Saarland University
Faculty of Natural Sciences and Technology I
Department of Computer Science

Selected Topics of Information Security and Cryptography

Seminar WS 08/09

Automated Checking of Observational Equivalence for an Extended Spi Calculus

Georgel Călin* Markus Rabe† Raphael Reischuk‡

May 4, 2009

Advisors:
Dr. Matteo Maffei,
Cătălin Hriţcu

*s9gecali@stud.uni-saarland.de
†markus.norman.rabe@gmx.de
‡raphael@stud.uni-saarland.de

Contents

Abstract

Borgström et al. have proposed a notion of symbolic bisimilarity for an extended Spi Calculus [12]. They developed a prototype tool, called SBC (Symbolic Bisimulation Checker) [10] implementing observational equivalence checking for arbitrary processes using shared-key cryptography. However, it is based on hard-coded equational theories.

We have partially extended SBC towards checking processes using a large class of equational theories for the message algebra. With our extension, the user can freely specify such an equational theory.

1 Introduction

Aiming to automatically analyze properties of cryptographic protocols, like secrecy or anonymity, one usually models such protocols as processes in process calculi and formulates the properties in terms of equivalence of different processes. The typical equivalence relation for this use is *observational equivalence*, one of the most significant representatives of *contextual equivalences*.

For example, a protocol $P(m)$ transferring a message m from some participant A to another participant B keeps the message m secret, if no adversary can observe a difference between $P(m)$ and $P(m')$ for two arbitrary messages $m \neq m'$.

Checking such equivalences is a hard task, since (as in the above example) quantification is usually done over *all* contexts (adversaries) and *all* messages m. Furthermore, if the message algebra is too expressive, even equivalence of expressions becomes undecidable. Even the very restricted process algebra of so called bi-processes [9] yields undecidability (a sound but incomplete method is implemented in the tool ProVerif [8]).

In the last years there has been substantial research on different proof techniques for checking observational equivalence of cryptographic protocols [14, 15, 7, 6, 9, 1]. Our work builds upon Johannes Borgström's thesis [11] which focuses on analyzing arbitrary processes (opposed to bi-processes in ProVerif [8, 6]), using a limited equational theory (see 2.1) and very late evaluation (symbolic semantics), as first proposed in [12]. Johannes Borgström and Sébastien Briais developed a tool named SBC that is able to check symbolic bisimulation for the subset of the Spi Calculus using only symmetric key encryption. However, the theory is also valid for an extended Spi Calculus which was introduced by Borgström [11] and provides more flexibility in the expression algebra as it allows arbitrary constructors and destructors.

Outline

In the beginning of our report (Section 2) we provide the theory necessary to understand the method SBC uses for checking observational equivalence. We discuss different message algebras, an extended Spi Calculus with concrete and symbolic semantics, their corresponding bisimulations, and the hedged bisimulation which is the foundation for SBC. Section 3 covers our contribution, the extension of SBC. First, we give an overview of how the tool works. We proceed with a description of the rule definition language we introduced, followed by a discussion about the modifications we have made to the original code. We close the section with a number of examples describing the current capabilities of the tool. In Section 4 we draw conclusions, present possible future extensions and discuss relevant related work.

Contributions

Our contribution is the partial extension of the SBC tool towards checking observational equivalence for processes specified in an extended Spi Calculus, allowing for arbitrary constructors and destructors. It provides a combination of specification freedom in equational theories and processes that is unmatched so far. Although not finished, the extended tool is already capable of checking a larger set of examples.

Related work

There are many approaches for automated checking of bisimulations in the Spi Calculus and the Applied π-Calculus [15, 14, 9, 6, 16, 10, 12, 17]. In Figure 1 we depict two degrees of freedom that theories and tools can provide for the user. One axis states which processes can be specified while the other indicates the freedom in equational theories.

The theories of Baudet et al. and Blanchet both allow for a large set of equational theories (as they build upon the Applied π-Calculus) but restrict to bi-processes [6, 9]. Bi-processes enable the comparison of processes that differ in their data terms only — structural differences between the processes are not allowed. Blanchet et al. implemented this approach as an extension to the well-known tool ProVerif. Baudet's method is based on symbolic semantics and is implemented in the tool YAPA [7].

The approach of Borgström et al. imposes only few restrictions on the processes to check. However, there are strictly fewer equational theories allowed in the extended Spi Calculus than in the Applied π-Calculus [12]. Still, the implementation available, SBC [10], works only for symmetric key encryption.

Very recently, Véronique Cortier and Stéphanie Delaune developed a method for checking observational equivalence in the Applied π-Calculus for a large set of processes — the class of *simple processes* [15].

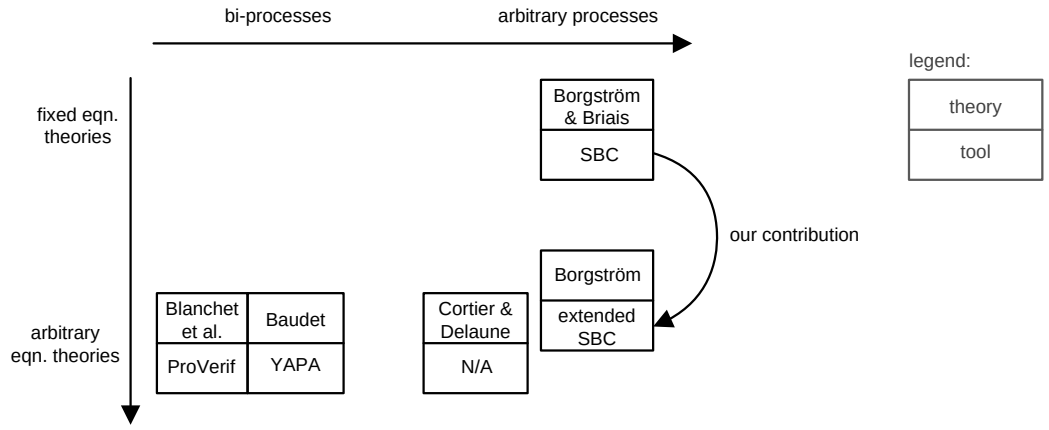

Figure 1: Our Contribution in Relation to Existing Work.

2 Verification of Cryptographic Protocols

In this section we provide a gentle introduction to automated verification of cryptographic protocols. We begin by discussing various notions of message algebras which are the differentiating factor of the calculi we introduce later in the section. Finally, we present certain bisimulations with the aim of automated checking.

2.1 Message Algebras

Message algebras determine which expressions can be build and analyzed in the process calculus and play a fundamental role in reseach on security protocols. Their properties determine the expressiveness of the calculus and therefore the range of possible applications. On the other hand, a message algebra that is too expressive might lead to undecidable equivalence problems on expressions, making the process calculus unsuitable for automated checking.

For example, the original Spi Calculus proposed by Abadi and Gordon in 1997 [3] starts with a minimal message language containing symmetric encryption and pairing for constructing composite messages (hashing and asymmetric encryption are added later in the paper). The (implicit) equational theory is fairly simple and consists of the following three equations:

$$\begin{aligned} \mathsf{Fst}(\mathsf{Pair}(x,y)) &= x, \\ \mathsf{Snd}(\mathsf{Pair}(x,y)) &= y, \\ \mathsf{Dec}(\mathsf{Enc}(x,y),y) &= x. \end{aligned}$$

An equation $\mathsf{Dec}(\mathsf{Enc}(x,y),y) = x$ means that a message $\mathsf{Dec}(\mathsf{Enc}(M,k),k')$ can be replaced by M whenever $k = k'$. The original version of the Spi Calculus [3] additionally imposes that the keys have to be names.

Recent research aims at finding a sufficiently large class of message algebras with

uniform decision procedures for security properties [1, 6]. In the following we will discuss the properties of several classes of equational theories.

Messages and Expressions

Before we discuss different message algebras, we need to introduce their foundations.

For definitions of names $\mathcal{N}$, variables $\mathcal{V}$, messages $\mathcal{M}$ and expressions $\mathcal{E}$ we refer to Section 2.2. In the following we will use M, N to range over messages and F, G to range over expressions. Further, we assume a set of constructors $\mathcal{F}^+$ and a set of destructors $\mathcal{F}^-$ whose elements are denoted with f resp. g and have arities $ar(f) \in \mathbb{N}$ resp. $ar(g) \in \mathbb{N}^+$. It is necessary to introduce this differentiation in order to disallow sending meaningless terms such as $\mathsf{Snd}(\mathsf{Enc}(m, k))$.

