduction with another renaming of rule* (5.8)*, namely $V \oplus V \oplus W \to W$, and substitution $\sigma = \{V \mapsto X \oplus X', W \mapsto Y'\}$. Note that the theory has the finite variant property (FV), since it is VP and the right hand sides of all the equations are constants or variables, which trivially satisfies the FVNS property.*

5.6.2 Checking Finiteness of Variant-Preserving Narrowing Sequences

In this section, we approximate the absence of infinite variant-preserving narrowing sequences by a checkable condition using the dependency pairs technique of [62] for the modulo case. Note that we do not really extend the dependency pairs technique to narrowing, since we do not allow extra variables in right-hand sides of rules; see [5] for an extension of the dependency pairs technique to narrowing, and [99] for termination of narrowing using the dependency pair technique. Termination of narrowing is a much harder problem than that of termination of rewriting [6] and we do not prove that narrowing or folding variant narrowing terminate; indeed recall that we are only interested in termination of the variant generation process rather than termination of narrowing strategies in general. In this section, we reuse the dependency pair technique and approximate the property of the absence of infinite variant-preserving narrowing sequences by avoiding any possible cycle in function calls. For avoiding cycles we use the dependency graph and adapt the notion of dependency pair chain to the variant case.

First, we need to extend the notion of a defined symbol. An equation $u = v$ is called *collapsing* if $v \in \mathcal{X}$ or $u \in \mathcal{X}$. We say a theory is *collapse-free*[3] if all its equations are non-collapsing.

Definition 32 (Defined Symbols for Rewriting Modulo Equations) [62] *Let (Σ, Ax, R) be an order-sorted rewrite theory with Ax collapse-free. Then the set of defined symbols D is the smallest set such that $D = \{root(l) \mid l \to r \in R\} \cup \{root(v) \mid u = v \in Ax \text{ or } v = u \in Ax, root(u) \in D\}$.*

In order to correctly approximate the dependency relation between defined symbols in the theory, we need to extend the equational theory in the following way.

Definition 33 (Adding Instantiations) [62] *Given an order-sorted rewrite theory $\mathcal{R} = (\Sigma, Ax, R)$ with Ax collapse-free, let $Ins_{Ax}(R)$ be a set containing only rules of the form $l\sigma \to r\sigma$ (where σ is a substitution and $l \to r \in R$). $Ins_{Ax}(R)$ is called an instantiation of R for the equations Ax iff $Ins_{Ax}(R)$ is the smallest set such that: (a) $R \subseteq Ins_{Ax}(R)$, (b) for all $l \to r \in R$, all v such that $u = v \in Ax$ or $v = u \in Ax$, and all $\sigma \in CSU_{Ax}(v = l)$, there exists a rule $l' \to r' \in Ins_{Ax}(R)$ and a variable renaming ρ such that $l\sigma =_{Ax} l'\rho$ and $r\sigma =_{Ax} r'\rho$.*

Note that when $Ax = \emptyset$ or Ax contains only AC or C axioms, $Ins_{Ax}(R) = R$. Dependency pairs are obtained as follows. Since we are dealing with the modulo case, it will be notationally more convenient to use terms directly in dependency pairs, without the usual capital letters for the top symbols.

Definition 34 (Dependency Pair) [62] *Let $\mathcal{R} = (\Sigma, Ax, R)$ be an order-sorted rewrite theory with Ax collapse-free. Let $Ins_{Ax}(R)$ be the instantiations of R for the equations Ax. If $l \to C[g(t_1, \ldots, t_m)]$ is a rule of $Ins_{Ax}(R)$*

[3]Note that regularity does not imply collapse-free, e.g., equation (5.6) of Example 3 is regular but also collapsing. Note also that if Ax contains collapsing axioms such as the identity axiom (5.6), it may be possible to use the variant based technique in [42] (see also the discussion in Section 5.8) to transform a decomposition (Σ, Ax, R) into a semantically equivalent one $(\Sigma, Ax_0, R \cup \overrightarrow{A}_{clps})$ where Ax_0 is collapse-free and $\overrightarrow{A}_{clps}$ are rewrite rules for the collapse axioms.

with C a context and g a defined symbol in $Ins_{Ax}(R)$, then $\langle l, g(t_1, \ldots, t_m)\rangle$ is called a dependency pair *of $\mathcal{R}$.*

Example 27 (Abelian Group) *The following presentation of the Abelian group theory, called $\mathcal{R}_* = (\Sigma, Ax, E)$, has been shown to satisfy the finite variant property in [32]. The operators Σ are $\_*\_$, $(\_)^{-1}$, and 1. The set of equations Ax consists of associativity and commutativity for $\_*\_$. The rules E are:*

$$\begin{aligned}
x * 1 &\rightarrow x && (5.16)\\
1^{-1} &\rightarrow 1 && (5.17)\\
x * x^{-1} &\rightarrow 1 && (5.18)\\
x^{-1} * y^{-1} &\rightarrow (x * y)^{-1} && (5.19)\\
(x * y)^{-1} * y &\rightarrow x^{-1} && (5.20)\\
x^{-1^{-1}} &\rightarrow x && (5.21)\\
(x^{-1} * y)^{-1} &\rightarrow x * y^{-1} && (5.22)\\
x * (x^{-1} * y) &\rightarrow y && (5.23)\\
x^{-1} * (y^{-1} * z) &\rightarrow (x * y)^{-1} * z && (5.24)\\
(x * y)^{-1} * (y * z) &\rightarrow x^{-1} * z && (5.25)
\end{aligned}$$

The AC-dependency pairs for this rewrite theory are as follows.

$$\begin{array}{ll}
(5.19)a: & \langle x^{-1} * y^{-1} , (x * y)^{-1}\rangle\\
(5.19)b: & \langle x^{-1} * y^{-1} , x * y\rangle\\
(5.22)a: & \langle (x^{-1} * y)^{-1} , x * y^{-1}\rangle\\
(5.22)b: & \langle (x^{-1} * y)^{-1} , y^{-1}\rangle\\
(5.20)a: & \langle (x * y)^{-1} * y , x^{-1}\rangle\\
(5.24)a: & \langle x^{-1} * y^{-1} * z , (x * y)^{-1} * z\rangle\\
(5.24)b: & \langle x^{-1} * y^{-1} * z , (x * y)^{-1}\rangle\\
(5.24)c: & \langle x^{-1} * y^{-1} * z , x * y\rangle\\
(5.25)a: & \langle (x * y)^{-1} * y * z , x^{-1} * z\rangle\\
(5.25)b: & \langle (x * y)^{-1} * y * z , x^{-1}\rangle
\end{array}$$

We have used the AProVE tool [63] to generate the dependency pairs. AProVE first applies the coherence algorithm of [62] to this example, which is unnecessary here and thus we drop the dependency pairs created that way.

The relevant notions from the dependency pairs technique are chains of dependency pairs and the dependency graph.

Definition 35 (Chain) [12] *Let $\mathcal{R} = (\Sigma, Ax, R)$ be an order-sorted rewrite theory with Ax collapse-free. A sequence of dependency pairs $\langle s_1, t_1\rangle\langle s_2, t_2\rangle \cdots \langle s_n, t_n\rangle$ of $\mathcal{R}$ is an $\mathcal{R}$-*chain *if there is a substitution σ such that $t_j\sigma \to^*_{R,Ax} s_{j+1}\sigma$ holds for every two consecutive pairs $\langle s_j, t_j\rangle$ and $\langle s_{j+1}, t_{j+1}\rangle$ in the sequence.*

Definition 36 (Dependency Graph) [12] *Let $\mathcal{R} = (\Sigma, Ax, R)$ be an order-sorted rewrite theory with Ax collapse-free. The dependency graph of $\mathcal{R}$ is the directed graph whose nodes (vertices) are the dependency pairs of R and there is an arc (directed edge) from $\langle s, t\rangle$ to $\langle u, v\rangle$ if $\langle s, t\rangle\langle u, v\rangle$ is a chain.*

Chains are not computable in general and an approximation must be performed. The notions of *connectable terms* and the *estimated dependency graph* as defined in [12] provide a useful approximation of the dependency graph. The estimated dependency graph can be computed using the CAP and REN procedures [12]: For any term $t \in \mathcal{T}_\Sigma(\mathcal{X})$, let CAP($t$) replace each proper subterm rooted by a defined symbol by a fresh variable and let REN(t) independently rename all occurrences of variables in t by fresh variables. Note that such an estimated dependency graph has been used in all examples in this section.

Example 28 *The dependency graph for Example 27 is shown in Figure 5.4. It was created with AProVE [63]. We see that there are self-loops on* (5.19)*b,* (5.22)*b,* (5.24)*a,* (5.24)*c and* (5.25)*a.* (5.19)*a has a loop with* (5.22)*a,* (5.22)*a has a loop with* (5.24)*b, and so on. It is a very highly connected graph.*

The most important notion for the absence of infinite narrowing sequences is that of a cycle in the dependency graph.

Definition 37 (Cycle) [12] *A nonempty set $\mathcal{P}$ of dependency pairs is called a* cycle *if, for any two dependency pairs $\langle s, t\rangle, \langle u, v\rangle \in \mathcal{P}$, there is a nonempty path from $\langle s, t\rangle$ to $\langle u, v\rangle$ and from $\langle u, v\rangle$ to $\langle s, t\rangle$ in the dependency graph that traverses dependency pairs from $\mathcal{P}$ only.*

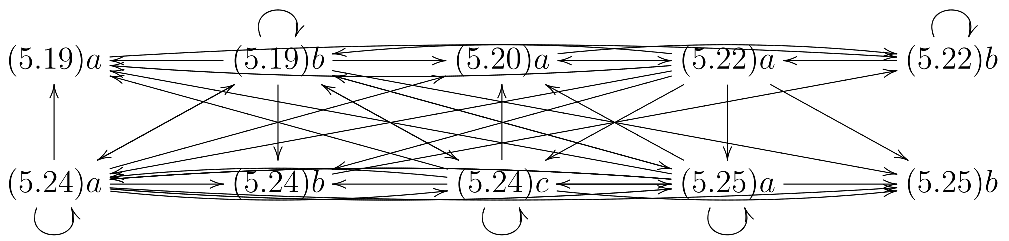


Figure 5.4: Dependency graph of Abelian group

As already demonstrated in the previous section, not all the rewriting (narrowing) sequences are relevant for the finite variant property, so that we can restrict the dependency graph only to variant-preserving rewriting (narrowing) sequences.

Definition 38 (Variant-preserving chain) *Let $\mathcal{R} = (\Sigma, Ax, E)$ be a variant-preserving decomposition of an equational theory $(\Sigma, \mathcal{E})$. A chain of dependency pairs $\langle s_1, t_1\rangle\langle s_2, t_2\rangle \cdots \langle s_n, t_n\rangle$ of $\mathcal{R}$ is a* variant-preserving chain *if there is a substitution σ such that σ is $\rightarrow_{E,Ax}$-normalized and the following rewrite sequence $s_1\sigma \rightarrow_{E,Ax} C_1[t_1]\sigma \rightarrow^*_{E,Ax} C_1[s_2]\sigma \rightarrow_{E,Ax} C_1[C_2[t_2]]\sigma \rightarrow^*_{E,Ax} \cdots \rightarrow^*_{E,Ax} C_1[C_2[\cdots C_{n-1}[s_n]]]\sigma \rightarrow_{E,Ax} C_1[C_2[\cdots C_{n-1}[C_n[t_n]]]]\sigma$ obtainable from the chain $\langle s_1, t_1\rangle\langle s_2, t_2\rangle \cdots \langle s_n, t_n\rangle$ is variant-preserving.*

The notions of a cycle, dependency graph, and estimated dependency graph are easily extended to the variant-preserving case. The following result approximates the absence of infinite narrowing sequences. We simply approximate such property by avoiding any cycle. We do not use any of the dependency pair processors of the dependency pair framework (see [12, 64]) and we do not require any term ordering. Obviously, there may be more specific techniques based on termination of narrowing for deciding the termination of variant-preserving narrowing sequences but this is left for future work.

Proposition 3 (Finiteness Check of the VP Narrowing sequences) *Let $\mathcal{R} = (\Sigma, Ax, E)$ be a variant-preserving decomposition of an equational theory $(\Sigma, \mathcal{E})$. Let Ax contain only linear, non-collapsing equations. If the estimated dependency graph does not contain any variant-preserving cycle, then there are no infinite variant-preserving narrowing sequences.*

Proof. *We prove this result by contradiction. Assume that the estimated dependency graph does not contain any variant-preserving cycle but there*

$$\begin{tikzcd}
(5.19)a & (5.19)b & (5.20)a & (5.22)a & (5.22)b \\
(5.24)a & (5.24)b & (5.24)c & (5.25)a \arrow[ulll] \arrow[ull] \arrow[l] \arrow[ll, bend left] \arrow[lll, bend left] & (5.25)b
\end{tikzcd}$$

Figure 5.5: Variant-preserving dependency graph

is an infinite variant-preserving narrowing sequence $\alpha : t_0 \rightsquigarrow_{p_1,\sigma_1,E,Ax} t_1 \cdots \rightsquigarrow_{p_n,\sigma_n,E,Ax} t_n \cdots$. *From* α *we can obtain an infinite number of finite variant-preserving rewrite sequences of the form* $t_0\theta_i \rightarrow_{p_1,E,Ax} t_1\theta_i \cdots \rightarrow_{p_i,E,Ax} t_i\theta_i$ *with* $\theta_i = \sigma_1 \cdots \sigma_i$. *For each variant-preserving rewrite sequence* $t_0\theta_i \rightarrow_{p_1,E,Ax} t_1\theta_i \cdots \rightarrow_{p_i,E,Ax} t_i\theta_i$*, there is a variant-preserving chain corresponding to such rewrite sequence. Since the number of dependency pairs is finite, there is a natural number* k *such that for the variant-preserving rewrite sequence* $t_0\theta_k \rightarrow_{p_1,E,Ax} t_1\theta_k \cdots \rightarrow_{p_k,E,Ax} t_k\theta_k$*, the variant-preserving chain associated to it is a cycle. Thus, the conclusion follows, because we assume that there is no variant-preserving cycle.* □

Note that the conditions that the axioms are non-collapsing and linear are necessary for completeness of the dependency graph, we refer the reader to [62] for explanations.

Example 29 (AG variant-preserving dependency pair graph) *We can show the variant-preserving dependency graph of Example 27 in Figure 5.5. One can see in the picture that all the cycles have disappeared, because they involved non-normalized substitutions, or terms without a variant-pattern, or could be shortened. Detailed reasons are provided next.*

For the dependency pair $(5.19)b$ *and its self-loop we need a substitution* σ *for which* $(X * Y)\sigma =_{AC} (X'^{-1} * Y'^{-1})\sigma$. *But then, e.g.,* $\sigma = \{X \mapsto X'^{-1}, Y \mapsto Y'^{-1}\}$ *and the left-hand side of the dependency pair becomes* $(X'^{-1})^{-1} * (Y'^{-1})^{-1}$*, which does not have a variant-pattern, as* $(X'^{-1})^{-1}$ *is reducible, so the self-loop is not a variant-preserving sequence and thus not a variant-preserving chain.*

For the dependency pairs $(5.24)a$*, i.e.,* $\langle s_1, t_1\rangle = \langle X^{-1} * Y^{-1} * Z, (X * Y)^{-1} * Z\rangle$*, and* $(5.25)a$*, i.e.,* $\langle s_2, t_2\rangle = \langle (X' * Y')^{-1} * Y' * Z', X'^{-1} * Z'\rangle$ *let us consider both directions. For one direction we have* $((X*Y)^{-1}*Z)\sigma =_{AC} ((X'*Y')^{-1}*Y'*Z')\sigma$ *so for example* $\sigma = \{Z \mapsto Y'*Z', X \mapsto X', Y \mapsto Y'\}$. *Then* $s_1\sigma =_{AC} X'^{-1}*Y'^{-1}*Y'*Z'$ *which has a variant-pattern and for which the rewriting sequence is* $X'^{-1}*Y'^{-1}*Y'*Z' \rightarrow (X'*Y')^{-1}*Y'*Z' \rightarrow X'^{-1}*Z'$.

$$\begin{tikzcd}
(5.19)a & (5.19)b & (5.20)a & (5.22)a & (5.22)b \\
(5.24)a & (5.24)b & (5.24)c & (5.25)a \arrow[ulll] \arrow[ull] \arrow[l] \arrow[ll, bend left] \arrow[lll, bend left] & (5.25)b \\
 & (5.27)a & (5.27)b \arrow[ull] \arrow[uull] \arrow[ul] \arrow[uul] \arrow[u] \arrow[uu, bend right] \arrow[ur] \arrow[urr] & &
\end{tikzcd}$$

Figure 5.6: Variant-preserving dependency graph for Diffie-Hellman

Nevertheless, it is not a variant-preserving sequence as there is a shorter rewriting sequence using rule (5.23), $X'^{-1} * Y'^{-1} * Y' * Z' \to X'^{-1} * Z'$, *so there is no variant-preserving chain here.*

Similarly for the chain from $(5.24)a$ *to* $(5.25)b$ *as the only difference is in* t_2, *so that* $t_2\sigma = X'^{-1}$ *but that will be padded with the context of* $\_*\_([], Z')$ *(where* $[]$ *is the hole) and so the same shorter rewriting sequence exists.*

In the other direction, from $(5.25)a$ *to* $(5.24)a$, *we have* $(X'^{-1} * Z')\sigma =_{AC} (X^{-1} * Y^{-1} * Z)\sigma$ *so then for example* $\sigma = \{Z' \mapsto Y^{-1}Z, X' \mapsto X\}$ *and* $s_2\sigma =_{AC} (X * Y')^{-1} * Y' * Y^{-1} * Z$ *which has a variant-pattern and the rewriting sequence* $(X * Y')^{-1} * Y' * Y^{-1} * Z \to X^{-1} * Y^{-1} * Z \to (X * Y)^{-1} * Z$. *The alternative rewriting sequence applying the rules in reverse order is* $(X * Y')^{-1} * Y' * Y^{-1} * Z \to (X * Y' * Y)^{-1} * Y' * Z \to (X * Y)^{-1} * Z$ *which is not shorter, so this is a variant-preserving sequence and thus we have a variant-preserving chain.*

Let us first introduce a representation of the Diffie-Hellman theory and then show the VP property for the theories of Abelian groups and Diffie-Hellman exponentiation, and also the finite variant property for the Diffie-Hellman theory.

Example 30 (Diffie-Hellman) *We get a rewrite theory representing the Diffie-Hellman theory, called* $\mathcal{R}_{\mathcal{DH}}$, *by extending the theory* $\mathcal{R}_*$ *from Example 27 by adding a new binary symbol exp and the following two rules:*

$$exp(x, 1) \to x \tag{5.26}$$

$$exp(exp(x, y), z) \to exp(x, y * z) \tag{5.27}$$

We can compute the dependency pairs and the associated graph using the results we already have from Example 29. Also note, that the rewrite theories

$\mathcal{R}_*$ and $\mathcal{R}_{\mathcal{DH}}$ both have the variant-preserving property, which we will check in Example 31, respectively Example 32. The following additional dependency pairs are required:

$$\begin{array}{lcl} (5.27)a & : & \langle exp(exp(x,y),z) \quad , \quad exp(x, y * z)\rangle \\ (5.27)b & : & \langle exp(exp(x,y),z) \quad , \quad y * z\rangle \end{array}$$

As shown in Figure 5.6, for rule (5.27) there are a lot of possibilities to go from (5.27)b, but the longest possible path has length 2. Let us show that there is actually a chain for the path from (5.27)b via (5.25)a to (5.19)a. After substituting as needed for this in the left-hand side of (5.27) we get $exp(exp(X,(U*V)^{-1}), V*W^{-1}) \to exp(X,(U*V)^{-1}*V*W^{-1})$, let us call this term t. Then from there we have $t \to exp(X, U^{-1}*W^{-1}) \to exp(X,(U*W)^{-1})$ and alternatively $t \to exp(X,(U*V*W)^{-1}*V) \to exp(X,(U*W)^{-1})$ which is not shorter. So this is really a variant-preserving chain and the longest chain from (5.27)b is length 2.

We show VP for our Abelian group representation next.

Example 31 *Let us check variant-preservingness for $\mathcal{R}_*$ by using Theorem 16. For rule (5.16) and any other rule there is no variant-pattern for $l\theta$ where θ substitutes another left-hand side into X. The reason is that the constant 1 needs to stay isolated, since otherwise a rewrite is possible, and so the left-hand side that was inserted stays together and is reducible. As rule (5.17) does not have any variable, the property holds trivially.*

For all following rules let us note that instantiating a variable that is a subterm of an inverse operator $^{-1}$ with a left-hand side of another rule, immediately results in a term that has no variant-pattern as that left-hand side stays together underneath. Thus the rules (5.18)–(5.22) do not need to be considered as all variables appear at least once underneath an inverse operator.

In this vein for rule (5.23) we only need to consider the terms created when instantiating Y. Only combination with (5.18),(5.20), (5.23), and (5.25) results in a term that has a variant-pattern. Let us show for example (5.23) with (5.25) (renamed to primed variables). The resulting term is $X * X^{-1} * (X' * Y')^{-1} * Y' * Z'$ which can be reduced by rule (5.24) (renamed to doubly primed variables) with substitution $\{X'' \mapsto X, Y'' \mapsto X'*Y', Z'' \mapsto X*Y'*Z'\}$ which is normalized.

For rule (5.24) *the only useful (i.e., with a chance of having a variant-pattern) instantiations are for Z, but also as there are already two appearances of a term headed by the inverse only left-hand sides with no inverse have a chance at having a variant-pattern. That only leaves rule* (5.16) *which results in term $X^{-1} * Y^{-1} * X' * 1$ which also does not have a variant-pattern.*

Finally, for rule (5.25) *we only need to instantiate the variable Z. There are variant-patterns for the combinations with* (5.18), (5.20), (5.23), *and* (5.25), *let us just show the last of these combinations,* (5.25) *with itself. The resulting term is $(X * Y)^{-1} * Y * (X' * Y')^{-1} * Y' * Z'$, which has a variant-pattern but also can rewrite with rule* (5.24) *(renamed with two primes) with the normalized substitution $\{X'' \mapsto X * Y, Y'' \mapsto X' * Y', Z'' \mapsto Y * Y' * Z'\}$.*

Therefore, $\mathcal{R}_*$ has the variant-preserving property.

Based on VP for Abelian groups we can check VP for Diffie-Hellman. It also turns out that Diffie-Hellman has the finite-variant property.

Example 32 *Variant-preservingness of the Diffie-Hellman theory $\mathcal{R}_{\mathcal{DH}}$ can be shown using Theorem 16 based upon the variant-preservingness of $\mathcal{R}_*$ shown in Example 31. Let us just observe that $\mathcal{R}_{\mathcal{DH}}$ is obtained by just adding a new symbol exp and rules for it. Putting this into any variable of any of the prior rules results in a term that has no variant-pattern. The other way around, any left-hand side put into any of the variables of the left-hand sides of one of the two new rules results in a term that has no variant-pattern. So $\mathcal{R}_{\mathcal{DH}}$ has the variant-preserving property, too.*

The proof of our final result for this section is trivial: since if there are no cycles in the estimated dependency graph, then we know for sure that there is no infinite variant-preserving rewrite sequence.

Theorem 17 (Approximation for the finite variant property) *Let $\mathcal{R} = (\Sigma, Ax, E)$ be a variant-preserving decomposition of an equational theory $(\Sigma, \mathcal{E})$ such that Ax contains only linear, non-collapsing equations. If the estimated dependency graph does not contain any variant-preserving cycle, then $\mathcal{R}$ has the finite variant property.*

Proof. *By Proposition 3 and Theorem 15.* □

5.6.3 Disproving the Finite Variant Property

If there are infinite variant-preserving narrowing sequences, we are done, because the finite variant property does not hold by Theorem 15. We can give a simple sufficient condition, a consequence of Theorem 15.

Theorem 18 (Non-termination of narrowing) *Let $\mathcal{R} = (\Sigma, Ax, E)$ be a variant-preserving decomposition of an equational theory $(\Sigma, \mathcal{E})$. Let Ax contain only linear, non-collapsing equations. If the estimated dependency graph does contain a variant-preserving chain $\langle s,t\rangle\langle s,t\rangle$ such that $s \sqsubseteq_{Ax} t$, called a* self-cycle*, and the* Cap *and* Ren *procedures were not necessary for obtaining term t, then there is an infinite variant-preserving narrowing sequence starting from term s.*

Proof. *The estimated dependency graph contains the chain $\langle s,t\rangle\langle s,t\rangle$ for the dependency pair $\langle s,t\rangle$. The dependency pair $\langle s,t\rangle$ comes from a rule $s \to C[t]_p$. Let σ be such that $s =_{Ax} t\sigma$. Since the* Cap *and* Ren *procedures have not been applied to term t, we have the infinite narrowing sequence $s \rightsquigarrow_{\Lambda,id,E,Ax} C[t]_p \rightsquigarrow_{p,\sigma,E,Ax} C[C'[t']_p]_p \rightsquigarrow_{p.p,\sigma',E,Ax} C[C'[C''[t'']_p]_p]_p \cdots$ where C' and C'' are properly renamed versions of C, t' and t'' are properly renamed versions of t, and σ' is a properly renamed version of σ.* □

Example 33 (ACUNh) [32] *Let us present the ACU example with nilpotence and homomorphism as discussed by Comon and Delaune.*[4] *This is $\mathcal{R}_{ACUNh}$, with $+$ AC, which has the variant-preserving property:*

$$\begin{array}{rcll} X+0 & \to & X & (5.28)\\ X+X & \to & 0 & (5.29)\\ X+X+Y & \to & Y & (5.30)\end{array} \qquad \begin{array}{rcll} h(0) & \to & 0 & (5.31)\\ h(X+Y) & \to & h(X)+h(Y) & (5.32)\end{array}$$

For the last rule we get three dependency pairs:

$$\begin{array}{lcl} (5.32)a & : & \langle h(x+y) \quad , \quad h(x)+h(y)\rangle\\ (5.32)b & : & \langle h(x+y) \quad , \quad h(x)\rangle\\ (5.32)c & : & \langle h(x+y) \quad , \quad h(y)\rangle \end{array}$$

[4]There is another, alternative term rewriting system representing this theory, which suffers from the same problems.

It is easy to see that there are self-cycles in (5.32)b *and* (5.32)c *using the substitution* $x \mapsto x_1 + z_1$*, which also allows going back and forth between them. This gives rise to the following graph:*

$$\begin{tikzcd}
(5.32)a & (5.32)b \arrow[l] \arrow[r, leftrightarrow] \arrow[loop below] & (5.32)c \arrow[ll, bend right] \arrow[loop below]
\end{tikzcd}$$

By Theorem 15, this theory does not have the finite variant property, as also proved in a different way in [32].

5.7 Variant-based Equational Unification

The intimate connection between variants and $\mathcal{E}$-unification is then as follows.

Definition 39 *For* $\mathcal{R} = (\Sigma, Ax, E)$ *with poset of sorts* $(\mathsf{S}, \leq)$ *being a decomposition of an equational theory* $(\Sigma, \mathcal{E})$*, we extend* (Σ, Ax, E) *and* $(\mathsf{S}, \leq)$ *to* $(\widehat{\Sigma}, Ax, \widehat{E})$ *and* $(\widehat{\mathsf{S}}, \leq)$ *as follows:*

1. *we add a new sort* Truth *to* $\widehat{\mathsf{S}}$*, not related to any sort in* Σ*,*
2. *we add a constant operator* `tt` *of sort* Truth *to* $\widehat{\Sigma}$*,*
3. *for each top sort of a connected component* [s]*, we add an operator* `eq` $: [\mathsf{s}] \times [\mathsf{s}] \to \mathsf{Truth}$ *to* $\widehat{\Sigma}$*, and*
4. *for each top sort* [s]*, we add a variable* X:[s] *and an extra rule* $\mathsf{eq}(X{:}[\mathsf{s}], X{:}[\mathsf{s}]) \to \mathtt{tt}$ *to* $\widehat{E}$*.*

Then, given any two Σ-terms t, t', if θ is an $\mathcal{E}$-unifier of t and t', then the E,Ax-canonical forms of $t\theta$ and $t'\theta$ must be Ax-equal and therefore the pair $(\mathtt{tt}, \theta)$ must be a variant of the term $\mathsf{eq}(t, t')$. Furthermore, if the term $\mathsf{eq}(t, t')$ has a finite set of most general variants, then we are *guaranteed* that the set of most general $\mathcal{E}$-unifiers of t and t' is *finite*.

Corollary 7 *Let* $\mathcal{R} = (\Sigma, Ax, E)$ *with poset of sorts* $(\mathsf{S}, \leq)$ *be a finite variant decomposition of an equational theory* $(\Sigma, \mathcal{E})$*. The equational theory* $(\widehat{\Sigma}, Ax, \widehat{E})$ *with poset of sorts* $(\widehat{\mathsf{S}}, \leq)$ *of Definition 39 is a finite decomposition.*

Proof. *Given a term* $\mathtt{eq}(t,t')$, *for any variant* $(u,\sigma) \in [\![\mathtt{eq}(t,t')]\!]_{E,Ax}$, *either* $u = \mathtt{tt}$ *or* $u = \mathtt{eq}(v,v')$ *such that* $(v,\phi) \in [\![t]\!]_{E,Ax}$ *and* $(v',\phi') \in [\![t']\!]_{E,Ax}$ *for some substitutions* ϕ *and* ϕ'. *Since* $[\![t]\!]_{E,Ax}$ *and* $[\![t']\!]_{E,Ax}$ *are finite, we conclude that* $[\![\mathtt{eq}(t,t')]\!]_{E,Ax}$ *is finite.* □

Let us make explicit the relation between variants and $\mathcal{E}$-unification. Given a decomposition (Σ, Ax, E) of an equational theory, two Σ-terms t_1 and t_2 such that $W_\cap = \mathit{Var}(t_1) \cap \mathit{Var}(t_2)$ and $W_\cup = \mathit{Var}(t_1) \cup \mathit{Var}(t_2)$, and two sets V_1 and V_2 of variants of t_1 and t_2, respectively, we define $V_1 \cap V_2 = \{(u_1\sigma, \theta_1\sigma \cup \theta_2\sigma \cup \sigma) \mid (u_1,\theta_1) \in V_1 \wedge (u_2,\theta_2) \in V_2 \wedge \exists\sigma : \sigma \in \mathit{CSU}^{W_\cup}_{Ax}(u_1 = u_2) \wedge (\theta_1\sigma)|_{W_\cap} =_{Ax} (\theta_2\sigma)|_{W_\cap}\}$.

Proposition 4 (Variant-based Unification) *Let* $\mathcal{R} = (\Sigma, Ax, E)$ *be a decomposition of an equational theory* $(\Sigma,\mathcal{E})$. *Let* t_1, t_2 *be two* Σ-*terms. Then,* ρ *is an* $\mathcal{E}$-*unifier of* t_1 *and* t_2 *iff* $\exists(t',\rho) \in [\![t_1]\!]^\star_{E,Ax} \cap [\![t_2]\!]^\star_{E,Ax}$.

Proof. ($\Rightarrow$) *If* ρ *is an* $\mathcal{E}$-*unifier of* t_1 *and* t_2, *then* $(t_1\rho)\downarrow_{E,Ax} =_{Ax} (t_2\rho)\downarrow_{E,Ax}$. *Let* $t'_1 = (t_1\rho)\downarrow_{E,Ax}$ *and* $t'_2 = (t_2\rho)\downarrow_{E,Ax}$. *We also have that* $(t'_1,\rho) \in [\![t_1]\!]^\star_{E,Ax}$, $(t'_2,\rho) \in [\![t_1]\!]^\star_{E,Ax}$, $(t'_1,\rho) \in [\![t_2]\!]^\star_{E,Ax}$, *and* $(t'_2,\rho) \in [\![t_2]\!]^\star_{E,Ax}$.

($\Leftarrow$) *If* $\exists(t',\rho) \in [\![t_1]\!]^\star_{E,Ax} \cap [\![t_2]\!]^\star_{E,Ax}$, *then* $t' =_{Ax} (t_1\rho)\downarrow_{E,Ax} =_{Ax} (t_2\rho)\downarrow_{E,Ax}$ *and clearly* ρ *is an* $\mathcal{E}$-*unifier of* t_1 *and* t_2. □

Proposition 5 (Minimal and Complete $\mathcal{E}$-unification) *Let* $\mathcal{R} = (\Sigma, Ax, E)$ *with poset of sorts* $(\mathsf{S}, \leq)$ *be a decomposition of an equational theory* $(\Sigma,\mathcal{E})$. *Let* t, t' *be two* Σ-*terms. Then,* $U = \{\theta \mid (\mathtt{tt},\theta) \in [\![\mathtt{eq}(t,t')]\!]_{\widehat{E},Ax}\}$ *is a minimal and complete set of* $\mathcal{E}$-*unifiers for* $t = t'$, *where* $\mathtt{eq}$ *and* $\mathtt{tt}$ *are new symbols as defined in Definition 39 and* $\widehat{E} = E \cup \{\mathtt{eq}(X{:}[\mathsf{s}], X{:}[\mathsf{s}]) \to \mathtt{tt} \mid \mathsf{s} \in \mathsf{S}\}$.

Proof. *We have to prove that for each* $\mathcal{E}$-*unifier* ρ *of* t *and* t', *there is an* $\mathcal{E}$-*unifier* σ *in* U *such that* $\rho \sqsubseteq_{\mathcal{E}} \sigma$. *First, it is clear by definition of* $\mathtt{eq}$ *and* $\mathtt{tt}$ *that* $\widehat{E}$ *satisfies properties* (1)–(4) *(see Section 5.1). Let* $U^* = \{\theta \mid (\mathtt{tt},\theta) \in [\![\mathtt{eq}(t,t')]\!]^\star_{\widehat{E},Ax}\}$. *If* ρ *is an* $\mathcal{E}$-*unifier of* t *and* t', *then* $\rho \in U^*$, *since for* $\bar{t} = (t\rho)\downarrow_{E,Ax}$ *and* $\bar{t'} = (t'\rho)\downarrow_{E,Ax}$, *we have that* $\bar{t} =_{Ax} \bar{t'}$ *and* $\mathtt{eq}(\bar{t},\bar{t'}) \to_{\widehat{E},Ax} \mathtt{tt}$. *If* $\rho \in U^*$, *then* ρ *is an* $\mathcal{E}$-*unifier of* t *and* t', *since* $\mathtt{eq}(t\rho, t'\rho) \to^*_{\widehat{E},Ax} \mathtt{tt}$ *and, by properties* (1)–(4), *we have that there are* $\bar{t}, \bar{t'}$ *s.t.* $\bar{t} = (t\rho)\downarrow_{E,Ax}$, $\bar{t'} = (t'\rho)\downarrow_{E,Ax}$, *and the following rewrite step exists* $\mathtt{eq}(\bar{t},\bar{t'}) \to_{\widehat{E},Ax} \mathtt{tt}$.

