Proving More Observational Equivalences with ProVerif

Vincent Cheval and Bruno Blanchet

October 2012

Research report LSV-12-19

Laboratoire Spécification & Vérification

École Normale Supérieure de Cachan
61, avenue du Président Wilson
94235 Cachan Cedex France

Proving More Observational Equivalences with ProVerif

Vincent Cheval[1] and Bruno Blanchet[2]

[1] LSV, ENS Cachan & CNRS & INRIA Saclay le-de-France, France
[2] INRIA Paris-Rocquencourt, France

Abstract. This paper presents an extension of the automatic protocol verifier ProVerif in order to prove more observational equivalences. ProVerif can prove observational equivalence between processes that have the same structure but differ by the messages they contain. In order to extend the class of equivalences that ProVerif handles, we extend the language of terms by defining more functions (destructors) by rewrite rules. In particular, we allow rewrite rules with inequalities as side-conditions, so that we can express tests "if then else" inside terms. Finally, we provide an automatic procedure that translates a process into an equivalent process that performs as many actions as possible inside terms, to allow ProVerif to prove the desired equivalence. These extensions have been implemented in ProVerif and allow us to automatically prove anonymity in the private authentication protocol by Abadi and Fournet.

1 Introduction

Today, many applications that manipulate private data incorporate a cryptographic protocol, in order to ensure that such private information is never disclosed to anyone but the entitled entities. However, it has been shown that some currently used cryptographic protocols are flawed, *e.g.* the e-passport protocols [5]. It is therefore essential to obtain as much confidence as possible in the correctness of security protocols. To this effect, many works have relied on symbolic methods to precisely analyse the security of cryptographic protocols. In the case of a bounded number of sessions, secrecy preservation was shown to be co-NP-complete [21], and for an unbounded number of sessions, several decidable classes have been identified (e.g. [20]). Moreover, several tools have been developed to automatically verify security properties on cryptographic protocols, *e.g.* CSP/FDR [22], AVISPA [6], ProVerif [8], SCYTHER [14]. Until recently, most tools focused on reachability properties (or trace properties), such as authentication and secrecy, which specify that the protocols cannot reach a bad state. However, privacy-type properties cannot be naturally expressed as reachability property and require the notion of behavioural equivalence that specifies the indistinguishability of instances of the protocols. In the literature, the notion of *may-testing equivalence* was first introduced in [19] and has been studied for several calculi, *e.g.* spi-calculus [3,16]. Typically, two processes P and Q are may-testing equivalent if for any process O, the processes $P \mid O$ and $Q \mid O$ can both emit on the same channels. However, the high difficulty of deciding this equivalence led to the introduction of stronger equivalences such as *observational equivalence* that additionally checks the bisimilarity of the process P and Q. This notion was the focus of several works, *e.g.* [9,15]. In this paper, we focus on the automatisation of the proofs of observational equivalence.

Related Work. The automated verification of equivalence properties for security protocols was first considered for bounded number of sessions with a fixed set of basic primitives and without else branches [17,16], but their complexity was too large for practical implementations. [7] showed that diff-equivalence, a strong equivalence between processes that have the same structure but differ by the terms they contain, is also decidable for bounded processes without else-branches; this result applies in particular to the detection of off-line guessing attacks against password-based protocols and to the proof of strong secrecy. However, the procedure does not seem to be well-suited for an implementation. Recently, more practical algorithms were designed for bounded processes with else branches, non-determinism and a fixed set of primitives [11] but there is no available implementation. These techniques rely on a symbolic semantics: in a symbolic semantics, such as [10,15,18], the messages that come from the adversary are represented by variables, to avoid an unbounded case distinction on these messages.

To our knowledge, only three works resulted in automatic tools for the verification of equivalence properties: PROFVERIF [8], SPEC [23] and AKISS [12]. The tool SPEC provides a decision procedure for observational equivalence for processes in the spi-calculus. However, the scope is limited to bounded determinate processes without non-trivial else branches, that is, processes whose executions are entirely determined by the adversary inputs. The tool AKISS was developed to decide a weaker equivalence close to the may-testing equivalence for a wide variety of primitives. Nevertheless, the scope is also limited to bounded determinate processes without non-trivial else branches. At last, the tool PROVERIF was first a protocol analyser for trace properties but, since [9], it can also check the diff-equivalence between processes written in the applied pi calculus [1]. Although the diff-equivalence is stronger than the observational equivalence, this is the only tool that accepts unbounded processes with else branches and any cryptographic primitives that can be represented by an equational theory and/or rewrite rules. Even if PROVERIF does not always terminate, it was shown very efficient for many case studies (e.g. proving the absence of guessing attacks in EKE, proving the core security of JFK [9] or proving anonymity and unlinkability of the *Active Authentication protocol* from the *electronic passport protocol* [4]). Hence the present paper focuses on the tool PROVERIF.