2.1.1 Subterm Convergent Rewriting Rules

An algebra-defining equation that is oriented from left to right is called *rewrite rule*. A rewrite system (i.e. a set of rewrite rules) is *subterm convergent*, if rewriting of any term converges to one of its subterms under repeated application of the rewrite rules.

Note that convergent rewrite systems in which deduction is decidable, can yield undecidable static equivalence (cf. [11, sect. 3.2]). However, Abadi and Cortier have shown decidability of static equivalence for the class of subterm convergent rewrite systems [1]. Borgström proposes the use of a subclass of subterm convergent rewrite systems which is suited for automated checking [11]. This subclass is called *constructor-destructor languages*.

2.1.2 Constructor-Destructor Languages

A constructor-destructor language Σ is a rewriting system consisting of the following components:

- a finite set of **constructors** $\mathcal{F}^+$and a finite set of **destructors** $\mathcal{F}^-$
  There is exactly one rewrite rule for each destructor $g \in \Sigma$ but there can be several destructors for one constructor. Thus, each destructor is associated to a single constructor.
- a set of **destructor rewrite rules**
  These rules are of the form $g(f(\widetilde{M}), \widetilde{N}) \to M'$ with $\mathrm{n}(g(f(\widetilde{M}), \widetilde{N})) = \emptyset$, $ar(f) = |\widetilde{M}|$, $ar(g) = |\widetilde{N}| + 1$ and $M' \in \{\widetilde{M}\} \cup \{\widetilde{N}\}$.

where $\mathrm{n}(F)$ are the names occurring in expression F.

Given such a language Σ, we define top-level rewriting on terms, written $F \to_\Sigma G$, as $\exists\, (F' \to G') \in \Sigma$ with an instantiation σ such that $F'\sigma = F$ and $G'\sigma = G$.

The uniqueness of destructor rules is chosen in order to ensure deterministic evaluation, although it would be preferable to allow multiple rules per destructor. We refer to Section 4.1 for further discussion.

In the original Spi Calculus [3] keys are restricted to be names. As Borgström stated in his thesis [11], the constructor-destructor languages do not allow for specifying such restrictions. However, we believe that it does not lead to problems if permit to specify such constraints, as there is already a similar restriction for channel names.

Example 2.1. In this example we provide a list of rules in order to depict the limitations of expressiveness of constructor-destructor languages.

- **Nondeterministic choice** rules like the following:

  $$\mathsf{Either}(\mathsf{Pair}(x, y)) \to x \qquad \text{and} \qquad \mathsf{Either}(\mathsf{Pair}(x, y)) \to y$$

  cannot both be present in a constructor-destructor language. They also do not yield a convergent rewrite system.

- In contrast, the **parameterized choice** rules

  $$\mathsf{Pick}(\mathsf{Pair}(x, y), 1) \to x \qquad \text{and} \qquad \mathsf{Pick}(\mathsf{Pair}(x, y), 2) \to y$$

  are not permitted in a constructor-destructor language, but yield a (subterm) convergent rewrite system. However, we can resolve this problem by introducing distinct destructor symbols Pick_1 and Pick_2 that each implement one of the rules:

  $$\mathsf{Pick}_1(\mathsf{Pair}(x, y)) \to x \qquad \text{and} \qquad \mathsf{Pick}_2(\mathsf{Pair}(x, y)) \to y$$

- The **limited inverse rule**

  $$g(f_1(f_2(x))) \to f_2(x)$$

  is allowed in a constructor-destructor language.

- The **idempotent rule**

  $$f(f(x)) \to f(x)$$

  as well as the **self-inverse rule**

  $$f(f(x)) \to x$$

  are not allowed in a constructor-destructor language. However, they can be part of a subterm convergent rewrite system.

2.1.3 Common Cryptographic Primitives

In this section we show how to realize some common cryptographic primitives in constructor-destructor languages.

- **Hash functions** work without any rule. In this way the attacker cannot analyze a received message $x = \mathsf{Hash}(k)$ and thus does not learn anything about secret k. However, if the components of the hash are known to the attacker can build the expression himself and can learn something about the content of x by checking for equality: $x = \mathsf{Hash}(k)$.
- **Symmetric encryption**
$$\mathsf{Dec}_s(\mathsf{Enc}_s(x, y), y) \to x$$
and **asymmetric encryption**
$$\mathsf{Dec}_a(\mathsf{Enc}_a(x, \mathsf{Pk}(y)), \mathsf{Sk}(y)) \to x$$
are defined in a straightforward way.
- **Signatures**
$$\begin{aligned} \mathsf{ExtractMsg}(\mathsf{Sign}(x, \mathsf{Sk}(y))) &\to x \\ \mathsf{Verify}(\mathsf{Sign}(x, \mathsf{Sk}(x)), \mathsf{Pk}(y)) &\to \mathsf{True} \end{aligned}$$
Actually, the second rewrite rule needs a trick: we have to append a dummy argument True to the destructor arguments.
$$\mathsf{Verify}(\mathsf{Sign}(x, \mathsf{Sk}(x)), \mathsf{Pk}(y), \mathsf{True}) \to \mathsf{True}$$

2.2 Extended Spi Calculus

The original Spi Calculus [3] is rooted in the tradition of the π-Calculus [19] which was designed for describing parallel processes and their interaction. What distinguishes the Spi Calculus from its ancestor, is the focus on cryptographic protocols by adding a simple term algebra to guards (called *matches* in [19]). However, this language provides limited flexibility as all its cryptographic primitives are hardcoded in the term algebra. The Applied Pi Calculus [2] generalizes this approach by allowing arbitrary equational theories of terms on the cost of possibly undecidable equivalence of terms.

In this report we consider the *Extended Spi Calculus* [11] which is a similar generalization of the Spi Calculus. It aims at automated checking, thus restricts equational theories to constructor-destructor languages.

2.2.1 Syntax

We assume a countably infinite set of names $\mathcal{N}$ with a, b, c, k, l, m, n ranging over its elements. We use x, y, z to range over the infinite set of variables $\mathcal{V}$ and u, v, w to range

F, G	::=	$x \mid a \mid f(\widetilde{F}) \mid g(\widetilde{F})$	expressions $\mathcal{E}$
M, N	::=	$a \mid f(\widetilde{M})$	messages $\mathcal{M}$
ϕ, ψ	::=	$\mathsf{true} \mid \phi \wedge \phi$	guards $\mathcal{G}$
	$\mid$	$[G = F] \mid [F : \mathcal{N}]$	
P, Q	::=	$\mathbf{0} \mid F(x).P \mid \overline{F}\langle F\rangle.P$	processes $\mathcal{P}$
	$\mid$	$!F(x).P \mid P + P \mid P \mid P$	
	$\mid$	$(\nu a)P \mid \phi P$	

Table 1: Syntax of the Extended Spi Calculus.

P	$\mathrm{fn}(P)$	$\mathrm{fv}(P)$
0	$\emptyset$	$\emptyset$
$F(x).Q$	$\mathrm{n}(F) \cup \mathrm{fn}(Q)$	$\mathrm{v}(F) \cup (\mathrm{fv}(Q) \setminus \{x\})$
$\overline{F}\langle G\rangle.Q$	$\mathrm{n}(F) \cup \mathrm{n}(G) \cup \mathrm{fn}(Q)$	$\mathrm{v}(F) \cup \mathrm{v}(G) \cup \mathrm{fv}(Q)$
$!F(x).Q$	$\mathrm{n}(F) \cup \mathrm{fn}(Q)$	$\mathrm{v}(F) \cup (\mathrm{fv}(Q) \setminus \{x\})$
$Q_1 + Q_2$	$\mathrm{fn}(Q_1) \cup \mathrm{fn}(Q_2)$	$\mathrm{fv}(Q_1) \cup \mathrm{fv}(Q_2)$
$Q_1 \mid Q_2$	$\mathrm{fn}(Q_1) \cup \mathrm{fn}(Q_2)$	$\mathrm{fv}(Q_1) \cup \mathrm{fv}(Q_2)$
$(\nu a)Q$	$\mathrm{fn}(Q) \setminus \{a\}$	$\mathrm{fv}(Q)$
ϕQ	$\mathrm{n}(\phi) \cup \mathrm{fn}(Q)$	$\mathrm{v}(\phi) \cup \mathrm{fv}(Q)$

Table 2: Free Names and Variables.

over the union of $\mathcal{N}$ and $\mathcal{V}$. The definitions of expressions, messages, guards and processes are listed in Table 1. As in the Applied Pi calculus there is no *let* construct — only an *if*-like guard.