Now, completeness means that for each $\mathcal{E}$-unifier ρ of t and t', there is an $\mathcal{E}$-unifier σ in U such that $\rho|_{t,t'} \sqsubseteq_{\mathcal{E}} \sigma|_{t,t'}$; and minimality means that for each $\mathcal{E}$-unifier σ in U there is no σ' in U such that $\sigma|_{t,t'} \sqsubseteq_{Ax} \sigma'|_{t,t'}$. Finally, by completeness and minimality of $[\![\mathtt{eq}(t,t')]\!]_{\widehat{E},Ax}$ w.r.t. $[\![\mathtt{eq}(t,t')]\!]^{\star}_{\widehat{E},Ax}$, we conclude completeness and minimality of U w.r.t U^*. □

Finally, it is clear that when we consider a finite variant decomposition, we obtain a decidable, finitary unification algorithm.

Corollary 8 (Finitary $\mathcal{E}$-unification) *Let $\mathcal{R} = (\Sigma, Ax, E)$ be a finite variant decomposition of an equational theory $(\Sigma, \mathcal{E})$. Then, for any two given terms t, t', $U = \{\theta \mid (\mathtt{tt}, \theta) \in [\![\mathtt{eq}(t,t')]\!]_{\widehat{E},Ax}\}$ is a* finite, minimal, and complete *set of $\mathcal{E}$-unifiers for $t = t'$, where $\widehat{E}$, $\mathtt{eq}$, and $\mathtt{tt}$ are defined in Definition 39.*

Note that the opposite does not hold: given two terms t, t' that have a finite, minimal, and complete set of $\mathcal{E}$-unifiers, the equational theory $\mathcal{R} = (\Sigma, \mathcal{E})$ may not have a finite variant decomposition (Σ, Ax, E). An example is the unification under homomorphism (or one-side distributivity), where there is a finite number of unifiers of two terms but the theory does not satisfy the finite variant property (see Example 33); the key reason for this is that the term $\mathtt{eq}(t,t')$ may have an infinite number of variants, even though there is only a finite set of most general variants of the form $(\mathtt{tt}, \theta)$.

Once we have clarified the intimate relation between variants and equational unification, we can consider how to compute a complete set of variants of a term using the variant minimality of $VN^{\circlearrowleft}_{\mathcal{R}}$. The minimality property of Definition 14 motivates the following corollary.

Corollary 9 *Let $\mathcal{R} = (\Sigma, Ax, E)$ be a decomposition of an equational theory $(\Sigma, \mathcal{E})$. For any two terms t, t' with the same top sort, the set $S = \{\theta \mid (\mathtt{tt}, \theta) \in [\![\mathtt{eq}(t,t')]\!]^{VN^{\circlearrowleft}_{\mathcal{R}}}_{\widehat{E},Ax}\}$ is a complete set of $\mathcal{E}$-unifiers for $t = t'$, where $\widehat{E}$, $\mathtt{eq}$, and $\mathtt{tt}$ are defined in Definition 39. If, in addition, $\mathcal{R}$ is a finite decomposition, then the set S is a finite set of $\mathcal{E}$-unifiers for $t = t'$.*

5.8 Applications

A first obvious application is in the area of unification algorithms. The key distinction is one between *dedicated* algorithms for a given theory T, for which a special-purpose algorithm exists, and *generic* algorithms such as folding variant narrowing, which can be applied to a wide range of theories not having a dedicated algorithm. The tradeoff is one of flexibility versus performance: a dedicated unification algorithm for a given theory T uses intimate knowledge of the theory's details and is typically much more efficient; but a special-purpose algorithm has to be developed for each such T, and combinations, though possible, are computationally expensive. By contrast, variant-based unification, being a generic method, is much more flexible and, as already mentioned and illustrated by several of our examples, if T and T' enjoy FV, $T \cup T'$ often does so as well, so that obtaining unification algorithms for combined theories is typically easy and does not require an explicit combination infrastructure. Of course, both methods should be used *together:* dedicated algorithms should be used whenever possible; variant-based unification can then be used to *extend the range of theories* that can be treated as follows: as soon as the theory Ax has a dedicated unification algorithm under minimal assumptions on Ax, we can *automatically* derive a unification algorithm for any theory $T = E \cup Ax$ such that E is confluent, terminating, sort-decreasing and coherent modulo Ax, and such an algorithm is guaranteed to be finitary if T enjoys FV.

This is exactly the approach that has been followed for analyzing cryptographic protocols modulo algebraic properties in the Maude-NPA tool [48, 113]. Such protocols can be modeled as rewrite theories $\mathcal{P} = (\Sigma, E, R)$, where the algebraic properties of the cryptographic functions are specified by equations E, and the protocol's transition rules are specified by the rewrite rules R. If E can be decomposed as $G \cup Ax$, where G is confluent, terminating, sort-decreasing and coherent modulo Ax and Ax has a finitary unification algorithm, we can perform symbolic reachability analysis on $\mathcal{P}$ by narrowing its symbolic states with the transition rules R *modulo* E, where E-unification can be carried out by folding variant narrowing with G modulo Ax and therefore does not need a dedicated E-unification algorithm. In this way, the Maude-NPA has been able to analyze a substantial collection of cryptographic protocols modulo their algebraic properties, see [48]. What makes the application of folding variant narrowing to cryptographic protocol verification interesting is its flexibility for accepting different equational theories specified by the user and its order-sorted nature, which is essential for realistic protocol specification. The following paragraph from the conclusions of a survey of algebraic properties used in cryptographic protocols [34] summarizes the actual situation in protocol verification:

> In this survey, we have identified many algebraic properties that are particularly relevant for the analysis of cryptographic protocols. ... Many recent results consider some algebraic properties. However, the existing results presented in this survey have two main weaknesses. Firstly, they are mostly theoretical: very few practical implementations enable to automatically verify protocols with algebraic properties. Secondly, in most of the cases, each paper develops an ad hoc decision procedure for a particular property.

Besides being the first practical narrowing strategy we are aware of for narrowing modulo axioms, the usefulness of folding variant narrowing goes way beyond the case of providing finitary unification algorithms for FV theories, such as those used in the Maude-NPA tool to analyze cryptographic protocols, and even beyond the case of providing a complete unification algorithm for equational theories modulo axioms. As demonstrated by its recent applications to termination algorithms modulo axioms in [42], and to algorithms for checking confluence and coherence of rewrite theories modulo axioms, such as those used in the most recent Maude CRC and ChC tools [44], computing the $E \cup Ax$-*variants* of a term may be just as important as computing $E \cup Ax$-unifiers. In particular, even for theories such as the theory of associativity, which lacks a finitary unification algorithm and *a fortiori* cannot be FV, the variants of a term (particularly in an order-sorted setting, and for terms typically used in left-hand sides of rules) can be finite quite often in practice and can provide a method to prove termination, and to check the local confluence and the coherence of rewrite rules, modulo associativity.

The key idea of why variant narrowing is important for termination, confluence, and coherence proofs, as demonstrated in [42] and in [44], is the following. Suppose that $R \cup Ax$ is a collection of rewrite rules modulo axioms Ax for which we want to prove, say, termination, or confluence, or coherence with some equations E (see [44] for an explanation of the coherence case). We may not have any tools checking such properties that can work modulo the given set of axioms Ax. For example, we are not aware of any termination tools that can handle termination modulo the commonly occurring theory ACU of associativity, commutativity and identity. What can we do? We can *decompose* Ax as a disjoint union $E \cup Ax'$, where E is confluent, terminating, sort-decreasing and coherent modulo Ax', and where we have methods to prove, e.g., termination or confluence modulo Ax'. For example, ACU decomposes in this way as $U \cup AC$ and enjoys FV. As shown in [42], we can transform $R \cup Ax$ into a semantically equivalent[5] theory $\widehat{R} \cup E \cup Ax'$, where now the set of rules is $\widehat{R} \cup E$, modulo the much simpler axioms Ax, where $\widehat{R}$ specializes each rule in R to the family of variants of their left-hand sides. If $E \cup Ax'$ has the finite variant property, we are sure that $\widehat{R}$ will be a finite set; but in practice $\widehat{R}$ can often be finite without the FV assumption. For example, Ax can be the theory A of associativity, for which unification is not even finitary. We can view A as a rule and decompose it as $A \cup \emptyset$. In an order-sorted setting, it turns out that many theories $\widehat{R} \cup A$ of practical interest can be decomposed as $(\widehat{R} \cup A) \cup \emptyset$ with $\widehat{R}$ finite, even though we know a priori that this is not possible in general, since A is not FV and does not even have a finitary unification algorithm. For example, we can often prove confluence modulo associativity of an equational specification in this way, while the usual approach to generate critical pairs may not be feasible because of the potentially infinite number of such pairs modulo A.

5.9 Related Work

Narrowing is a fundamental rewriting technique useful for many purposes, including equational unification and equational theorem proving [74], combinations of functional and logic programming [65, 69, 95], partial evaluation [4], symbolic reachability analysis of rewrite theories understood as transition systems [91], and symbolic model checking [51].

It is known that narrowing modulo axioms provides a complete unification algorithm [78], using full narrowing, which is hopelessly ineffient. The good completeness properties for standard narrowing extend naturally to similar completeness properties for narrowing modulo axioms. For effective strategies, like the *basic narrowing* strategy [74], it turns out that it is incomplete even modulo associativity-commutativity already [32], assuming the standard

[5]This semantic equivalence is very strong: that the original theory will be, e.g., terminating, confluent, and so on modulo Ax iff the transformed theory is so modulo Ax'.

definition of basic narrowing given in [73] is extended in a straightforward way to the modulo case.

With an empty set of axioms (the free case), basic narrowing is complete for unification in the sense of lifting all innermost rewriting sequences into basic narrowing sequences (see [94]). There are works such as [74, 7], which investigate conditions under which basic narrowing terminates, as well. Indeed, except for [78, 123], we are not aware of any studies about narrowing strategies in the modulo case. Furthermore, as work in [32, 123] shows, narrowing modulo axioms such as associativity-commutativity (AC) can very easily lead to non-terminating behavior.

The dependency pairs method [12] is a well-known technique for proving termination of rewriting (modulo axioms). It is extended to narrowing in [5], see [99] for termination of narrowing using the dependency pair technique. Termination of narrowing is a much harder problem than that of termination of rewriting [6] indeed. The AProVE tool [63] is a way to generate dependency pairs, based on [12, 62].

CHAPTER 6

PROTOCOL ANALYSIS MODULO COMBINATION OF THEORIES: A CASE STUDY IN MAUDE-NPA

This chapter is based on [113] and is joint work with Santiago Escobar, Catherine Meadows and José Meseguer. There is a growing interest in formal methods and tools to analyze cryptographic protocols *modulo* algebraic properties of their underlying cryptographic functions. It is well-known that an intruder who uses algebraic equivalences of such functions can mount attacks that would be impossible if the cryptographic functions did not satisfy such equivalences. In practice, however, protocols use a collection of well-known functions, whose algebraic properties can naturally be grouped together as a union of theories $E_1 \cup \ldots \cup E_n$. Reasoning symbolically modulo the algebraic properties $E_1 \cup \ldots \cup E_n$ requires performing $(E_1 \cup \ldots \cup E_n)$-unification. However, even if a unification algorithm for each individual E_i is available, this requires combining the existing algorithms by methods that are highly nondeterministic and have high computational cost. In this chapter we present an alternative method to obtain unification algorithms for combined theories based on *variant narrowing*. Although variant narrowing is less efficient at the level of a single theory E_i, it does not use any costly combination method. Furthermore, it does not require that each E_i has a dedicated unification algorithm in a tool implementation. We illustrate the use of this method in the Maude-NPA tool by means of several protocols, including a well-known protocol requiring the combination of three distinct equational theories.

In recent years there has been growing interest in the formal analysis of protocols in which the cryptographic algorithms satisfy different algebraic properties [27, 33, 88, 48]. Applications such as electronic voting, digital cash, anonymous communication, and even key distribution, all can profit from the use of such cryptosystems. Thus, a number of tools and algorithms have been developed that can analyze protocols that make use of these specialized cryptosystems [88, 85, 17, 11, 36].

Less attention has been paid to combinations of algebraic properties.

However, protocols often make use of more than one type of cryptosystem. For example, the Internet Key Exchange protocol [72] makes use of Diffie-Hellman exponentiation (for exchange of master keys), public and private key cryptography (for authentication of master keys), shared key cryptography (for exchange of session keys), and exclusive-or (used in the generation of master keys). All of these functions satisfy different equational theories. Thus it is important to understand the behavior of algebraic properties in concert as well as separately. This is especially the case for protocol analysis systems based on unification, where the problem of combining unification algorithms [13, 114] for different theories is known to be highly nondeterministic and complex, even when efficient unification algorithms exist for the individual theories, and even when the theories are disjoint (that is, share no common symbols).

The Maude-NPA protocol analysis tool, which relies on unification to perform backwards reachability analysis from insecure states, makes use of two different techniques to handle the combination problem. One is to use a general-purpose approach to unification called *variant narrowing* [56] (see also Chapter 5), which, although not as efficient as special purpose unification algorithms, can be applied to a broad class of theories that satisfy the *finite variant property* [32] (see Section 5.5). A second technique, applicable to special purpose algorithms or to theories that do not satisfy the finite variant property, uses a more general framework for combining unification algorithms.

One advantage of using variant narrowing is that there are well-known methods and tools for checking that a combination of theories has the finite variant property, including checking its local confluence and termination, and also its satisfaction of the finite variant property itself [52] (see Section 5.6). Furthermore, under appropriate assumptions some of these checks can be made modularly (see, e.g., [100] for a survey of modular confluence and termination proof methods). This makes variant narrowing easily applicable for unification combination and very suitable for experimentation with different theories. Later on, when the theory is better understood, it may be worth the effort to invest the time to implement and integrate more efficient special-purpose algorithms.

In this chapter we describe several case studies involving the use of variant narrowing to apply Maude-NPA to the analysis of several protocols involving exclusive-or and other theories; including our running example protocol that involves three theories: (i) an associative-commutative theory satisfied by symbols used in state construction, (ii) a cancellation theory for public key encryption and decryption, and (iii) the equational theory of the exclusive-or operator. This theory combination is illustrated in the analysis of a version of the Needham-Schroeder-Lowe protocol [85], denoted NSL$\oplus$, in which one of the concatenation operators is replaced by an exclusive-or [25].

In one of the other example protocols (Wired Equivalent Privacy protocol [1]), we find an attack with Maude-NPA. Then, we look at another version of the protocol which is supposed to fix that attack, which Maude-NPA indeed proves to be secure.

The rest of this chapter is organized as follows. In Section 6.1 we give an overview of Maude-NPA. In Section 6.2 we recall variant narrowing and explain how it is used in Maude-NPA, referring to Chapter 5 for some results. In Section 6.3 we describe our use of Maude-NPA on the three examples: (i) the NSL$\oplus$ protocol (in Section 6.3.1), (ii) a key exchange protocol based on exclusive-or and a central server (see Section 6.3.2) by Tatebayashi, Matsuzaki and Newman [118], and (iii) the Wired Equivalent Privacy protocol (WEP) standard by IEEE [1] using exclusive-or and other theories (see Section 6.3.3). Additionally, in Section 6.3.4 we look at a version of WEP which fixes the attack present in the original version, and prove the security of the revised protocol with Maude-NPA. In Section 6.4 we discuss related work, and in Section 6.5 we present some additional discussion.

6.1 Protocol Specification and Analysis in Maude-NPA

Given a protocol $\mathcal{P}$, we first explain how its states are modeled algebraically. The key idea is to model such states as elements of an initial algebra $T_{\Sigma_\mathcal{P}/E_\mathcal{P}}$, where $\Sigma_\mathcal{P}$ is the signature defining the sorts and function symbols for the cryptographic functions and for all the state constructor symbols and $E_\mathcal{P}$ is a set of equations specifying the *algebraic properties* of the cryptographic functions and the state constructors. Therefore, a state is an $E_\mathcal{P}$-equivalence class $[t] \in T_{\Sigma_\mathcal{P}/E_\mathcal{P}}$ with t a ground $\Sigma_\mathcal{P}$-term. However, since the number of states $T_{\Sigma_\mathcal{P}/E_\mathcal{P}}$ is in general infinite, rather than exploring concrete protocol states $[t] \in T_{\Sigma_\mathcal{P}/E_\mathcal{P}}$ we explore *symbolic state patterns* $[t(x_1, \ldots, x_n)] \in T_{\Sigma_\mathcal{P}/E_\mathcal{P}}(X)$

on the free $(\Sigma_{\mathcal{P}}, E_{\mathcal{P}})$-algebra over a set of variables X. In this way, a state pattern $[t(x_1, \ldots, x_n)]$ represents not a single concrete state but a possibly infinite set of such states, namely all the instances of the pattern $[t(x_1, \ldots, x_n)]$ where the variables $x_1, \ldots, x_n$ have been instantiated by concrete ground terms.

Let us introduce a motivating example that we will use to illustrate our approach based on exclusive–or. We use an exclusive–or version borrowed from [25] of the Needham-Schroeder-Lowe protocol [85] which we denote NSL⊕. In our analysis we use the protocol based on public key encryption, i.e., operators *pk* and *sk* satisfying the equations $pk(P, sk(P, M)) = M$ and $sk(P, pk(P, M)) = M$ and the messages are put together using concatenation and exclusive–or. Note that we use a representation of public-key encryption in which only principal P can compute $sk(P, X)$ and everyone can compute $pk(P, X)$. For exclusive–or we have the associativity and commutativity (AC) axioms for $\oplus$, plus the equations[1] $X \oplus 0 = X,\ X \oplus X = 0,\ X \oplus X \oplus Y = Y$.

1. $A \to B : pk(B, N_A; A)$

   A sends to B, encrypted under B's public key, a communication request containing a nonce N_A that has been generated by A, concatenated with its name.

2. $B \to A : pk(A, N_A; B \oplus N_B)$

   B answers with a message encrypted under A's public key, containing the nonce of A, concatenated with the exclusive–or combination of a new nonce created by B and its name.

3. $A \to B : pk(B, N_B)$

   A responds with B's nonce encrypted under B's public key.

A and B agree that they both know N_A and N_B and no one else does.

In the Maude-NPA [47, 48], a *state* in the protocol execution is a term t of sort *state*, $t \in T_{\Sigma_{\mathcal{P}}/E_{\mathcal{P}}}(X)_{state}$. A state is a multiset built by an associative and commutative union operator $\_\&\_$. Each element in the multiset can be a strand or the intruder knowledge at that state (intruder knowledge is

[1]The third equation follows from the first two. It is needed for coherence modulo AC.

wrapped by $\{\_\}$). A *strand* [58] represents the sequence of messages sent and received by a principal executing the protocol and is indicated by a sequence of messages $[msg_1^-,\ msg_2^+,\ msg_3^-, \ldots,\ msg_{k-1}^-,\ msg_k^+]$ where each msg_i is a term of sort Msg (i.e., $msg_i \in T_{\Sigma_{\mathcal{P}}}(X)_{\mathsf{Msg}}$), msg^- represents an input message, and msg^+ represents an output message. In Maude-NPA, strands evolve over time and thus we use the symbol $|$ to divide past and future in a strand, i.e., $[msg_1^\pm, \ldots, msg_{j-1}^\pm \mid msg_j^\pm, msg_{j+1}^\pm, \ldots, msg_k^\pm]$ where $msg_1^\pm, \ldots,$ $msg_{j-1}^\pm$ are the past messages, and $msg_j^\pm, msg_{j+1}^\pm, \ldots, msg_k^\pm$ are the future messages ($msg_j^\pm$ is the immediate future message). The *intruder knowledge* is represented as a multiset of facts unioned together with an associative and commutativity union operator `_,_`. There are two kinds of intruder facts: positive knowledge facts (the intruder knows m, i.e., $m{\in}\mathcal{I}$), and negative knowledge facts (the intruder *does not yet know* m but *will know it in a future state*, i.e., $m{\notin}\mathcal{I}$), where m is a message expression. Facts of the form $m{\notin}\mathcal{I}$ make sense in a backwards analysis, since one state can have $m{\in}\mathcal{I}$ and a prior state can have $m{\notin}\mathcal{I}$.

The strands associated to the three protocol steps above are given next. There are two strands, one for each principal in the protocol. Note that the first message passing $A \to B : pk(B, N_A; A)$ is represented by a message in Alice's strand sending $(pk(B, n(A, r); A))^+$, together with another message in Bob's strand that receives $(pk(B, N; A))^-$. When a principal cannot observe the contents of a concrete part of a received message (e.g., because a key is necessary to look inside), we use a generic variable for such part of the message in the strand (as with variable N of sort *Nonce* above, and similarly for X, Y below). We encourage the reader to compare the protocol in strand notation to the above presentation of the protocol. We also omit the initial and final *nil* in strands, which are needed in the tool but clutter the presentation.

- (Alice) :: r :: $[(pk(B, n(A, r); A))^+, (pk(A, n(A, r); B \oplus Y))^-, (pk(B, Y))^+]$
- (Bob) :: r' :: $[(pk(B, X; A))^-, (pk(A, X; B \oplus n(B, r')))^+, (pk(B, n(B, r')))^-]$

Note that r, r' are used for nonce generation (they are special variables handled as *unique constants* in order to obtain an infinite number of available constants).

There are also strands for initial knowledge and actions of the intruder, such as concatenation, deconcatenation, encryption, decryption, etc. Concatenation by the intruder is described by the strand $[(X)^-, (Y)^-, (X;Y)^+]$, for example. We will show the full list of intruder capabilities in Section 6.3.1.

Our protocol analysis methodology is then based on the idea of *backward reachability analysis*, where we begin with one or more state patterns corresponding to *attack states*, and want to prove or disprove that they are *unreachable* from the set of initial protocol states. In order to perform such a reachability analysis we must describe how states change as a consequence of principals performing protocol steps and of intruder actions. This can be done by describing such state changes by means of a set $R_{\mathcal{P}}$ of *rewrite rules*, so that the rewrite theory $(\Sigma_{\mathcal{P}}, E_{\mathcal{P}}, R_{\mathcal{P}})$ characterizes the behavior of protocol $\mathcal{P}$ modulo the equations $E_{\mathcal{P}}$. The following rewrite rules describe the general state transitions, where each state transition implies moving rightwards the vertical bar of one strand:

$$\begin{array}{lcl} SS \mathbin{\&} [L \mid M^-, L'] \mathbin{\&} \{M{\in}\mathcal{I}, IK\} & \to & SS \mathbin{\&} [L, M^- \mid L'] \mathbin{\&} \{IK\} \\ SS \mathbin{\&} [L \mid M^+, L'] \mathbin{\&} \{IK\} & \to & SS \mathbin{\&} [L, M^+ \mid L'] \mathbin{\&} \{IK\} \\ SS \mathbin{\&} [L \mid M^+, L'] \mathbin{\&} \{M{\notin}\mathcal{I}, IK\} & \to & SS \mathbin{\&} [L, M^+ \mid L'] \mathbin{\&} \{M{\in}\mathcal{I}, IK\} \end{array}$$

variables L, L' denote lists of input and output messages (m^+,m^-) within a strand, IK denotes a set of intruder facts ($m{\in}\mathcal{I}$,$m{\notin}\mathcal{I}$), and SS denotes a set of strands. An unbounded number of sessions is handled by another rewrite rule introducing an extra strand $[m_1^{\pm}, \ldots, m_{j-1}^{\pm} \mid m_j^+, msg_{j+1}^{\pm}, \ldots, m_k^{\pm}]$ for an intruder knowledge fact of the form $m_j{\in}\mathcal{I}$. See [47] for further information.

The way to analyze *backwards* reachability is then relatively easy, namely to run the protocol "in reverse." This can be achieved by using the set of rules $R_{\mathcal{P}}^{-1}$, where $v \longrightarrow u$ is in $R_{\mathcal{P}}^{-1}$ iff $u \longrightarrow v$ is in $R_{\mathcal{P}}$. Reachability analysis can be performed *symbolically*, not on concrete states but on symbolic state patterns $[t(x_1, \ldots, x_n)]$ by means of *narrowing modulo $E_{\mathcal{P}}$* (see Definition 10 in Section 5.3, and [78, 91]).

$E_{\mathcal{P}}$-unification precisely models all the different ways in which an intruder could exploit the algebraic properties $E_{\mathcal{P}}$ of $\mathcal{P}$ to break the protocol; therefore, if an initial state can be shown unreachable by backwards reachability analysis modulo $E_{\mathcal{P}}$ from an attack state pattern, this ensures that, even if the intruder uses the algebraic properties $E_{\mathcal{P}}$, the attack cannot be mounted. Therefore, efficient support for $E_{\mathcal{P}}$-unification is a crucial feature of symbolic reachability analysis of protocols modulo their algebraic properties $E_{\mathcal{P}}$.

6.2 A Unification Algorithm for $XOR \cup pk\text{-}sk \cup AC$

In general, combining unification algorithms for a theory $E = E_1 \cup E_2 \cup \ldots \cup E_n$ is computationally quite expensive, and typically assumes that the symbols in E_i and E_j are pairwise disjoint for each $i \neq j$. This is due to the substantial amount of non–determinism involved in the inference systems supporting such combinations (see [13]). In our NSL⊕ example, $E = E_1 \cup E_2 \cup E_3$, where E_1 is the *XOR* theory, E_2 is the theory *pk-sk* given by the two public key encryption equations $pk(K, sk(K, M)) = M$ and $sk(K, pk(K, M)) = M$, and E_3 is the *AC* theory for each of the state constructors `_,_` and `_&_`, explained in Section 6.1. To further complicate the matter, we need to combine not just *untyped* unification algorithms, but typed, and more precisely *order-sorted* ones.

Fortunately, the variant–narrowing–based approach that we use in this chapter avoids all these difficulties by obtaining the ($XOR \cup pk\text{-}sk \cup AC$)-unification algorithm as an instance of the *variant narrowing* methodology supported by Maude-NPA. The point is that if an equational theory E has the *finite variant property* [32], then a *finitary* E-unification algorithm can be obtained by *variant narrowing* [56, 54], as already explained in Section 5.4.1. In our case, the equations in the theory *pk-sk* are confluent and terminating and, furthermore, have the finite variant property. Likewise, the equations in the *XOR* theory presented in Section 6.1 are confluent, terminating and coherent modulo the *AC* axioms of ⊕ and also have the finite variant property. Finally, the theory of *AC* for the state-building constructors `_,_` and `_&_` is of course finitary and can be viewed as a trivial case of a theory with the finite variant property (decomposed with no rules and only axioms). Note that all these three equational theories are disjoint, i.e., they do not share any symbols. The good news is that the following disjoint union theory $XOR \cup pk\text{-}sk \cup AC$ with $\Sigma_{NSL\oplus}$ being the entire (order-sorted) signature of our NSL⊕ protocol example is also confluent, terminating and coherent modulo the *AC* axioms[2], and satisfies the finite variant property:

[2]All these conditions are easily checkable. Indeed, coherence modulo the combined AC axioms is immediate, and we can use standard methods and tools to check the local confluence and termination of the combined theory; similarly, the method described in [52] can be used to check the finite variant property of the combined theory. Alternatively, one can use *modular* methods to check that a combined theory satisfies all these properties under certain assumptions: see [100] for a good survey of modularity results for confluence and termination. Likewise, the finite variant property can also be checked modularly under

1. *Rules*:

    - $pk(K, sk(K, M)) = M$, $sk(K, pk(K, M)) = M$,
    - $X \oplus 0 = X$, $X \oplus X = 0$, $X \oplus X \oplus Y = Y$,

2. *Axioms*: AC for $\oplus$, AC for $\_,\_$ and AC for $\_\&\_$

Therefore, Maude-NPA can analyze the NSL$\oplus$ protocol using variant narrowing. In the following we will recall the notions of Chapter 5 that are crucial for the use of variant narrowing in this chapter, to allow the reader to read this chapter without having to have read the previous chapter in full.

Let us now motivate the key notions in an intuitive fashion. All definitions linked below are from Chapter 5.

- The decomposition (Σ, Ax, E) of an equational theory $(\Sigma, \mathcal{E})$ is such that $\mathcal{E} = E \cup Ax$, with a number of extra conditions to assure that the decomposition behaves like the original equational theory when executed, see Definition 2 for details.

- An E, Ax-variant of a term t is a pair (t', σ) such that t' is the E, Ax-canonical form of $t\sigma$.

- The variant semantics of a term, $[\![t]\!]^{\star}_{E,Ax}$, is the set of all normalized variants of t, see Definition 5.

- For comparing variants, we write $(t_1, \theta_1) \sqsubseteq_{E,Ax} (t_2, \theta_2)$ to denote that variant (t_2, θ_2) is *more general* than variant (t_1, θ_1).

- We call $[\![t]\!]_{E,Ax}$ the most general and complete variant semantics of t when: (i) it is a subset of the variant semantics $[\![t]\!]^{\star}_{E,Ax}$ from above, and (ii) every possible variant of t is an instance of at least one element of $[\![t]\!]_{E,Ax}$, see Definition 9 for all the details.

- Note that, by definition, all the substitutions in $[\![t]\!]_{E,Ax}$ are E,Ax-normalized. Moreover, $[\![t]\!]_{E,Ax}$ is unique up to equivalence modulo Ax and provides a very succinct description of $[\![t]\!]^{\star}_{E,Ax}$.

- We get a minimal and complete $\mathcal{E}$-unification procedure by intersecting the variant semantics of the terms in question, the details are given in Proposition 5.

appropriate assumptions, but a discussion of this topic is beyond the scope of this chapter.

- The finite variant property for an equational theory decomposed as $E \cup Ax$ means that for each term t, $[\![t]\!]_{E,Ax}$ is a finite set. See details in Section 5.5

- For a theory with the finite variant property we have a finitary $\mathcal{E}$-unification algorithm (giving a finite, minimal, and complete set of unifiers modulo the theory), by the intersection of the most general and complete variant semantics of the terms of interest. This is shown in detail in Corollary 8.

Let us now look at an example using these notions, in particular that of a variant semantics $[\![t]\!]^{\star}_{E,Ax}$.

Example 34 *Let us consider the equational theory $XOR \cup pk\text{-}sk$, which, together with AC for* $\_,\_$ *and* $\_\&\_$ *is used for our NSL$\oplus$ protocol presented in Section 6.1. This equational theory is relevant because none of our previously defined unification procedures is directly applicable to it, e.g. unification algorithms for exclusive–or such as [68] do not directly apply if extra equations are added.*

For (Σ, Ax, E) a decomposition of $XOR \cup pk\text{-}sk$, and for terms $t = M \oplus sk(K, pk(K, M))$ and $s = X \oplus sk(K, pk(K, Y))$, we have that $[\![t]\!]^{\star}_{E,Ax} = \{(0, id), \ldots\}$ and

$$\begin{aligned}
[\![s]\!]^{\star}_{E,Ax} = \{&(X \oplus Y, id),\\
&(Z, \{X \mapsto 0, Y \mapsto Z\}), (Z, \{X \mapsto Z, Y \mapsto 0\}),\\
&(Z, \{X \mapsto Z \oplus U, Y \mapsto U\}), (Z, \{X \mapsto U, Y \mapsto Z \oplus U\}),\\
&(0, \{X \mapsto U, Y \mapsto U\}), (Z_1 \oplus Z_2, \{X \mapsto U \oplus Z_1, Y \mapsto U \oplus Z_2\}),\\
&(0, \{X \mapsto V \oplus W, Y \mapsto V \oplus W\}), \ldots\}
\end{aligned}$$

Note the similarities, and differences, between this example and Example 2.

A follow-up example looks at the most general and complete variant semantics $[\![t]\!]_{E,Ax}$.

Example 35 *Continuing Example 34 it is obvious that the following variants* *are most general w.r.t.* $\sqsubseteq_{E,Ax}$*:* $[\![t]\!]_{E,Ax} = \{(0, id)\}$ *and*

$$\begin{aligned}[\![s]\!]_{E,Ax} = \{&(X \oplus Y, id),\\ &(Z, \{X \mapsto 0, Y \mapsto Z\}), (Z, \{X \mapsto Z, Y \mapsto 0\}),\\ &(Z, \{X \mapsto Z \oplus U, Y \mapsto U\}), (Z, \{X \mapsto U, Y \mapsto Z \oplus U\}),\\ &(0, \{X \mapsto U, Y \mapsto U\}), (Z_1 \oplus Z_2, \{X \mapsto U \oplus Z_1, Y \mapsto U \oplus Z_2\})\}.\end{aligned}$$

Actually, this example is just Example 4 revisited.

Currently, Maude-NPA restricts itself to a subset of theories satisfying the finite variant property:

1. The axioms Ax can declare some binary operators in Σ to be commutative (with the `comm` attribute), or associative-commutative (with the `assoc` and `comm` attributes).

2. The set of rewrite rules E is *strongly right irreducible*, that is no instance of the right-hand side of a rule in E by a normalized substitution can be further simplified by the application the equations in E modulo Ax.

The reasons for restricting ourselves in this way is for efficiency and ease of implementation. Maude currently supports unification modulo commutative and associative-commutative theories, as well as syntactic unification, so this is what drives our choice of Ax. Furthermore, the restriction of E to strongly right irreducible theories means that the depth of the narrowing tree is bounded by the number of symbols in a term. Moreover, many of the finite variant theories that arise in cryptographic protocol analysis satisfy strong right irreducibility. These include encryption-decryption cancellation, exclusive-or, and modular exponentiation. The major exception is Abelian groups (other than those described by exclusive-or). We are currently working on implementing full variant narrowing in Maude-NPA to handle these and other cases not currently covered by strong right irreducibility.

6.3 Finding attacks modulo $XOR \cup pk\text{-}sk \cup AC$ in Maude-NPA

Now we present the three different protocol case studies: NSL⊕, TMN, and WEP. The following subsections deal with each of the protocols in that order, but note that there are two subsections for WEP: one which finds the attack in the original protocol and another that verifies the security of a modified version of the protocol. That last subsection also includes explanation about how the Maude-NPA can prove the security of a protocol in the first place.

6.3.1 NSL$\oplus$

We have analyzed the NSL$\oplus$ protocol presented in Section 6.1 modulo its equational theory $\mathit{XOR}\cup \mathit{pk}\text{-}\mathit{sk}\cup \mathit{AC}$ in Maude-NPA using variant narrowing.

We now explain in more detail all the operations available to the intruder. Its capabilities are all given in strand notation. Note that we are omitting the position marker | which is assumed to be at the beginning.