For the sake of readability, we will use the abbreviation $\widetilde{A}$ for a sequence $A_1, \ldots, A_n$ for some $n \in \mathbb{N}^+$. Further, common notations such as $|\widetilde{A}|$ for the length of a sequence and $f(\widetilde{A})$ for component-wise application of some function f will be used.

We define the names $\mathrm{n}(\cdot)$ and variables $\mathrm{v}(\cdot)$ of a term to be the names resp. variables occurring in it. The *free names* $\mathrm{fn}(\cdot)$ and *free variables* $\mathrm{fv}(\cdot)$ are defined in Table 2. The *bound names* $\mathrm{bn}(\cdot)$ resp. *bound variables* $\mathrm{bv}(\cdot)$ are the complement of the free names resp. free variables in the set of used names $\mathrm{n}(\cdot)$ resp. used variables $\mathrm{v}(\cdot)$.

Substitutions σ can be instantiations (idempotent functions $\{^F/_x\}$ of type $\mathcal{V} \to \mathcal{E}$) or renamings (injective functions of type $\mathcal{N} \to \mathcal{N}$). They apply the usual way to processes, expressions and guards.

2.2.2 Semantics

There are various notions of semantics available, which differ in the time at which constraints are evaluated but are still equivalent. We will take a look at *late* and *very late* semantics, which we call *concrete* respectively *symbolic* semantics. In the following, we provide an intuitive explanation of the concrete semantics before discussing the

Structure of expression F:	$\mathsf{e}(F)$:
$F \in \mathcal{N}$	F
$F = f(\widetilde{M}),\ f \in \mathcal{F}^+$	$f(\mathsf{e}(\widetilde{M}))$
$F = g(\widetilde{G}),\ g \in \mathcal{F}^-,\ g(\mathsf{e}(\widetilde{G})) \rightarrow_\Sigma M$	M
otherwise	$\perp$

Table 3: Evaluation of Expressions.

ϕ :	evaluates to:
$[\![\mathsf{true}]\!]$	true
$[\![\psi_1 \wedge \psi_2]\!]$	$[\![\psi_1]\!]$ and $[\![\psi_2]\!]$
$[\![[F = G]]\!]$	$\mathsf{e}(F) = \mathsf{e}(G) \neq \perp$
$[\![[F : \mathcal{N}]]\!]$	$\mathsf{e}(F) \in \mathcal{N}$

Table 4: Evaluation of Guards.

principles of symbolic semantics:

- **0**: nil process. It does nothing and may as well be omitted.
- $c(x).P$: *input* on channel c and execute process P with x bound to the input.
- $\overline{c}\langle M\rangle.P$: *output* message M on channel c and execute P.
- $P \mid Q$: *parallel composition* of P and Q. The processes can execute their actions independently or they choose to communicate over a common channel if there is a matching pair of input and output actions available.
- $(va)P$: create a *new name* a and execute P. The new name is not known outside the process P.
- ϕP: if the *guard* ϕ evaluates to true, execute process P.
- $!F(x).P$: *replication* behaves like an unbounded number of copies of $F(x).P$ running in parallel.

We use $\rightarrow$ to denote the relation on processes induced by the concrete semantics. Note that in the following we consider only the subset of the Extended Spi Calculus not using replication.

Since expressions may contain destructors, we need to define semantics for evaluation of expressions under an equational theory Σ (see Table 3). As mentioned above, we restrict the possible equational theories to be constructor-destructor languages. Evaluation of guards is defined in Table 4.

In case of an input like $c(x).P$ an adversary (in form of a context) can send any term it can construct resulting in infinitely many possibilities to instantiate the variable. The symbolic semantics avoids this problem by postponing instantiation as far as possible.

$$\text{Cin}\quad \frac{\mathsf{e}(G) = a}{G(x).P \xrightarrow{a(x)} P} \qquad \text{Sin}\quad \frac{}{G(x).P \xrightarrow[[G:\mathcal{N}]]{G(x)} P}$$

$$\text{Sout}\quad \frac{}{\overline{G}\langle F\rangle.P \xrightarrow[[G:\mathcal{N}]\wedge[F:\mathcal{M}]]{\overline{G}\langle F\rangle} P}$$

$$\text{Scom}\quad \frac{P \xrightarrow[\phi_P]{(\nu\widetilde{b})\,\overline{G}\langle F\rangle} P' \qquad Q \xrightarrow[\phi_Q]{(\nu\widetilde{c})\,G'(x)} Q'}{P \mid Q \xrightarrow[\phi_P\wedge\phi_Q\wedge[G=G']]{(\nu\widetilde{bc})\,\tau} P' \mid Q'\{F/x\}}$$

Table 5: Concrete (Cin) and Symbolic (Sin, Sout, Scom) Semantics.

Instead of directly instantiating and evaluating the conditions of a transition, we add constraints that have to hold on the path in question. Thereby, existence of a path becomes a satisfaction problem on the conjunction of the constraints of all its transitions, making symbolic semantics a helpful step towards the quest of automated checking.

In Table 5 we present a selection of inference rules for the symbolic semantics opposed to the inference rule Cin of the concrete semantics. Cin has the precondition $\mathbf{e}(G) = a$ whereas Sin has no preconditions but introduces the equivalent path condition $[G : \mathcal{N}]$.

The exact definitions of the symbolic semantics are not necessary to understand the insights to follow. We refer to Borgström's thesis [11] for these definitions.

2.3 Environment-Sensitive Bisimulations

In this section we give a brief introduction to environment-sensitive bisimulations with a focus on those suited for automated checking.

Environment-sensitive bisimulations require behavioral equivalence under all environments additionally to their bisimulation properties. Our target relation is *observational equivalence*, an important representative of these equivalences, which is equivalent to *barbed equivalence* [11, 20].

Definition 2.2. *Observational equivalence* is the largest symmetric relation $\mathcal{R}$ between closed extended processes with the same domain such that $A\,\mathcal{R}\,B$ implies:

1. if $A \Downarrow c$, then $B \Downarrow c$;
2. if $A \to^* A'$, then $B \to^* B'$ and $A'\,\mathcal{R}\,B'$ for some B';
3. $C[A]\,\mathcal{R}\,C[B]$ for all evaluation contexts C.

where $P \Downarrow c$ denotes possible output of process P on channel c and $\to^*$ denotes the transitive closure of the transition relation $\to$. Evaluation contexts C are built from $[]$, $C \mid P$, $P \mid P$, $(\nu a)C$.

Observational equivalence is hard to check, since the definition quantifies over all (infinitely many) contexts. In order to overcome this problem, we need a sound relation that does not rely on this type of quantification. Therefore, we move to the corresponding relation in the symbolic world, the *symbolic bisimulation*. Symbolic bisimulation cirumvents the problem of infinite branching by replacing the contexts in definition 2.2 by symbolic environments distinguishing only finitely many cases.

We will not provide the definition of symbolic bisimulation and refer to the literature [11, 16, 12, 18]. In his thesis, Borgström proves equivalence between the version of concrete and symbolic bisimulation we use.

Hedged Bisimulation

With the aim of a checking method, Borgström introduced the *Hedged bisimulation* being sound to symbolic bisimulation. The idea of this approach is to maintain a set of message pairs each representing a fragment of information being exposed to the environment in the respective processes at a common point of time. If the environment can distinguish the messages of such a pair, the corresponding processes are not equivalent. The bisimilarity check works by searching for a series of "consistent" hedges (see Definition 2.7) that correspond to the evaluation steps of the processes.

Definition 2.3. A *hedge* is a finite subset of $\mathcal{M} \times \mathcal{M}$. We denote by $\mathcal{H}$ the set of all hedges.

Example 2.4. This example gives corresponding hedges for a pair of processes.

$$\begin{aligned} & (\nu a)\ \overline{c}\langle \mathsf{Hash}(a)\rangle.\overline{c}\langle a\rangle.0, \\ \text{and} \quad & (\nu a)\ \overline{c}\langle \mathsf{Enc}(a,a)\rangle.\overline{c}\langle a\rangle.0 \end{aligned}$$

The initial hedge contains the pairs of free names of the processes, in this case only (c, c). After executing the first step of each process the hedge additionally contains the pair $(\mathsf{Hash}(a), \mathsf{Enc}(a, a))$ which is still consistent as a is not known to the environment and thus neither of the messages can be analyzed. Executing the next step will add the pair (a, a) to the hedge. This enables the environment to analyze (decrypt) and distinguish the previously received messages rendering the hedge inconsistent. The processes are therefore not observationally equivalent.

Definition 2.5. For a hedge h, we obtain $\mathcal{S}^+(h)$ by applying constructors to the elements of h. The *synthesis* $\mathcal{S}(h)$ is the union of $\mathcal{S}^+(h)$ and h. Note that the intersection of $\mathcal{S}^+(h)$ and h is not necessarily empty.

The *analysis* of a hedge $\mathcal{A}(h)$ is defined as the union of h and the hedge resulting after application of destructors with parameters that are constructible out of $\mathcal{A}(h)$ itself.

The *irreducibles* $\mathcal{I}(\cdot)$ of a hedge are defined as

$$\mathcal{I}(h) := \mathcal{A}(h) \setminus \mathcal{S}^+(\mathcal{A}(h))$$

representing the minimal set of elements of h that are necessary to reconstruct $\mathcal{S}(\mathcal{A}(h))$.

Example 2.6. We use the equational theory resulting from the union of all rewriting rules defined in Section 2.1.3. Let

$$h = \{(\mathsf{Hash}(k), \mathsf{Hash}(k)),\ (\mathsf{Pair}(a,a), \mathsf{Pair}(b,b)),\ (\mathsf{Enc}(m, \mathsf{Hash}(a)), \mathsf{Enc}(m, \mathsf{Hash}(b)))\}.$$

Then the analysis of h is $\mathcal{A}(h) = h \cup \{(a,b),\ (m,m)\}$. Note that we can decrypt the encrypted message, since we have $(\mathsf{Hash}(a), \mathsf{Hash}(b)) \in \mathcal{S}(\mathcal{A}(h))$.

The set of irreducibles is:

$$\mathcal{I}(h) = \{(\mathsf{Hash}(k), \mathsf{Hash}(k)),\ (a,b),\ (m,m)\}$$

Definition 2.7. An irreducible hedge h is *left consistent* iff for all $(M,N) \in h$ it holds that

1. if $M \in \mathcal{N}$ then $N \in \mathcal{N}$, and
2. there is no $N' \neq N$ with $(M,N') \in \mathcal{S}(h)$
3. $\forall(\widetilde{M',N'}) \in \mathcal{S}(h), g \in \mathcal{F}^-$: if $g(M,\widetilde{M'})$ succeeds then $g(N,\widetilde{N'})$ succeeds as well.

where $(\widetilde{M',N'})$ denotes a sequence of pairs and $\widetilde{M'}$ as well as $\widetilde{N'}$ are their respective projections. Note that we simplified condition 3 in order to avoid introducing several hard to understand definitions. We call a hedge h *consistent* iff both h and h^{-1} are left consistent.

Consistency is the most important definition in this report. It is a machine checkable property of hedges (according to [11, p.105]) providing a sound indicator for observational equivalence.

Definition 2.8. A *hedged relation* $\mathcal{R}$ is a subset of $\mathcal{H} \times \mathcal{P} \times \mathcal{P}$. We say that $\mathcal{R}$ is consistent if $h \vdash P\ \mathcal{R}\ Q$ implies that h is consistent. A consistent hedged relation is a *hedged simulation* if whenever $h \vdash P\ \mathcal{R}\ Q$ we have

1. if $P \xrightarrow{\tau} P'$ then there exists Q' such that $Q \xrightarrow{\tau} Q'$ and $h \vdash P'\ \mathcal{R}\ Q'$.
2. if $P \xrightarrow{a(x)} P'$, $h \vdash a \leftrightarrow b$, $B \subset \mathcal{N}$ is finite, $B \cap (\mathrm{fn}(P,Q) \cup n(h)) = \emptyset$, $M, N \in \mathcal{M}$, and $h \cup Id_B \vdash M \leftrightarrow N$, then there exists Q' such that $Q \xrightarrow{b(x)} Q'$ and $h \cup Id_B \vdash P'\{^M/_x\}\ \mathcal{R}\ Q'\{^N/_x\}$.
3. if $P \xrightarrow{(\nu\widetilde{c})\overline{a}\langle M\rangle} P'$, $h \vdash a \leftrightarrow b$ and $\{\widetilde{c}\} \cap (fn(P) \cup n(\pi_1(h))) = \emptyset$ there exist Q', N, $\widetilde{d}$ with $\{\widetilde{d}\} \cap (\mathrm{fn}(Q) \cup n(\pi_2(h))) = \emptyset$ such that $Q \xrightarrow{(\nu\widetilde{d})\overline{b}\langle N\rangle} Q'$ and $\mathcal{I}(h \cup \{(M,N)\}) \vdash P'\ \mathcal{R}\ Q'$.

$\mathcal{R}$ is a *hedged bisimulation* if both $\mathcal{R}$ and $\mathcal{R}^{-1}$ are hedged bisimulations.

This definition does not rely on quantification over environments. The automated checking method of hedged bisimilarity builds the hedge in a stepwise manner as described in the definition above and checks for consistency in each step.

3 Extensions to SBC

Johannes Borgström and Sebastian Briais created a prototype tool 'SBC' to check observational equivalence for the Spi Calculus in an automated fashion. According to Borgström's thesis the checker is based on an unproven method [11, Appendix B]. Nevertheless, we decided to extend the checker, since it runs reasonable stable (it was successfully tested on the subset of the *Security Protocols Online REpository* [13] using only symmetric key encryption).

Our contribution is the extension of SBC towards checking protocol specifications in the Extended Spi Calculus. As it was written for Spi Calculus constructs only, most functions of the original tool had hardcoded parts for symmetric encryption and pairing. The core task was to generalize each function to the case of arbitrary constructor-destructor languages.

In the following, we give an overview of the tool's method to check bisimilarity, discuss the language extension necessary for specifying rewrite rules, explain selected code excerpts, and take a look at the current state of the tool.

3.1 Overview

In the following, we give a high-level view on how the tool checks observational equivalence. Little has changed in the general structure of the code, so the description fits on both the original SBC as well as our extended version.

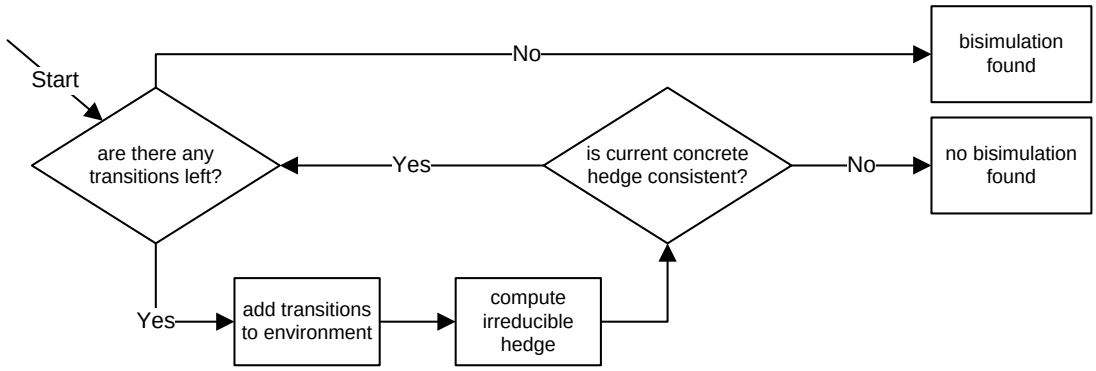

Figure 2: Schematic Main Loop of SBC.

We have to stress that the tool's method cannot be complete since equality in observational equivalence is undecidable (not even semi-decidable).

The tool works with symbolic transitions and for a given pair of processes builds a hedge and a symbolic environment as described in Borgström's thesis [11, pp.122-134]. The main loop of the program (depicted schematically in Figure 2) repeatedly adds the next transition constraints to the symbolic environment, computes the corresponding

irreducible hedge, and checks for consistency.

If this procedure comes to an end, i.e. there are no unhandled transitions left, we found a bisimulation for the process pair. Up to correct implementation of the theory, soundness is guaranteed by Borgström's theoretical results concerning symbolic bisimulations [11].