(s1) $[(X)^-, (Y)^-, (X;Y)^+]$ Concatenation

(s2) $[(X;Y)^-, (X)^+]$ Left-deconcatenation

(s3) $[(X;Y)^-, (Y)^+]$ Right-deconcatenation

(s4) $[(X)^-, (Y)^-, (X \oplus Y)^+]$ Exclusive–or

(s6) $[(X)^-, (sk(i,X))^+]$ Encryption with i's private key

(s7) $[(X)^-, (pk(A,X))^+]$ Encryption with any public key

(s8) $[(0)^+]$ Generate the exclusive–or neutral element

(s9) $[(A)^+]$ Generate any principal's name.

The attack state pattern from which we start the backwards narrowing search in this example is given by one strand, representing Bob (b) wanting to communicate with Alice (a)

$$:: r :: [(pk(b,X;a))^-, (pk(a,X;b \oplus n(b,r)))^+, (pk(b,n(b,r)))^- | nil]$$

together with requiring the intruder (i) to have learned Bob's nonce, i.e., $n(b,r) {\in} \mathcal{I}$. What this represents is an attack in which Bob has properly executed the protocol and believes to be talking to Alice, while the intruder has obtained the nonce that Bob created and considers a secret shared between Alice and him.

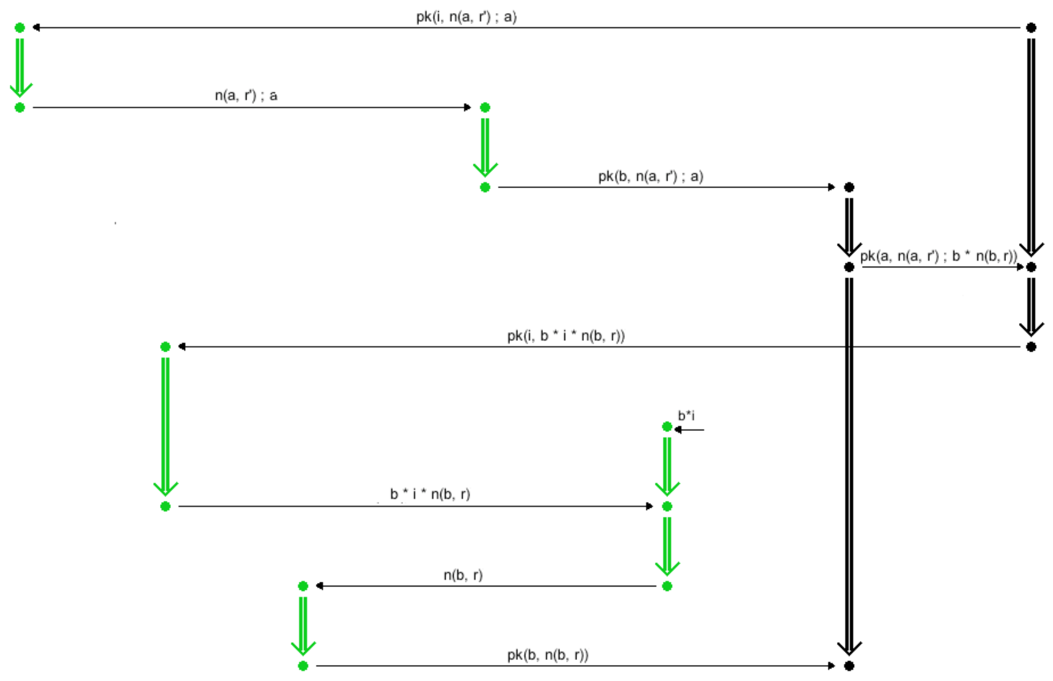


Figure 6.1: Pictorial representation of the initial state, leading to an attack on the NSL⊕ protocol

See Figure 6.1 for a pictorial representation of the strand space and messages sent and received, depicting the attack found by Maude-NPA. This attack agrees with the one described in [25]. The figure has been created with the help of the Maude-NPA GUI [111], with the exclusive–or symbol $\oplus$ textually represented as $*$ in the figure.

6.3.2 TMN - Key Exchange Protocol

The TMN protocol is a symmetric key distribution protocol [118, 86] initially proposed by Tatebayashi, Matsuzaki and Newman. The purpose is for A and B to share a key K_B. The server also checks that neither K_A nor K_B have been used in prior sessions. The Avispa [11] and XOR-ProVerif [81] (based on ProVerif [17]) tools are both able to deal with this protocol as well, according to [83].

The protocol has three principals, which are Alice (A), Bob (B) and the server (S). There is only a single public key and private key pair in use which belongs to the server, so we assume that the server is the only one that can decrypt messages that are encrypted by it. We will write $enc(\_)$ for that encryption using the public key of S. The fresh symmetric keys that are being exchanged are K_A and K_B. Here is the protocol:

1. $A \rightarrow S : B, enc(K_A)$

   A sends to S the pair containing the name of the intended communication partner B, and, encrypted by S's public key, the freshly chosen symmetric key K_A.

2. $S \rightarrow B : A$

   S sends to B a notification that A wants to establish a shared key.

3. $B \rightarrow S : A, enc(K_B)$

   B answers to S with the pair of the name A and, encrypted under S's public key, its own freshly chosen symmetric key K_B.

4. $S \rightarrow A : B, K_B \oplus K_A$

   S sends to A the pair with the name of B, and the exclusive-or combination of the two keys provided by A and B, i.e., $K_B \oplus K_A$.

At the end of this protocol, A and B share the fresh key K_B, as A can compute $K_B = (K_B \oplus K_A) \oplus K_A$, as it knows K_A. There is an attack on this protocol by an intruder I that can be described as follows:

1. $A \rightarrow S : B, enc(K_A)$

   A starts a normal session with B.

2. $S \rightarrow I : A$

   I intercepts the message sent by S that was intended for B.

3. $I(B) \rightarrow S : A, enc(K_I)$

   I impersonates B and sends his own symmetric key to the server.

4. $S \rightarrow I : B, K_I \oplus K_A$

   Finally, the intruder intercepts the message intended for A, including $K_I \oplus K_A$, and as the intruder knows K_I, he can find K_A by computing $K_A = (K_I \oplus K_A) \oplus K_I$. Finally, I can re-transmit the pair $B, K_I \oplus K_A$ to A.

In our notation for Maude-NPA we have three strands, one for each of the principals. Note that K_A will be represented by a nonce $n(A, r)$, similarly for K_B. The variables NA and NB are used to capture unknown nonces and the pairing is made explicit using $pair(\_, \_)$:

- (Alice) $:: r :: [(pair(B, enc(n(A, r))))^+, (pair(B, n(A, r) \oplus NB))^-]$
- (Bob) $:: r' :: [(A)^-, (pair(A, enc(n(B, r'))))^+]$
- (Server) $\begin{array}{l}[(pair(B, enc(NA)))^-, (A)^+,\\ (pair(A, enc(NB)))^-, (pair(B, NA \oplus NB))^+]\end{array}$

Let us show the capabilities of the intruder, and note that we are omitting any leading or trailing nil and the position marker | which is assumed at the beginning:

(s1) $[(X)^-, (Y)^-, (X \oplus Y)^+]$ Exclusive–or

(s2) $[(pair(X, Y))^-, (X)^+]$ Left–projection

(s3) $[(pair(X, Y))^-, (Y)^+]$ Right–projection

(s4) $[(X)^-, (Y)^-, (pair(X, Y))^+]$ Pairing

(s5) $:: r :: [(n(i, r))^+]$ Generate a key for i

(s6) $[(N)^-, (enc(N))^+]$ Encrypt with S public key

(s7) $[(A)^+]$ Generate any principal's name.

To find the attack that is listed above, we start the Maude-NPA with the following attack state pattern, from which we start the backwards narrowing search, representing Alice's (a) attempt to communicate with Bob (b) where the intruder is able to learn the nonce (i.e., key) of Alice, i.e., $n(a, r) \in \mathcal{I}$:

$$:: r :: [(pair(b, enc(n(a, r))))^+, (pair(b, n(a, r) \oplus NB))^- | nil]$$

See Figure 6.2 for a pictorial representation of the strand space and messages sent and received, depicting the attack found by Maude-NPA, which is essentially the attack described above. The figure has been created with the help of the Maude-NPA GUI [111], with the exclusive–or symbol $\oplus$ textually represented as $*$ in the figure.

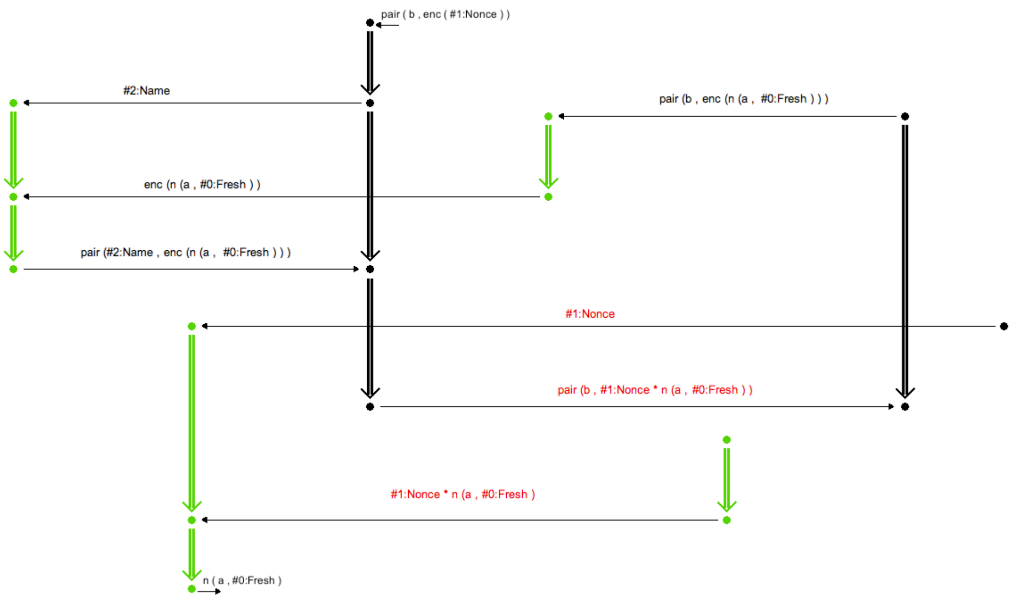

pair (b , enc (#1:Nonce))
#2:Name
pair (b , enc (n (a , #0:Fresh)))
enc (n (a , #0:Fresh))
pair (#2:Name , enc (n (a , #0:Fresh)))
#1:Nonce
pair (b , #1:Nonce * n (a , #0:Fresh))
#1:Nonce * n (a , #0:Fresh)
n (a , #0:Fresh)

Figure 6.2: Pictorial representation of the initial state, leading to an attack on the TMN protocol

6.3.3 Wired Equivalent Privacy Protocol

The Wired Equivalent Privacy Protocol (WEP) is defined in [1]. The purpose of WEP is to protect data during wireless transmission. The protocol encrypts a message M to be sent from principal A to another principal B. It is a single step protocol with no response:

1. $A \rightarrow B : V, ([M, C(M)] \oplus RC4(V, KAB))$

   A sends a vector V, paired with the exclusive-or combination of the message with a checksum (i.e., $[M, C(M)]$) and $RC4(V, K_{AB})$ where RC4 is a public one-way algorithm using the initial vector V and a symmetric key K_{AB}.

   The receiver can then compute the message M as it knows the key K_{AB} and gets V, so it can compute $RC4(V, K_{AB})$ and then make use of the exclusive-or cancellation property to get $[M, C(M)]$. Verification with the checksum yields the message M.

The purpose is for no one, except for A and B, to be able to know M. It turns out there is an attack, using the fact that V can be reused and one message M_1 can be sent to different recipients and thus endanger the secrecy of a message M_2 sent later:

1. $A \rightarrow B : V, ([M_1, C(M_1)] \oplus RC4(V, K_{AB}))$

   A sends the same message M_1 to B.

2. $A \rightarrow I : V, ([M_1, C(M_1)] \oplus RC4(V, K_{AI}))$

   Then, A send the same message M_1 to I. Now, I is able to determine $RC4(V, K_{AI})$ as K_{AI} is known. Then, by exclusive-or combining the two payloads of 1 and 2 and adding $RC4(V, K_{AI})$ it gets $RC4(V, K_{AB})$ as the result of the term $([M_1, C(M_1)] \oplus RC4(V, K_{AB})) \oplus ([M_1, C(M_1)] \oplus RC4(V, K_{AI})) \oplus RC4(V, K_{AI})$. With that in hand, all further messages that A sends to B can be accessed by the intruder.

3. $A \rightarrow I : V, ([M_2, C(M_2)] \oplus RC4(V, K_{AB}))$

   Intercepting this message intended for B, the intruder can indeed compute $[M_2, C(M_2)]$ and thus M_2 by simple exclusive-or combination: $([M_2, C(M_2)] \oplus RC4(V, K_{AB})) \oplus RC4(V, K_{AB})$.

In our strand notation for Maude-NPA we give two possible strands, one with just the single message as described above, and another one sending the same message twice (similarly to [83]). Note that the secret messages will be represented by nonces created by the sender, so we can be sure no one knows them ahead of time, or can guess them. We also make the pairing of a vector and the remainder of the message explicit, by using $pair(\_, \_)$:

- (Alice) :: r :: $[(pair(V, ([n(A, r), c(n(A, r))] \oplus rc4(V, k(A, B)))))^+]$
- (Alice) :: r' :: $[\ (pair(V, ([n(A, r'), c(n(A, r'))] \oplus rc4(V, k(A, B)))))^+,$
  $(pair(V, ([n(A, r'), c(n(A, r'))] \oplus rc4(V, k(A, C)))))^+]$

Let us show the capabilities of the intruder, and note that we are omitting any leading or trailing nil and the position marker | which is assumed to be placed at the beginning:

(s1) $[(X)^-, (Y)^-, (X \oplus Y)^+]$ Exclusive–or

(s2) $[(A)^-, (k(A, i))^+]$ Symmetric keys for all principals with intruder i

(s3) $[(pair(V, X))^-, (V)^+]$ Left–projection

(s4) $[(pair(V, X))^-, (X)^+]$ Right–projection

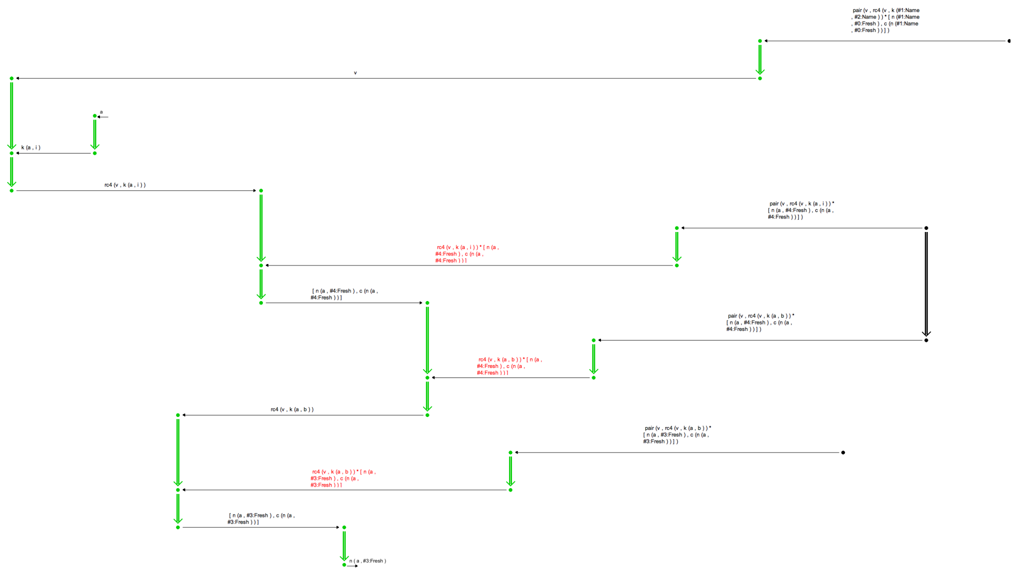

pair (v , rc4 (v , k (#1:Name , #2:Name)) * [n (#1:Name , #0:Fresh) , c (n (#1:Name , #0:Fresh))])
v
a
k (a , i)
rc4 (v , k (a , i))
pair (v , rc4 (v , k (a , i)) * [n (a , #4:Fresh) , c (n (a , #4:Fresh))])
rc4 (v , k (a , i)) * [n (a , #4:Fresh) , c (n (a , #4:Fresh))]
[n (a , #4:Fresh) , c (n (a , #4:Fresh))]
pair (v , rc4 (v , k (a , b)) * [n (a , #4:Fresh) , c (n (a , #4:Fresh))])
rc4 (v , k (a , b)) * [n (a , #4:Fresh) , c (n (a , #4:Fresh))]
rc4 (v , k (a , b))
pair (v , rc4 (v , k (a , b)) * [n (a , #3:Fresh) , c (n (a , #3:Fresh))])
rc4 (v , k (a , b)) * [n (a , #3:Fresh) , c (n (a , #3:Fresh))]
[n (a , #3:Fresh) , c (n (a , #3:Fresh))]
n (a , #3:Fresh)

Figure 6.3: Pictorial representation of the initial state, leading to an attack on the WEP protocol

(s5) $[(V)^-, (k(A,B))^-, (rc4(V,k(A,B)))^+]$ RC4 generation

(s6) $[(N)^-, (c(N))^+]$ Checksum generation

(s7) $[([N, c(N')])^-, (N)^+]$ Message extraction

(s8) $[(A)^+]$ Generate any principal's name

To find the attack that is listed above, we start the Maude-NPA with the following attack state pattern, from which we start the backwards narrowing search, representing Alice's (a) message to Bob (b) where the intruder is able to learn the nonce (i.e., message) that Alice is sending, i.e., $n(a,r) {\in} \mathcal{I}$:

$$:: r :: [(pair(v, ([n(a,r), c(n(a,r))] \oplus rc4(v, k(a,b)))))^+ | nil]$$

The Maude-NPA is able to find the attack described above in 20 backwards steps for this initial state. The attack is presented in Figure 6.3. That figure has been created with the help of the Maude-NPA GUI [111], with the exclusive–or symbol $\oplus$ textually represented as $*$ in the figure.

6.3.4 Fixed Version of Wired Equivalent Privacy Protocol

It is possible to fix the WEP protocol quite easily. The suggested fix is to change the initial vector that is used for each message. The fix has been proposed before, in [83], and the Maude-NPA is able to verify the security of the fixed protocol, which is demonstrated in this section.

In the attempt to verify protocols the Maude-NPA tool explores the whole state space of a protocol, starting its backwards exploration from a symbolic description of a set of potential attack states and searching for an initial state of the protocol. In general, this search state space is infinite and the search would not terminate, and thus no decision about security of protocols could be made. To deal with this issue the Maude-NPA is equipped with powerful state space reduction techniques [49, 50], that enable the tool to cut the state space down. The usual infinite state space actually gets reduced to a finite state space most of the time but not always.[3] In practice, this reduced state space can often be fully explored in a moderate amount of time.

Of course, for these state space reductions to be worthwhile they need to ensure that completeness is maintained, as otherwise the absence of attacks in the reduced state space allows no conclusions about the existence or absence of attacks in the full state space. The main, and oldest, state space reduction technique in the tool is that of grammars. Grammars are used to cut down the search space by identifying non-terminating search paths. Usually the grammars alone are able to reduce the infinite state space to a finite one.

There are further techniques that remove unreachable states, which eliminates the cost of exploring them, e.g., (i) the idea of public data, (ii) limiting the dynamic introduction of new strands, (iii) prioritizing input messages, and (iv) detecting inconsistent states early. There are also techniques for removing redundant states, namely, (v) a subsumption partial order reduction, and (vi) the super-lazy intruder [49, 50].

These techniques are needed for the efficiency as well as for achieving full verification. As we will see in this example, no attack is found and the (reduced) state space is finite and completely explored by the tool. By the completeness of the state space reductions we can then conclude that there actually is no attack and the protocol is proven secure modulo the given

[3]Since the Maude-NPA analyzes protocols assuming an *unbounded* number of sessions, such analysis is in general undecidable.

algebraic properties.

As described above, a fix for this protocol is to require all the vectors that are used to be new, i.e., no vector can be re-used. We will only present the protocol in strand notation, the change needed for the textbook notation is then obvious. This new protocol definition uses a nonce as the vector and thus no two vectors used by the participants will ever be the same.

- (Alice) $$:: r, r2 :: [(\ \ pair(vec(n(A,r2),([n(A,r),c(n(A,r))] \oplus rc4(vec(n(A,r2),k(A,B)))))^{+}]$$

- (Alice) $$\begin{aligned} :: r', r3, r4 :: [(\ \ & pair(vec(n(A,r3),([n(A,r'),c(n(A,r'))] \\ & \oplus rc4(vec(n(A,r3),k(A,B)))))^{+}, \\ (\ \ & pair(vec(n(A,r4),([n(A,r'),c(n(A,r'))] \\ & \oplus rc4(vec(n(A,r4),k(A,C)))))^{+}] \end{aligned}$$

Let us show the additional capability of the intruder, and note that we are omitting any leading or trailing *nil* and the position marker | which is assumed to be placed at the beginning:

(s9) $:: r :: [(vec(n(i,r)))^{+}]$ Generate a fresh vector

We start the Maude-NPA with the same attack state pattern as in the basic version of the protocol. Note how v still just represents any vector, which we do not need to specify as a nonce. This pattern represents Alice's (a) message to Bob (b) where the intruder is able to learn the nonce (i.e., message) that Alice is sending, i.e., $n(a,r) \in \mathcal{I}$:

$$:: r :: [(pair(v,([n(a,r),c(n(a,r))] \oplus rc4(v,k(a,b)))))^{+}|nil]$$

The search started from the above attack state pattern results in a finite search space, without finding an attack. Therefore, the Maude-NPA has verified that this fixed version of WEP is indeed secure.

In [83] it is shown that both the Avispa [11] and XOR-ProVerif [81] (based on ProVerif [17]) tools are able to deal with this protocol, finding the attack in the initial version presented in the previous subsection and showing its absence in the fixed version presented here.

6.4 Related Work

There is a substantial amount of research on formal verification of cryptographic protocols. Much of it abstracts away from any equational theories obeyed by the cryptographic operators, but there is a growing amount of work addressing this problem. The earliest was the NRL Protocol Analyzer [88], which, like Maude-NPA, was based on unification and backwards search, implemented via narrowing over confluent equational theories. This was sufficient to handle, for example, the cancellation of encryption and decryption, although there were many theories of interest it did not address, such as exclusive-or and other Abelian group operators.

More recently, tools have begun to offer support for specification and, to some degree, analysis of protocols involving equational theories. These tools include, for example, ProVerif [17], OFMC [15], and CL-Atse [121]. Both OFMC and CL-Atse work in the bounded session model, while ProVerif uses abstraction and unbounded sessions. Both OFMC and CL-Atse support exclusive-or and Diffie-Hellman exponentiation. ProVerif can also be used to analyze these, but the equational theories it is known to work well with are more limited, e.g., not supporting associativity-commutativity or Diffie-Hellman exponentiation. However, Küsters and Truderung [81, 82] have developed algorithms that can, under certain restrictions, translate protocols using exclusive-or or Diffie-Hellman exponentiation to protocols that can be analyzed by ProVerif in a free algebra model; for exclusive-or they can handle protocols satisfying the $\oplus$-linearity property. According to a study by Lafourcade et al. [83], this produces analysis times that are only slightly slower than analyses by OFMC and CL-Atse, mainly because of the translation time.

There is also a growing amount of theoretical work on cryptographic protocol analysis using equational theories, e.g. [3, 26, 21, 28, 16]. This concentrates on the decidability of problems of interest to cryptographic protocol analysis, such as deducibility, which means that it is possible (e.g. for an intruder) to deduce a term from a set of terms, and static equivalence, which means that an intruder cannot tell the difference between two sets of terms. However, there is much less work on the combination of different theories, although Arnaud, Cortier, and Delaune [33] have considered the problem in terms of decidability of the problem for combination of disjoint theories, showing that if any two disjoint theories have decidable static equivalence problems, then so does their combination. More recently Chevalier and Rusinowitch analyze the security of cryptographic protocols via constraint systems and have also studied composition of theories. In [27], they give a general method for combining disjoint theories that is based on the Baader-Schulz combination algorithm for unification algorithms for different theories [13]. This can be thought of as a constraint-based analogue of the Maude-NPA combination framework, which is also based on the Baader-Schulz combination algorithm [13].

6.5 Discussion

To gain high assurance about cryptographic protocols using formal methods requires reasoning modulo the algebraic properties of the underlying cryptographic functions. In symbolic analyses this typically necessitates performing unification *modulo* such algebraic properties. However, since a protocol may use a variety of different functions —so that different protocols typically require reasoning modulo different theories— it is unrealistic to expect that a fixed set of unification algorithms will suffice for such analyses. That is, *combination methods* that obtain unification algorithm for a composition of theories out of a family of such algorithm for each of them, are unavoidable. Standard methods for obtaining a unification algorithm for a combined theory $E_1 \cup \ldots \cup E_n$ [13] are computationally costly due to the high degree of non-determinism in the combination method; furthermore, they require the existence of a unification algorithm for each individual theory E_i, which in practice may not be available in a tool's infrastructure. In Chapter 6 we have proposed an alternative method based on *folding variant narrowing* to obtain a $(E_1 \cup \ldots \cup E_n)$-unification algorithm under simpler requirements. Specifically, dedicated implementations of unification algorithms for each of the theories E_i are not needed: in our example, only a dedicated AC-unification algorithm was used: no dedicated algorithms for *XOR* or *pk-sk* were needed. Furthermore, even though narrowing is less efficient than a dedicated algorithm for each individual theory E_i, the costly computational overhead of a standard combination method is avoided. The case studies presented have shown that variant narrowing, as supported by the Maude-NPA, is indeed an effective method to deal with nontrivial combinations of equational theories; and for analyzing many protocols with even a modest infrastructure of built-in unification algorithms.

We should emphasize that standard combination methods such as those described in [13], and the alternative variant narrowing method presented here are not rival methods. Instead they are highly *complementary* methods which, when used *in tandem*, allow a tool to analyze a much wider range of protocols than those analyzable by each method in isolation. Let us use our example theory $XOR \cup pk\text{-}sk \cup AC$ to illustrate this important point. Variant narrowing decomposed this combined theory into: (i) three rewrite rules for XOR and two rewrite rules for *pk-sk* plus, (ii) three instances of AC: one for $\oplus$, another for $\_,\_$ and another for $\_\&\_$. That is, variant narrowing with the rules in (i) was performed *modulo* the axioms in (ii). But the axioms in (ii) are themselves a *combined theory* (in fact, also combined with all the other function symbols in the protocol specification as free function symbols). The Maude infrastructure used by Maude-NPA has in fact used an order-sorted version of a standard combination method in the style of [13] to support unification with the combined axioms of (ii). Therefore, the advantages of using standard combination methods and variant narrowing in tandem are the following:

1. A given tool infrastructure can only have a finite number of predefined (finitary) unification algorithms for, say, theories $T_1, \ldots, T_k$; however, it should also be able to support any combination of such built-in theories by a standard combination method.

2. A given protocol may require performing unification modulo a combination of theories $E_1 \cup \ldots \cup E_n$, but some of the E_i may not belong to the library $T_1, \ldots, T_k$, so that the standard combination method cannot be used.

3. However, if $E_1 \cup \ldots \cup E_n$ can be *refactored* as a theory decomposition (Σ, B, R) that: (i) it has the finite variant property; and (ii) B is a combination of the theories $T_1, \ldots, T_k$ supported by the current library, then a *finitary* $(E_1 \cup \ldots \cup E_n)$-unification algorithm can be obtained by variant narrowing.

CHAPTER 7

CONCLUSIONS AND FUTURE WORK

This chapter first presents the conclusions of this dissertation, followed by a discussion of future research directions.

7.1 Conclusions

Three aspects of browser security have been addressed in this dissertation: (i) the *machine-to-user communication*, (ii) *internal browser security concerns* and (iii) the *machine-to-machine communication*. First, we deal with the machine-to-user communication for two browsers, IE and IBOS, by showing a methodology of creating models for browsers that are amenable to formal analysis of security properties in areas (i) and (ii). Regarding (i), this methodology does find many possible attacks in IE for both the address bar and the status bar. It also shows the absence of attacks for the address bar of IBOS. As for (ii), we use the same methodology to look at internal browser security in the case of IBOS and the same origin policy (SOP). There we check that the SOP holds and are able to find a bug in the display memory management.

For (iii), the machine-to-machine communication, we look into browser-generic cryptographic protocols, and are able to analyze a number of them *modulo their algebraic properties*. We find bugs in some protocols while showing the security of others. To be able to analyze such protocols modulo their algebraic properties we use a new generic method we have developed in this dissertation which allows the effective computation of unifiers modulo equational theories, based on a new narrowing strategy, which we call *folding variant narrowing*. Related to that we have presented a new automatic method for checking the finite variant property for a given theory and paved the way for further applications based on this new narrowing strategy.

Comparing to existing methods, like the ProVerif tool [17] and its adaptations for analysis modulo specific algebraic theories like exclusive-or or Diffie-Hellman [81, 82], it turns out that using the unification based on our new narrowing strategy in the Maude-NPA tool [47, 48] is quite more general. It simply does not need to be specifically tailored to the exact theory modulo which the protocol is supposed to work.

The key advancements in this dissertation are: (i) the new methodology for formal modeling and analysis of web browsers; (ii) the case studies with that methodology on IE GUI and IBOS GUI and internals; (iii) a new narrowing strategy called folding variant narrowing that is used for the computation of unification modulo axioms; (iv) an automatic check for the finite variant property; (v) cryptographic protocol case studies based on the new unification algorithm; and (vi) paving the way for further automated deduction uses of folding variant narrowing.

7.1.1 Browser Analysis Conclusions

GUI logic flaws are a real and pressing security problem – these flaws can be exploited to lure even security-conscious users to visit malicious web pages. We have formulated GUI logic correctness as a new research problem, and have proposed a systematic approach to pro-actively uncover logic flaws in browser GUI design/implementation that lead to spoofing attacks.

Specifically, based upon an in-depth study of the logic of key subsets of IE source code, we have developed a formal model of the browser logic and have applied formal reasoning to uncover important new spoofing scenarios. This has been done for both the status bar and the address bar. The knowledge obtained from our approach offers an in-depth understanding of potential logic flaws in the graphical interface implementation. The IE development team has confirmed that all thirteen flaws reported by us are indeed exploitable, and has fixed eleven of them in the latest build. Through this work, we demonstrate the feasibility and the benefit of applying a rigorous formal approach to GUI design and implementation.

Despite the fact that the analysis approach is systematic, it only provides *relative completeness*: relative to the kind of spoofing scenarios being considered, the IE code subset currently modeled, and our search spaces.

We have also developed a model of the logic of IBOS in which we have proven the correctness of the address bar handling, just like we analyzed the address bar for IE.

In that model of IBOS we have shown that the SOP holds, by analyzing a number of properties that altogether imply SOP. However, we were able to find a bug in the display memory management as originally designed as well. We have proposed a straightforward fix and we have shown that in the model this fix works correctly.

Finding flaws like these, or showing their absence, is important for the level of trust a user can place on the browser being used. The kind of analysis we do is of course not only relevant for IE and IBOS, but can provide valuable insight into any browser, or any other highly networked application or GUI-based application.

7.1.2 Folding Variant Narrowing Conclusions

We have presented a self-contained and extended exposition of the key concepts, results, and algorithms for variant narrowing and variant-based unification; and we have illustrated the main ideas with a rich collection of examples. What these new techniques achieve is to bring narrowing modulo axioms from a theoretical possibility with hopeless practical prospects into a practically useful technique with many potential applications, some of which have already been exploited in actual tools such as the Maude-NPA, the Maude Chuch-Rosser Checker (CRC) and Maude Coherence Checker (ChC) tools; see the future work section for more applications.

7.1.3 Protocol Analysis Conclusions

The case studies presented have shown that folding variant narrowing, as supported by the Maude-NPA, is indeed an effective method to deal with nontrivial combinations of equational theories. Using Maude-NPA allows us to analyze multiple cryptographic protocols, showcased by the four protocol case studies, proving correctness for one protocol and finding flaws in the remaining three protocols.

7.2 Future Work

We split the future work into three subsections, dealing with browser analysis in Section 7.2.1, folding variant narrowing in Section 7.2.2 and cryptographic protocol analysis in Section 7.2.3

7.2.1 Browser Analysis Future Work

Regarding the IBOS web browser there are a few follow up projects that would be useful. First, now that a design that has been analyzed in detail exists, it would make sense to analyze the implementation using (semi-) automatic source code verification tools [107, 109, 108]. Another way of increasing the confidence in IBOS would be to actually use the proven secure microkernel seL4 or to develop a new proof of security for IBOS' underlying microkernel.

Another important task ahead is to obtain a precise high-level specification of more IE modules, and to extend our current formal models and analyses to cover most IE functionality. For example, the model should accommodate the tab browsing logic and the hosting mechanisms for document types other than HTML, such as PDF, Microsoft Word, Macromedia Flash, etc. Our methodology can be extended to tackling this pending challenge in the future.

GUI logic flaws affect all web browsers, not just IE and IBOS. We believe that the methodology presented in this dissertation can be equally applied to systematically identify vulnerabilities in other browsers. More broadly, non-browser applications, e.g., email clients and digital identity management tools [93], have similar graphical interface integrity issues. Thus, ensuring GUI logic correctness is a research direction with significant practical relevance.

7.2.2 Folding Variant Narrowing Future Work

Folding variant narrowing has allowed work in the direction of asymmetric unification, where the instantiation of one of the sides of a unification problem must remain in canonical form. This does turn out to be very useful for cryptographic protocol analysis and is, in some way, already done in many approaches but has not been formalized in a general way before. Together with a number of colleagues we have just published work on this [46]. Once this asymmetric unification problem was made explicit, work has also been done on dedicated unification algorithms, e.g., for a combination of theories with exclusive-or, that would have higher performance over our more generic, narrowing based approach.

Folding variant narrowing has already been used in some automated deduction tools, specifically for checking the confluence and coherence of rewrite theories modulo axioms in the Maude CRC and ChC tools [44] and to reason about termination modulo combination of associativity, commutativity and identity in the Maude Termination Tool [41]. Note that a variant based approach can actually be used even for theories that do not have the finite variant property as a given term may have a finite number of variants anyway. In this way we envision many other applications of variant narrowing in automated deduction.

A few issues with great potential for improvement are: (i) better variant generation strategies and (ii) better algorithms for ensuring that a theory has the finite variant property. For example, the current implementation of folding variant narrowing and variant-based unification available in Maude [40] and used by the Maude-NPA only supports a subclass of FV theories, and could be substantially optimized in many ways. Here lazy narrowing strategies may be useful but no notion of needed or demanded evaluation step has been defined for the modulo case. Another promising direction is to further advance the proof techniques for checking FV and implement tools for such checking. There is recent work on extending techniques for termination of rewriting to termination of narrowing which could be adapted to prove FV. Modularity results for modular combination of theories enjoying the finite variant property are also interesting, similarly to modularity results for termination of basic narrowing [7].