Once we found an inconsistent hedge, we report that no bisimulation could be found. However, this does not necessarily mean that there is none, as the method is incomplete. This result usually hints towards a violation of the property which we analyze, i.e. an error in the protocol.

3.2 Parser Extension

We have extended the parser of SBC to give the possibility of specifying an arbitrary constructor-destructor language via a set of rules.

The rules defining the equational theory are subject to the following grammar:

```
RULE     ::= 'rule' BODY '->' EXPR
BODY     ::= DES '(' CONS '(' EXPRLIST ')' ',' EXPRLIST ')'
EXPRLIST ::= EXPR [',' EXPRLIST]
EXPR     ::= VARIABLE | CONS '(' EXPRLIST ')'
```

where variables (VARIABLE), constructors (CONS), and destructors (DES) are strings consisting of alphanumeric characters. Constructor names have a leading '#', destructor names begin with '-'. Both constructors and destructors are introduced through their occurrences in the rules.

The definition of constructor-destructor languages restricts the right-hand side of the rewrite rule to occur as child expressions of the constructor or of the destructor (except for its first child). We have decided to allow for specifying arbitrary subexpressions, as it seems to cause no problems. This helps to specify some primitives like blinding which have deeply nested return values.

When parsing a destructor rule we search for the following information in the destructor arguments:

(i) which subexpression is the return value?

(ii) which variables must be equal?

(iii) which constructor symbols occur at which locations?

Information (ii) is necessary to check for key matchings when a destructor is applied. The third piece of information describes the structure of the destructor.

Storing Arbitrary Destructor Rules

For extending SBC to constructor-destructor languages, the information for each destructor rule has to be stored for later access by the checker. For this purpose, we introduce a new datatype `destructorinfo` consisting of three main components:

(i) `address`: the position of the return value in the destructor tree, stored as a list of descendant node numbers.

(ii) `matchlist`: (`address`,`address`) pairs describing addresses in the destructor rule where expressions have to be equal in order to apply the rule.

(iii) `constructorlist`: (`address`,`constructor`) pairs indicating the structure of the expression tree: for a rule to be applied, the respective constructor has to occur at the given address.

These three components are stored in a table of type `destructor * destructorinfo`.

Example 3.1. Consider the rule

$$\mathsf{des}(\mathsf{cons1}(a, b, c), \mathsf{cons2}(a, d), b) \to c$$

The `destructorinfo` contains the return address as `[1;3]`, where `1` represents `cons1` and `3` the variable c. The `matchlist` consists of the pairs (`[1;1]`,`[2;1]`) and (`[1;2]`,`[3]`) for the matchings of a and b. The `constructorlist` contains two elements: for `cons1` the pair (`[1]`,`cons1`) and for `cons2` the pair (`[2]`,`cons2`).

3.3 Code Excerpts

In this section we describe some relevant functions of the extended SBC code. We explain the role of the respective code excerpts and the changes that were necessary in order to support arbitrary constructors and destructors.

Note that we did not remove the hard-coded primitives of the original version — we even kept them up to date. Beyond possibilities for testing, this enables us to provide simple cases that help the reader to get the rough idea before we discuss the general cases.

The following three procedures are described. We start with a simple example, `eval_concrete`, implementing the evaluation of expressions described in Table 3. The second code fragment `consistency_of_concr_hedge` covers one of the most important procedures of the tool: the consistency check according to Definition 2.7. The last excerpt shows how the analysis (Definition 2.5) for an expression pair is computed.

3.3.1 Concrete Expression Evaluation

The function `eval_concrete` evaluates a concrete term according to the definitions in Table 3. It is used in multiple locations in the program. The method's only parameter is an expression F. It returns the evaluated expression if the evaluation of F is defined — otherwise `None`. In the following listing we see some of the cases for hard-coded functions implementing the rewriting rule $\mathsf{Fst}(\mathsf{Pair}(x, y)) \to x$.

```
1  let rec eval_concrete = function
2      Name(n) -> Some(Name(n))
3    | Pair(f,g) ->            (* hard-coded constructor *)
4        match eval_concrete f, eval_concrete g with
5          Some(m), Some(n) -> Some(Pair(m,n))
6        | _, _ -> None
7    | Fst(f) ->               (* hard-coded destructor *)
8        match eval_concrete f with
9          Some(Pair(m,_)) -> Some(m)
10       | _ -> None
```

Listing 1: Concrete Evaluation for Pairs.

The constructor `Pair` and destructor `Fst` succeed only if their subexpressions evaluate successfully. The destructor `Fst` additionally requires the evaluated subexpression to be a `Pair`, as it is necessary to apply the according rewrite rule.

Listing 2 shows the case for general constructors.

```
11   | Cons(c,args) ->
12       let rec eval_args = function
13           [] -> []
14         | x::xs ->
15           match eval_concrete x with
16               None -> []
17             | Some(ev) -> ev::(eval_args xs)
18       in
19         match eval_args args with
20           [] -> None
21         | vs -> Some(Cons(c,vs))
```

Listing 2: Concrete Evaluation for Arbitrary Constructors.

All constructor arguments `args` are evaluated using the recursive procedure `eval_args` (line 19). If the evaluation of all subexpressions succeeds (line 21), we return the reduced expression including the constructor — otherwise we return `None` (line 20).

Evaluation for destructors is more involved:

```
22   | Des(d,args) ->
23       match args with
24         [] -> ERROR!
25       | c::da ->
```

```
26            match eval_concrete c with
27              Some(Cons(cons,ca)) ->
28                let evaluated_des_args = List.fold_left
29                  (fun evlist arg ->
30                    match eval_concrete arg with
31                        Some(eval) -> eval :: evlist
32                      | None -> ERROR!
33                  )
34                  [] da
35                in
36                  apply_destr_concrete d (Cons(cons,ca)::evaluated_des_args)
37            | _ -> None
```

Listing 3: Concrete Evaluation for Arbitrary Destructors.

Line 24 implements the requirement for destructors to have at least one argument. The first argument c should evaluate to a constructor cons (lines 25 to 27). In this case all other destructor arguments da are evaluated using `eval_concrete` (lines 28 to 34). Otherwise the destructor cannot be evaluated (line 37). All of the destructor arguments should be successfully evaluated in order to apply the destructor rule (line 32).

Assuming all evaluations succeeded, the destructor rule is applied (line 36) using procedure `apply_destr_concrete` which is described below. Note that this method was written with a possible further extension in mind — it can handle multiple rewrite rules per destructor.

```
1   let rec apply_destr_concrete destr destr_args =
2     let Cons(c,cons_args)::_ = destr_args in
3     let rec check_rules = function
4         [] -> None
5       | (addr, constrList, matchList)::rules ->
6           if (constructors_match constrList destr_args)
7           && (arguments_match matchList destr_args)
8           then Some(get_subexp_at_address addr destr_args)
9           else check_rules rules
10    in
11      check_rules (destructors.lookup_destructor destr)
```

Listing 4: Concrete Application of a Destructor.

Given the destructor symbol `destr` and its arguments `destr_args` (in which are contained the constructor symbol `c` with arguments `cons_args`, line 2), application of the destructor is performed by calling the recursive procedure `check_rules` (line 11) with all corresponding rules for destructor `destr`.

If no rules are available (line 4) the destructor cannot be applied. Otherwise, it is checked whether the constructors occur at the specified positions (line 6) and that corresponding arguments match (line 7). If this check succeeds the return value of the destructor is returned (line 8). Otherwise the next rule is examined (line 9).

3.3.2 Concrete Hedge Consistency

The procedure `consistency_of_concr_hedge` is the very heart of SBC. Loosely speaking, the consistency check reports whether two processes can be distinguished by the environment.

The input is a concrete irreducible hedge `h`, the names of the left part of the current knowledge of the environment and for the right part we have messages and variables.

The procedure iterates over all elements of the hedge (line 3). The current expression pair is referred to as `(exp1,exp2)`.

```
1  let consistency_of_concr_hedge h left_names right_name_msgs right_vars =
2    let (proj1,proj2) = Hedge.projections_of_hedge h in
3    Hedge.for_all
4      (fun (exp1,exp2) -> match (exp1,exp2) with
```

The case distinction for `(exp1,exp2)` is listed below. Consider the simple (hard-coded) cases first.

```
5          Expression.Enc(m1,k1), Expression.Enc(m2,k2) ->
6            not (ExpressionSet.mem k1 proj1) && not (ExpressionSet.mem k2 proj2)
```

Given two encryptions, the environment is consistent if no key is available in the hedge `h` (line 6).[1]

Intuitively, if key `k1` is available in the hedge then decryption for the left process could be applied, hence the message `m1` would be added to the knowledge of the environment. Consequently, the attacker could distinguish the processes since the destructor only succeeds for one of the processes. (Note that the keys cannot occur both in `h`. If so, the pair `(exp1,exp2)` would not occur in the irreducible hedge `h` since it can be constructed from `h`.)

More formally, line 6 implements part 3 of Definition 2.7.

```
7      | Expression.Name(a1), Expression.Name(a2) ->
8          not (NameSet.mem a1 left_names)
9          ||  (ExpressionSet.mem (Expression.Name(a2)) right_name_msgs)
10         ||  not (NameSet.mem a2 right_vars)
```

Given two names `a1` and `a2`, we check the implication

$$\texttt{a1} \in \texttt{left\_names} \quad \Rightarrow \quad \texttt{a2} \in \texttt{right\_name\_msgs} \vee \texttt{a2} \notin \texttt{right\_vars}$$

This corresponds to case 1 of the consistency definition (Definition 2.7).

[1] The attentive reader might ask why we only test for membership of the hedge `h` and not of the synthesis of `h`: for hard-coded encryptions, compound keys are not allowed.

The arbitrary case for constructors looks as follows:

```
11      | Expression.Cons(c1,ca1), Expression.Cons(c2,ca2) ->
12          let all_args1 = List.for_all (fun e -> is_in_synth_h e proj1) ca1 in
13          let all_args2 = List.for_all (fun e -> is_in_synth_h e proj2) ca2
14          in
15            not all_args1 && not all_args2
16            && (des_can_be_applied exp1 h = des_can_be_applied exp2 h)
```

For both constructors `c1` and `c2` we check that there exists at least one argument that is not available in the synthesis of hedge `h`. Otherwise, if all arguments for one constructor are available then condition 2 of Definition 2.7 is violated. Recall that we assume `h` to be irreducible, hence not all arguments for both constructors will be in the synthesis for `h`.

Additionally, we have to check that application of destructors must be possible for both constructors or not possible at all. The procedure `des_can_be_applied` (line 16) performs this check by iterating over all destructors. This implements part 3 of definition 2.7.

We stress that in general there is neither an implication between the condition in line 15 to the one in line 16 nor in the converse direction. Consider a case in which not all arguments of a constructor are available, hence the attacker cannot construct the object himself. However, a destructor might be applied when all required destructor arguments are available. On the other hand, consider asymmetric deterministic encryption and decryption. The attacker might reconstruct the ciphertext, but as the private key is not known, may not decrypt it.

Note that we do not explicitly treat pairs here. As the destructor for pairs has no restriction, the pair's components can always be reconstructed. Hence, pairs should never occur in any irreducible hedge. The following rule covers pairing constructors as well as any destructor. Further, it implements the cases in which we get a pair of two different constructs (e.g. a pair and a name).

```
17      | _ -> false
```

Besides the cases above, the for-all construct in line 18 to 20 additionally checks the implication `exp1=exp1'` $\Rightarrow$ `exp2=exp2'` implementing part 2 of Definition 2.7.

```
18    && Hedge.for_all
19         (fun (exp1',exp2')  ->  not (exp1 = exp1') || (exp2 = exp2'))
20         h
```

3.3.3 Analysis for Hedges

The procedure `check_analyse` tries to apply destructors in order to compute the analysis of a hedge. It is iteratively called with the sets of process pairs `doneset`, `to_check`, and

`newset` together with the current expression pair `(e1,e2)` from `to_check`. At the beginning, `doneset` and `newset` are empty, `to_check` contains the knowledge of the environment, i.e. is the hedge to analyze. Over time, processed expression pairs move from `to_check` to `doneset` while newly discovered pairs are added to `newset`. At the end, the hedges `doneset`, `to_check`, and `newset` are returned. `Check_analyse` is re-executed unless a fixed point has been computed, i.e. no more elements of `to_check` can be analyzed.

Again, let us consider the simplest cases first.

```
1  let check_analyse (e1,e2) (doneset,to_check,newset) =
2    match e1,e2 with

4      Pair(f1,g1), Pair(f2,g2) ->
5        let to_check' = Hedge.remove (e1,e2) to_check in
6        let doneset' = Hedge.add (e1,e2) doneset in
7        let newset' =
8          if (Hedge.mem (f1,f2) to_check) || (Hedge.mem (f1,f2) doneset)
9          then newset
10         else Hedge.add (f1,f2) newset in
11       let newset' =
12         if (Hedge.mem (g1,g2) to_check) || (Hedge.mem (g1,g2) doneset)
13         then newset'
14         else Hedge.add (g1,g2) newset'
15       in
16         (doneset', to_check', newset')
```

Listing 5: `check_analyse` for Pairs.

The analysis procedure for pairs returns their components since the destructors `Fst` and `Snd` can always be applied. Hence both pairs `(e1,e2)` are moved from `to_check` to `doneset` (line 5 to 6). The components are only added to `newset` if not already contained in the analysis (line 7 to 14).

Next, consider the analysis for encryptions.

```
17   | Enc(m1,k1), Enc(m2,k2) ->
18       if is_in_synth_h (k1,k2) (Hedge.union to_check doneset)
19       then   (* keys are available *)
20         let to_check' = Hedge.remove (e1,e2) to_check in
21         let doneset' = Hedge.add (e1,e2) doneset in
22         let newset' =
23           if (Hedge.mem (m1,m2) to_check) || (Hedge.mem (m1,m2) doneset)
24           then newset
25           else Hedge.add (m1,m2) newset
26         in
27           (doneset', to_check', newset')
28       else   (* keys are not available *)
29         (doneset,to_check,newset)
```

Listing 6: `check_analyse` for Encryptions.

First, we have to check whether both keys `(k1,k2)` are available in the synthesis of the

current knowledge (line 18). If this is the case then the decryption destructor can be applied by the environment. Hence, as above, we have to move both encryptions from `to_check` to `doneset` (line 20 to 21). If the messages `(m1,m2)` are not yet contained in the analysis, add them to `newset` (line 22 to 25).

If the keys are not both available in the current knowledge then the three hedges are not modified (line 29).

In the following, we show a simplified version of the case for arbitrary constructors. Loosely speaking, we have to check if there are destructors to be applied. In particular, we have to check if all required arguments of the destructor are in the synthesis. If destructors can be applied we add the values they return to the analysis, hence to `newset`.

We iterate over all available destructor rules (line 33 to 73) in order to find the destructors corresponding to constructor `c` (line 30). As the current knowledge `know` is invariant we compute it outside the iteration (line 31).

```
30  | Cons(c,es), Cons(c',fs) when c = c' ->
31      let know = Hedge.union to_check doneset in

33      Hashtbl.fold    (* iterate over all destructor rules *)
34      (fun destructor destructorinfo (doneset',to_check',newset') ->
35        let (address,constructorlist,matchlist) = destructorinfo in
36        let rec tryconstructors = function
37          [] -> (doneset', to_check', newset')
```

For each rule (consisting of `destructor`, `address`, `constructorlist`, `matchlist`, line 34 to 35) the function `tryconstructors` is called with the `constructorlist` (lines 36 and 71) containing information at which address certain constructors must appear. If the required constructor `c` cannot be found in this list (line 37) then the analysis cannot be extended, hence the hedges are not modified.

Now assume, the required constructor `cons` is found to appear as first argument of a destructor (line 38). We then have to check if all *required arguments* of the corresponding destructor are in the analysis. A required destructor argument contains at least one subexpression whose address has to occur in the `matchlist`, and exactly one component of this `matchlist` entry has to refer to a subexpression of a constructor argument. As an example, consider the destructor rule from page 16 (Example 3.1) again:

$$\mathsf{des}(\mathsf{cons1}(a,b,c),\mathsf{cons2}(a,d),b) \to c$$

The first required argument is `cons2` (with argument a) since the subexpression a occurs in the constructor arguments. This is encoded in the `destructorinfo` by the fact that the address of the subexpression a occurs in the `matchlist`, and the corresponding partner in the `matchlist` occurs as address inside the constructor arguments. The second required argument is b as it occurs directly in the constructor.