Furthermore, a promising direction is the study of symbolic, narrowing-based, reachability analysis techniques for rewrite theories $\mathcal{R} = (\Sigma, E \cup Ax, R)$, where E is confluent, terminating, sort-decreasing and coherent modulo Ax and a finitary Ax-unification exists, but $E \cup Ax$ need not be FV. And an even more ambitious future task is to extend these techniques to new techniques for the development of finitary unification algorithms for theories that have such algorithms but do not enjoy FV.

7.2.3 Protocol Analysis Future Work

The Maude-NPA actually is currently restricted to only applying folding variant narrowing to the case of theories that are strongly right irreducible. It would be very useful to extend the Maude-NPA's unification features to the full capabilities of the folding variant narrowing theory as given here and to remove the strongly right irreducible requirement. Indeed, current work on the Maude implementation should make this possible in the near future.

A very important future direction is work in formal tools supporting symbolic protocol analysis modulo equational properties. Such work should include: (i) developing methods for expanding a tool's built-in unification infrastructure based on a finite number of predefined (finitary) unification algorithms for theories $T_1, \ldots, T_k$ (with any combination of them done by standard combination methods), to make it as efficient and extensible as possible; and (ii) improving and optimizing the methods for efficient variant narrowing modulo such infrastructure. Good candidates for new theories T_j to be added to the built-in infrastructure include commonly used theories, with high priority given to theories that lack the finite variant properties. For example, the theory of homomorphic encryption, which lacks the finite variant property, has been recently added to Maude-NPA for exactly this purpose.

APPENDIX A

EXPLAINED MAUDE SPECIFICATION OF THE IE MODEL

All the data source files for this chapter are available at [112]. We are using Maude 2.6 for all of these experiments.

A.1 Status Bar - Explained Specification

In this section we list and describe the source code of the Maude model of the status bar, including its execution and the initial state space generation.

```
fmod OBJECT is
including QID .

sort Attribute .
sort AttributeSet .
subsort Attribute < AttributeSet .
op nil : -> AttributeSet .
op _,_ : AttributeSet AttributeSet
         -> AttributeSet [assoc comm id: nil] .

sort ObjectName .
subsort Qid < ObjectName .
op nil : -> ObjectName .

sort Object .
op <_|_> : ObjectName AttributeSet -> Object .
```

Objects have attributes that are stored as associative-commutative sets and they do have an object name.

```
sort ClassName .
ops AbstractElement Anchor Button Form
    Image InputField Label
    : -> ClassName .
```

```
sort URL .
ops maliciousUrl wantedUrl emptyUrl arbitraryUrl
    : -> URL .

sort InputType .
ops htmlInputButton htmlInputText
    htmlInputImage htmlInputSubmit
    : -> InputType .

op className:_ : ClassName -> Attribute .
op targetURL:_ : URL -> Attribute .
op upLink:_    : ObjectName -> Attribute .
op container:_ : ObjectName -> Attribute .
op inputType:_ : InputType -> Attribute .
endfm
```

Class names, URLs and the input type are all possible attributes of an object. Also, each object has an uplink to a parent object and may have a link to an object containing it.

```
fmod ACTION is
including OBJECT .

sort Action .
sort ActionList .
subsort Action < ActionList .
op noOp : -> Action .
op _;_ : ActionList ActionList
         -> ActionList [assoc id: noOp] .
```

Actions are stored as a list, due to their nature of being executed in order.

```
sort MessageType .
sort RawMessageType .
subsort RawMessageType < MessageType .
ops MOUSEMOVE LBUTTONDOWN LBUTTONUP
    : -> RawMessageType .
ops MOUSEOVER MOUSELEAVE : -> MessageType .
```

We differentiate between raw mouse messages, i.e., movement of the mouse and the button being pressed and released, and mouse messages, like mousing over an element or leaving an element on the screen.

```
op onMouseMessage : ObjectName RawMessageType
                    -> Action .
op pumpMessage : ObjectName MessageType -> Action .
op fireJScriptNonClick : ObjectName MessageType
                         -> Action .
op fireJScriptClick : ObjectName -> Action .
op bubbleHandleMessage  : ObjectName MessageType
                          -> Action .
op bubbleClickAction : ObjectName -> Action .
op doClick : ObjectName -> Action .
op handleMessage : ObjectName MessageType -> Action .
op clickAction : ObjectName -> Action .
op cancelBubble : -> Action .

ops setStatusText FollowHyperlink : URL -> Action .
op eyeInspection : -> Action .
endfm
```

Actions can be to pass messages of different types to objects, to simulate what JavaScript does[1], the bubbling mechanism of passing the message on to the parent HTML object, and click messages.

Also, setting of the status bar, following a hyper link and the user inspecting the status bar are all explicitly shown as actions which allows our search later on to find these easily.

```
fmod STATE-MULTI-SET is
including ACTION .

sort StateMultiSet .
sort StateMultiSetElement .
subsort StateMultiSetElement < StateMultiSet .
op nil : -> StateMultiSet .
op __ : StateMultiSet StateMultiSet
        -> StateMultiSet [assoc comm id: nil] .

sort ObjectMultiSet .
subsort Object < ObjectMultiSet .
op nil : -> ObjectMultiSet .
op __ : ObjectMultiSet ObjectMultiSet
        -> ObjectMultiSet [assoc comm id: nil] .
```

[1]The JavaScript action does get translated to nothing later on, but it is included here so that malicious use of JavaScript could be modeled if so desired. In the end we decided to only care about static HTML.

```
op {_} : ObjectMultiSet -> StateMultiSetElement .
op [_] : ActionList -> StateMultiSetElement .
op statusBar : URL -> StateMultiSetElement .
op memorizedUrl : URL -> StateMultiSetElement .
endfm
```

A multiset of objects is used to represent all the HTML objects; it is then wrapped by curly braces to become a part of the state multi set. That state multiset also includes the list of actions, wrapped by square brackets, the actual status bar value, and the user memorized URL value.

```
mod GENERAL-MOUSE-RULES is
including STATE-MULTI-SET .

var A : Action . var AL : ActionList .
var M : MessageType . vars RM RM' : RawMessageType .
vars O O' O'' : ObjectName .
vars Atts Atts' : AttributeSet . vars Url Url' : URL .
var C : ClassName . var OMS : ObjectMultiSet .

**** We are interested in two consecutive mouse
**** messages, be they moves or clicks, this means that
**** the 'old' HTML object is always getting the first
**** message.
eq [ onMouseMessage(O, RM) ;
     onMouseMessage(O, RM') ; AL ]
 = [ pumpMessage(O, RM) ; pumpMessage(O, RM') ; AL ] .
```

If there are two mouse messages on the same HTML object, then they can simply be pumped directly to that HTML object. If the messages are for two different HTML objects `O` and `O'` the next equation will take care of them:

```
ceq [onMouseMessage(O,RM) ; onMouseMessage(O',RM') ;
     AL]  {OMS}
  = [pumpMessage(O,RM) ; pumpMessage(O,MOUSELEAVE) ;
     if not (childOfAnchor(O', OMS))
      then setStatusText(emptyUrl)
      else noOp fi ;
     pumpMessage(O',RM') ; pumpMessage(O',MOUSEOVER) ;
     AL] {OMS}
  if O =/= O' .
```

On the other hand, for mouse messages on two different objects, a MOUSELEAVE message on the first object is added after the first message and a MOUSEOVER message on the second object is added after the second message. Additionally, if the second object is not the child of an anchor, the status bar is set to the empty URL; otherwise this step is skipped.

```
**** checking property of being a child of an anchor
**** (direct, or indirect, i.e., transitive)
op childOfAnchor : ObjectName ObjectMultiSet -> Bool .
eq childOfAnchor(O,
     < O | upLink: O' , Atts >
     < O' | className: C , upLink: O'' , Atts' > OMS)
   = if C == Anchor
        then true
        else childOfAnchor(O,
               < O | upLink: O'' , Atts > OMS)
        fi .

eq childOfAnchor(O, < O | upLink: nil , Atts >
                 OMS)
 = false .
```

The check for being a child of an anchor is transitive and goes to the top, until there are no further parent objects available, unless an anchor object is found.

```
eq [pumpMessage(O, M) ; AL]
 = [fireJScriptNonClick(O, M) ;
    bubbleHandleMessage(O, M) ;
    if M == LBUTTONUP then doClick(O) else noOp fi ;
    AL] .
```

Pumping a message results in the JavaScript non-click message being triggered (which, as mentioned before, will disappear; also see code later) and the start of the bubble mechanism for that object and message. Afterwards, if it was the mouse button being released, i.e., LBUTTONUP, then the method for a click is started. Bubbling is described next:

```
rl [bubbleHandleMessage(O, M) ; AL]
   {< O | upLink: O' , Atts > OMS}
=> [handleMessage(O, M) ;
    if O' == nil
```

```
       then noOp
       else bubbleHandleMessage(O', M)
       fi ;
    AL]
   {< O | upLink: O' , Atts > OMS} .
```

A bubble handle message is transformed to handling that message at the current object, and, if there is a parent object, passing the bubble handle message on to that object afterwards. The handling of a message can end the bubbling mechanism by posting a `cancelBubble` into the list of actions. The same happens to the bubbling of a click action instead of a handle message.

```
rl [bubbleClickAction(O) ; AL]
   {< O | upLink: O' , Atts > OMS}
=> [clickAction(O) ;
    if O' == nil then noOp
                 else bubbleClickAction(O') fi ; AL]
   {< O | upLink: O' , Atts > OMS} .

eq [cancelBubble ; bubbleHandleMessage(O, M) ; AL]
 = [AL] .

eq [cancelBubble ; bubbleClickAction(O) ; AL]
 = [AL] .

eq [cancelBubble ; AL]
 = [AL] [owise] .
```

Canceling the bubble removes the bubbling handle message as well as the bubbling click action. Note that only the first action in the list of actions is actively executed, these dormant bubbling actions can be removed easily and there is no chance their actions could be executed already. The last equation, which allows `cancelBubble` to be removed, is marked with `[owise]`, which means it will only be executed when none of the other equations is applicable.

```
rl [setStatusText(Url) ; AL]
   statusBar(Url')
=> [AL]
   statusBar(Url) .

rl [eyeInspection ; AL]
   statusBar(Url) memorizedUrl(Url')
```

```
=> [AL]
   statusBar(Url) memorizedUrl(Url) .

eq [FollowHyperlink(Url) ; A ; AL]
 = [FollowHyperlink(Url)] if A =/= noOp .
endm
```

The action of setting the status bar URL is completed by actually changing the URL in the status bar to the new URL. The user's inspection of the status bar copies the value of the status bar into the memorized URL wrapper, and following a hyperlink ends the action execution by dropping all further actions that should follow, as this is the point where the decision can be made whether the URL the navigation actually goes to is the same as the one in the user's memory, the status bar, both, or neither of them.

We assume from now on that the JavaScript that is being used does nothing too outrageous, like changing the class name or the whole DOM tree, as we are ultimately interested in security in the absence of JavaScript for the status bar anyway. This allows us to model any `XHandleMessage` (with `X` any of the elements) with a simple equation instead of requiring rules.

```
mod ABSTRACT-ELEMENT is
including STATE-MULTI-SET .

var AL : ActionList . var O : ObjectName .
var M : MessageType .

op AbstractElementHandleMessage
    : ObjectName MessageType -> Action .
op AbstractElementDoClick : ObjectName -> Action .
op AbstractElementClickAction : ObjectName -> Action .

eq [AbstractElementHandleMessage(O,M) ; AL]
 = [AL] .

eq [AbstractElementClickAction(O) ; AL]
 = [AL] .

eq [AbstractElementDoClick(O) ; AL]
 = [fireJScriptClick(O) ; bubbleClickAction(O) ; AL] .
endm
```

The HandleMessage and ClickAction of AbstractElement do nothing, while the DoClick does start the JavaScript click and bubbles, i.e., continues, the click action. Note that these will be overwritten in each more specific class with the actual implementation behavior.

```
mod ANCHOR is
including STATE-MULTI-SET .
including ABSTRACT-ELEMENT .

var AL : ActionList . var O : ObjectName .
var M : MessageType . var Atts : AttributeSet .
var Url : URL . var OMS : ObjectMultiSet .

op AnchorHandleMessage : ObjectName MessageType
     -> Action .
op AnchorDoClick : ObjectName -> Action .
op AnchorClickAction : ObjectName -> Action .

eq [handleMessage(O, M) ; AL]
   {< O | className: Anchor , Atts > OMS}
 = [AnchorHandleMessage(O, M) ; AL]
   {< O | className: Anchor , Atts > OMS} .

eq [doClick(O) ; AL]
   {< O | className: Anchor , Atts > OMS}
 = [AnchorDoClick(O) ; AL]
   {< O | className: Anchor , Atts > OMS} .

eq [clickAction(O) ; AL]
   {< O | className: Anchor , Atts > OMS}
 = [AnchorClickAction(O) ; AL]
   {< O | className: Anchor , Atts > OMS} .


ceq [AnchorHandleMessage(O, M) ; AL]
  = [cancelBubble ; AL]
 if M == LBUTTONDOWN .

crl [AnchorHandleMessage(O,M) ; AL]
    {< O | targetURL: Url , Atts > OMS}
 => [setStatusText(Url) ; AL]
    {< O | targetURL: Url , Atts > OMS}
 if M == MOUSEOVER .
```

```
ceq [AnchorHandleMessage(O,M) ; AL]
  = [AL]
 if M == MOUSELEAVE or M == MOUSEMOVE or
    M == LBUTTONUP .

rl [AnchorClickAction(O) ; AL]
   {< O | targetURL: Url , className: Anchor , Atts >
    OMS}
=> [FollowHyperlink(Url) ; AL]
   {< O | targetURL: Url , className: Anchor , Atts >
    OMS} .

eq [AnchorDoClick(O) ; AL]
 = [AbstractElementDoClick(O) ; AL ] .
endm
```

For the Anchor element the three standard actions get instantiated with their specific Anchor version. If the message is a left button down, then the Anchor handles it by stopping the bubbling mechanism as itself will take of the click when it gets it. If the message is a mouse over message, it will similarly just set the status bar text to its URL. If it is a mouse move, the left button being released or the anchor is left (due to a move) then it just does nothing and the bubble continues. On a click action it actually starts navigation to its URL.

```
mod BUTTON is
including STATE-MULTI-SET .
including ABSTRACT-ELEMENT .

var AL : ActionList . vars O O' : ObjectName .
var M : MessageType . vars Atts Atts' : AttributeSet .
var Url : URL . var OMS : ObjectMultiSet .

op ButtonHandleMessage : ObjectName MessageType
     -> Action .
op ButtonDoClick : ObjectName -> Action .
op ButtonClickAction : ObjectName -> Action .

eq [handleMessage(O, M) ; AL]
   {< O | className: Button , Atts > OMS}
 = [ButtonHandleMessage(O,M) ; AL]
   {< O | className: Button , Atts > OMS} .
```

```
eq [doClick(O) ; AL]
   {< O | className: Button , Atts > OMS}
 = [ButtonDoClick(O) ; AL]
   {< O | className: Button , Atts > OMS} .

eq [clickAction(O) ; AL]
   {< O | className: Button , Atts > OMS}
 = [ButtonClickAction(O) ; AL]
   {< O | className: Button , Atts > OMS} .

eq [ButtonHandleMessage(O,M) ; AL]
 = [AbstractElementHandleMessage(O,M) ; AL] .

rl [ButtonClickAction(O) ; AL]
   {< O | container: O' , className: Button , Atts >
    < O' | targetURL: Url , className: Form , Atts' >
    OMS}
=> [FollowHyperlink(Url) ; cancelBubble ;  AL]
   {< O | container: O' , className: Button , Atts >
    < O' | targetURL: Url , className: Form , Atts' >
    OMS} .

eq [ButtonClickAction(O) ; AL]
   {< O | container: nil , className: Button , Atts >
    OMS}
 = [cancelBubble ; AL]
   {< O | container: nil , className: Button , Atts >
    OMS} .

eq [ButtonDoClick(O) ; AL]
 = [AbstractElementDoClick(O) ; AL ] .
endm
```

The `Button` works like the `Anchor`, in that it instantiates all actions to its class-specific actions. Message handling is then left to the default version of an abstract element, similarly for `DoClick`. A `ClickAction` triggers navigation to the URL stored in the object associated to the button. If there is no associated object the button cancels the bubble and does nothing instead.

```
mod FORM is
including STATE-MULTI-SET .
including ABSTRACT-ELEMENT .
```

```
var AL : ActionList . var O : ObjectName .
var M : MessageType . var Atts : AttributeSet .
var OMS : ObjectMultiSet .

op FormHandleMessage : ObjectName MessageType
     -> Action .
op FormDoClick : ObjectName -> Action .
op FormClickAction : ObjectName -> Action .

eq [handleMessage(O, M) ; AL]
   {< O | className: Form , Atts > OMS}
 = [FormHandleMessage(O,M) ; AL]
   {< O | className: Form , Atts > OMS} .

eq [doClick(O) ; AL]
   {< O | className: Form , Atts > OMS}
 = [FormDoClick(O) ; AL]
   {< O | className: Form , Atts > OMS} .

eq [clickAction(O) ; AL]
   {< O | className: Form , Atts > OMS}
 = [FormClickAction(O) ; AL]
   {< O | className: Form , Atts > OMS} .

eq [FormHandleMessage(O,M) ; AL]
 = [AbstractElementHandleMessage(O,M) ; AL] .

eq [FormDoClick(O) ; AL]
 = [AbstractElementDoClick(O) ; AL] .

eq [FormClickAction(O) ; AL]
 = [AbstractElementClickAction(O) ; AL] .
endm
```

For a Form, all actions default to the appropriate action of the abstract element.

```
mod IMAGE is
including STATE-MULTI-SET .
including ABSTRACT-ELEMENT .
including ANCHOR .

var AL : ActionList . vars O O' : ObjectName .
var M : MessageType . vars Atts Atts' : AttributeSet .
var Url : URL . var OMS : ObjectMultiSet .

op ImageHandleMessage : ObjectName MessageType
     -> Action .
op ImageDoClick : ObjectName -> Action .
op ImageClickAction : ObjectName -> Action .

eq [handleMessage(O, M) ; AL]
   {< O | className: Image , Atts > OMS}
 = [ImageHandleMessage(O,M) ; AL]
   {< O | className: Image , Atts > OMS} .

eq [doClick(O) ; AL]
   {< O | className: Image , Atts > OMS}
 = [ImageDoClick(O) ; AL]
   {< O | className: Image , Atts > OMS} .

eq [clickAction(O) ; AL]
   {< O | className: Image , Atts > OMS}
 = [ImageClickAction(O) ; AL]
   {< O | className: Image , Atts > OMS} .

eq [ImageDoClick(O) ; AL]
 = [AbstractElementDoClick(O) ; AL] .

rl [ImageClickAction(O) ; AL]
   {< O | container: O' , className: Image , Atts >
    < O' | className: Anchor , targetURL: Url , Atts' >
    OMS}
=> [AnchorClickAction(O') ; cancelBubble ; AL]
   {< O | container: O' , className: Image , Atts >
    < O' | className: Anchor , targetURL: Url , Atts' >
    OMS} .

rl [ImageClickAction(O) ; AL]
   {< O | container: nil , className: Image , Atts >
    OMS}
=> [cancelBubble ; AL]
   {< O | container: nil , className: Image , Atts >
    OMS} .

ceq [ImageHandleMessage(O,M) ; AL]
 = [AL]
```

```
if M =/= MOUSEOVER .

rl [ImageHandleMessage(O,MOUSEOVER) ; AL]
   {< O | container: O' , className: Image , Atts >
    < O' | className: Anchor , targetURL: Url , Atts' >
    OMS}
=> [setStatusText(Url) ; AL]
   {< O | container: O' , className: Image , Atts >
    < O' | className: Anchor , targetURL: Url , Atts' >
    OMS} .

rl [ImageHandleMessage(O,MOUSEOVER) ; AL]
   {< O | container: nil , className: Image , Atts >
    OMS}
=> [AL]
   {< O | container: nil , className: Image , Atts >
    OMS} .
endm
```

For an `Image`, the `DoClick` action defaults to the way of being handled by the abstract element. The `ClickAction` either does nothing, if there is no associated `Anchor` for the `Image`, or calls `AnchorClickAction` on the associated anchor, if there is one. For message handling of anything but a `MOUSEOVER` nothing happens. In case of a `MOUSEOVER` message being handled, if there is an associated (i.e., containing) `Anchor` then the `targetURL` from there is set in the status bar. Without an associated `Anchor` this message just gets dropped.

```
mod INPUT-FIELD is
including STATE-MULTI-SET .
including ABSTRACT-ELEMENT .

var AL : ActionList . vars O O' : ObjectName .
var M : MessageType . vars Atts Atts' : AttributeSet .
var Url : URL . var OMS : ObjectMultiSet .

op InputFieldHandleMessage : ObjectName MessageType
     -> Action .
op InputFieldDoClick : ObjectName -> Action .
op InputFieldClickAction : ObjectName -> Action .

eq [handleMessage(O, M) ; AL]
   {< O | className: InputField , Atts > OMS}
```

```
 = [InputFieldHandleMessage(O,M) ; AL]
   {< O | className: InputField , Atts > OMS} .

eq [doClick(O) ; AL]
   {< O | className: InputField , Atts > OMS}
 = [InputFieldDoClick(O) ; AL]
   {< O | className: InputField , Atts > OMS} .

eq [clickAction(O) ; AL]
   {< O | className: InputField , Atts > OMS}
 = [InputFieldClickAction(O) ; AL]
   {< O | className: InputField , Atts > OMS} .

eq [InputFieldHandleMessage(O,M) ; AL]
 = [AbstractElementHandleMessage(O,M) ; AL] .
```

As usual, for `InputField` elements the element-specific functions are called. Message handling is left to the abstract element way of handling the message. Note that in what follows we assume sane JavaScript which does not change the DOM tree, similarly to what we required before, as we are ultimately interested in static HTML for the status bar.

```
eq [InputFieldClickAction(O) ; AL]
   {< O | inputType: htmlInputButton ,
          className: InputField , Atts > OMS}
 = [AL]
   {< O | inputType: htmlInputButton ,
          className: InputField , Atts > OMS} .

eq [InputFieldClickAction(O) ; AL]
   {< O | inputType: htmlInputText ,
          className: InputField , Atts > OMS}
 = [AbstractElementClickAction(O) ; AL]
   {< O | inputType: htmlInputText ,
          className: InputField , Atts > OMS} .

crl [InputFieldClickAction(O) ; AL]
   {< O | inputType: htmlInputImage , container: O' ,
          className: InputField , Atts >
    < O' | targetURL: Url , className: Form , Atts' >
    OMS}
=> [FollowHyperlink(Url) ; cancelBubble ; AL]
   {< O | inputType: htmlInputImage , container: O' ,
          className: InputField , Atts >
      < O' | targetURL: Url , className: Form , Atts' >
      OMS}
if O' =/= nil .

eq [InputFieldClickAction(O) ; AL]
   {< O | inputType: htmlInputImage , container: nil ,
          className: InputField , Atts > OMS}
 = [cancelBubble ; AL]
   {< O | inputType: htmlInputImage , container: nil ,
          className: InputField , Atts > OMS} .

crl [InputFieldClickAction(O) ; AL]
   {< O | inputType: htmlInputSubmit , container: O' ,
          className: InputField , Atts >
    < O' | targetURL: Url , className: Form , Atts' >
    OMS}
=> [FollowHyperlink(Url) ; cancelBubble ; AL]
   {< O | inputType: htmlInputSubmit , container: O' ,
          className: InputField , Atts >
    < O' | targetURL: Url , className: Form , Atts' >
    OMS}
if O' =/= nil .

eq [InputFieldClickAction(O) ; AL]
   {< O | inputType: htmlInputSubmit , container: nil ,
          className: InputField , Atts > OMS}
 = [cancelBubble ; AL]
   {< O | inputType: htmlInputSubmit , container: nil ,
          className: InputField , Atts > OMS} .

eq [InputFieldDoClick(O) ; AL]
 = [AbstractElementDoClick(O) ; AL] .
endm
```

For the `InputField` in the case of a `ClickAction` there are many different possible outcomes. First, if the `inputType` is that of a `htmlInputButton` it is simply dropped. If the `inputType` is a `htmlInputText` then the abstract elements handling is called.

In case of an `htmlInputImage` the way it is handled depends on whether there is an associated `Form`, and in case there is one then navigation to the URL of that `Form` is started. Without an associated `Form`, the bubble is canceled. The same thing happens for the case of `htmlInputSubmit`, in that an associated `Form` triggers navigation to the URL of that `Form`, with the bubble being canceled otherwise.

The `DoClick` is then just handled by the abstract element base case.

```
mod LABEL is
including STATE-MULTI-SET .
including ABSTRACT-ELEMENT .

var AL : ActionList . var O : ObjectName .
var M : MessageType . var Atts : AttributeSet .
var OMS : ObjectMultiSet .

op LabelHandleMessage : ObjectName MessageType
     -> Action .
op LabelDoClick : ObjectName -> Action .
op LabelClickAction : ObjectName -> Action .

eq [handleMessage(O, M) ; AL]
   {< O | className: Label , Atts > OMS}
 = [LabelHandleMessage(O,M) ; AL]
   {< O | className: Label , Atts > OMS} .

eq [doClick(O) ; AL]
   {< O | className: Label , Atts > OMS}
 = [LabelDoClick(O) ; AL]
   {< O | className: Label , Atts > OMS} .

eq [clickAction(O) ; AL]
   {< O | className: Label , Atts > OMS}
 = [LabelClickAction(O) ; AL]
   {< O | className: Label , Atts > OMS} .

ceq [LabelHandleMessage(O,M) ; AL]
  = [cancelBubble ; AL]
 if M == MOUSEOVER or M == MOUSELEAVE .

ceq [LabelHandleMessage(O,M) ; AL]
  = [AbstractElementHandleMessage(O,M) ; AL]
 if M == LBUTTONUP or M == LBUTTONDOWN or
    M == MOUSEMOVE .

eq [LabelDoClick(O) ; AL]
 = [AbstractElementDoClick(O) ; AL] .
```

```
eq [LabelClickAction(O) ; AL]
 = [FollowHyperlink(maliciousUrl) ; cancelBubble ; AL].
endm
```

Handling a `MOUSEOVER` or `MOUSELEAVE` message for a `Label` leads to a canceling of the bubble mechanism, while button up and down, as well as mouse moves, get relayed to the abstract element handling. The `DoClick` gets passed to the abstract handler as well, but the `ClickAction` leads to navigation to a malicious URL immediately. The reasoning for this is that the URL of the `Label` is never set as a status text, and thus any URL the `Label` navigates to is malicious by definition.

```
mod COMPLETE-MOUSE-RULES is
including STATE-MULTI-SET .
including GENERAL-MOUSE-RULES .
including ABSTRACT-ELEMENT .
including ANCHOR .
including BUTTON .
including FORM .
including IMAGE .
including INPUT-FIELD .
including LABEL .

var AL : ActionList . var O : ObjectName .
var M : MessageType .

eq [fireJScriptNonClick(O,M) ; AL] = [noOp ; AL] .
eq [fireJScriptClick(O) ; AL] = [noOp ; AL] .
endm
```

Here we have combined all the different element-handling mechanisms into one module, and this is also the point where we define that any JavaScript is simply ignored. Changing this here would lead to partial handling of JavaScript by what is given here, but for full JavaScript handling one would also need to take care to allow transformations of the whole DOM tree at run time, which we do not support.

```
fmod NATSET is
including INT .
sort NatSet .
subsort Nat < NatSet .
```

```
op nil : -> NatSet .
op __ : NatSet NatSet -> NatSet [assoc comm id: nil] .
endfm
```

These are just the standard natural numbers with a set constructor that we use later for the initial generation of all possible test cases.

```
mod WRAPPING is
including GENERAL-MOUSE-RULES .
including NATSET .

var SMS : StateMultiSet . var C : ClassName .
var KO : [ObjectName] .

op wrap : StateMultiSet -> StateMultiSet [frozen] .
eq wrap(SMS) = SMS .
```

Let us first explain the intention and effects of this wrapper. As `wrap` is declared to be *frozen*, no rewrite underneath is possible. Every possible rewrite has to happen at the top-level, that is, on a term of the form `wrap(X)`. Also, note that the equation given requires the state multiset `SMS` to be of the appropriate sort, and that the terms we use to generate (exhaustively!) the search space are only of the kind, and not of the sort. That is, as long as there is at least one `Any...` term (see directly below) inside, this equation will not be applicable.

```
op AnyClassName : -> [ClassName] .
op AnyUrl : -> [URL] .
op AnyContainer : NatSet -> [ObjectName] .
op AnyInputType : -> [InputType] .

op e : Nat -> ObjectName .

ceq className: C, container: KO
  = className: C
if C =/= Button /\ C =/= Image /\ C =/= InputField .

ceq className: C, targetURL: AnyUrl
  = className: C
 if C =/= Anchor /\ C =/= Form .

ceq className: C , inputType: AnyInputType
```

```
  = className: C
 if C =/= InputField .
endm
```

The first four operators declared are all declared to go to the kind of the sort they are there to create. This means that, until they have actually been transformed to proper terms of their sort, the `wrap` cannot be dissolved. The equations given here just remove irrelevant portions of the state, i.e., an object that is neither a `Button`, an `Image` nor an `InputField`, then there is no need to have a field for a `container`. Similarly, only `Anchor`s and `Form`s have a `targetURL` and only `InputFields`s have an `inputType`.

The next module limits each connected component in the DOM tree to have at most one anchor. To ensure at most one anchor per connected component this module works on a copy of the DOM tree in which it checks all elements the anchor in question have an `upLink` to (including transitively), as well as walking down the graph. As there are no explicit links down, one is added here and the DOM tree is obviously manipulated, which is no problem as we are dealing with a copy.

```
mod ONE-ANCHOR-LIMIT is
including GENERAL-MOUSE-RULES .

var KOMS : [ObjectMultiSet] . var AL : ActionList .
vars U U' : URL . vars O O' O'' : ObjectName .
vars KAtts KAtts' : [AttributeSet] .
var KC : [ClassName] .

op noAnchorInCC : ObjectName StateMultiSet -> Bool .
op downLink:_ : ObjectName -> Attribute .

eq noAnchorInCC(O, [AL]  {< O | KAtts > KOMS}
                   statusBar(U) memorizedUrl(U'))
 = noAnchorInCC(O, {< O | downLink: O , KAtts > KOMS}).

eq noAnchorInCC(O, {< O | KAtts >}) = true .
```

First, everything that is not an object is dropped, as it is not relevant, and a `downLink` to itself is added. That will be compared to other element's `upLink`s and thus used for a transitive descent. The next equation works if there is only one element, then obviously there is no second anchor.

```
eq noAnchorInCC(O, {< O | upLink: O' , KAtts >
                    < O' | className: Anchor , KAtts' >
                    KOMS})
   = false .

ceq noAnchorInCC(O, {< O | upLink: O' , KAtts >
                     < O' | className: KC ,
                            upLink: O'' , KAtts' >
                     KOMS})
  = noAnchorInCC(O, {< O | upLink: O'' , KAtts >
                     KOMS})
if KC =/= Anchor .
```

Here the DOM tree is manipulated, in that the `upLink` of the object in question is changed to its transitive `upLink`, which is its grandfather or an even earlier ancestor. At the same time that intermediate element is dropped as it has no further use.

```
eq noAnchorInCC(O,
     {< O | downLink: O'' , KAtts >
      < O' | upLink: O'' , className: Anchor , KAtts' >
                    KOMS})
 = false .

ceq noAnchorInCC(O,
      {< O | downLink: O'' , KAtts >
       < O' | upLink: O'' , className: KC , KAtts' >
                     KOMS})
  = noAnchorInCC(O, {< O | downLink: O' , KAtts >
                     KOMS})
if KC =/= Anchor .

eq noAnchorInCC(O, {< O' | KAtts > KOMS})
 = true [owise] .
endm
```

In the first equation, if an element has an `upLink` that is an `Anchor` and matches the current descent level (`downLink`) of the object, then we found a second anchor in the connected component. If on the other hand that element is not an anchor, then we change the `downLink` to be that element, and drop that element from the tree and continue.