The necessary expressions that occur top-level, not nested in a constructor, (like the b above) are obtained by the folding (line 39 to 50).

```
38          | ([1],cons)::cl when c = cons ->
39              let (es',fs') = List.fold_left
40                (fun (es'',fs'') (addr_a,addr_b) ->
41                  match addr_a, addr_b with
42                    (1::addr_a'), [b] when (b>1) ->
43                      (get_subexp_at_address addr_a' es) :: es'',
44                      (get_subexp_at_address addr_a' fs) :: fs''
45                  | [a], (1::addr_b') when (a>1) ->
46                      (get_subexp_at_address addr_b' es) :: es'',
47                      (get_subexp_at_address addr_b' fs) :: fs''
48                  | _, _ -> es'', fs''
49                )
50                ([],[]) matchlist
```

We iterate over all pairs in the `matchlist`. Note that exactly one component must refer to a constructor argument, hence the address must begin with 1 to be an argument of the constructor which is the first destructor argument (lines 42, 45). In particular, as the addresses `addr_a` and `addr_b` refer to destructor arguments while we search the constructor arguments `es` and `fs`, we must omit the leading 1 by using `addr_a'` and `addr_b'` (lines 43,44 and 46,47).

The top-level arguments to be checked are now contained in `(es',fs')` (line 39). We still need the required constructors. These are added to `(es',fs')` by the procedure `get_required_constructors` (lines 51 and 52).

```
51              in let (es',fs') = get_required_constructors (es',fs')
52                                 constructorlist matchlist c (es,fs)
```

The latter iterates over the constructors in `constructorlist` and does the following:

- for each constructor `c`: all addresses in the `matchlist` whose second component in the corresponding `matchlist` pair refers to a constructor argument are collected
- from these addresses, two constructors are built using the corresponding expressions from `(es,fs)`.
- the constructor pair is added to `(es',fs')`.

The procedure `hedge_is_in_synth_h` (line 54) checks whether all expression pairs built from `es'` and `fs'` are in the synthesis of the current knowledge `know` (line 31).

```
53              in
54              if hedge_is_in_synth_h (es',fs')
55              then
56                let to_check'' = Hedge.remove (e1,e2) to_check' in
57                let doneset'' = Hedge.add (e1,e2) doneset' in
58                let newset'' =
59                  let 1::consaddr = address in
60                  let retval = (get_subexp_at_address consaddr es,
61                                get_subexp_at_address consaddr fs) in
62                  if not (is_in_synth_h retval know)
```

```
63                    then Hedge.add retval newset'
64                    else newset'
65                in
66                  (doneset'', to_check'', newset'')
```

If this check succeeds, as seen before, the current expression pair `(e1,e2)` is marked as processed by moving it from `to_check` to `doneset` (lines 56 and 57). If not yet available, the expression to which the destructor evaluates is added to the analysis, hence to `newset` (line 58 to 64). Note that the first part of the address has to be truncated (line 59) since `address` refers to the position in the destructor arguments.

If not all arguments are available in the synthesis the destructor cannot be applied, hence the hedges are returned without modifications (line 68).

```
67              else
68                (doneset',to_check',newset')
```

If in the matching of line 36 a constructor in the `constructorlist` for the current destructor does not appear as first argument of the destructor we jump over it (line 69).

```
69          | _::cl -> tryconstructors cl
70        in
71          tryconstructors constructorlist
72      )
73      (destructors#table()) (doneset,to_check,newset)
```

The last case in the matching of line 2 covers everything else, hence destructors or pairs of different constructors (line 74). In these cases no analysis is possible.

```
74    | _ -> (doneset,to_check,newset)
```

3.4 Tool Assessment

The extension of SBC is not yet completed. There are still some functions and cases not extended. However, the most relevant procedures are extended so that we are able to provide a number of running examples. The following sections give an overview of the current state of the tool.

3.4.1 Examples

Our test suite contains about 30 examples which were not expressible with the original SBC as they contain rewrite rules not available in the restricted Spi Calculus [3]. Our extended version of the tool yields correct results for these examples. Some of them are

described in the following. We provide insights into the functionality of hedge analysis, synthesis and irreducibles.

Example 3.2. This example shows a processing for hash functions (i.e. functions which have no rewrite rules). Even if the structure of the message being hashed is different, no environment can detect this difference, as the basic ingredients (the fresh names a and b) are not available.

```
agent P(c) = (^a)'c<#hash(a)>
agent Q(c) = (^b)'c<#hash(<b,b>)>

sbisim P(c) Q(c)
```

After executing the output action `'c<·>` the hedge consists of the pairs (c, c) and $(\#hash(a), \#hash(\langle b, b\rangle))$ where $\langle\cdot,\cdot\rangle$ denotes the build-in pair constructor and `(^a)` a fresh name a. This hedge is already irreducible and is consistent. Our extended tool outputs the corresponding bisimulation.

Example 3.3. In contrast to the previous example, all names in the following specification are free. Thus, the environment may use them to reconstruct messages.

```
rule -symdec(#symenc(m,k),k) -> m

agent P(c,m,k,l) = 'c<#hash(#symenc(m,k))>
agent Q(c,m,k,l) = 'c<#hash(#symenc(m,l))>

sbisim P(c,m,k,l) Q(c,m,k,l)
```

The hedge contains not only the sent messages and the pair of channel names, but also the pairs (m, m), (k, k), and (l, l) as they are not restricted by the processes. As the message pair $(\#hash(\#symenc(m, k)), \#hash(\#symenc(m, l)))$ can be constructed by the environment, we have two non-identical pairs with equal left components in the synthesis of the hedge. Hence, consistency is violated.

Example 3.4. Consider the following specification of processes P and Q using *deterministic* asymmetric encryption:

```
rule -asymdec(#asymenc(m,#pk(k)),#sk(k)) -> m

agent P(c) = (^a)(^b)'c<#asymenc(a,#pk(a))>.'c<<a,a>>
agent Q(c) = (^a)(^b)'c<#asymenc(a,#pk(b))>.'c<<b,b>>

sbisim P(c) Q(c)
```

We consider the hedge after the first send operation in the processes P and Q which consists of only two tuples (c, c) and $(\#symenc(a, \#pk(a)), \#symenc(a, \#pk(b)))$. As the names a and b are not known to the environment, the ciphertexts cannot be decrypted. However, after the execution of the second send operation the environment gains the pair

$(\langle a, a\rangle, \langle b, b\rangle)$. After applying irreducibles, this pair gets replaced by (a, b), as this suffices to reconstruct the original pairs.

Additionally, the irreducible hedge contains the pair (a, a) by a direct analysis step: when the operation tries to apply the destructor $-asymdec$ it checks whether the second argument can be constructed by the environment. As the pair (a, b) is available, the environment can also construct the secret keys $(\#sk(a), \#sk(b))$. Thus, we have the final hedge $\{(a, b), (a, a), (c, c)\}$ which is not consistent. Hence, the two processes are not bisimilar.

Example 3.5. In this example we show the tool's ability to find the right instantiations of input variables to distinguish two processes with different guard conditions. Additionally, it shows that the tool makes use of the knowledge gained during process execution.

```
agent P(c) = (^a)'c<a>.c(x).{[x=#hash1(a)]}'c<x>
agent Q(c) = (^a)'c<a>.c(x).{[x=#hash2(a)]}'c<x>

sbisim P(c) Q(c)
```

Both processes reveal their private symbol a and read on channel c afterwards. As c is a public channel, the environment can instantiate variable x as needed for distinguishing the processes. For example, #hash1(a) is a distinguishing instantiation, as the guard expression of P will succeed, whereas process Q gets stuck.

Our test suite contains examples that are significantly more complex. For the sake of simplicity we choose not to present them in detail here.

3.4.2 Limitations of the Tool

The approach chosen by Borgström and extended by us has some limitations. In this section we describe a false negative example whose incompleteness is caused by the approach.

For the following example, the original as well as the extended version of the tool fails:

```
agent P(c,x) = {[x=c]}'c<x> + 'c<x>
agent Q(c,x) = 'c<x>

sbisim P(c,x) Q(c,x)
```

The processes P and Q are clearly bisimilar as both of them can do only one thing, namely sending x over channel c — no matter whether the guard evaluates successfully or not. However, the tool considers them not to be equivalent since it does not find the correct matching partners. This example shows that the tool cannot handle branching over the choice operator correctly. Note that this error does not lead to unsound results.

4 Conclusions

After introducing message algebras and the extended version of Spi Calculus used by SBC, we recapitulated the important parts of Johannes Borgström's theory necessary for checking observational equivalence.