If none of the equations before the last one applied, then we have searched and discarded the whole connected component and there is no second anchor in that connected component. Note how that equation is marked with [owise], meaning it only applies if all other equations fail.

```
mod INPUTFIELD-ONLY-LEAF is
including GENERAL-MOUSE-RULES .

vars O O' : ObjectName . var KAtts : [AttributeSet] .
var KSMS : [StateMultiSet] .
var KOMS : [ObjectMultiSet] .

op isLeaf : ObjectName StateMultiSet -> Bool .

eq isLeaf(O, {< O' | upLink: O , KAtts > KOMS} KSMS)
 = false .

eq isLeaf(O, KSMS)
 = true [owise] .
endm
```

The InputField elements are only allowed as leaves, so we check if there is any element O' that has an upLink to the element O. If so, it is not a leaf, otherwise it is.

```
mod ANCHOR-CREATION is
including WRAPPING .
including ONE-ANCHOR-LIMIT .

var O : ObjectName . var KAtts : [AttributeSet] .
var KSMS : [StateMultiSet] .
var KOMS : [ObjectMultiSet] .

crl wrap( {< O | className: AnyClassName , KAtts >
           KOMS} KSMS)
 => wrap( {< O | className: Anchor , KAtts >
           KOMS} KSMS)
 if noAnchorInCC(O, {< O | className: AnyClassName ,
                          KAtts > KOMS} KSMS) .

rl wrap( {< O | className: Anchor ,
                targetURL: AnyUrl , KAtts > KOMS} KSMS)
=> wrap( {< O | className: Anchor ,
                targetURL: maliciousUrl , KAtts > KOMS}
          KSMS) .
```

```
rl wrap( {< O | className: Anchor ,
                targetURL: AnyUrl , KAtts > KOMS} KSMS)
=> wrap( {< O | className: Anchor ,
                targetURL: wantedUrl , KAtts > KOMS}
          KSMS) .
endm
```

To create an anchor we allow an element that is not defined (`className: AnyClassName`) to become an anchor only if there is no other anchor in the connected component, shown in the first (conditional) rule. If we have an anchor, we allow, by the second and third rule, for the undefined URL to become `maliciousURL` or `wantedURL`.

```
mod BUTTON-CREATION is
including WRAPPING .

var N : Nat . var NS : NatSet . var O : ObjectName .
vars KAtts KAtts' : [AttributeSet] .
var KSMS : [StateMultiSet] .
var KOMS : [ObjectMultiSet] .

rl wrap(
     {< O | className: AnyClassName ,
            container: AnyContainer(N NS) , KAtts >
      < e(N) | className: Form , KAtts' >
      KOMS}
     KSMS)
=> wrap(
     {< O | className: Button ,
            container: e(N) , KAtts >
      < e(N) | className: Form , KAtts' >
      KOMS}
      KSMS) .
endm
```

For an element that is not yet fixed in the DOM tree (see `container: AnyContainer(N NS)`) we allow it to become a `Button` if there is already a `Form` that we can connect it to directly (`container: e(N)`). Also note that technically it has to connect to the closest `Form`, which we do not enforce, thus potentially allowing false positives that do not actually happen.

```
mod FORM-CREATION is
```

```
including WRAPPING .

var O : ObjectName . var KAtts : [AttributeSet] .
var KSMS : [StateMultiSet] .
var KOMS : [ObjectMultiSet] .

rl wrap( {< O | className: AnyClassName , KAtts >
          KOMS} KSMS)
=> wrap( {< O | className: Form , KAtts > KOMS} KSMS) .

rl wrap( {< O | className: Form ,
                targetURL: AnyUrl , KAtts > KOMS} KSMS)
=> wrap( {< O | className: Form ,
                targetURL: maliciousUrl , KAtts > KOMS}
         KSMS) .

rl wrap( {< O | className: Form ,
                targetURL: AnyUrl , KAtts > KOMS} KSMS)
=> wrap( {< O | className: Form ,
                targetURL: wantedUrl , KAtts > KOMS}
         KSMS) .
endm
```

A `Form` can be created at any time, and naturally, its URL can be set to `maliciousUrl` or `wantedUrl`, similar to an `Anchor`.

```
mod IMAGE-CREATION is
including WRAPPING .

var N : Nat . var NS : NatSet . var O : ObjectName .
vars KAtts KAtts' : [AttributeSet] .
var KSMS : [StateMultiSet] .
var KOMS : [ObjectMultiSet] .

rl wrap(
     {< O | className: AnyClassName ,
            container: AnyContainer(N NS) , KAtts >
      < e(N) | className: Anchor , KAtts' >
      KOMS}
     KSMS)
=> wrap(
     {< O | className: Image ,
            container: e(N) , KAtts >
      < e(N) | className: Anchor , KAtts' >
      KOMS}
     KSMS) .
endm
```

For an Image, similarly to a button linking to a form, we require it to link to an Anchor.

```
mod INPUT-FIELD-CREATION is
including WRAPPING .
including INPUTFIELD-ONLY-LEAF .

var N : Nat . var NS : NatSet . var O : ObjectName .
vars KAtts KAtts' : [AttributeSet] .
var KSMS : [StateMultiSet] .
var KOMS : [ObjectMultiSet] .

crl wrap(
      {< O | className: AnyClassName ,
             container: AnyContainer(N NS) , KAtts >
       < e(N) | className: Form , KAtts' >
       KOMS}
      KSMS)
=> wrap(
      {< O | className: InputField ,
             container: e(N) , KAtts >
       < e(N) | className: Form , KAtts' >
       KOMS}
      KSMS)
if isLeaf(O, {< e(N) | className: Form , KAtts' >
             KOMS} KSMS) .


rl wrap( {< O | className: InputField ,
                inputType: AnyInputType , KAtts >
                KOMS} KSMS)
=> wrap( {< O | className: InputField ,
                inputType: htmlInputButton , KAtts >
                KOMS} KSMS) .

rl wrap( {< O | className: InputField ,
                inputType: AnyInputType , KAtts >
                KOMS} KSMS)
=> wrap( {< O | className: InputField ,
                inputType: htmlInputImage , KAtts >
                KOMS} KSMS) .

rl wrap( {< O | className: InputField ,
                inputType: AnyInputType , KAtts >
                KOMS} KSMS)
=> wrap( {< O | className: InputField ,
                inputType: htmlInputSubmit , KAtts >
                KOMS} KSMS) .

rl wrap( {< O | className: InputField ,
                inputType: AnyInputType , KAtts >
                KOMS} KSMS)
=> wrap( {< O | className: InputField ,
                inputType: htmlInputText , KAtts >
                KOMS} KSMS) .
endm
```

We allow creation of an `InputField` if there is already a `Form` it can be linked to, and it has to be a leaf. Then, the remaining rules fix its `inputType` to any one of the possibilities. Note that, technically, an `InputField` has to connect to the nearest `Form`, but we relax this, potentially allowing false positives, but they can be easily adjusted and actually do not appear in our experiments.

```
mod LABEL-CREATION is
including WRAPPING .

var O : ObjectName . var KAtts : [AttributeSet] .
var KSMS : [StateMultiSet] .
var KOMS : [ObjectMultiSet] .

rl wrap( {< O | className: AnyClassName , KAtts >
          KOMS} KSMS)
=> wrap( {< O | className: Label , KAtts > KOMS} KSMS).
endm
```

The creation of a `Label` is allowed for any element in the DOM tree without restriction.

Next we put together all the different element-creation mechanisms as explained above and link them together and initialize them.

```
mod CREATE-STARTS+TESTS is
```

```
including COMPLETE-MOUSE-RULES .
including WRAPPING .
including ANCHOR-CREATION .
including BUTTON-CREATION .
including FORM-CREATION .
including IMAGE-CREATION .
including INPUT-FIELD-CREATION .
including LABEL-CREATION .

vars O O' : ObjectName . var AL : ActionList .
var KOMS : [ObjectMultiSet] . vars N N' : Nat .
var NS : NatSet . var KSMS : [StateMultiSet] .

op createAnyObject : Nat ObjectName NatSet -> [Object].
eq createAnyObject(N, O, NS)
 = < e(N) | className: AnyClassName ,
            targetURL: AnyUrl, inputType: AnyInputType,
            container: AnyContainer(NS), upLink: O > .
```

This creates an object with the name `e(N)`, an `upLink` of `O` and a container that is any element of the set `NS`.

Then a `try` consists of having the mouse over an object `O` for which we trigger `MOUSEOVER`, so it updates the status bar and then moving from that object to another object (possibly the same one), doing a manual inspection by the user (modeled as `eyeInspection` and then clicking that object by pressing and releasing the mouse button.

```
op try : ObjectName ObjectName -> ActionList .

eq try (O,O')
 = pumpMessage(O , MOUSEOVER) ;
   onMouseMessage(O , MOUSEMOVE) ;
   onMouseMessage(O', MOUSEMOVE) ;
   eyeInspection ;
   onMouseMessage(O', LBUTTONDOWN) ;
   onMouseMessage(O', LBUTTONUP) .

op try : NatSet -> ActionList .

rl wrap([try(N NS)] KSMS)
=> wrap(
   try(e(N), e(N))
   KSMS) .
```

```
rl wrap([try(N N' NS)] KSMS)
=> wrap(
   try(e(N), e(N'))
   KSMS) .
```

We also allow a `try` from a set of naturals, which simply means to pick two of them (or just one) and have a try from there.

For the initial state we hard code empty URLs into both the status bar and the memorized URL. It uses a given list of actions on a given set of objects.

```
op initialState : ActionList ObjectMultiSet
     -> StateMultiSet .
eq initialState(AL, KOMS)
 = [AL] {KOMS}
   statusBar(emptyUrl) memorizedUrl(emptyUrl) .
endm
```

Now let us look at an example and what it means. For simplicity let us just take a DOM tree with two elements that are connected, on which we move the mouse from anywhere to anywhere. We of course are searching from that start state to a final state in which the navigation goes to a `maliciousUrl` while the user memorized `wantedUrl`. The initial state consists of the list of actions as given by `try` and the two objects being created with possible links between.

```
search
wrap(initialState(try(1 2),
                  createAnyObject(1, e(2), 2)
                  createAnyObject(2, nil, 1)))
=>!
[FollowHyperlink(maliciousUrl)]
memorizedUrl(wantedUrl)
X:StateMultiSet .
```

The result of that run is the following, which represents two possible attacks that are very similar:

```
Solution 1 (state 1751)
X:StateMultiSet -->
    statusBar(wantedUrl)
    < e(1) | className: Anchor,
             targetURL: wantedUrl,upLink: e(2) >
    < e(2) | className: Label,upLink: nil >

Solution 2 (state 1752)
X:StateMultiSet -->
    statusBar(wantedUrl)
    < e(1) | className: Label,upLink: e(2) >
    < e(2) | className: Anchor,
             targetURL: wantedUrl,upLink: nil >

No more solutions.
states: 1765  rewrites: 20460 in 290ms cpu
(590ms real) (70551 rewrites/second)
```

The two possible states show the combination, in either order, of an `Anchor` being a child or the parent of a `Label`.

Now, let us look at an example with three elements in one connected component, which is created this way:

```
search
wrap(initialState(try(1 2 3),
                  createAnyObject(1, e(2), 2 3)
                  createAnyObject(2, e(3), 3)
                  createAnyObject(3, nil, nil)))
=>!
[FollowHyperlink(maliciousUrl)]
memorizedUrl(wantedUrl)
X:StateMultiSet .
```

There are actually 25 results, so let us pick one to explore in more detail:

```
Solution 20 (state 26386)
X:StateMultiSet -->
    statusBar(wantedUrl)
    < e(1) | className: InputField,upLink: e(2),
             container: e(3),
             inputType: htmlInputImage >
    < e(2) | className: Anchor,
             targetURL: wantedUrl, upLink: e(3) >
    < e(3) | className: Form,
             targetURL: maliciousUrl, upLink: nil >
```

This is actually the DOM tree layout that is presented in Figure 3.5, where an `InputField` will take the click and navigate to its URL. On mousing over though, the URL of the `Anchor` is being displayed.

For more examples, and their resulting output, please see the actual code, respectively the results, in the files included in the code distribution.

A.2 Address Bar - Explained Specification

Whenever there is a boolean variable that, depending on its value, leads to different possible outcomes we have modeled this not by including this variable and setting it one way or the other, but rather by including two rules: one that has the effect of the variable being true, the other having the effect of the variable being false. As Maude will explore all possible paths (via the `search` command), and the trace includes the names of the rules used, we know which value was chosen for each variable relevant to an attack.

Then let us note that the parts of the code that are commented out via `---(...)` and inside they state `**** We force this to be BOOL **** because otherwise there is a bad trace.` (where `BOOL` is `TRUE` or `FALSE`) correspond to one of those choices. That means that the model was first run with this code included and it allowed us to find attacks based on this choice of value for the underlying boolean variable. We then removed any thus identified attack causes, until the remaining model was safe and no more attacks were found. Thus, we have found all possible attacks and analyzing the associated traces as seen in Section 3.3 and shown in more detail below is enough.

The file `list-of-flags.txt` includes all the variables, and its value, which can lead to attacks.

We do not add a sort or wrapper for a `BROWSERINSTANCE`. As we are only dealing with a single such instance, we can just include all its elements directly into the state space as elements of sort `StateElement`. Similarly, there is only one `VIEW`, so all its attributes are added to the general state space as well.

```
fmod CLASSES is
extending QID .
extending INT .

sort Name .
subsort Qid < Name .
op n : Nat -> Name .
op nil : -> Name .

sort Frame .
sort FrameName .
op f : Name -> FrameName .

sort Markup .
sort MarkupName .
op m : Name -> MarkupName .
```

First we define that all quoted identifiers, `Qid`, e.g., `'a` and `'b` are names, as well as the indexed name `n(X)` with `X` a natural number. Then, `Frame` and `Markup` have names.

```
sort Url .
subsort Qid < Url .
op noUrl : -> Url .
op someHistoryUrl : Nat -> Url .

sort DomTree .
op nil : -> DomTree .
op dl : Url -> DomTree .
```

We define a `Url` to be a quoted identifier, the empty URL, or some previously visited URL parametric on a natural number.

```
sort Attribute .
sort AttributeSet .
subsort Attribute < AttributeSet .
op nil : -> AttributeSet .
op _,_ : AttributeSet AttributeSet
        -> AttributeSet [assoc comm id: nil] .
```

The state is made up of sets of attributes in the standard way.

```
sort ObjectName .
subsorts FrameName MarkupName < ObjectName .

sort Object .
```

```
op <_|_> : ObjectName AttributeSet -> Object .

**** FRAME's attributes
op currentMarkup:_ : MarkupName -> Attribute .
op pendingMarkup:_ : MarkupName -> Attribute .

**** MARKUP's attributes
op url:_ : Url -> Attribute .
op frame:_ : FrameName -> Attribute .
op tree:_ : DomTree -> Attribute .
endfm
```

Both `FrameName`s and `MarkupName`s are object names, and an object is the combination of a name and an attribute set. Potential attributes are the current and pending markups, identified by name, the URL, the frame and the associated DOM tree.

```
fmod EVENT is
including QID .
including CLASSES .

sort Event .

op startNavigation : Url FrameName -> Event .
op ready : MarkupName -> Event .
op ensure : -> Event .
op onPaint : -> Event .

op mark : Event -> Event .
endfm
```

Events are the start of navigation, the ready event for a specific markup, and the painting on the screen. Note that we do single out events representative of downloading (by using `mark` on them), and allow other events to bypass them, as the download can take a long time. See the `METHOD-CALLS` module for details.

```
mod EVENTQUEUE is
including EVENT .

sort EventQueue .
subsort Event < EventQueue .
```

```
op noq : -> EventQueue .
op _,_ : EventQueue EventQueue
         -> EventQueue [assoc id: noq] .
endm
```

The queue of events is a standard queue that is associative and has an empty neutral element.

```
fmod METHOD-DEF is
including CLASSES .

sort Method .

op FollowHyperlink : Url FrameName -> Method .
op PostMan : Url FrameName -> Method .
op SetInteractive : MarkupName -> Method .
op NavigationComplete : FrameName -> Method .
op FireNavigationComplete : FrameName -> Method .
```

All the expected function calls are listed as `Method` elements above, see Figure 3.9 in Chapter 3.3.2.

```
op GetPidlForDisplay : -> Method .
op SwitchMarkup : MarkupName FrameName -> Method .
```

Switching the markup requires a new markup, identified by its name, and the frame this switch happens on, also identified by its name.

```
op EnsureSize : -> Method .
op EnsureView : -> Method .
op RenderView : -> Method .
op HistoryBack : -> Method .
op BROWSERINSTANCE::Travel : Int -> Method .
op CTravelLog::Travel : Int -> Method .
op Invoke : -> Method .
op LoadHistory : -> Method .
op CreateMarkup : Bool -> Method .
op CreateFrameHelper : MarkupName -> Method .
```

All these are methods as seen in Figure 3.9 in Chapter 3.3.2.

```
sort HelpingMethod .
subsort HelpingMethod < Method .
```

```
op SetInteractive-BoolExp : MarkupName
     -> HelpingMethod .
op FireNavigationComplete-pidl : Url
     -> HelpingMethod .
op FireNavigationComplete-!fViewActivated : Url
     -> HelpingMethod .
op SetAddressBar : Url -> HelpingMethod .
op SwitchMarkup-PrimarySwitch : MarkupName FrameName
     -> HelpingMethod .
op SwitchMarkup-AllButLastIf :
     MarkupName FrameName Bool -> HelpingMethod .
op SwitchMarkup-SwapInNewMarkup :
     MarkupName FrameName Bool -> HelpingMethod .
op Invoke-PostEvent : -> HelpingMethod .
op CreateMarkup-PrimaryFrame-pMarkup->frame :
     MarkupName -> HelpingMethod .
endfm
```

Now, the methods above are not actual methods in the source code of IE, but partial versions of them. Generally, the part before the "-" is the method this happens in, and the part after the "-" shows which path inside it went down.

```
fmod METHOD-LIST is
including METHOD-DEF .

sort MethodList .
subsort Method < MethodList .
op nop : -> MethodList .
op _;_ : MethodList MethodList
          -> MethodList [assoc id: nop] .
endfm
```

The execution of methods happens in order, so we define lists of methods, as usual, being associative.

```
mod STATE is
including EVENTQUEUE .
including METHOD-LIST .

sort State .
sort StateElement .
```

```
subsort StateElement < State .
op nil : -> State .
op __ : State State -> State [assoc comm id: nil] .

sort ObjectMultiSet .
subsort Object < ObjectMultiSet .
op nil : -> ObjectMultiSet .
op __ : ObjectMultiSet ObjectMultiSet
         -> ObjectMultiSet [assoc comm id: nil] .

op {_} : ObjectMultiSet -> StateElement .
op [_] : EventQueue -> StateElement .
op [_] : MethodList -> StateElement .
```

The state consists of the event queue, and the list of methods being called as well as all of the objects that are wrapped inside { }.

```
op freshNameCounter : Nat -> StateElement .
op historyAccesses : Nat -> StateElement .
op painted : Bool -> StateElement .
```

The first two elements are counters internal to the model that are used to create new, fresh names. The `painted` element is used to remember whether an `onPaint` has already been injected at the very end.

```
op primaryFrame : FrameName -> StateElement .
op urlOfView : Url -> StateElement .
```

The above two elements are modeling the primary frame we are looking at as well as the URL of the current view. Technically speaking it should be indirected twice, that is, the browser instance has view, and that view has an associated URL. But, as we noted above that we are limiting ourselves to just one browser instance with only one view, we can avoid the extra hassle.

```
op addressBar : Url -> StateElement .
op urlPaintedOnScreen : Url -> StateElement .
endm
```

Finally, this represents the content of the address bar, respectively the URL where the content of the screen came from.

In the following, whenever there are two `[]` wrappers, the first will contain the method list, separated internally by ; while the second will be the event queue, separated by , internally.

```
mod METHOD-CALLS is
including STATE .

vars S S' : State . vars U U' : Url .
vars Q Q' : EventQueue . vars ML ML' : MethodList .
vars F F' : FrameName . vars N N' : Nat .
vars Atts Atts' : AttributeSet .
vars M M' M'' : MarkupName . vars D D' : DomTree .
vars OMS OMS' : ObjectMultiSet . vars B B' : Bool .
vars I I' : Int .

eq [FollowHyperlink(U, F) ; ML] [Q]
 = [ML] [Q, startNavigation(U, F)] .
```

If the next method call is a `FollowHyperlink` then that is replaced by adding a `startNavigation` event with the same arguments in the event queue, see Figure 3.9 in Chapter 3.3.2.

```
eq [PostMan(U, F) ; ML] [Q] freshNameCounter(N)
   {< F | pendingMarkup: M , Atts > OMS}
 = [ML] [Q, mark(ready(m(n(N))))]
   freshNameCounter(N + 1)
   {< F | pendingMarkup: m(n(N)) , Atts >
    < m(n(N)) | frame: F , url: U , tree: nil > OMS} .
```

A `PostMan` call requires an appropriate frame for which it will change the `pendingMarkup`, increase the counter of fresh names, add a new markup, with the specified URL, to the list of objects and put a `ready` event into the queue. Note that that event is marked, meaning it can be delayed, to simulate network transfer times.

```
---(
**** We force this to be FALSE
**** because otherwise there is a bad trace.
rl [_fIsInSetInteractive-TRUE] :
   [SetInteractive(M) ; ML]
=> [ML] .
)
```

```
rl [_fIsInSetInteractive-FALSE] :
   [SetInteractive(M) ; ML]
   {< M | frame: F , Atts > OMS}
=> [SwitchMarkup(M, F) ; SetInteractive-BoolExp(M) ;
    ML] {< M | frame: F , Atts > OMS} .
```

The first case simulates the possibility of immediately exiting method `SetInteractive`, by simply dropping it from the method list. This would happen in case of re-entry. It is commented out, as with that code active it is possible to get an attack because `primaryFrame->currentMarkup == NULL` is possible.

The other case is where the function properly executes, which leads to a call of `SwitchMarkup` and `SetInteractive-BoolExp`.

```
rl [BOOLEXP1-TRUE] :
   [SetInteractive-BoolExp(M) ; ML]
   {< M | frame: F , Atts > OMS}
=> [NavigationComplete(F) ; ML]
   {< M | frame: F , Atts > OMS} .

---(
**** We force this to be TRUE
**** because otherwise there is a bad trace.
rl [BOOLEXP1-FALSE] :
   [SetInteractive-BoolExp(M) ; ML]
=> [ML] .
)
```

In turn, `SetInteractive-BoolExp` goes to `NavigationComplete` in the proper execution with `BOOLEXP1` being true.

Alternatively, if `BOOLEXP1` is false, the method aborts right away and nothing happens. This can lead to a different potential attack scenario.

```
---(
**** We force this to be FALSE
**** because otherwise there is a bad trace.
rl [BOOLEXP2-TRUE] :
   [NavigationComplete(F) ; ML]
=> [ML] .
)

rl [BOOLEXP2-FALSE] :
   [NavigationComplete(F) ; ML]
=> [FireNavigationComplete(F) ; ML] .
```

Under the condition that BOOLEXP2 is true there is a silent return of NavigationComplete, which leads to an attack; otherwise normal execution continues and FireNavigationComplete is called as expected.

```
rl [bstrUrl-TRUE] :
   [FireNavigationComplete(F) ; ML]
   {< F | currentMarkup: M , Atts >
    < M | url: U , Atts' > OMS}
=> [GetPidlForDisplay ;
    FireNavigationComplete-pidl(U) ; ML]
   {< F | currentMarkup: M , Atts >
    < M | url: U , Atts' > OMS} .

---(
**** We force this to be TRUE
**** because otherwise there is a bad trace.
rl [bstrUrl-FALSE] :
   [FireNavigationComplete(F) ; ML]
=> [ML] .
)
```

For the bstrURL condition we have a silent return if it is false, leading to issues, and regular execution otherwise.

```
rl [pidl-TRUE] :
   [FireNavigationComplete-pidl(U) ; ML]
=> [FireNavigationComplete-!fViewActivated(U) ; ML] .

---(
**** We force this to be TRUE
**** because otherwise there is a bad trace.
rl [pidl-FALSE] :
   [FireNavigationComplete-pidl(U) ; ML]
=> [ML] .
)
```

If the pidl condition is false, there is another silent return; otherwise the FireNavigationComplete method is executed further. The decision about the pidl condition indeed happens only here, and it does not happen in GetPidlForDisplay.

```
rl [!fViewActivated-TRUE] :
   [FireNavigationComplete-!fViewActivated(U) ; ML]
=> [SetAddressBar(U) ; ML] .

---(
**** We force this to be TRUE
**** because otherwise there is a bad trace.
rl [!fViewActivated-FALSE] :
   [FireNavigationComplete-!fViewActivated(U) ; ML]
=> [ML] .
)
```

The `!fViewActivated` condition needs to be true for regular execution, calling `SetAddressBar`, otherwise it returns silently, leading to an attack and that is excluded here. Note that all of the above were rules, so Maude's search space exploration would explore all possibilities, until we took them out.

```
eq [GetPidlForDisplay ; ML]
 = [ML] .

rl [SetAddressBar] :
   [SetAddressBar(U) ; ML] addressBar(U')
=> [ML] addressBar(U) .
```

The `SetAddressBar` method actually does set the address bar, as expected.

```
---(
**** We force this to be FALSE
**** because otherwise there is a bad trace.
rl [!pMarkupNew->_fWindowPending-TRUE] :
   [SwitchMarkup(M, F) ; ML]
=> [ML] .
)

rl [!pMarkupNew->_fWindowPending-FALSE] :
   [SwitchMarkup(M, F) ; ML]
=> [SwitchMarkup-PrimarySwitch(M, F) ; ML] .
```

For `!pMarkupNew->_fWindowPending` we remove the true case as that silently returns from the `SwitchMarkup` method, leading to trouble. Otherwise it continues execution, having passed the first condition check. The potential trouble in the first case is that with `HistoryBack` the case that `primaryFrame->currentMarkup == NULL` is possible.

```
eq [SwitchMarkup-PrimarySwitch(M, F) ; ML]
   primaryFrame(F)
 = [SwitchMarkup-AllButLastIf(M, F, true) ; ML]
   primaryFrame(F) .

ceq [SwitchMarkup-PrimarySwitch(M, F) ; ML]
    primaryFrame(F')
  = [SwitchMarkup-AllButLastIf(M, F, false) ; ML]
    primaryFrame(F')
if F =/= F' .
```

These equations just check if the `SwitchMarkup` method has been called on the primary frame, and note that as `true` or `false` in a third argument of `SwitchMarkup-AllButLastIf`.

```
---(
**** We force this to be FALSE
**** because otherwise there is a bad trace.
rl [someIfStopsSwitchMarkup-TRUE] :
   [SwitchMarkup-AllButLastIf(M, F, B) ; ML]
=> [ML] .
)

rl [someIfStopsSwitchMarkup-FALSE] :
   [SwitchMarkup-AllButLastIf(M, F, B) ; ML]
=> [SwitchMarkup-SwapInNewMarkup(M, F, B) ; ML] .
```

The first case here represents the case when some of the remaining conditions are responsible for making it silently return; otherwise the execution continues.

```
eq [SwitchMarkup-SwapInNewMarkup(M, F, true) ; ML] [Q]
   {< F | currentMarkup: M' ,
          pendingMarkup: M'' , Atts > OMS}
 = [ML] [Q , ensure]
   {< F | currentMarkup: M ,
          pendingMarkup: m(nil) , Atts > OMS} .

eq [SwitchMarkup-SwapInNewMarkup(M, F, false) ; ML]
   {< F | currentMarkup: M' ,
          pendingMarkup: M'' , Atts > OMS}
 = [ML]
   {< F | currentMarkup: M ,
          pendingMarkup: m(nil) , Atts > OMS} .
```

Switching in a new markup will add `ensure` to the event queue if it is happening on the primary frame; otherwise it will not.

```
rl [IsActiveEnsureSize-TRUE] :
   [EnsureSize ; ML] urlOfView(U) primaryFrame(F)
   {< F | currentMarkup: M , Atts >
    < M | url: U' , Atts' > OMS}
=> [ML] urlOfView(U') primaryFrame(F)
   {< F | currentMarkup: M , Atts >
    < M | url: U' , Atts' > OMS} .

---(
**** We force this to be TRUE
**** because otherwise there is a bad trace.
rl [IsActiveEnsureSize-FALSE] :
   [EnsureSize ; ML]
=> [ML] .
)
```

Depending on `IsActiveEnsureSize` the `EnsureSize` method can silently return, leading to an attack, or change the content that will be made visible on screen shortly. In case you are wondering how this would lead to an attack, then note that in the attack case the things displayed on screen are not changing, but independently of this the URL displayed in the address bar will!

```
rl [IsActiveEnsureView-TRUE] :
   [EnsureView ; ML]
=> [EnsureSize ; ML] .

---(
**** We force this to be TRUE
**** because otherwise there is a bad trace.
rl [IsActiveEnsureView-False] :
   [EnsureView ; ML]
=> [ML] .
)
```

From EnsureView either EnsureSize gets called, or in the case that IsActiveEnsureView is false it just returns, leading to an attack.

```
rl [pRenderSurface!=NULL-TRUE] :
   [RenderView ; ML]
   urlOfView(U) urlPaintedOnScreen(U')
=> [ML] urlOfView(U) urlPaintedOnScreen(U) .

---(
**** We force this to be TRUE
**** because otherwise there is a bad trace.
rl [pRenderSurface!=NULL-FALSE] :
   [RenderView ; ML]
=> [ML] .
)
```

In RenderView the content of urlOfView will be painted on the screen, or if pRenderSurface!=NULL is false it will just return. Again leading to the attack mentioned just before and thus excluded.

```
eq [HistoryBack ; ML]
 = [BROWSERINSTANCE::Travel(-1) ; ML] .

rl [pTravelLog&&_pBrowserSvc-TRUE-and-LPAREN_
    fViewLinkedInWebOC-FALSE-or-hr-FALSE-RPAREN] :
   [BROWSERINSTANCE::Travel(I) ; ML]
=> [CTravelLog::Travel(I) ; ML] .

rl [LPAREN_pTravelLog&&_pBrowserSvc-TRUE-and-
    LPAREN_fViewLinkedInWebOC-FALSE-or-
    hr-FALSE-RPAREN-RPAREN--FALSE] :
   [BROWSERINSTANCE::Travel(I) ; ML]
=> [ML] .
```

Here, HistoryBack becomes BROWSERINSTANCE::Trave(-1). Then, there is two possible outcomes to BROWSERINSTANCE::Travel: it can either execute and call CTravelLog::Travel or just abort.

```
rl [SUCCEEDED-TRUE] :
   [CTravelLog::Travel(I) ; ML]
=> [Invoke ; ML] .

rl [SUCCEEDED-FALSE] :
   [CTravelLog::Travel(I) ; ML]
=> [ML] .
```

Depending on SUCCEEDED the travel will call Invoke or return.

```
rl [hGlobal!=NULL&&SUCCEEDED-TRUE] :
   [Invoke ; ML]
=> [LoadHistory ; Invoke-PostEvent ; ML] .

rl [hGlobal!=NULL&&SUCCEEDED-FALSE] :
   [Invoke ; ML]
=> [ML] .
```

To continue, Invoke can either call LoadHistory ; Invoke-PostEvent or return.

```
eq [Invoke-PostEvent ; ML] [Q]
   primaryFrame(F) historyAccesses(N)
 = [ML] [Q , startNavigation(someHistoryUrl(N), F)]
   primaryFrame(F) historyAccesses(N + 1) .

rl [pstm&&SUCCEEDED-TRUE-and-LPAREN_pHTMLDocument-FALSE
    -or-_dwDocFlags&&DOCFLAC_...-FALSE-RPAREN] :
   [LoadHistory ; ML]
=> [CreateMarkup(true) ; ML] .

rl [LPARENpstm&&SUCCEEDED-TRUE-and-LPAREN_
    pHTMLDocument-FALSE-or-_dwDocFlags&&
    DOCFLAC_...-FALSE-RPAREN-RPAREN--FALSE] :
   [LoadHistory ; ML]
=> [ML] .
```

Invoke-PostEvent will then add a new event StartNavigation to the event queue. Depending on the relevant condition, LoadHistory will either call CreateMarkup or silently return.

```
rl [!pMarkup-TRUE] :
   [CreateMarkup(B) ; ML]
=> [ML] .

rl [!pMarkup-FALSE] :
 [CreateMarkup(true) ; ML] freshNameCounter(N)
   {OMS}
=> [CreateFrameHelper(m(n(N))) ;
```

```
    CreateMarkup-PrimaryFrame-pMarkup->frame(m(n(N))) ;
    ML]
   freshNameCounter(N + 1)
   {< m(n(N)) | url: noUrl , frame: f(nil) ,
                tree: nil > OMS} .

rl [!pMarkup-FALSE] :
   [CreateMarkup(false) ; ML] freshNameCounter(N)
   {OMS}
=> [ML] freshNameCounter(N + 1)
   {< m(n(N)) | url: noUrl , frame: f(nil) ,
                tree: nil > OMS} .
```

In the first rule above, `!pMarkup` being true, the system is out of memory, so nothing has been created. In the second and third rule a new markup is created. Only in the second rule a method is called that will create an associated frame, and then the rest of the `CreateMarkup` method gets executed. In the third rule there is just a return from that method.

```
eq [CreateMarkup-PrimaryFrame-pMarkup->frame(M) ; ML]
   primaryFrame(F')
   {< M | frame: F , Atts > OMS}
 = [ML] primaryFrame(F)
   {< M | frame: F , Atts > OMS} .
```

This is the continuation of `CreateMarkup`, which simply changes the primary frame to that call's argument.

```
---(
**** We force this to be FALSE
**** because otherwise there is a bad trace.
rl [!pFrame-TRUE] :
   [CreateFrameHelper(M) ; ML]
=> [ML] .
)

rl [!pFrame-FALSE] :
   [CreateFrameHelper(M) ; ML] freshNameCounter(N)
   {< M | frame: F , Atts > OMS}
=> [ML] freshNameCounter(N + 1)
   {< M | frame: f(n(N)) , Atts >
    < f(n(N)) | currentMarkup: m(nil) ,
                pendingMarkup: M > OMS} .
```

```
endm
```

In the first rule we are out of memory, and thus an attack is possible as it just silently returns. The second one is the creation of the frame associated to the markup.

Now we have covered all the method calls, so it is time to switch over to the handling of the events.

```
mod EVENT-HANDLING is
including STATE .

vars S S' : State . vars U U' : Url .
vars Q Q' : EventQueue . vars ML ML' : MethodList .
vars F F' : FrameName . vars N N' : Nat .
vars Atts Atts' : AttributeSet .
vars M M' : MarkupName . vars D D' : DomTree .
vars OMS OMS' : ObjectMultiSet . vars E E' : Event .

eq [startNavigation(U, F) , Q] [nop] S
 = [Q] [PostMan(U, F)] S .
```

The `startNavigation` event will call the `PostMan` method, if the method call list is empty, and it is the first event in the queue.

```
eq [ready(M) , Q] [nop]
   {< M | tree: D , url: U , Atts > OMS} S
 = [Q] [SetInteractive(M)]
   {< M | tree: dl(U), url: U , Atts > OMS} S .

eq [ensure , Q] [nop] S
 = [Q] [EnsureView] S .

eq [onPaint , Q] [nop] S
 = [Q] [RenderView] S .
```

When `ready` is the first event (and is not marked, see below) the DOM tree associated to the markup parameter of ready will be downloaded from its URL, shown as `dl(U)`. The `tree` it gets put in has originally been initialized with the empty DOM tree.

The `ensure` event calls the `EnsureView` method, while the `onPaint` event triggers the `RenderView` method.

```
rl [marked-event-delayed] :
   [mark(E) , E' , Q]
=> [E' , mark(E) , Q] .

rl [marked-event-happens] :
   [mark(E) , Q]
=> [E , Q] .
```

Marked events, shown by `mark`, can be delayed. This means that the event following them can be moved in front of the marked event, or the mark can be removed, which means the event is now ready for execution.

```
eq [nop] [noq] painted(false)
 = [nop] [onPaint] painted(true) .
endm
```

When there are no more method calls and no more events in the event queue, and the screen has not yet been painted, then the `onPaint` event is added once.

Next we define the possible starting points for execution and define what it means for a state to be a good state.

```
mod CREATE-ALL-STARTS is
including METHOD-CALLS .
including EVENT-HANDLING .

op goodState : State -> Bool .
op badState : State -> Bool .

vars S S' : State . vars U U' : Url .
vars Q Q' : EventQueue . vars ML ML' : MethodList .
vars F F' : FrameName . vars N N' : Nat .
vars Atts Atts' : AttributeSet .
vars M M' M'' : MarkupName . vars D D' : DomTree .
vars OMS OMS' : ObjectMultiSet .
vars B B' : Bool . vars I I' : Int .

eq badState(S) = not(goodState(S)) .

eq goodState(addressBar(U) urlOfView(U)
             urlPaintedOnScreen(U) primaryFrame(F)
           {< F | currentMarkup: M , Atts >
            < M | url: U , tree: dl(U) , Atts' >
            OMS} S)
 = true .

eq goodState(S)
 = false [owise] .
```

More specifically, a good state is a state where the address bar matches with what is painted on the screen, i.e., the content on the screen is downloaded from the URL in the address bar. Also, the internal `urlOfView` is the same and the URL of the current markup of the primary frame matches it as well.