We have documented some relevant code fragments of the existing SBC tool. We have partially extended the tool to allow for checking processes defined in the Extended Spi Calculus with an arbitrary constructor-destructor language. For some procedures, we have documented the extensions and have given references to the theory.

Our extended tool is now able to handle a large class of examples that was not checkable before. Most of the examples are now processable by the extended SBC because of the increased expressiveness of freely specifiable rewrite rules.

Still missing are the extensions for arbitrary constructors and destructors in the procedures `Hedge.check_ast`, the application of destructors on names in `Formula.reduce_expression`, and the extensions for constructors in `Formula.q`.

4.1 Future Work

As mentioned above, there are still procedures that are not yet completely extended. In order to finish the extension, one would have to look very carefully at the indicated locations and invest more time to understand them. Testing how the tool behaves with bigger examples would be the most immediate improvement that could be brought. Further, it would be necessary to prove correctness of the tools algorithm. In the following we describe additional possible extensions.

Multiple rules per destructor. An idea which we want to take a closer look at is the specification of multiple rewrite rules per destructor. While experimenting with different rewrite systems, we encountered that the restriction to one rule makes it hard to specify some cryptographic primitives. For example a logical 'or' or some Zero Knowledge primitives can hardly be realized without the possibility for multiple rules:

$$\mathsf{or}(x, \mathsf{true}) \longrightarrow \mathsf{true}$$
$$\mathsf{or}(\mathsf{true}, x) \longrightarrow \mathsf{true}$$

However, we discovered some elementary difficulties with the implementation when we tried to drop this limitation. As an example, let us consider an equational theory with the above specified destructor or and the processes

$$\begin{aligned} P &= c(x).c(y).[\mathsf{or}(x, y) = \mathsf{true}]\,\overline{c}\langle a\rangle \\ Q &= c(x).c(y).[\mathsf{or}(x, y) = \mathsf{true}]\,\overline{c}\langle b\rangle \end{aligned}$$

The tool then needs the guard to evaluate to true in order to proceed with the search (and obtain an inconsistent hedge). Obviously, there are different instantiations that

lead to this effect. The problem is that currently the tool cannot branch over different instantiations; it fully relies on the uniqueness of the instantiations which holds under constructor-destructor languages with the restriction in question.

Yet not impossible to solve, we believe that fundamental changes in the program structure are necessary in order to handle this additional type of branching. Furthermore, the theoretical soundness of this extension would need to be preserved. Hence, we decided to stay with the restriction of one rule per destructor for the scope of this seminar and leave this point as a possible extension to the ideas we realized.

Replication operator. Borgström's theory does not cover replication. One would have to find a finite abstraction for the infinite number of replications.

Protocol phases. As an optimization, we could allow for specification of protocol phases as it is possible in ProVerif [8]. This would require to adapt the used syntax and semantics as well as the related theory and implementation.

Allowing a larger class of equational theories. An extension to the tool could be brought by using Baudet's NP algorithm for checking symbolic environment consistency. This would allow SBC to work with arbitrary subterm-convergent rewrite systems.

Extending completeness. So far there are false positive tests. It would be important to extend the completeness of the tool such that we obtain less or no false positives.

Negation in guards. Negation is not permitted in the semantics for Boolean guards. It would be helpful in order to specify branchings in the processes.

Private function symbols. The user can already specify private function symbols. However, it is still cumbersome to do this by hand and implementing a syntactic extension to deal with it would prove to be useful.

Zero Knowledge primitives. It is possible to implement Zero Knowledge primitives as a constructor-destructor language using a technique similar to the one introduced by Backes, Maffei, Unruh [5]. It would be beneficial to provide a syntactical extension supporting their handling. This extension would allow trying SBC on electronic voting protocols such as Civitas or DAA (Direct Anonymous Attestation).

4.2 Acknowledgments

We would like to thank Catalin Hriţcu and Matteo Maffei for the fruitful discussions we had during the semester and for giving us such an interesting and challenging topic.

We are also grateful to Johannes Borgström for providing helpful hints and suggestions as well as the source code of SBC.

References

[1] Martín Abadi and Véronique Cortier. Deciding Knowledge in Security Protocols under Equational Theories. *Theoretical Computer Science*, 367(1):2–32, 2006.

[2] Martín Abadi and Cédric Fournet. Mobile Values, New Names, and Secure Communication. *SIGPLAN Notices*, 36(3):104–115, 2001.

[3] Martín Abadi and Andrew D. Gordon. A Calculus for Cryptographic Protocols: The Spi Calculus. In *Fourth ACM Conference on Computer and Communications Security*, pages 36–47. ACM Press, 1997.

[4] Michael Backes, Cătălin Hriţcu, and Matteo Maffei. Automated Verification of Remote Electronic Voting Protocols in the Applied Pi-calculus. In *21th IEEE Symposium on Computer Security Foundations (CSF 2008)*, pages 195–209. IEEE Computer Society Press, June 2008.

[5] Michael Backes, Matteo Maffei, and Dominique Unruh. Zero-Knowledge in the Applied Pi-calculus and Automated Verification of the Direct Anonymous Attestation Protocol. In *IEEE Symposium on Security and Privacy, Proceedings of SSP'08*, pages 202–215, May 2008. Preprint on IACR ePrint 2007/289.

[6] Mathieu Baudet. *Sécurité des Protocoles Cryptographiques : Aspects Logiques et Calculatoires*. PhD thesis, École Normale Supérieure de Cachan, January 2007.

[7] Mathieu Baudet, Véronique Cortier, and Stéphanie Delaune. YAPA: A Generic Tool for Computing Intruder Knowledge. Research Report LSV-09-03, Laboratoire Spécification et Vérification, ENS Cachan, France, February 2009.

[8] Bruno Blanchet. Proverif. http://www.proverif.ens.fr, 2009.

[9] Bruno Blanchet, Martín Abadi, and Cédric Fournet. Automated Verification of Selected Equivalences for Security Protocols. *Journal of Logic and Algebraic Programming*, 75(1):3–51, February – March 2008.

[10] Johannes Börgstrom. Spi Bisimulation Checker. http://lamp.epfl.ch/ jobo/sbc, 2006.

[11] Johannes Borgström. *Equivalences and Calculi for Formal Verification of Cryptographic Protocols*. PhD thesis, École Polytechnique Fédérale de Lausanne, March 2008.

[12] Johannes Borgström, Sébastien Briais, and Uwe Nestmann. Symbolic Bisimulation in the Spi Calculus. In Philippa Gardner and Nobuko Yoshida, editors, *Proc. CONCUR 2004*, volume 3170 of *Lecture Notes in Computer Science*, pages 161–176. Springer, 2004.

[13] LSV Cachan. Security Protocols Online REpository. http://www.lsv.ens-cachan.fr/Software/spore, 2009.

[14] Ştefan Ciobâcă, Stéphanie Delaune, and Steve Kremer. Computing Knowledge in Security Protocols under Convergent Equational Theories. Research Report LSV-09-05, Laboratoire Spécification et Vérification, ENS Cachan, France, March 2009.

[15] Véronique Cortier and Stéphanie Delaune. A Method for Proving Observational Equivalence. Research report lsv-09-04, Ecole Normale Supérieur de Cachan, February 2009.

[16] Stéphanie Delaune, Steve Kremer, and Mark D. Ryan. Symbolic Bisimulation for the Applied Pi-Calculus. In V. Arvind and Sanjiva Prasad, editors, *Proceedings of the 27th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'07)*, volume 4855 of *Lecture Notes in Computer Science*, pages 133–145, New Delhi, India, December 2007. Springer.

[17] Anders Strandløv Elkjær, Michael Höhle, Hansel Hüttel, and Kasper Overgård Nielsen. Towards Automatic Bisimilarity Checking in the Spi Calculus. *Australian Computer Science Communications*, 21(3):175–189, 1999.

[18] Matthew Hennessy and Huimin Lin. Symbolic Bisimulations. *Theoretical Computer Science*, 138(2):353–389, 1995.

[19] Robin Milner, Joachim Parrow, and David Walker. A Calculus of Mobile Processes, Part I,II. *Information and Computation*, 100:1–77, September 1992.

[20] Robin Milner and Davide Sangiorgi. Barbed Bisimulation. In *ICALP '92: Proceedings of the 19th International Colloquium on Automata, Languages and Programming*, pages 685–695, London, UK, 1992. Springer-Verlag.