In the following `FH` stands for “follow hyperlink” while `HB` stands for “history back”.

```
op consistent-state : -> State .
op startFH : -> State .
op startHB : -> State .
op startFH-HB : -> State .
op startHB-FH : -> State .
op startFH-FH : -> State .
op startHB-HB : -> State .

eq consistent-state
 = primaryFrame(f('f0)) [noq]
   { < f('f0) | currentMarkup: m('m0) ,
                pendingMarkup: m(nil) >
     < m('m0) | url: 'urlA , frame: f('f0) ,
                tree: dl('urlA) > }
   addressBar('urlA) urlOfView('urlA)
   urlPaintedOnScreen('urlA) freshNameCounter(0)
   historyAccesses(0) painted(false) .
```

The consistent state from which all our experiments start is given above. The starting state is then customized by the method call given, see below.

```
eq startFH
 = consistent-state
   [FollowHyperlink('urlB, f('f0))] .

eq startHB
 = consistent-state
   [HistoryBack] .
```

```
eq startFH-HB
 = consistent-state
   [FollowHyperlink('urlB, f('f0)) ; HistoryBack] .

eq startHB-FH
 = consistent-state
   [HistoryBack ; FollowHyperlink('urlB, f('f0))] .

eq startFH-FH
 = consistent-state
   [FollowHyperlink('urlB, f('f0)) ;
    FollowHyperlink('urlC, f('f0))] .

eq startHB-HB
 = consistent-state
   [HistoryBack ; HistoryBack] .

endm
```

We either only give one command to the browser, like in the first case, or two commands that can be any combination of following a hyperlink and navigating back in the history.

Let us look back at Table 3.5 in Chapter 3.3.4 and let us pick the scenario based on condition No. 2, which is a silent return of the method `FireNavigationComplete`. We find it as an attack scenario in the first search, with just one call to `FollowHyperlink`, as such:

```
search startFH =>! S:State such that badState(S:State).
```

This uses the rule we have labeled with `bstrUrl-FALSE`, which is the condition triggering this silent return and is based on a bad format of the URL. The search result we get is the following

```
state 0, State:
{< f('f0) | currentMarkup: m('m0),
            pendingMarkup: m(n(0)) >
< m('m0) | url: 'urlA,frame: f('f0),tree: dl('urlA) >
< m(n(0)) | url: 'urlB, frame: f('f0),tree: nil >}
[mark(ready(m(n(0))))] [nop]
freshNameCounter(1) historyAccesses(0)
painted(false) primaryFrame(f('f0))
addressBar('urlA)
```

```
urlOfView('urlA) urlPaintedOnScreen('urlA)
===[ rl [mark(E:Event),Q] => [E:Event,Q]
[label marked-event-happens] . ]===>
===[ rl {OMS < M | Atts,frame: F >}
        [SetInteractive(M) ; ML] =>
        {OMS < M | Atts,frame: F >}
        [SwitchMarkup(M, F) ;
        SetInteractive-BoolExp(M) ; ML]
[label _fIsInSetInteractive-FALSE] . ]===>
===[ rl [SwitchMarkup(M, F) ; ML] =>
        [SwitchMarkup-PrimarySwitch(M, F) ; ML]
[label !pMarkupNew->_fWindowPending-FALSE] . ]===>
===[ rl [SwitchMarkup-AllButLastIf(M, F, B) ; ML] =>
        [SwitchMarkup-SwapInNewMarkup(M, F, B) ; ML]
[label someIfStopsSwitchMarkup-FALSE] . ]===>
===[ rl {OMS < M | Atts,frame: F >}
        [SetInteractive-BoolExp(M) ; ML] =>
        {OMS < M | Atts,frame: F >}
        [NavigateComplete(F) ; ML]
[label BOOLEXP1-TRUE] .  ]===>
===[ rl [NavigateComplete(F) ; ML] =>
        [FireNavigateComplete ; ML]
[label BOOLEXP2-FALSE] . ]===>
===[ rl [FireNavigateComplete ; ML] => [ML]
```

The next rule being applied (labeled `bstrUrl-FALSE`) simply drops the `FireNavigateComplete`, which is a silent return that does not change the address bar URL ultimately, while the content is changed. This is the crucial step in this execution.

```
[label bstrUrl-FALSE] . ]===>
===[ rl [EnsureView ; ML] => [EnsureSize ; ML]
[label IsActiveEnsureView-TRUE]  . ]===>
===[ rl {OMS < F | Atts,currentMarkup: M >
        < M | Atts',url: U' >} [EnsureSize ; ML]
        primaryFrame(F) urlOfView(U) =>
        [ML] urlOfView(U')
        {< F | Atts, currentMarkup: M > OMS
         < M | Atts',url: U' >} primaryFrame(F)
[label IsActiveEnsureSize-TRUE] . ]===>
===[ rl [RenderView ; ML]
        urlOfView(U) urlPaintedOnScreen(U') =>
        [ML] urlOfView(U) urlPaintedOnScreen(U)
[label pRenderSurface!=NULL-TRUE] .  ]===>
```

```
state 22, State:
{< f('f0) | currentMarkup: m(n(0)),
            pendingMarkup: m(nil) >
< m('m0) | url: 'urlA,frame: f('f0),
           tree: dl('urlA) >
< m(n(0)) | url: 'urlB, frame: f('f0),
            tree: dl('urlB) >}
[noq] [nop]
freshNameCounter(1) historyAccesses(0)
painted(true) primaryFrame(f('f0))
addressBar('urlA)
urlOfView('urlB) urlPaintedOnScreen('urlB)
```

As you can see in the final state, the URL in the address bar has not been updated according to the rest of the execution, but the content of the page the user sees did update. Thus, changing the content and changing the address bar URL need to be linked, or even be one atomic action.

APPENDIX B

EXPLAINED MAUDE SPECIFICATION OF THE IBOS MODEL

In this chapter we list and describe the detailed specification of the Maude model of the IBOS browser. All the data source files are available at [112]. We are using Maude 2.6 for all of these experiments.

First, we explain the model architecture of IBOS in Section B.1. Then we explain the extension for and analysis of the display memory in Section B.3. The analysis for the address bar and SOP will be described in Section B.4.

In this section, whenever we write *kernel* we are referring to the *IBOS kernel*. We will refer to the underlying L4KA::Pistachio microkernel operating system with more detail.

B.1 IBOS - Model Architecture

The code explained in this section is found in file `ibos.maude`. You can see the overall state structure in Figure 4.2.

We start with process identifiers, which are given in module `PROC-ID`. As there is only a single kernel, and a single web app manager, the process id `kernel-id`, respectively `webappmgr-id` will be that one object's id. On the other hand, `webapp-id` and `network-id` are only place holders, which are used in policies exclusively. Any actual web app or network process will have a natural number as id, which is between 1024 and 1055 for web apps, and between 256 and 1023 for network processes. Note that the limitation to 32 web apps is due to memory concerns in the IBOS implementation.

```
mod PROC-ID is
including CONFIGURATION .
protecting INT .

sort ProcId .
```

```
subsort Int < ProcId  < Oid .

op kernel-id : -> ProcId .
op webappmgr-id : -> ProcId .

op webapp-id : -> ProcId .
op network-id : -> ProcId .

op cache-id : -> ProcId .
op cookie-id : -> ProcId .
op vesafb-server-id : -> ProcId .
op mouse-server-id : -> ProcId .
op network-server-id : -> ProcId .
op dns-server-id : -> ProcId .
op ui-id : -> ProcId .
op mouse-intr-id : -> ProcId .
op network-intr-id : -> ProcId .
op storage-id : -> ProcId .
```

Now all elements in our object soup are going to be processes, identified by `proc` as their class id, with the exception of the kernel, addressed as `< kernel-id : kernel |`, the NIC, which will be addressed as `< 0 : nic |` as there is only one, and the DMA memory used for communication from network process to NIC, addressed as `< N : nic |`, with `N` the process id of that network process.

```
op proc : -> Cid [ctor] .
op kernel : -> Cid [ctor] .
op nic : -> Cid [ctor] .
op mem : -> Cid [ctor] .
endm
```

There are different types of messages as well as values for messages, shown in the `MSG-TYPE` module.

```
mod MSG-TYPE is
protecting INT .
sort MsgType .
sort MsgVal .
subsort Int < MsgVal .

op MSG-NEW-URL                   : -> MsgType .
op MSG-FETCH-URL                 : -> MsgType .
op MSG-RETURN-URL                  : -> MsgType .
op MSG-FETCH-URL-ABORT             : -> MsgType .
op MSG-SWITCH-TAB                  : -> MsgType .
op MSG-RETURN-URL-METADATA         : -> MsgType .

op MSG-COOKIE-SET                  : -> MsgType .
op MSG-COOKIE-GET                  : -> MsgType .
op MSG-COOKIE-GET-RETURN           : -> MsgType .

op MSG-DOM-COOKIE-SET              : -> MsgType .
op MSG-DOM-COOKIE-GET              : -> MsgType .
op MSG-DOM-COOKIE-GET-RETURN       : -> MsgType .

op MSG-WRITE-FILE                  : -> MsgType .
op MSG-READ-FILE                   : -> MsgType .
op MSG-READ-FILE-RETURN            : -> MsgType .
op MSG-DOWNLOAD-INFO               : -> MsgType .

op MSG-WEBAPP-MSG                  : -> MsgType .
op MSG-UI-MSG                      : -> MsgType .
endm
```

The different types of system calls that are available due to using L4Ka::Pistachio. The system call `OPOS-SYSCALL-FD-SEND-MESSAGE` is the one that is used most often.

```
mod SYSCALL-TYPE is
sort SyscallType .
op OPOS-SYSCALL-FD-SEND-MESSAGE : -> SyscallType .
op OPOS-SYSCALL-CREATE-PROCESS : -> SyscallType .
op OPOS-SYSCALL-REGISTER-IRQ-THREAD : -> SyscallType .
op OPOS-SYSCALL-GET-RESERVE-MEM : -> SyscallType .
op OPOS-SYSCALL-GET-SERVICE-TID : -> SyscallType .
op OPOS-SYSCALL-REGISTER-SERVICE : -> SyscallType .
op OPOS-SYSCALL-ALLOCATE-DMA-MEMORY : -> SyscallType .
op OPOS-SYSCALL-POLL : -> SyscallType .
op OPOS-SYSCALL-FD-RECEIVE-MSSAGE : -> SyscallType .
op OPOS-SYSCALL-E1000-SEND-ETHERNET-PACKET :
     -> SyscallType .
op OPOS-SYSCALL-E1000-PARSE-INTERRUPT-RESULT :
     -> SyscallType .
op OPOS-SYSCALL-E1000-IF-UP : -> SyscallType .
op OPOS-SYSCALL-REGISTER-SUBSYSTEM : -> SyscallType .
op OPOS-SYSCALL-GET-FB-MEMORY : -> SyscallType .
op OPOS-SYSCALL-IS-WINDOW-MGR : -> SyscallType .
op OPOS-SYSCALL-NET-IS-PORT-AVAILABLE : -> SyscallType .
op OPOS-SYSCALL-NET-ALLOCATE-PORT : -> SyscallType .
op OPOS-SYSCALL-NET-FREE-PORT : -> SyscallType .
op OPOS-SYSCALL-TOUCH : -> SyscallType .
endm
```

The module `PAYLOAD` includes the wrapper for browser level messages, called `payload`, which takes seven arguments. The first two arguments are of type `Oid`, representing the sender and receiver of the message. Note again that the sender `Oid` is enforced to be correct by the kernel when the message is passed on. In some cases the receiver `Oid` can be changed by the kernel as well. The type of the message, of sort `MsgType`, is the next argument, with a more specific message value `MsgVal` if needed. The `String` argument will take, e.g., the URL of any message that has one. We are going to ignore the `typed` and `untyped` argument, but they could store additional data, if needed.

```
mod PAYLOAD is
protecting MSG-TYPE .
protecting PROC-ID .
protecting STRING .
sort typed .
op mtTyped : -> typed [ctor] .
sort untyped .
op mtUntyped : -> untyped [ctor] .
sort Payload .
op payload : Oid Oid MsgType MsgVal String typed untyped
               -> Payload [ctor] .
endm
```

We are using pipes for different objects to communicate with each other. Each pipe is bidirectional between the two objects it connects. Note that for all pipes one of those objects is always the kernel, as all communication is forced to go through the kernel.

Therefore, `pipe` is an operator that is used as object class, `Cid`. Each pipe will be identified by an id, which is the same as that of the process the pipe is connecting to the kernel. The incoming messages are stored inside the wrapper `fromKernel`, while outgoing messages are put into the wrapper `toKernel`. Messages are stored in a list of messages, `MessageList`, while each message itself is made up of a system call and the payload.

```
mod MSG-PIPE-BASICS is
including CONFIGURATION .
protecting PROC-ID .
protecting MSG-TYPE .
protecting SYSCALL-TYPE .
protecting PAYLOAD  .
protecting STRING .
protecting INT .

op pipe : -> Cid [ctor] .

op fromKernel : MessageList -> Attribute [ctor] .
op toKernel : MessageList -> Attribute [ctor] .

sort Message .
sort MessageList .
subsort Message < MessageList .
op msg : SyscallType Payload -> Message [ctor] .
op mt : -> Message [ctor] .
op _,_ : MessageList MessageList
        -> MessageList [ctor assoc id: mt] .
```

A helper function is defined here, that given an identifier and a message changes the sender identifier in that message's payload to the given identifier.

```
op changeRecipient : Oid Message -> Message .
eq changeRecipient(Num'':Nat,
     msg(ST:SyscallType,
         payload(Num:Nat, N':ProcId, M:MsgType, V:MsgVal,
                 S:String, T:typed, U:untyped)))
   = msg(ST:SyscallType,
     payload(Num:Nat, Num'':Nat, M:MsgType, V:MsgVal,
             S:String, T:typed, U:untyped)) .
endm
```

The web app manager is keeping track of the next unused process id number for a web app, wrapped in `nextWAN`. Note that web apps start at id `1024` and end at `1055`, due to memory limits in the actual IBOS implementation. Note that, even though this is defined in its own module, ultimately in our model this is an attribute of the kernel.

```
mod WEBAPPMGR is
inc MSG-PIPE-BASICS .
op nextWAN : Int -> Attribute [ctor] .
endm
```

A URL is represented by a `Label`, wrapped inside `l`, made up of a protocol, a domain and a port. We subsort `Label` to `String` for practical reasons, noting that obviously a string representation would be possible for the whole URL, but less convenient to work with. Lists of labels follow the usual conventions.

```
mod LABEL is
inc MSG-PIPE-BASICS .

sort Domain .
op dom : String -> Domain [ctor] .

sort Port .
op port : Nat -> Port [ctor] .

sort Protocol .
op http : -> Protocol [ctor] .
op https : -> Protocol [ctor] .

sort Label .
op about-blank : -> Label [ctor] .
op l : Protocol Domain Port -> Label [ctor] .
subsort Label < String .

sort LabelList .
subsort Label < LabelList .
op mtLL : -> LabelList [ctor] .
op _,_ : LabelList LabelList
         -> LabelList [ctor assoc id: mtLL] .
endm
```

Note that we represent data put on the screen only by its source, in the form of an URL. Web apps are represented with a number of attributes. The URL of the data that will be rendered on the screen (if this is the active web app) is stored in `rendered`, the URL where the web app will load its data from is stored in `URL`. The marker inside `loading` shows whether this web app has already sent out the request for data, if it needs to refresh or has not yet loaded any data. A 0 represents that such a message was not yet sent. The check `isWebapp` is used to see if the given `Oid` is the id of an actual web app or the web app place holder, or not.

```
mod WEBAPP is
inc LABEL .

op rendered : Label -> Attribute [ctor] .
op URL : Label -> Attribute [ctor] .
op loading : Nat -> Attribute [ctor] .

op isWebapp : Oid -> Bool .
eq isWebapp(Num:Nat)
 = (1024 <= Num:Nat) and (Num:Nat < 1056) .
eq isWebapp(webapp-id) = true .
eq isWebapp(O:Oid) = false [owise] .
```

The equation below sends the message to load the data from the given URL `L'`, in case it has not yet been sent (`loading(0)`). The receiver of this message is the `Oid` representing any network process `network-id`, the kernel will find the right network process in this case.

The rule below changes the URL that is (or would be) rendered appropriately with the return message. Note that the checks for whether this return message is acceptable happen elsewhere, at `checkConnection`.

```
vars N N' : Nat .
vars ML : MessageList .
vars L L' : Label .
vars Att Att2 : AttributeSet .

eq < N : proc | rendered(L) , URL(L') , loading(0) , Att >
   < N : pipe | toKernel(ML) , Att2 >
 = < N : proc | rendered(L) , URL(L') , loading(1) , Att >
   < N : pipe | toKernel(ML,
      msg(OPOS-SYSCALL-FD-SEND-MESSAGE,
          payload(N, network-id, MSG-FETCH-URL, 0, L',
                  mtTyped, mtUntyped))) , Att2 > .

  rl < N : proc | rendered(L) , URL(L') ,loading(1) , Att >
     < N : pipe | fromKernel(
        msg(OPOS-SYSCALL-FD-SEND-MESSAGE,
           payload(N', N, MSG-RETURN-URL, V:MsgVal,
                 LL:Label, T:typed, U:untyped)),
        ML) , Att2 >
   => < N : proc | rendered(LL:Label) , URL(L') ,
                 loading(1) , Att >
      < N : pipe | fromKernel(ML) , Att2 > .
endm
```

The network process receives messages and sends them out to the network via the NIC. Part of the necessary steps can be found below in the actual kernel, as they are done by the kernel. In this model, we do not include latency or anything like that, but just allow each message sent by the NIC to immediately trigger a response. Step by step it goes like this: (i) network process gets a request, (ii) network process forms an ethernet frame, (iii) kernel checks that frame and gives it to the NIC, (iv) NIC generates answer immediately as an ethernet frame, (v) that return ethernet frame gets handed to the (correct) network process again, (vi) which then returns it to the sender of the original request.

Step by step, we first need to be able to identify network processes by their ids which go from 256 to 1023 and the generic place holder `network-id`. A network process has lists of labels `in` and `out`, representing incoming and outgoing data, as well as the id of the process it wants to return data to `returnTo`.

```
mod NETWORK is
inc LABEL .

****  need to be able to check if it is a network process
op isNetProc : Oid -> Bool .
eq isNetProc(Num:Nat)
 = (256 <= Num:Nat) and (Num:Nat < 1023) .
eq isNetProc(network-id) = true .
eq isNetProc(O:Oid) = false [owise] .

var N : Nat .
vars ML ML' : MessageList .
vars Att Att2 : AttributeSet .

op returnTo : ProcId -> Attribute [ctor] .
op in : LabelList -> Attribute [ctor] .
op out : LabelList -> Attribute [ctor] .
```

This rule is a request from a web app and how it is received by the network process. Note that this has gone through the kernel before getting to this point, as it is in the in-bound message queue for the network process. The network process takes the URL to be fetched, `L:Label`, into its out-bound queue.

```
**** request from a webapp:
crl < N : proc | returnTo(SomeProcNum:Nat) ,
                 out(Ll:LabelList) , Att >
    < N : pipe | toKernel(ML) , fromKernel(
        msg(OPOS-SYSCALL-FD-SEND-MESSAGE,
            payload(Num:Nat, N, MSG-FETCH-URL, V:MsgVal,
                    L:Label, T:typed, U:untyped)),
        ML') , Att2 >
    => < N : proc | returnTo(Num:Nat) ,
                    out(Ll:LabelList, L:Label) , Att >
      < N : pipe | toKernel(ML) , fromKernel(ML') , Att2 >
 if isNetProc(N) .
```

Then the network process writes the ethernet frame into the assigned memory for pick-up by the NIC; note that the kernel will be involved in the following step.

```
rl < N : proc | out(L:Label, Ll:LabelList) , Att >
   < N : mem | out(mtLL) , Att2 >
=> < N : proc | out(Ll:LabelList) , Att >
   < N : mem | out(L:Label) , Att2 > .
```

Looking at the general explanation above, the following step is missing and can be found in the kernel. See this DMA related rule there. The NIC will create a return message right away, but note that the order of returns is not guaranteed.

```
rl < 0 : nic | out(Ll:LabelList , L:Label , Ll':LabelList) ,
               in(Ll'':LabelList) , Att >
=> < 0 : nic | out(Ll:LabelList , Ll':LabelList) ,
               in(Ll'':LabelList, L:Label) , Att > .
```

Again, there is another step missing now, to be found in the kernel, which assigns the incoming ethernet frame to the appropriate network process, you can see this DMA related rule below. Then the network process can just read the return from its memory for NIC contact, but remember it went through the kernel and the appropriate check in the step before.

```
  rl < N : proc | in(Ll:LabelList) , Att >
     < N : mem | in(L:Label) , Att2 >
  => < N : proc | in(Ll:LabelList, L:Label) , Att >
     < N : mem | in(mtLL) , Att2 > .
```

This equation now sends the return message from the network process to the web app, which of course will be subject to all the usual checks by the kernel later on.

```
ceq < N : proc | returnTo(Num:Nat) ,
                 in(L:Label, Ll:LabelList) , Att >
    < N : pipe | toKernel(ML) , fromKernel(ML') , Att2 >
  = < N : proc | returnTo(Num:Nat) , in(Ll:LabelList) , Att >
    < N : pipe | toKernel(ML,
       msg(OPOS-SYSCALL-FD-SEND-MESSAGE,
           payload(N, Num:Nat, MSG-RETURN-URL, 0, L:Label,
                  mtTyped, mtUntyped))) ,
                 fromKernel(ML') , Att2 >
if isNetProc(N) .
endm
```

The following module, `KERNEL-POLICIES` catches the vast majority of all things the kernel is doing actually. First, we do deal with policies. Sets of policies are defined as usual. The set of policies is wrapped by `msgPolicy` and stored as an attribute of the kernel. A policy itself is a sender and receiver `Oid` and a `MsgType`. Note that the sender and/or the receiver can be the catch-all identifiers for, e.g., all web apps or all network processes.

```
mod KERNEL-POLICIES is
inc MSG-PIPE-BASICS .
inc WEBAPPMGR .
inc LABEL .
inc WEBAPP .
inc NETWORK .

sort Policy .
sort PolicySet .
subsort Policy < PolicySet .
op mtPS : -> PolicySet .
op _,_ : PolicySet PolicySet
        -> PolicySet [assoc comm id: mtPS] .

op msgPolicy : PolicySet -> Attribute [ctor] .
op policy : Oid Oid MsgType -> Policy [ctor] .
```

The next available process id for a network process is wrapped by the operator `nextNetworkProc`. The wrapper `handledCurrently` is the attribute in the kernel that stores the message that the kernel is currently working on. There is only ever going to be one message in there, or potentially no message if the kernel is waiting for other processes.

```
**** the next available proc id for a network proc
op nextNetworkProc : Nat -> Attribute [ctor] .

**** the message currently handled by the kernel
op handledCurrently : Message -> Attribute [ctor] .
```

This is the mapping of web apps to URLs. It is stored as an attribute in the kernel, called `weblabels`, and it is a set of process ids being mapped to labels. The label given to a web app here, needs to match the first label of the mapping for network processes.

```
sort WebappProcInfo .
op pi : ProcId Label -> WebappProcInfo [ctor] .

sort WebappProcInfoSet .
subsort WebappProcInfo < WebappProcInfoSet .
op mtWPIS : -> WebappProcInfoSet [ctor] .
op _,_ : WebappProcInfoSet WebappProcInfoSet
        -> WebappProcInfoSet [assoc comm id: mtWPIS] .
op weblabels : WebappProcInfoSet -> Attribute [ctor] .
```

This is the mapping for network processes, called `networklabels`, which is similarly stored as a set. It maps each process id to two labels, where the first label has to match the label of the communicating web app, and the second label is where messages are actually sent to in the network (or NIC here).

```
sort NetworkProcInfo .
op pi : ProcId Label Label -> NetworkProcInfo [ctor] .
```

```
sort NetworkProcInfoSet .
subsort NetworkProcInfo < NetworkProcInfoSet .
op mtNPIS : -> NetworkProcInfoSet [ctor] .
op _,_ : NetworkProcInfoSet NetworkProcInfoSet
         -> NetworkProcInfoSet [assoc comm id: mtNPIS] .
op networklabels : NetworkProcInfoSet -> Attribute [ctor] .
```

Now we are getting back to the DMA rules that we could not put in the network process. This rule checks that the target is what is allowed for the network process by matching `L':Label` in the outgoing DMA memory to the destination stored in the kernel, before passing it on to the NIC. Similarly for the second rule in reverse, for the return message in the NIC, the right network process is found in the kernel stored mapping and then the incoming data is given to that DMA block.

```
rl < N : mem | out(L':Label) , Att >
   < kernel-id : kernel |
     networklabels(pi(N, L:Label, L':Label),
                   NPIS:NetworkProcInfoSet) ,
     Att2 >
   < 0 : nic | out(Ll:LabelList) , Att3 >
=> < N : mem | out(mtLL) , Att >
   < kernel-id : kernel |
     networklabels(pi(N, L:Label, L':Label),
                   NPIS:NetworkProcInfoSet) ,
     Att2 >
   < 0 : nic | out(Ll:LabelList, L':Label) , Att3 > .

rl < N : mem | in(mtLL) , Att >
   < kernel-id : kernel |
     networklabels(pi(N, L:Label, L':Label),
                   NPIS:NetworkProcInfoSet) ,
     Att2 >
   < 0 : nic | in(L':Label, Ll:LabelList) , Att3 >
=> < N : mem | in(L':Label) , Att >
   < kernel-id : kernel |
     networklabels(pi(N, L:Label, L':Label),
                   NPIS:NetworkProcInfoSet) ,
     Att2 >
   < 0 : nic | in(Ll:LabelList) , Att3 > .
```

Once policy checking is completed, e.g., that a `MSG-FETCH-URL` can be sent from a web app to a network process, a second step is needed which checks whether that web app and network process are allowed to communicate by looking into the kernel stored mapping of processes to URLs. This operator `checkConnection` is used to do just that. Its argument are two identifiers and the message (3 argument version), and depending on the message, the relevant URL is already extracted (4 argument version). In case a connection can be found for an incoming message, from a network process to a web app, witnessed by their first label matching, the check succeeds and the message is forwarded.

```
op checkConnection : Oid Oid Message -> Message .
op checkConnection : Oid Oid Label Message -> Message .

eq < kernel-id : kernel |
    handledCurrently(checkConnection(Num:Nat, Num':Nat, M)) ,
    weblabels(pi(Num':Nat, L:Label), WPIS:WebappProcInfoSet) ,
    networklabels(pi(Num:Nat, L:Label, L':Label),
                  NPIS:NetworkProcInfoSet) ,
    Att >
 =  < kernel-id : kernel |
     handledCurrently(M) ,
     weblabels(pi(Num':Nat, L:Label), WPIS:WebappProcInfoSet) ,
     networklabels(pi(Num:Nat, L:Label, L':Label),
                   NPIS:NetworkProcInfoSet) ,
     Att > .
```

In the direction of an outgoing message, going to `L':Label`, from a web app to a network process, the given network process address is simply ignored. If a matching network process, with the same first URL, and the actual target `L':Label` as its second URL, is found, then the message is passed onto that destination. That is done by using operator `changeRecipient` (shown above) with that network process id and the message.

```
eq < kernel-id : kernel |
    handledCurrently(checkConnection(
                      Num:Nat, Num':Nat, L':Label, M)) ,
    weblabels(pi(Num:Nat, L:Label), WPIS:WebappProcInfoSet) ,
    networklabels(pi(Num'':Nat, L:Label, L':Label),
                  NPIS:NetworkProcInfoSet) ,
    Att >
 = < kernel-id : kernel |
    handledCurrently(changeRecipient(Num'':Nat, M)) ,
    weblabels(pi(Num:Nat, L:Label), WPIS:WebappProcInfoSet) ,
    networklabels(pi(Num'':Nat, L:Label, L':Label),
                  NPIS:NetworkProcInfoSet) ,
    Att > .
```

The last possible case here is that no appropriate network process exists. This is quite possible as the first message sent from a web app to the network is what creates its associated network process. The network process, its DMA memory and its pipe are all created in this step.

```
eq < kernel-id : kernel |
    handledCurrently(checkConnection(
                       Num:Nat, Num':Oid, L':Label, M)) ,
    weblabels(pi(Num:Nat, L:Label), WPIS:WebappProcInfoSet) ,
    networklabels(NPIS:NetworkProcInfoSet) ,
    nextNetworkProc(Num'':Nat) ,
    Att >
 = < kernel-id : kernel |
    handledCurrently(changeRecipient(Num'':Nat, M)) ,
    weblabels(pi(Num:Nat, L:Label), WPIS:WebappProcInfoSet) ,
    networklabels(pi(Num'':Nat, L:Label, L':Label),
                  NPIS:NetworkProcInfoSet) ,
    nextNetworkProc(s(Num'':Nat)) ,
    Att >
   < Num'':Nat : proc | returnTo(Num:Nat) ,
                        in(mtLL) , out(mtLL) >
   < Num'':Nat : mem | in(mtLL) , out(mtLL) >
   < Num'':Nat : pipe | toKernel(mt), fromKernel(mt) >
 [owise] .
```

This rule models the kernel receiving a message on some pipe. It subjects the incoming OP message to the policy checking against the policy set MP, and sets the sender's process id correctly to ID, independent of the claimed id N in the actual message payload.

```
var M : Message .
var MP : PolicySet .
vars Att Att2 Att3 : AttributeSet .
vars ID N N' : ProcId .
var ML : MessageList .

rl [kernelReceivesOPMessage] :
  < kernel-id : kernel |
      handledCurrently(mt) ,
      msgPolicy(MP), Att >
  < ID : pipe | toKernel(
            msg(ST:SyscallType,
                payload(N, N', M:MsgType, V:MsgVal, S:String,
                        T:typed, U:untyped)), ML) ,
                Att2 >
  =>
  < kernel-id : kernel |
      handledCurrently(policyAllows(
            msg(ST:SyscallType,
                payload(ID, N', M:MsgType, V:MsgVal, S:String,
                        T:typed, U:untyped)), MP)) ,
      msgPolicy(MP), Att >
  < ID : pipe | toKernel(ML) , Att2 > .
```

Once the policy checking has been completed, the OP message can be forwarded by the kernel in this rule.

```
rl [kernelForwardsOPMessage] :
  < kernel-id : kernel | handledCurrently(
            msg(ST:SyscallType,
                payload(ID, N', M:MsgType, V:MsgVal, S:String,
                        T:typed, U:untyped))) ,
      Att >
  < N' : pipe | fromKernel(ML) , Att2 >
=>
  < kernel-id : kernel |
      handledCurrently(mt) ,
      Att >
  < N' : pipe | fromKernel(ML,
            msg(ST:SyscallType,
                payload(ID, N', M:MsgType, V:MsgVal, S:String,
                        T:typed, U:untyped))) ,
            Att2 > .
```

The `Label` stored in the `displayedTopBar` is the address bar that is shown on the UI; it is an attribute of the kernel, to simplify it being part of the secure UI. The actual content of the screen is modeled in the process with the `display-id` identifier, which has the attribute `displayedContent` which stores the URL of the currently displayed content. Only the active web app, identified by the process id in `activeWebapp`, is able to change the content that is displayed on screen. Initially it is empty, operator `none` is used for that. There is also one more wrapper for messages, called `kernelDo`, it is used for messages that the kernel is handling, but where it has to do more than just forward the message, but actually needs to take action instead.

```
op displayedTopBar : Label -> Attribute [ctor] .

op display-id : -> ProcId .
op activeWebapp : ProcId -> Attribute [ctor] .
op none : -> ProcId .
op displayedContent : Label -> Attribute [ctor] .

op kernelDo : Message -> Message .
```

This next rule is used after the policy checking completes, in case the kernel has to do something. Here, the kernel switches the active tab. For that, it changes the address bar representation `displayedTopBar`, it changes the active web app in `activeWebApp` and it empties the displayed memory, which the new owner will need to refresh.

```
rl [kernelHandlesTabSwitch] :
  < kernel-id : kernel |
      handledCurrently(kernelDo(
         msg(ST:SyscallType,
             payload(ui-id, N', MSG-SWITCH-TAB, V:MsgVal,
                     S:String, T:typed, U:untyped)))) ,
      displayedTopBar(L:Label),
      weblabels(pi(N', L':Label), WPIS:WebappProcInfoSet) ,
      Att >
  < display-id : proc |
    activeWebapp(P:ProcId),
    displayedContent(L'':Label),
    Att2 >
  =>
  < kernel-id : kernel |
      handledCurrently(mt) ,
      displayedTopBar(L':Label),
      weblabels(pi(N', L':Label), WPIS:WebappProcInfoSet) ,
      Att >
  < display-id : proc |
    activeWebapp(N'),
    displayedContent(about-blank),
    Att2 > .
```

The currently active web app can change the display whenever it wants to. The following rule models the abstract version; see the display memory modeling section, Section B.3, for a more concrete, but buggy, version corresponding to a design error which is then corrected.

```
crl  < display-id : proc |
       activeWebapp(N),
       displayedContent(LOld:Label),
       Att2 >
     < N : proc |
       rendered(L:Label),
       Att3 >
  =>
     < display-id : proc |
       activeWebapp(N),
       displayedContent(L:Label),
       Att2 >
    < N : proc |
       rendered(L:Label),
       Att3 >
if LOld:Label =/= L:Label .
```

For the creation of a new web app, after policy checking, the kernel does all the required steps here. It changes the address bar, the display memory access, clearing it out first, and then lets the new owner refresh it later. Note that for any new URL a new web app is created, as the label of an existing web app never changes.

```
rl [kernelHandlesNewUrl] :
  < kernel-id : kernel |
      handledCurrently(kernelDo(
         msg(ST:SyscallType,
             payload(ui-id, webapp-id, MSG-NEW-URL, V:MsgVal,
                     URL:Label, T:typed, U:untyped)))) ,
      displayedTopBar(L:Label),
      weblabels(WPIS:WebappProcInfoSet) ,
      Att >
  < display-id : proc |
    activeWebapp(P:ProcId),
    displayedContent(L'':Label),
    Att2 >
  < webappmgr-id : proc | nextWAN(NewWA:Nat) , Att3 >
  =>
  < kernel-id : kernel |
      handledCurrently(mt) ,
      displayedTopBar(URL:Label),
      weblabels(pi(NewWA:Nat, URL:Label),
                WPIS:WebappProcInfoSet) ,
      Att >
  < display-id : proc |
    activeWebapp(NewWA:Nat),
    displayedContent(about-blank),
    Att2 >
  < webappmgr-id : proc | nextWAN(s(NewWA:Nat)) , Att3 >
  < NewWA:Nat : proc |
      rendered(about-blank) ,
      URL(URL:Label) ,
      loading(0) >
  < NewWA:Nat : pipe |
      fromKernel(mt),
      toKernel(mt) > .
```

The operator `policyAllows` is responsible for checking whether a given message is permissible with respect to a given set of policies. The set of policies will be defined in the initial configuration for any model execution. The first equation below allows a message through if the process identifiers and message type have a match in the policy set.

```
op policyAllows : Message PolicySet -> Message .

eq policyAllows(
       msg(ST:SyscallType,
           payload(N, N’, M:MsgType, V:MsgVal, S:String,
                   T:typed, U:untyped)),
       (policy(N, N’, M:MsgType), MP))
   =  msg(ST:SyscallType,
          payload(N, N’, M:MsgType, V:MsgVal, S:String,
                  T:typed, U:untyped)) .
```

In case a generic policy, e.g., for all web apps, see `webapp-id`, is used, we check that the process id in that argument slot (the first one), namely `Num:Nat` is indeed a web app process id. In this particular case, this is for messages from web apps to a non-network process.

```
ceq policyAllows(
     msg(ST:SyscallType,
         payload(Num:Nat, N', M:MsgType, V:MsgVal, S:String,
                 T:typed, U:untyped)),
     (policy(webapp-id, N', M:MsgType), MP))
  =  msg(ST:SyscallType,
         payload(Num:Nat, N', M:MsgType, V:MsgVal, S:String,
                 T:typed, U:untyped))
if isWebapp(Num:Nat) /\ not isNetProc(N') .
```

This is for policies sending from a process that is not a network process to a web app.

```
ceq policyAllows(
     msg(ST:SyscallType,
         payload(N, Num':Nat, M:MsgType, V:MsgVal, S:String,
                 T:typed, U:untyped)),
     (policy(N, webapp-id, M:MsgType), MP))
  =  msg(ST:SyscallType,
         payload(N, Num':Nat, M:MsgType, V:MsgVal, S:String,
                 T:typed, U:untyped))
if isWebapp(Num':Nat) /\ not isNetProc(N) .
```

Similarly now for a network process communicating with a non-web app.

```
ceq policyAllows(
     msg(ST:SyscallType,
         payload(Num:Nat, N', M:MsgType, V:MsgVal, S:String,
                 T:typed, U:untyped)),
     (policy(network-id, N', M:MsgType), MP))
  =  msg(ST:SyscallType,
         payload(Num:Nat, N', M:MsgType, V:MsgVal, S:String,
                 T:typed, U:untyped))
if isNetProc(Num:Nat)  /\ not isWebapp(N') .
```

And again similarly for a non-web app communicating with a network process.

```
ceq policyAllows(
     msg(ST:SyscallType,
         payload(N, Num':Nat, M:MsgType, V:MsgVal, S:String,
                 T:typed, U:untyped)),
      (policy(N, network-id, M:MsgType), MP))
  =  msg(ST:SyscallType,
         payload(N, Num':Nat, M:MsgType, V:MsgVal, S:String,
                 T:typed, U:untyped))
if isNetProc(Num':Nat) /\ not isWebapp(N) .
```

And now for the connection between a web app and a network process. Note that this requires further checking by virtue of the `checkConnection` operator.

```
ceq policyAllows(
     msg(ST:SyscallType,
         payload(Num:Nat, Num':Oid, M:MsgType, V:MsgVal,
                 L:Label, T:typed, U:untyped)),
     (policy(webapp-id, network-id, M:MsgType), MP))
  = checkConnection(Num:Nat, Num':Oid, L:Label,
     msg(ST:SyscallType,
         payload(Num:Nat, Num':Oid, M:MsgType, V:MsgVal,
                 L:Label, T:typed, U:untyped)))
if isWebapp(Num:Nat) /\ isNetProc(Num':Oid) .
```

For the reverse direction from a network process to a web app we have this equation. This requires a check using `checkConnection` as well.

```
ceq policyAllows(
     msg(ST:SyscallType,
         payload(Num:Nat, Num':Nat, M:MsgType, V:MsgVal,
                 S:String, T:typed, U:untyped)),
     (policy(network-id, webapp-id, M:MsgType), MP))
  = checkConnection(Num:Nat, Num':Nat,
     msg(ST:SyscallType,
         payload(Num:Nat, Num':Nat, M:MsgType, V:MsgVal,
                 S:String, T:typed, U:untyped)))
if isNetProc(Num:Nat) /\ isWebapp(Num':Nat) .
```

This equation is to allow the UI to tell a web app to switch the tab. Ultimately the kernel executes this actually. Even though this looks like a very specific equation, it is still contingent on the actual policy to be in the set of policies!

```
ceq policyAllows(
     msg(ST:SyscallType,
         payload(ui-id, Num':Nat, MSG-SWITCH-TAB, V:MsgVal,
                 S:String, T:typed, U:untyped)),
     (policy(ui-id, webapp-id, MSG-SWITCH-TAB), MP))
  = kernelDo(msg(ST:SyscallType,
         payload(ui-id, Num':Nat, MSG-SWITCH-TAB, V:MsgVal,
                 S:String, T:typed, U:untyped)))
if isWebapp(Num':Nat) .
```

The UI can send a message to create a new web app with a new URL, being allowed by this equation. Note that the requisite policy needs to be in the initial configuration for this equation to be applicable, like above.

```
eq policyAllows(
    msg(ST:SyscallType,
        payload(ui-id, webapp-id, MSG-NEW-URL, V:MsgVal,
                   URL:Label, T:typed, U:untyped)),
       (policy(ui-id, webapp-id, MSG-NEW-URL), MP))
  = kernelDo(msg(ST:SyscallType,
        payload(ui-id, webapp-id, MSG-NEW-URL, V:MsgVal,
                URL:Label, T:typed, U:untyped))) .
```

If it is not explicitly allowed, it is implicitly disallowed. This equation takes care of that, by deleting all messages that are not conformant with a policy.

```
eq policyAllows(M, MP) = mt [owise] .
endm
```

The UI does not have a separate module. The relevant pieces of the UI are the URL of what is on the screen, `displayedContent`, and the address bar, `displayedTopBar`. Those are all included in the kernel as we saw already.

This module `KERNEL` collects all prior pieces, including specifically the module `KERNEL-POLICIES` which in fact contains more than just the policies.

```
mod KERNEL is
inc MSG-PIPE-BASICS .
inc KERNEL-POLICIES .
inc WEBAPPMGR .
inc WEBAPP .
endm
```

The module `RUN` contains the `KERNEL` and defines the initial state using a number of helper operators that will all be defined further on.

```
mod RUN is
inc MSG-PIPE-BASICS .
inc KERNEL-POLICIES .
inc KERNEL .
op init-proc : -> Configuration .
op initialPS : -> PolicySet .

ops init-kernel init-display
    init-webappmgr init-cache init-cookie init-vesafb-server
    init-mouse-server init-network-server init-dns-server
    init-ui init-mouse-intr init-network-intr init-nic :
    -> Configuration .
```

The following is the whole set of initial processes, and each of the elements is defined afterwards:

```
eq init-proc = init-display init-webappmgr init-cache
     init-cookie init-vesafb-server init-mouse-server
     init-network-server init-dns-server init-ui
     init-mouse-intr init-network-intr init-nic .

eq init-display =
     < display-id : proc | activeWebapp(none) ,
                           displayedContent(about-blank) > .
eq init-webappmgr =
    < webappmgr-id : proc | nextWAN(1024) >
    < webappmgr-id : pipe | toKernel(mt) , fromKernel(mt) > .
eq init-cache =
    < cache-id : proc | none >
    < cache-id : pipe | toKernel(mt), fromKernel(mt) > .
eq init-cookie =
    < cookie-id : proc | none >
    < cookie-id : pipe | toKernel(mt), fromKernel(mt) > .
eq init-vesafb-server =
    < vesafb-server-id : proc | none >
    < vesafb-server-id : pipe |
       toKernel(mt), fromKernel(mt) > .
eq init-mouse-server =
    < mouse-server-id : proc | none >
    < mouse-server-id : pipe |
       toKernel(mt), fromKernel(mt) > .
eq init-network-server =
    < network-server-id : proc | none >
    < network-server-id : pipe |
       toKernel(mt), fromKernel(mt) > .
eq init-dns-server =
    < dns-server-id : proc | none >
    < dns-server-id : pipe | toKernel(mt), fromKernel(mt) > .
eq init-ui =
     < ui-id : proc | none >
     < ui-id : pipe | toKernel(mt), fromKernel(mt) > .
eq init-mouse-intr =
    < mouse-intr-id : proc | none >
    < mouse-intr-id : pipe | toKernel(mt), fromKernel(mt) > .
eq init-network-intr =
    < mouse-intr-id : proc | none >
    < mouse-intr-id : pipe | toKernel(mt), fromKernel(mt) > .
eq init-nic =
     < 0 : nic | in(mtLL) , out(mtLL) > .
```

Now we define the initial set of policies, and each of the individual policies thereafter.

```
eq initialPS = msg-webapp-ui , msg-webapp-cookie ,
     msg-webapp-network , msg-webapp-storage ,
     msg-ui-webapp , msg-network-cookie ,
     msg-network-webapp , msg-cookie-webapp ,
     msg-cookie-network , msg-storage-webapp ,
     msg-storage-ui .

op msg-webapp-ui : -> PolicySet .
eq msg-webapp-ui = policy(webapp-id, ui-id, MSG-UI-MSG) .
op msg-webapp-cookie : -> PolicySet .
eq msg-webapp-cookie =
    policy(webapp-id, cookie-id, MSG-DOM-COOKIE-SET) ,
    policy(webapp-id, cookie-id, MSG-DOM-COOKIE-GET) .
op msg-webapp-network : -> PolicySet .
eq msg-webapp-network =
    policy(webapp-id, network-id, MSG-FETCH-URL) ,
    policy(webapp-id, network-id, MSG-FETCH-URL-ABORT) .
op msg-webapp-storage : -> PolicySet .
eq msg-webapp-storage =
    policy(webapp-id, storage-id, MSG-WRITE-FILE) ,
    policy(webapp-id, storage-id, MSG-READ-FILE) .
op msg-ui-webapp : -> PolicySet .
eq msg-ui-webapp =
    policy(ui-id, webapp-id, MSG-WEBAPP-MSG) ,
    policy(ui-id, webapp-id, MSG-SWITCH-TAB) ,
    policy(ui-id, webapp-id, MSG-NEW-URL) .
op msg-network-cookie : -> PolicySet .
```

```
eq msg-network-cookie =
    policy(network-id, cookie-id, MSG-COOKIE-SET) ,
    policy(network-id, cookie-id, MSG-COOKIE-GET) .
op msg-network-webapp : -> PolicySet .
eq msg-network-webapp =
    policy(network-id, webapp-id, MSG-RETURN-URL) ,
    policy(network-id, webapp-id, MSG-RETURN-URL-METADATA) .
op msg-cookie-webapp : -> PolicySet .
eq msg-cookie-webapp =
    policy(cookie-id, webapp-id, MSG-DOM-COOKIE-GET-RETURN) .
op msg-cookie-network : -> PolicySet .
eq msg-cookie-network =
    policy(cookie-id, network-id, MSG-COOKIE-GET-RETURN) .
op msg-storage-webapp : -> PolicySet .
eq msg-storage-webapp =
    policy(storage-id, webapp-id, MSG-READ-FILE-RETURN) .
op msg-storage-ui : -> PolicySet .
eq msg-storage-ui =
    policy(storage-id, ui-id, MSG-DOWNLOAD-INFO) .
endm
```

Now we add the module `TEST-INSTRUMENTATION` that is useful for us to be able to give test drive commands to the whole configuration, without needing a mouse or keyboard model. Initially we define the sort `Cmd` to be one particular command, for which we also define lists. The whole instrumentation will be inside the new object with object identifier `testMsg` and class identifier `testMsg`. The actual commands are `switch-tab`, and `new-url` which takes a URL as argument.

```
mod TEST-INSTRUMENTATION is
inc RUN .

sort Cmd .
sort CmdList .
subsort Cmd < CmdList .
op mtCmdList : -> CmdList .
op _,_ : CmdList CmdList
         -> CmdList [assoc comm id: mtCmdList] .
op cmd : CmdList -> Attribute [ctor] .
op testMsg : -> Cid .
op testMsg : -> Oid .

op new-url : Label -> Cmd .
op switch-tab : -> Cmd .
```

We are defining a number of generic but fixed URLs. Then we define further new URLs based on a natural number argument, i.e., `new-url`.

```
ops Url1 Url2 Url3 Url4 : -> Label .
vars Att Att2 Att3 : AttributeSet .

op url : Nat -> Label .
```

For a URL mis-match, two different URLs are enough, so we allow `new-url` to expand in three different ways based on these rules, to be able to have two URLs involved in a mis-match and an independent further URL. Note that we are going to use search, so all possible combinations will be explored.

The `inspect` operator is a command that we use here to rewrite to `inspect(3)`, which means a three step inspection. It would be possible to use a different number in this rule, or simply give the operator with an argument of the users choice right away. Then, each step will either be to switch the tab, or get sent a new URL. Keeping with our modeling methodology of a soup of objects, `inspect-space` creates that test driver object with the three step inspection.

```
op inspect : -> Cmd .
op inspect : Nat -> Cmd .
rl inspect => inspect(3) .
rl inspect(0) => mtCmdList .
rl inspect(s(N:Nat)) => new-url(url(1)) , inspect(N:Nat) .
rl inspect(s(N:Nat)) => new-url(url(2)) , inspect(N:Nat) .
rl inspect(s(N:Nat)) => new-url(url(3)) , inspect(N:Nat) .
rl inspect(s(N:Nat)) => switch-tab , inspect(N:Nat) .

op inspect-space :  -> Configuration .
eq inspect-space =
     < testMsg : testMsg | cmd( inspect(3) ) > .
```

The `new-url` is resolved like this.

```
rl [testNewUrl] :
  < kernel-id : kernel |
     handledCurrently(mt) ,
     Att >
  < testMsg : testMsg | cmd( new-url(L:Label) ,
                               CMDList:CmdList ) >
=>
  < kernel-id : kernel |
      handledCurrently(kernelDo(
        msg(OPOS-SYSCALL-FD-SEND-MESSAGE,
            payload(ui-id, webapp-id, MSG-NEW-URL, 0,
                    L:Label, mtTyped, mtUntyped)))) ,
      Att >
  < testMsg : testMsg | cmd( CMDList:CmdList ) >
.
```

And the switch-tab is resolved like this.

```
rl [testTabSwitch] :
  < testMsg : testMsg | cmd( switch-tab , CMDList:CmdList ) >
  < kernel-id : kernel |
      handledCurrently(mt) ,
      weblabels(pi(N':Nat, L':Label),
                WPIS:WebappProcInfoSet) ,
      Att >
  =>
  < testMsg : testMsg | cmd( CMDList:CmdList ) >
  < kernel-id : kernel |
      handledCurrently(kernelDo(
        msg(OPOS-SYSCALL-FD-SEND-MESSAGE,
            payload(ui-id, N':Nat, MSG-SWITCH-TAB, 0,
                    about-blank, mtTyped, mtUntyped)))) ,
      weblabels(pi(N':Nat, L':Label),
                WPIS:WebappProcInfoSet) ,
      Att > .
```

The base for an initial kernel for tests is the following.

```
op init-simp-kernel : -> Configuration .
eq init-simp-kernel
    = < kernel-id : kernel |
          handledCurrently(mt) ,
          msgPolicy(initialPS) ,
          nextNetworkProc(256) ,
          weblabels(mtWPIS) ,
          networklabels(mtNPIS) ,
          displayedTopBar(about-blank) >
       init-proc .
```

An example initial kernel is defined like this.

```
*****  experimental driver for messages!
eq init-kernel
    = < kernel-id : kernel |
          handledCurrently(mt) ,
          msgPolicy(initialPS) ,
          nextNetworkProc(256) ,
          weblabels(pi(1050,l(http,dom("test"),port(80)))) ,
          networklabels(mtNPIS)  ,
          displayedTopBar(about-blank) >
      < 1050 : proc | rendered(about-blank) ,
          URL(l(http, dom("test"), port(81))) , loading(1) >
      < 1050 : pipe | toKernel(
              msg(OPOS-SYSCALL-FD-SEND-MESSAGE,
              payload(1050, 500,
                      MSG-FETCH-URL, 0,
                      l(http,dom("test"),port(81)),
                      mtTyped, mtUntyped))
               ) ,
                      fromKernel(mt) >
      < 0 : nic | in(mtLL) , out(mtLL) >
    .
endm
```

Analysis of the above code happens at the very end of this chapter, in Section B.4.

B.2 Internal Rules Termination

We are considering here the termination of the internal rules for our IBOS model. This is needed for the reordering of the rewrite sequence into normalizing with the internal rules between each single trigger rule execution to be sensible. For this goal we will present the internal rules in a descending order that leads to termination.

Let us present the order in which the rules will be executed in the system, and let us note that each data transfer that has used one rule will never be able to be used in that same rule (with the same argument, see the first rule's explanation), or any rule that is listed before, again. See Appendix B.1 for the whole specification and the context and explanation of the rules, our purpose here is to show the ordering of the rules.

The first rule shows how the kernel receives a browser related data transfer message and how it handles it. It gets passed to the policy checking and will then be treated further in the rules below. Note that this rule can possibly be used multiple times in a data transfer, but with different arguments (the `MsgType` in particular will be another one).

```
rl [kernelReceivesOPMessage] :
  < kernel-id : kernel |
      handledCurrently(mt) ,
      msgPolicy(MP), Att >
  < ID : pipe | toKernel(
            msg(ST:SyscallType,
                payload(N, N', M:MsgType, V:MsgVal, S:String,
                        T:typed, U:untyped)), ML) ,
                Att2 >
  =>
  < kernel-id : kernel |
      handledCurrently(policyAllows(
            msg(ST:SyscallType,
                payload(ID, N', M:MsgType, V:MsgVal, S:String,
                        T:typed, U:untyped)), MP)) ,
      msgPolicy(MP), Att >
  < ID : pipe | toKernel(ML) , Att2 > .
```

After policy checking the message can be forwarded to the recipient. Potentially one of the special cases below can apply instead.

```
rl [kernelForwardsOPMessage] :
  < kernel-id : kernel | handledCurrently(
            msg(ST:SyscallType,
                payload(ID, N', M:MsgType, V:MsgVal, S:String,
                        T:typed, U:untyped))) ,
      Att >
  < N' : pipe | fromKernel(ML) , Att2 >
=>
  < kernel-id : kernel |
      handledCurrently(mt) ,
      Att >
  < N' : pipe | fromKernel(ML,
            msg(ST:SyscallType,
                payload(ID, N', M:MsgType, V:MsgVal, S:String,
                        T:typed, U:untyped))) ,
            Att2 > .
```

If the data transfer was supposed to lead to the creation of a new web app, then the kernel does that. This rule will not be usable again after on that data transfer. The potential follow-up messages in this data transfer are based on the `loading(0)` property which will start the loading of that web site's content.

```
rl [kernelHandlesNewUrl] :
  < kernel-id : kernel |
      handledCurrently(kernelDo(
         msg(ST:SyscallType,
             payload(ui-id, webapp-id, MSG-NEW-URL, V:MsgVal,
                     URL:Label, T:typed, U:untyped)))) ,
      displayedTopBar(L:Label),
      weblabels(WPIS:WebappProcInfoSet) ,
      Att >
  < display-id : proc |
    activeWebapp(P:ProcId),
    displayedContent(L'':Label),
    Att2 >
  < webappmgr-id : proc | nextWAN(NewWA:Nat) , Att3 >
  =>
  < kernel-id : kernel |
      handledCurrently(mt) ,
      displayedTopBar(URL:Label),
      weblabels(pi(NewWA:Nat, URL:Label),
                WPIS:WebappProcInfoSet) ,
      Att >
  < display-id : proc |
    activeWebapp(NewWA:Nat),
    displayedContent(about-blank),
    Att2 >
  < webappmgr-id : proc | nextWAN(s(NewWA:Nat)) , Att3 >
  < NewWA:Nat : proc |
      rendered(about-blank) ,
      URL(URL:Label) ,
      loading(0) >
  < NewWA:Nat : pipe |
      fromKernel(mt),
      toKernel(mt) > .
```

In case the message was a tab switch this is handled here and this rule will not be re-used again for this data transfer as well.

```
rl [kernelHandlesTabSwitch] :
  < kernel-id : kernel |
      handledCurrently(kernelDo(
         msg(ST:SyscallType,
             payload(ui-id, N', MSG-SWITCH-TAB, V:MsgVal,
                     S:String, T:typed, U:untyped)))) ,
      displayedTopBar(L:Label),
      weblabels(pi(N', L':Label), WPIS:WebappProcInfoSet) ,
      Att >
  < display-id : proc |
    activeWebapp(P:ProcId),
    displayedContent(L'':Label),
    Att2 >
  =>
  < kernel-id : kernel |
      handledCurrently(mt) ,
      displayedTopBar(L':Label),
      weblabels(pi(N', L':Label), WPIS:WebappProcInfoSet) ,
      Att >
  < display-id : proc |
    activeWebapp(N'),
    displayedContent(about-blank),
    Att2 > .
```

The remaining rules represent the chain of messages started for a data transfer that loads data from a given web site, going through the NIC, getting data back from the NIC, going back to the network process and ultimately going to the web app for display purposes. Once a data transfer is at this stage these rules will be applied consecutively, with the kernel handling rule from above potentially involved, but due to the arguments the only further processing from there is with the remaining rules below this point. Note that of course each of these rules only applies once!

First, this is the request from a web app to a network process to get data from a given URL.

```
crl < N : proc | returnTo(SomeProcNum:Nat) ,
                 out(Ll:LabelList) , Att >
    < N : pipe | toKernel(ML) , fromKernel(
        msg(OPOS-SYSCALL-FD-SEND-MESSAGE,
            payload(Num:Nat, N, MSG-FETCH-URL, V:MsgVal,
                    L:Label, T:typed, U:untyped)),
        ML') , Att2 >
    => < N : proc | returnTo(Num:Nat) ,
                    out(Ll:LabelList, L:Label) , Att >
      < N : pipe | toKernel(ML) , fromKernel(ML') , Att2 >
 if isNetProc(N) .
```

This is the network process writing that request into the memory for NIC pickup, after kernel validation.

```
rl < N : proc | out(L:Label, Ll:LabelList) , Att >
   < N : mem | out(mtLL) , Att2 >
=> < N : proc | out(Ll:LabelList) , Att >
   < N : mem | out(L:Label) , Att2 > .
```

Kernel validating the network process data for the NIC in the next rule.

```
rl < N : mem | out(L':Label) , Att >
   < kernel-id : kernel |
      networklabels(pi(N, L:Label, L':Label),
                    NPIS:NetworkProcInfoSet) ,
      Att2 >
   < 0 : nic | out(Ll:LabelList) , Att3 >
=> < N : mem | out(mtLL) , Att >
   < kernel-id : kernel |
      networklabels(pi(N, L:Label, L':Label),
                    NPIS:NetworkProcInfoSet) ,
      Att2 >
   < 0 : nic | out(Ll:LabelList, L':Label) , Att3 > .
```

The NIC interaction with the outside world is given as an immediate response from that outside world.

```
rl < 0 : nic | out(Ll:LabelList , L:Label , Ll':LabelList) ,
               in(Ll'':LabelList) , Att >
=> < 0 : nic | out(Ll:LabelList , Ll':LabelList) ,
               in(Ll'':LabelList, L:Label) , Att > .
```

The NIC's return data is given to the memory accessible by the right network process.

```
rl < N : mem | in(mtLL) , Att >
   < kernel-id : kernel |
      networklabels(pi(N, L:Label, L':Label),
                    NPIS:NetworkProcInfoSet) ,
      Att2 >
   < 0 : nic | in(L':Label, Ll:LabelList) , Att3 >
=> < N : mem | in(L':Label) , Att >
   < kernel-id : kernel |
      networklabels(pi(N, L:Label, L':Label),
                    NPIS:NetworkProcInfoSet) ,
      Att2 >
   < 0 : nic | in(Ll:LabelList) , Att3 > .
```

That appropriate network process fetches the data from the memory.

```
rl < N : proc | in(Ll:LabelList) , Att >
   < N : mem | in(L:Label) , Att2 >
=> < N : proc | in(Ll:LabelList, L:Label) , Att >
   < N : mem | in(mtLL) , Att2 > .
```

Between the above rule and the following rule there is the use of an equation, and then the passing mechanism of the first and second rule (labeled `kernelReceivesOPMessage` and `kernelForwardsOPMessage`) at the top gets used, but this time with a `MSG-RETURN-URL` instead of the `MSG-FETCH-URL` argument in the payload, which leads to all the data fetching related rules not applying. So, from here the first two rules will apply once each and then the remaining rules below trigger in order.

```
  rl < N : proc | rendered(L) , URL(L') ,loading(1) , Att >
     < N : pipe | fromKernel(
        msg(OPOS-SYSCALL-FD-SEND-MESSAGE,
           payload(N', N, MSG-RETURN-URL, V:MsgVal,
                   LL:Label, T:typed, U:untyped)),
        ML) , Att2 >
   => < N : proc | rendered(LL:Label) , URL(L') ,
                   loading(1) , Att >
      < N : pipe | fromKernel(ML) , Att2 > .
```

The above rule potentially changes the URL in `rendered` and thus the next rule can apply to adjust the `displayedContent`, but it can only apply a single time per change, due to the condition.

```
crl  < display-id : proc |
       activeWebapp(N),
       displayedContent(LOld:Label),
       Att2 >
```

```
     < N : proc |
       rendered(L:Label),
       Att3 >
  =>
     < display-id : proc |
       activeWebapp(N),
       displayedContent(L:Label),
       Att2 >
    < N : proc |
       rendered(L:Label),
       Att3 >
if LOld:Label =/= L:Label .
```

At this point no further rule will apply to a data transfer that has taken all the steps to get through the rules to this point.

Thus, we have shown the order in which the rules can be applied and we note that this way we have explained that any sequence of steps using these rules will actually terminate.

B.3 Display Memory Modeling

Next we are looking at the contents of the file `memory.maude`. This is a model of the memory and page table interaction which finds a bug. Therefore it is not integrated into the rest of the model but requires fixing in the implementation of IBOS. Some simplification of processes from before have been done for this.

First, we add three types of memory, which represent all memory we care about. Specifically it is the empty memory, the current video memory, and any kind of other memory. We also define a page table. Its entries map process identifiers (that will turn out to be web apps) to memory locations.

```
in ibos.maude

mod MEMORY is
inc RUN .
inc TEST-INSTRUMENTATION .

sort Memory .
```

```
op nullMemory : -> Memory [ctor] .
op videoMemory : -> Memory [ctor] .
op otherMemory : -> Memory [ctor] .

sort PGTE .
op pg-table-entry : ProcId Memory -> PGTE [ctor] .
sort PGTESet .
subsort PGTE < PGTESet .
op mtPGTEs : -> PGTESet .
op _,_ : PGTESet PGTESet
         -> PGTESet [assoc comm id: mtPGTEs] .
op pg-table : PGTESet -> Attribute [ctor] .

op vidMem : Label -> Attribute [ctor] .
```

This is our testing instrumentation. The `initial-test` is what we base our further checking on.

```
op initial-test : -> Configuration .
eq initial-test
    = < kernel-id : kernel |
          handledCurrently(mt) ,
          msgPolicy(mtPS) ,
          nextNetworkProc(256) ,
          weblabels(mtWPIS) ,
          networklabels(mtNPIS)  ,
          displayedTopBar(Url1) ,
          pg-table(pg-table-entry(1050, videoMemory)) ,
          activeWebapp(1050) ,
          vidMem(Url1)
      >
      init-webappmgr
      < 1050 : proc | rendered(Url1), URL(Url1), loading(1) >
      < 1050 : pipe | toKernel(mt) ,
                      fromKernel(mt) >
   .
```

The `bug-trigger` message is one that we have found through experimentation to find the display memory bug of creating an empty display.

```
op bug-trigger : -> Configuration .
eq bug-trigger =
    < testMsg : testMsg | cmd( new-tab(Url2) ,
                               update(1050, Url3) ,
                              tab-switch(1050)) > .
```

For the above to make sense we of course require the following definitions for our test drivers. Similar to `inspect` from above we will define the elements for `explore` here.

```
op new-tab : Label -> Cmd .
op update : Oid Label -> Cmd .
op tab-switch : Oid -> Cmd .

op url : Nat -> Label .
op new-tab : -> Cmd .
rl new-tab => new-tab(url(1)) .
rl new-tab => new-tab(url(2)) .
rl new-tab => new-tab(url(3)) .
op update : -> Cmd .
op update : Label -> Cmd .
rl update => update(url(1)) .
rl update => update(url(2)) .
rl update => update(url(3)) .
op tab-switch : -> Cmd .
```

These are the rules that show what each of the commands actually does. Both of these only work with web apps.

```
rl  < testMsg : testMsg | cmd ( update(U:Label),
                                CMDList:CmdList) >
    < N:Nat : proc | rendered(U':Label),
                     URL(U'':Label), loading(N':Nat) >
 => < testMsg : testMsg | cmd ( update(N:Nat, U:Label),
                                CMDList:CmdList) >
    < N:Nat : proc | rendered(U':Label),
                     URL(U'':Label), loading(N':Nat) > .

rl  < testMsg : testMsg | cmd ( tab-switch,
                                CMDList:CmdList) >
    < N:Nat : proc | rendered(U':Label),
                     URL(U'':Label), loading(N':Nat) >
 => < testMsg : testMsg | cmd ( tab-switch(N:Nat),
                                CMDList:CmdList) >
    < N:Nat : proc | rendered(U':Label),
                     URL(U'':Label), loading(N':Nat) > .
```

This is the similar exploration setting that will be complete expanded due to the search commands we will run.

```
op explore : -> Cmd .
op explore : Nat -> Cmd .
rl explore => explore(3) .
rl explore(0) => mtCmdList .
rl explore(s(N:Nat)) => new-tab , explore(N:Nat) .
rl explore(s(N:Nat)) => update , explore(N:Nat) .
rl explore(s(N:Nat)) => tab-switch , explore(N:Nat) .

op explore-space :  -> Configuration .
eq explore-space =
      < testMsg : testMsg | cmd( explore(3) ) > .
```

This rule works with the `new-tab` command, to create a new process and pipe and to make it the active web app.

```
rl < testMsg : testMsg | cmd ( new-tab(U:Label),
                               CMDList:CmdList) >
   < kernel-id : kernel |
          Att1:AttributeSet ,
          displayedTopBar(U1:Label) ,
          pg-table(pg-table-entry(N2:Nat, videoMemory),
                   pg:PGTESet) ,
          activeWebapp(N2:Nat) ,
          vidMem(U1:Label) >
   < webappmgr-id : proc | nextWAN(N:Nat) >
=> < testMsg : testMsg | cmd ( CMDList:CmdList) >
   < kernel-id : kernel |
        Att1:AttributeSet ,
        displayedTopBar(U:Label) ,
        pg-table(pg-table-entry(N2:Nat, nullMemory),
                 pg-table-entry(N:Nat, videoMemory),
                 pg:PGTESet) ,
        activeWebapp(N:Nat) ,
        vidMem(U:Label) >
   < N:Nat : proc | rendered(U:Label),
                    URL(U:Label), loading(1) >
   < N:Nat : pipe | fromKernel(mt) , toKernel(mt) >
   < webappmgr-id : proc | nextWAN(s(N:Nat)) > .
```

A web app is being updated which is not currently the active web app, see the condition, so this web app will be assigned the memory `otherMemory` in the page table.

```
crl < testMsg : testMsg | cmd ( update(N:Nat, U:Label),
                                CMDList:CmdList) >
    < kernel-id : kernel |
          Att1:AttributeSet ,
          displayedTopBar(U1:Label) ,
          pg-table(pg-table-entry(N:Nat, nullMemory),
                   pg:PGTESet) ,
          activeWebapp(N2:Nat) ,
          vidMem(U1:Label)
      >
      < N:Nat : proc | rendered(U2:Label),
                       URL(U2:Label), loading(1) >
=> < testMsg : testMsg | cmd ( CMDList:CmdList) >
   < kernel-id : kernel |
          Att1:AttributeSet ,
          displayedTopBar(U1:Label) ,
          pg-table(pg-table-entry(N:Nat, otherMemory),
                   pg:PGTESet) ,
          activeWebapp(N2:Nat) ,
          vidMem(U1:Label)
      >
      < N:Nat : proc | rendered(U:Label),
                       URL(U:Label), loading(1) >
 if N:Nat =/= N2:Nat .
```

In case the web app that is getting switched to is currently mapped to the null memory, then there is a page fault, and it will be updated to point at the actual video memory, which means the content of the display will get refreshed and normal operation can continue.

```
crl < testMsg : testMsg | cmd ( tab-switch(N:Nat),
                                CMDList:CmdList) >
    < kernel-id : kernel |
          Att1:AttributeSet ,
          displayedTopBar(U1:Label) ,
          pg-table(pg-table-entry(N:Nat, nullMemory),
                   pg-table-entry(N2:Nat, videoMemory),
                   pg:PGTESet) ,
          activeWebapp(N2:Nat) ,
          vidMem(U1:Label)
      >
      < N:Nat : proc | rendered(U2:Label),
                       URL(U2:Label), loading(1) >
=>  < testMsg : testMsg | cmd (CMDList:CmdList) >
    < kernel-id : kernel |
          Att1:AttributeSet ,
          displayedTopBar(U2:Label) ,
          pg-table(pg-table-entry(N:Nat, videoMemory),
          pg-table-entry(N2:Nat, nullMemory), pg:PGTESet) ,
          activeWebapp(N:Nat) ,
          vidMem(about-blank)
      >
      < N:Nat : proc | rendered(U2:Label),
                       URL(U2:Label), loading(1) >
 if N:Nat =/= N2:Nat .
endm
```

Now when the tab is switched to a web app for which a mapping to some memory already exists, it is not re-mapped to the memory. This is actually a crucial step to the bug we encounter. The video memory remapping is based on page faults, but as there is no page fault, it does not get remapped. Originally, this rule was part of the MEMORY module, but to make the error more obvious and to be able to concisely present the fix below, we have pushed this one rule into its own module, called CRUCIAL-RULE-BAD.

```
mod CRUCIAL-RULE-BAD is
inc MEMORY .

crl < testMsg : testMsg | cmd ( tab-switch(N:Nat),
                                CMDList:CmdList) >
    < kernel-id : kernel |
          Att1:AttributeSet ,
          displayedTopBar(U1:Label) ,
          pg-table(pg-table-entry(N:Nat, otherMemory),
                   pg-table-entry(N2:Nat, videoMemory),
                   pg:PGTESet) ,
          activeWebapp(N2:Nat) ,
          vidMem(U1:Label)
      >
      < N:Nat : proc | rendered(U2:Label),
                       URL(U2:Label), loading(1) >
=>  < testMsg : testMsg | cmd (CMDList:CmdList) >
    < kernel-id : kernel |
          Att1:AttributeSet ,
          displayedTopBar(U2:Label) ,
          pg-table(pg-table-entry(N:Nat, otherMemory),
                   pg-table-entry(N2:Nat, nullMemory),
                   pg:PGTESet) ,
          activeWebapp(N:Nat) ,
          vidMem(about-blank)
      >
      < N:Nat : proc | rendered(U2:Label),
                       URL(U2:Label), loading(1) >
 if N:Nat =/= N2:Nat .
endm
```

This search finds the cause of the empty display issues by exploring the whole space spanned due to `explore-space`. After seeing the output, we can simplify this for the next search. That simpler search uses the `bug-trigger` instead, which is enough to see the bug. When running these search commands, finding any solution means there is a bug, if there were no solution than we would have not found any bug.

```
search initial-test explore-space
=>!
X:Configuration
< kernel-id : kernel | Att:AttributeSet ,
    vidMem(about-blank) , activeWebapp(N:Nat) ,
    pg-table(pg-table-entry(N:Nat, otherMemory),
             pg:PGTESet) > .

search initial-test bug-trigger
=>!
X:Configuration
< kernel-id : kernel | Att:AttributeSet ,
    vidMem(about-blank) , activeWebapp(N:Nat) ,
    pg-table(pg-table-entry(N:Nat, otherMemory),
             pg:PGTESet) > .
```

The fix for this is to not depend on page faults, and the rest of our analysis uses a model without this issue, see the next section for the fix.

B.3.1 Fixed Display Memory

Now we are looking at the contents of file `memory-fixed.maude`. We only need to change the one rule in the module `CRUCIAL-RULE-BAD`, to fix the issue. Replacing that module with the module `CRUCIAL-RULE-FIXED`, where the remapping of the video memory always happens independent of page faults, is enough.

```
mod CRUCIAL-RULE-FIXED is
inc MEMORY .

crl < testMsg : testMsg | cmd ( tab-switch(N:Nat),
                                CMDList:CmdList) >
    < kernel-id : kernel |
          Att1:AttributeSet ,
          displayedTopBar(U1:Label) ,
          pg-table(pg-table-entry(N:Nat, M:Memory),
                   pg-table-entry(N2:Nat, videoMemory),
                   pg:PGTESet) ,
          activeWebapp(N2:Nat) ,
          vidMem(U1:Label)
      >
      < N:Nat : proc | rendered(U2:Label),
                       URL(U2:Label), loading(1) >
=>  < testMsg : testMsg | cmd (CMDList:CmdList) >
    < kernel-id : kernel |
          Att1:AttributeSet ,
          displayedTopBar(U2:Label) ,
          pg-table(pg-table-entry(N:Nat, videoMemory),
                   pg-table-entry(N2:Nat, nullMemory),
                   pg:PGTESet) ,
          activeWebapp(N:Nat) ,
          vidMem(about-blank)
      >
      < N:Nat : proc | rendered(U2:Label),
                       URL(U2:Label), loading(1) >
 if N:Nat =/= N2:Nat .
endm
```

Note that the difference is that `N:Nat` is always mapped to `videoMemory` in this rule. Then we can run the exact `search` command from before, which found errors when using the bad rule. It is the command looking for errors, but now it finds none.

```
search initial-test explore-space
=>!
X:Configuration
```

```
< kernel-id : kernel | Att:AttributeSet ,
    vidMem(about-blank) , activeWebapp(N:Nat) ,
    pg-table(pg-table-entry(N:Nat, otherMemory),
             pg:PGTESet) > .
```

With this, we know that a solution not depending on page table faults does work for the display memory as shown in the `search`.

B.4 IBOS Analysis

Now we are looking at the content of the file `analysis.maude`. All references in this section are to elements in Chapter 4.

This first command is just to check that the model works as expected.

```
in ibos.maude

search init-simp-kernel
inspect-space
=>*
X:Configuration
< N:Nat : pipe |
    toKernel(ML:MessageList) ,
    fromKernel(msg(OPOS-SYSCALL-FD-SEND-MESSAGE,
          payload(Num:Nat, N:Nat, MSG-FETCH-URL,
                  V:MsgVal, L:Label, T:typed, U:untyped)),
          ML':MessageList) , Att:AttributeSet >
< kernel-id : kernel | Att2:AttributeSet ,
    weblabels(pi(Num:Nat,L':Label),WAPIS:WebappProcInfoSet) ,
    networklabels(pi(N:Nat, L':Label, L:Label),
                  NPIS:NetworkProcInfoSet)  ,
    displayedTopBar(URL:Label) > .
```

Next we look at the address bar correctness from Section 4.3.2. If there is no solution, this means there is no mismatch and thus the address bar correctness holds.

```
search init-simp-kernel
inspect-space
=>*
  X:Configuration
  < kernel-id : kernel | Att:AttributeSet ,
          displayedTopBar(URL:Label) >
  < display-id : proc |
    displayedContent(URL':Label),
    Att2:AttributeSet >
such that URL:Label =/= URL':Label
      and URL:Label =/= about-blank
      and URL':Label =/= about-blank .
```

Now we are going into the code relevant for Section 4.3.3. This is the check that all network requests from web page instances go to the proper network process, property (1). This command will have no solution, so no mismatched labeling exists and thus the property does hold.

```
search init-simp-kernel
inspect-space
=>*
X:Configuration
< N:Nat : pipe |
   toKernel(ML:MessageList) ,
   fromKernel(msg(OPOS-SYSCALL-FD-SEND-MESSAGE,
                  payload(Num:Nat, N:Nat, MSG-FETCH-URL,
                          V:MsgVal, L1:Label,
                          T:typed, U:untyped)),
              ML':MessageList) , Att:AttributeSet >
< kernel-id : kernel | Att2:AttributeSet ,
   weblabels(pi(Num:Nat,L1':Label), WAPIS:WebappProcInfoSet) ,
   networklabels(pi(N:Nat, L2':Label, L2:Label),
                 NPIS:NetworkProcInfoSet)  ,
   displayedTopBar(URL:Label) >
such that  L1:Label =/= L2:Label
       or L1':Label =/= L2':Label .
```

This checks the property that an incoming ethernet frame gets handed to the right network process memory, property (2). Again, this search will have no result, meaning no mismatch, meaning the property holds.

```
search init-simp-kernel
inspect-space
=>*
X:Configuration
< N:Nat : mem | in(L1:Label, Ll:LabelList),
                Att:AttributeSet >
< kernel-id : kernel |
   networklabels(pi(N:Nat, L':Label, L2:Label),
                 NPIS:NetworkProcInfoSet) ,
   Att2:AttributeSet >
such that L1:Label =/= L2:Label .
```

The next search command checks that any ethernet frame that is given by a network process to the NIC via the DMA memory matches the marking of the network process, i.e., property (3). Again, there is no solution for this, meaning no mismatch, meaning the property holds.

```
search init-simp-kernel
inspect-space
=>*
X:Configuration
< N:Nat : mem | out(L1:Label) , Att:AttributeSet >
< kernel-id : kernel |
   networklabels(pi(N:Nat, L':Label, L2:Label),
                 NPIS:NetworkProcInfoSet) ,
   Att2:AttributeSet >
such that L1:Label =/= L2:Label .
```

The return message from a network process to a web page instance only has data from an appropriate source, property (4). No solution is found, there is thus no mismatch between the labeling of the web app, the network process and the data, so the property holds.

```
search init-simp-kernel
inspect-space
=>*
X:Configuration
< N:Nat : proc | rendered(L11:Label) , URL(L'':Label) ,
                 loading(1) , Att:AttributeSet >
< N:Nat : pipe | fromKernel(
       msg(OPOS-SYSCALL-FD-SEND-MESSAGE,
           payload(N':Nat, N:Nat, MSG-RETURN-URL, V:MsgVal,
                   L2:Label, T:typed, U:untyped)),
       ML:MessageList) , Att2:AttributeSet >
< kernel-id : kernel |
  weblabels(pi(N:Nat, L':Label), WAPIS:WebappProcInfoSet) ,
  networklabels(pi(N':Nat, L':Label, L1:Label),
                NPIS:NetworkProcInfoSet) ,
  Att3:AttributeSet >
such that L1:Label =/= L2:Label
```

.

Last, we have the property (9), that is that the URL of the current tab is displayed to the user. That means, there is never a mismatch between the URL of the currently active web app and the address bar. As there is no solution found this property holds again.

```
search init-simp-kernel
inspect-space
=>*
  X:Configuration
  < kernel-id : kernel | Att:AttributeSet ,
          displayedTopBar(URL:Label) >
  < display-id : proc |
    activeWebapp(W:ProcId),
    Att2:AttributeSet >
  < W:ProcId : proc |
    URL(URL':Label),
    Att3:AttributeSet >
such that URL:Label =/= URL':Label  .
```

The correctness of the other properties has been argued in Section 4.3.3, so there is no need to repeat those arguments here.

REFERENCES

[1] *IEEE 802.11 Local and Metropolitan Area Networks: Wireless LAN Medium Access Control (MAC) and Physical (PHY) Specifications*, 1999.

[2] *Proceedings of the Network and Distributed System Security Symposium, NDSS 2006, San Diego, California, USA*. The Internet Society, 2006.

[3] Martín Abadi and Véronique Cortier. Deciding knowledge in security protocols under equational theories. *Theoretical Computer Science*, 367(1-2):2–32, 2006.

[4] M. Alpuente, M. Falaschi, and G. Vidal. Partial evaluation of functional logic programs. *ACM Transactions on Programming Languages and Systems*, 20(4):768–844, 1998.

[5] María Alpuente, Santiago Escobar, and José Iborra. Termination of narrowing using dependency pairs. In Maria Garcia de la Banda and Enrico Pontelli, editors, *Logic Programming, 24th International Conference, ICLP 2008, Udine, Italy, December 9-13 2008, Proceedings*, volume 5366 of *Lecture Notes in Computer Science*, pages 317–331. Springer, 2008.

[6] María Alpuente, Santiago Escobar, and José Iborra. Termination of narrowing revisited. *Theoretical Computer Science*, 410(46):4608–4625, 2009.

[7] María Alpuente, Santiago Escobar, and José Iborra. Modular termination of basic narrowing and equational unification. *Logic Journal of the IGPL*, 2010. doi: 10.1093/jigpal/jzq009.

[8] Siva Anantharaman, Paliath Narendran, and Michaël Rusinowitch. Unification modulo CUI plus distributivity axioms. *J. Autom. Reasoning*, 33(1):1–28, 2004.

[9] S. Antoy. Evaluation strategies for functional logic programming. *Journal of Symbolic Computation*, 40:875—-903, 2005.

[10] S. Antoy, R. Echahed, and M. Hanus. A needed narrowing strategy. *Journal of the ACM*, 47(4):776–822, 2000.

[11] Alessandro Armando, David A. Basin, Yohan Boichut, Yannick Chevalier, Luca Compagna, Jorge Cuéllar, Paul Hankes Drielsma, Pierre-Cyrille Héam, Olga Kouchnarenko, Jacopo Mantovani, Sebastian Mödersheim, David von Oheimb, Michaël Rusinowitch, Judson Santiago, Mathieu Turuani, Luca Viganò, and Laurent Vigneron. The AVISPA tool for the automated validation of internet security protocols and applications. In *CAV*, pages 281–285, 2005.

[12] Thomas Arts and Jürgen Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236(1-2):133–178, 2000.

[13] Franz Baader and Klaus U. Schulz. Unification in the union of disjoint equational theories: Combining decision procedures. In Deepak Kapur, editor, *CADE*, volume 607 of *Lecture Notes in Computer Science*, pages 50–65. Springer, 1992.

[14] Thomas Ball and Sriram K. Rajamani. The SLAM project: debugging system software via static analysis. In *POPL*, pages 1–3, 2002.

[15] David A. Basin, Sebastian Mödersheim, and Luca Viganò. An on-the-fly model-checker for security protocol analysis. In Einar Snekkenes and Dieter Gollmann, editors, *ESORICS*, volume 2808 of *Lecture Notes in Computer Science*, pages 253–270. Springer, 2003.

[16] Mathieu Baudet, Véronique Cortier, and Stéphanie Delaune. YAPA: A generic tool for computing intruder knowledge. In Treinen [120], pages 148–163.

[17] Bruno Blanchet. An efficient cryptographic protocol verifier based on prolog rules. In *CSFW*, pages 82–96. IEEE Computer Society, 2001.

[18] Bugtraq list. Firefox visual spoofing flaws. `http://securityfocus.com/bid`. Bug IDs: 10532, 10832, 12153, 12234, 12798, 14526, 14919.

[19] Bugtraq list. Internet explorer visual spoofing flaws. `http://securityfocus.com/bid`. Bug IDs: 3469, 10023, 10943, 11561, 11590, 11851, 11855, 1254.

[20] Bugtraq list. Netscape navigator visual spoofing flaws. `http://securityfocus.com/bid`. Bug IDs: 7564, 10389.

[21] Sergiu Bursuc and Hubert Comon-Lundh. Protocol security and algebraic properties: Decision results for a bounded number of sessions. In Treinen [120], pages 133–147.

[22] Hao Chen, Drew Dean, and David Wagner. Model checking one million lines of C code. In *NDSS*. The Internet Society, 2004.

[23] Shuo Chen, José Meseguer, Ralf Sasse, Helen J. Wang, and Yi-Min Wang. A systematic approach to uncover security flaws in GUI logic. In *IEEE Symposium on Security and Privacy*, pages 71–85. IEEE Computer Society, 2007.

[24] Shuo Chen, David Ross, and Yi-Min Wang. An analysis of browser domain-isolation bugs and a light-weight transparent defense mechanism. In Peng Ning, Sabrina De Capitani di Vimercati, and Paul F. Syverson, editors, *ACM Conference on Computer and Communications Security*, pages 2–11. ACM, 2007.

[25] Yannick Chevalier, Ralf Küsters, Michaël Rusinowitch, and Mathieu Turuani. An NP decision procedure for protocol insecurity with XOR. In *LICS*, pages 261–270. IEEE Computer Society, 2003.

[26] Yannick Chevalier and Michaël Rusinowitch. Hierarchical combination of intruder theories. *Inf. Comput.*, 206(2-4):352–377, 2008.

[27] Yannick Chevalier and Michaël Rusinowitch. Symbolic protocol analysis in the union of disjoint intruder theories: Combining decision procedures. *Theoretical Computer Science*, 411(10):1261–1282, 2010.

[28] Stefan Ciobâca, Stéphanie Delaune, and Steve Kremer. Computing knowledge in security protocols under convergent equational theories. In Renate A. Schmidt, editor, *CADE*, volume 5663 of *Lecture Notes in Computer Science*, pages 355–370. Springer, 2009.

[29] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott. *All About Maude - A High-Performance Logical Framework*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.

[30] Manuel Clavel, Francisco Durán, Steven Eker, Santiago Escobar, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn L. Talcott. Unification and narrowing in Maude 2.4. In Ralf Treinen, editor, *Rewriting Techniques and Applications, 20th International Conference, RTA 2009, Brasília, Brazil, June 29 - July 1, 2009, Proceedings*, volume 5595 of *Lecture Notes in Computer Science*, pages 380–390. Springer, 2009.

[31] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn L. Talcott, editors. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.

[32] Hubert Comon-Lundh and Stéphanie Delaune. The finite variant property: How to get rid of some algebraic properties. In Giesl [61], pages 294–307.

[33] Véronique Cortier, Jérémie Delaitre, and Stéphanie Delaune. Safely composing security protocols. In Vikraman Arvind and Sanjiva Prasad, editors, *FSTTCS*, volume 4855 of *Lecture Notes in Computer Science*, pages 352–363. Springer, 2007.

[34] Véronique Cortier, Stéphanie Delaune, and Pascal Lafourcade. A survey of algebraic properties used in cryptographic protocols. *Journal of Computer Security*, 14(1):1–43, 2006.

[35] Richard S. Cox, Steven D. Gribble, Henry M. Levy, and Jacob Gorm Hansen. A safety-oriented platform for web applications. In *IEEE Symposium on Security and Privacy*, pages 350–364. IEEE Computer Society, 2006.

[36] Cas J. F. Cremers. The Scyther tool: Verification, falsification, and analysis of security protocols. In *CAV*, pages 414–418, 2008.

[37] Rachna Dhamija, J. D. Tygar, and Marti A. Hearst. Why phishing works. In Grinter et al. [67], pages 581–590.

[38] Rachna Dhamija and J. Doug Tygar. The battle against phishing: Dynamic security skins. In Lorrie Faith Cranor, editor, *SOUPS*, volume 93 of *ACM International Conference Proceeding Series*, pages 77–88. ACM, 2005.

[39] Danny Dolev and Andrew Chi-Chih Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–207, 1983.

[40] Francisco Durán, Steven Eker, Santiago Escobar, José Meseguer, and Carolyn L. Talcott. Variants, unification, narrowing, and symbolic reachability in Maude 2.6. In Manfred Schmidt-Schauss, editor, *Proceedings of the 22nd International Conference on Rewriting Techniques and Applications, RTA 2011, May 30 - June 1, Novi Sad, Serbia*, LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2011. To appear.

[41] Francisco Durán, Salvador Lucas, and José Meseguer. Mtt: The maude termination tool (system description). In Alessandro Armando, Peter Baumgartner, and Gilles Dowek, editors, *IJCAR*, volume 5195 of *Lecture Notes in Computer Science*, pages 313–319. Springer, 2008.

[42] Francisco Durán, Salvador Lucas, and José Meseguer. Termination modulo combinations of equational theories. In Silvio Ghilardi and Roberto Sebastiani, editors, *FroCos*, volume 5749 of *Lecture Notes in Computer Science*, pages 246–262. Springer, 2009.

[43] Francisco Durán and José Meseguer. A Maude coherence checker tool for conditional order-sorted rewrite theories. In Ölveczky [101], pages 86–103.

[44] Francisco Durán and José Meseguer. On the Church-Rosser and coherence properties of conditional order-sorted rewrite theories. *Journal of Logic and Algebraic Programming*, 2012.

[45] Jeremy Epstein, John McHugh, Rita Pascale, Hilarie Orman, Glenn Benson, and et al. A prototype B3 trusted X Window System. In *Computer Security Applications Conference*, 1991.

[46] Serdar Erbatur, Santiago Escobar, Deepak Kapur, Zhiqiang Liu, Christopher Lynch, Catherine Meadows, José Meseguer, Paliath Narendran, Sonia Santiago, , and Ralf Sasse. Effective symbolic protocol analysis via equational irreducibility conditions. 2012. Accepted at ESORICS 2012.

[47] Santiago Escobar, Catherine Meadows, and José Meseguer. A rewriting-based inference system for the NRL protocol analyzer and its meta-logical properties. *Theoretical Computer Science*, 367(1-2):162–202, 2006.

[48] Santiago Escobar, Catherine Meadows, and José Meseguer. Maude-NPA: Cryptographic protocol analysis modulo equational properties. In Alessandro Aldini, Gilles Barthe, and Roberto Gorrieri, editors, *FOSAD*, volume 5705 of *Lecture Notes in Computer Science*, pages 1–50. Springer, 2007.

[49] Santiago Escobar, Catherine Meadows, and José Meseguer. State space reduction in the Maude-NRL protocol analyzer. In Sushil Jajodia and Javier López, editors, *ESORICS*, volume 5283 of *Lecture Notes in Computer Science*, pages 548–562. Springer, 2008.

[50] Santiago Escobar, Catherine Meadows, and José Meseguer. State space reduction in the Maude-NRL protocol analyzer. *Information and Computation*, 2012. In Press.

[51] Santiago Escobar and José Meseguer. Symbolic model checking of infinite-state systems using narrowing. In Franz Baader, editor, *RTA*, volume 4533 of *Lecture Notes in Computer Science*, pages 153–168. Springer, 2007.

[52] Santiago Escobar, José Meseguer, and Ralf Sasse. Effectively checking or disproving the finite variant property. Technical Report UIUCDCS-R-2008-2960, Department of Computer Science - University of Illinois at Urbana-Champaign, April 2008.

[53] Santiago Escobar, José Meseguer, and Ralf Sasse. Effectively checking the finite variant property. In Andrei Voronkov, editor, *RTA*, volume 5117 of *Lecture Notes in Computer Science*, pages 79–93. Springer, 2008.

[54] Santiago Escobar, José Meseguer, and Ralf Sasse. Variant narrowing and equational unification. *Electronic Notes Theoretical Computer Science*, 238(3):103–119, 2009.

[55] Santiago Escobar, José Meseguer, and Prasanna Thati. Natural narrowing for general term rewriting systems. In Giesl [61], pages 279–293.

[56] Santiago Escobar, Ralf Sasse, and José Meseguer. Folding variant narrowing and optimal variant termination. In Ölveczky [101], pages 52–68.

[57] Santiago Escobar, Ralf Sasse, and José Meseguer. Folding variant narrowing and optimal variant termination. *Journal of Logic and Algebraic Programming*, 2012. DOI: 10.1016/j.jlap.2012.01.002.

[58] F. J. Thayer Fabrega, J. Herzog, and J. Guttman. Strand Spaces: What Makes a Security Protocol Correct? *Journal of Computer Security*, 7:191–230, 1999.

[59] Edward W. Felten, Dirk Balfanz, Drew Dean, , and Dan S. Wallach. Web spoofing: An internet con game. In *20th National Information Systems Security Conference*, 1996.

[60] Dinei Florencio and Cormac Herley. Stopping a phishing attack, even when the victims ignore warnings. Technical report, Microsoft Resarch, MSR-TR-2005-142.

[61] Jürgen Giesl, editor. *Term Rewriting and Applications, 16th International Conference, RTA 2005, Nara, Japan, April 19-21, 2005, Proceedings*, volume 3467 of *Lecture Notes in Computer Science*. Springer, 2005.

[62] Jürgen Giesl and Deepak Kapur. Dependency pairs for equational rewriting. In Aart Middeldorp, editor, *RTA*, volume 2051 of *Lecture Notes in Computer Science*, pages 93–108. Springer, 2001.

[63] Jürgen Giesl, Peter Schneider-Kamp, and René Thiemann. Automatic termination proofs in the dependency pair framework. In Ulrich Furbach and Natarajan Shankar, editors, *IJCAR*, volume 4130 of *Lecture Notes in Computer Science*, pages 281–286. Springer, 2006.

[64] Jürgen Giesl, René Thiemann, Peter Schneider-Kamp, and Stephan Falke. Mechanizing and improving dependency pairs. *Journal of Automated Reasoning*, 37(3):155–203, 2006.

[65] Joseph A. Goguen and José Meseguer. Equality, types, modules, and (why not ?) generics for logic programming. *Journal of Logic Programming*, 1(2):179–210, 1984.

[66] Chris Grier, Shuo Tang, and Samuel T. King. Designing and implementing the OP and OP2 web browsers. *TWEB*, 5(2):11, 2011.

[67] Rebecca E. Grinter, Tom Rodden, Paul M. Aoki, Edward Cutrell, Robin Jeffries, and Gary M. Olson, editors. *Proceedings of the 2006 Conference on Human Factors in Computing Systems, CHI 2006, Montréal, Québec, Canada, April 22-27, 2006*. ACM, 2006.

[68] Qing Guo, Paliath Narendran, and David A. Wolfram. Unification and matching modulo nilpotence. In Michael A. McRobbie and John K. Slaney, editors, *CADE*, volume 1104 of *Lecture Notes in Computer Science*, pages 261–274. Springer, 1996.

[69] Michael Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *Journal of Logic Programming*, 19&20:583–628, 1994.

[70] Michael Hanus. Lazy narrowing with simplification. *Journal of Computer Languages*, 23(2-4):61–85, 1997.

[71] Michael Hanus. Multi-paradigm declarative languages. In Verónica Dahl and Ilkka Niemelä, editors, *ICLP*, volume 4670 of *Lecture Notes in Computer Science*, pages 45–75. Springer, 2007.

[72] D. Harkins and D. Carrel. The Internet Key Exchange (IKE), November 1998. IETF RFC 2409.

[73] Steffen Hölldobler. *Foundations of Equational Logic Programming*, volume 353 of *Lecture Notes in Computer Science*. Springer, 1989.

[74] Jean-Marie Hullot. Canonical forms and unification. In Wolfgang Bibel and Robert A. Kowalski, editors, *CADE*, volume 87 of *Lecture Notes in Computer Science*, pages 318–334. Springer, 1980.

[75] Galen Hunt and Doug Brubacher. Detours: Binary interception of Win32 functions. In *Proceedings of the 3rd USENIX Windows NT Symposium*, pages 135–143, Seattle, WA, July 1999.

[76] Symantec Inc. Symantec global internet security threat report: Trends for 2008, volume xiv, April 2009. `http://www.symantec.com/threatreport/archive.jsp`.

[77] Symantec Inc. Symantec global internet security threat report: Trends for 2009, volume xv, April 2010. `http://www.symantec.com/threatreport/archive.jsp`.

[78] Jean-Pierre Jouannaud, Claude Kirchner, and Hélène Kirchner. Incremental construction of unification algorithms in equational theories. In Josep Díaz, editor, *ICALP*, volume 154 of *Lecture Notes in Computer Science*, pages 361–373. Springer, 1983.

[79] Jean-Pierre Jouannaud and Hélène Kirchner. Completion of a set of rules modulo a set of equations. *SIAM J. Comput.*, 15(4):1155–1194, 1986.

[80] Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an operating-system kernel. *Commun. ACM*, 53(6):107–115, 2010.

[81] Ralf Küsters and Tomasz Truderung. Reducing protocol analysis with XOR to the XOR-free case in the Horn theory based approach. In *ACM Conference on Computer and Communications Security*, pages 129–138, 2008.

[82] Ralf Küsters and Tomasz Truderung. Using ProVerif to analyze protocols with Diffie-Hellman exponentiation. In *CSF*, pages 157–171. IEEE Computer Society, 2009.

[83] Pascal Lafourcade, Vanessa Terrade, and Sylvain Vigier. Comparison of cryptographic verification tools dealing with algebraic properties. In *Formal Aspects in Security and Trust*, pages 173–185, 2009.

[84] V. Benjamin Livshits and Monica S. Lam. Finding security vulnerabilities in java applications with static analysis, USENIX Security Symposium 2005.

[85] Gavin Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *TACAS*, pages 147–166, 1996.

[86] Gavin Lowe and A. W. Roscoe. Using CSP to detect errors in the TMN protocol. *IEEE Trans. Software Eng.*, 23(10):659–669, 1997.

[87] Catherine Meadows. Formal verification of cryptographic protocols: A survey. In Josef Pieprzyk and Reihaneh Safavi-Naini, editors, *ASIACRYPT*, volume 917 of *Lecture Notes in Computer Science*, pages 135–150. Springer, 1994.

[88] Catherine Meadows. The NRL protocol analyzer: An overview. *Journal of Logic Programming*, 26(2):113–131, 1996.

[89] José Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[90] José Meseguer. Membership algebra as a logical framework for equational specification. In Francesco Parisi-Presicce, editor, *WADT*, volume 1376 of *Lecture Notes in Computer Science*, pages 18–61. Springer, 1997.

[91] José Meseguer and Prasanna Thati. Symbolic reachability analysis using narrowing and its application to verification of cryptographic protocols. *Higher-Order and Symbolic Computation*, 20(1–2):123–160, 2007.

[92] L4Ka::Pistachio microkernel, 2010. `http://l4ka.org/projects/pistachio`.

[93] Microsoft Corporation. Microsoft's vision for an identity metasystem. http://msdn.microsoft.com/.

[94] Aart Middeldorp and Erik Hamoen. Completeness results for basic narrowing. *Journal of Applicable Algebra in Engineering, Communication, and Computing*, 5:213–253, 1994.

[95] Juan Carlos González Moreno, Maria Teresa Hortalá-González, Francisco Javier López-Fraguas, and Mario Rodríguez-Artalejo. An approach to declarative programming based on a rewriting logic. *Journal of Logic Programming*, 40(1):47–87, 1999.

[96] Alexander Moshchuk, Tanya Bragin, Steven D. Gribble, and Henry M. Levy. A crawler-based study of spyware in the web. In *NDSS* [2].

[97] MSDN. Changing element styles. `http://msdn.microsoft.com/`.

[98] MSDN. Ole background. `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vccore/html/_core_ole_background.asp`.

[99] Naoki Nishida and Germán Vidal. Termination of narrowing via termination of rewriting. *Appl. Algebra Eng. Commun. Comput.*, 21(3):177–225, 2010.

[100] E. Ohlebusch. *Advanced Topics in Term Rewriting.* Springer, 2002.

[101] Peter Csaba Ölveczky, editor. *Rewriting Logic and Its Applications - 8th International Workshop, WRLA 2010, Held as a Satellite Event of ETAPS 2010, Paphos, Cyprus, March 20-21, 2010, Revised Selected Papers*, volume 6381 of *Lecture Notes in Computer Science*. Springer, 2010.

[102] Gerald E. Peterson and Mark E. Stickel. Complete sets of reductions for some equational theories. *J. ACM*, 28(2):233–264, 1981.

[103] Niels Provos, Panayiotis Mavrommatis, Moheeb Abu Rajab, and Fabian Monrose. All your iframes point to us. In Paul C. van Oorschot, editor, *USENIX Security Symposium*, pages 1–16. USENIX Association, 2008.

[104] Charles Reis, John Dunagan, Helen J. Wang, Opher Dubrovsky, and Saher Esmeir. BrowserShield: Vulnerability-driven filtering of dynamic html. In *OSDI*, pages 61–74. USENIX Association, 2006.

[105] Mario Rodríguez-Artalejo. Functional and constraint logic programming. In Hubert Comon, Claude Marché, and Ralf Treinen, editors, *CCL*, volume 2002 of *Lecture Notes in Computer Science*, pages 202–270. Springer, 1999.

[106] Blake Ross, Collin Jackson, Nicholas Miyake, and et al. Stronger password authentication using browser extensions. In *Usenix Security Symposium*, 2005.

[107] Grigore Rosu and Andrei Stefanescu. Matching logic: a new program verification approach. In Richard N. Taylor, Harald Gall, and Nenad Medvidovic, editors, *ICSE*, pages 868–871. ACM, 2011.

[108] Grigore Rosu and Andrei Stefanescu. From hoare logic to matching logic. In *Proceedings of the 18th International Symposium on Formal Methods (FM'12)*, LNCS. Springer, 2012. To appear.

[109] Grigore Rosu and Andrei Stefanescu. Towards a unified theory of operational and axiomatic semantics. In *Proceedings of the 39th International Colloquium on Automata, Languages and Programming (ICALP'12)*, LNCS. Springer, 2012. To appear.

[110] Peter Y. A. Ryan and Steve A. Schneider. An attack on a recursive authentication protocol. A cautionary tale. *Inf. Process. Lett.*, 65(1):7–10, 1998.

[111] S. Santiago, Carolyn L. Talcott, Santiago Escobar, Catherine Meadows, and José Meseguer. A graphical user interface for Maude-NPA. *Electronic Notes Theoretical Computer Science*, 258(1):3–20, 2009.

[112] Ralf Sasse. Source code for dissertation: Security models in rewriting logic for cryptographic protocols and browsers, July 2012. `http://formal.cs.illinois.edu/rsasse/dissertation-code/`.

[113] Ralf Sasse, Santiago Escobar, Catherine Meadows, and José Meseguer. Protocol analysis modulo combination of theories: A case study in Maude-NPA. In Jorge Cuéllar, Javier Lopez, Gilles Barthe, and Alexander Pretschner, editors, *STM*, volume 6710 of *Lecture Notes in Computer Science*, pages 163–178. Springer, 2010.

[114] Manfred Schmidt-Schauß. Unification in a combination of arbitrary disjoint equational theories. *J. Symb. Comput.*, 8(1/2):51–99, 1989.

[115] Kapil Singh, Alexander Moshchuk, Helen J. Wang, and Wenke Lee. On the incoherencies in web browser access control policies. In *IEEE Symposium on Security and Privacy*, pages 463–478. IEEE Computer Society, 2010.

[116] SpoofStick. `http://www.spoofstick.com`.

[117] Shuo Tang, Haohui Mai, and Samuel T. King. Trust and protection in the illinois browser operating system. In Remzi H. Arpaci-Dusseau and Brad Chen, editors, *OSDI*, pages 17–32. USENIX Association, 2010.

[118] Makoto Tatebayashi, Natsume Matsuzaki, and David B. Newman Jr. Key distribution protocol for digital mobile communication systems. In Gilles Brassard, editor, *CRYPTO*, volume 435 of *Lecture Notes in Computer Science*, pages 324–334. Springer, 1989.

[119] TeReSe, editor. *Term Rewriting Systems.* Cambridge University Press, Cambridge, 2003.

[120] Ralf Treinen, editor. *Rewriting Techniques and Applications, 20th International Conference, RTA 2009, Brasília, Brazil, June 29 - July 1, 2009, Proceedings*, volume 5595 of *Lecture Notes in Computer Science.* Springer, 2009.

[121] Mathieu Turuani. The CL-Atse protocol analyser. In *RTA*, pages 277–286, 2006.

[122] Laurent Vigneron. Automated deduction techniques for studying rough algebras. *Fundamenta Informaticae*, 33(1):85–103, 1998.

[123] Emanuele Viola. E-unifiability via narrowing. In Antonio Restivo, Simona Ronchi Della Rocca, and Luca Roversi, editors, *ICTCS*, volume 2202 of *Lecture Notes in Computer Science*, pages 426–438. Springer, 2001.

[124] Patrick Viry. Equational rules for rewriting logic. *Theoretical Computer Science*, 285(2):487–517, 2002.

[125] Yi-Min Wang, Doug Beck, Xuxian Jiang, Roussi Roussev, Chad Verbowski, Shuo Chen, and Samuel T. King. Automated web patrol with Strider HoneyMonkeys: Finding web sites that exploit browser vulnerabilities. In *NDSS* [2].

[126] Min Wu, Robert C. Miller, and Simson L. Garfinkel. Do security toolbars actually prevent phishing attacks? In Grinter et al. [67], pages 601–610.

[127] Junfeng Yang, Paul Twohey, Dawson R. Engler, and Madanlal Musuvathi. Using model checking to find serious file system errors (awarded best paper!). In *OSDI*, pages 273–288, 2004.

[128] Zishuang (Eileen) Ye and Sean W. Smith. Trusted paths for browsers. In Dan Boneh, editor, *USENIX Security Symposium*, pages 263–279. USENIX, 2002.

[129] Ka-Ping Yee and Kragen Sitaker. Passpet: convenient password management and phishing protection. In Lorrie Faith Cranor, editor, *SOUPS*, volume 149 of *ACM International Conference Proceeding Series*, pages 32–43. ACM, 2006.