Motivation. Since the notion of equivalence used by PROVERIF is stronger than the observational equivalence, it may yield some false attacks on the protocols. Indeed, PROVERIF focuses on proving equivalences $P \approx Q$ in which P and Q are two variants of the same process obtained by selecting different terms for P and Q. Moreover, PROVERIF requires that all tests yield the same result in both processes, in particular the tests of conditional branchings. Thus, for a protocol that does not satisfy this condition, PROVERIF will fail to prove equivalence. Unfortunately, many indistinguishable processes do not satisfy this condition. Consider for example the two naives processes:

$$P \stackrel{def}{=} c(x).\mathsf{if}\ x = \mathsf{pk}(sk_A)\ \mathsf{then}\ \overline{c}\langle\{s\}_{\mathsf{pk}(sk_A)}\rangle\ \mathsf{else}\ \overline{c}\langle\{N_p\}_{\mathsf{pk}(sk_A)}\rangle$$
$$Q \stackrel{def}{=} c(x).\mathsf{if}\ x = \mathsf{pk}(sk_B)\ \mathsf{then}\ \overline{c}\langle\{s\}_{\mathsf{pk}(sk_B)}\rangle\ \mathsf{else}\ \overline{c}\langle\{N_q\}_{\mathsf{pk}(sk_B)}\rangle$$

For simplicity, we omitted the name restriction but we assume that all names but c are private. The protocol P is simply waiting for the public key of the agent A ($\mathsf{pk}(sk_A)$)

on a channel c, and if P receives it then he sends some secret s encrypted with A's public key else a fresh nonce N_p encrypted with A's public key is sent on the channel c. On the other hand, the protocol Q does similar actions but is waiting for the public key of the agent B ($\mathsf{pk}(sk_B)$) instead of A. Assuming that the attacker does not have access to the private keys of A and B, then the two protocols are equivalent since the attacker cannot differentiate $\{s\}_{\mathsf{pk}(sk_A)}$, $\{N_p\}_{\mathsf{pk}(sk_A)}$, $\{s\}_{\mathsf{pk}(sk_B)}$ and $\{N_q\}_{\mathsf{pk}(sk_B)}$.

However, if the intruder sends the public key of the agent A ($\mathsf{pk}(sk_A)$), then the test of the conditional branching in P will succeed ($\mathsf{pk}(sk_A) = \mathsf{pk}(sk_A)$) whereas the test of the same conditional branching in Q will fail ($\mathsf{pk}(sk_A) \neq \mathsf{pk}(sk_B)$). Since this test does not yield the same result in both processes, PROVERIF will fail to prove the equivalence between P and Q. More realistic examples illustrating this false attack can be found in several cases studies, e.g. the *private authentication protocol* [2] and the *e-passport protocol* [5].

Our Contribution. Our main contribution consists in addressing the issue of false attacks due to conditional branchings. In particular, we allow function symbols defined by rewrite rules with inequalities as side-conditions, so that we can express tests of conditional branchings directly inside terms (Section 2), and we show how the original Horn clauses based algorithm of PROVERIF can be adapted to our new calculus (Sections 3 and 4). Moreover, we provide an automatic procedure that transforms a process into an equivalent process that contains as few conditional branchings as possible which allows ProVerif to prove equivalence on a larger class of processes. In particular, the implementation of our extension in ProVerif allowed us to automatically prove anonymity of the private authentication protocol for an unbounded number of sessions (Section 5). Our implementation is available as ProVerif 1.87beta, at `http://proverif.inria.fr`

2 Model

This section introduces our process calculus, by giving its syntax and semantics. As mentioned above, our work extends the behaviour of destructor symbols, so our syntax and semantics of terms change in comparison to the original calculus of PROVERIF [9]. However, we did not modify the syntax of processes thus the semantics of processes only differs from the changes coming from the modifications in the semantics of terms.

2.1 Syntax

The syntax of our calculus is summarised in Fig. 1. The messages sent on the network by agents in a protocol are modelled using an abstract term algebra. We assume an infinite set of names $\mathcal{N}$ and an infinite set of variables $\mathcal{X}$. We also consider a signature Σ consisting of a finite set of function symbols with their arity. We distinguish two categories of function symbols: constructors f and destructors g. Constructors build terms; destructors, defined by rewrite rules, manipulate terms, as detailed below. We denote by h a constructor or a destructor. *Messages* M are terms built from variables, names, and constructors applied to terms.

$M ::=$	message
x, y, z	variables
a, b, c	names
$\mathsf{f}(M_1, \ldots, M_n)$	constructor application
$U ::=$	may-fail message
M	message
fail	failure
u	may-fail variable
$D ::=$	term evaluation
U	may-fail message
$\text{eval } \mathsf{h}(D_1, \ldots, D_n)$	function evaluation
$P, Q, R ::=$	processes
0	nil
$M(x).P$	input
$\overline{M}\langle N\rangle.P$	output
$P \mid Q$	parallel composition
$!\,P$	replication
$(\nu a).P$	restriction
$\mathsf{let}\ x = D\ \mathsf{in}\ P\ \mathsf{else}\ Q$	term evaluation

Fig. 1. Syntax of terms and processes

We define an *equational theory* by a finite set of equations $M = N$, where M, N are terms without names. The equational theory is then obtained from these equations by reflexive, symmetric, and transitive closure, closure under application of function symbols, and closure under substitution of terms for variables. By identifying an equational theory with its signature Σ, we denote $M =_\Sigma N$ an equality modulo the equational theory, and $M \neq_\Sigma N$ an inequality modulo the equational theory. We write $M = N$ and $M \neq N$ for syntactic equality and inequality, respectively. In this paper, we only consider consistent equational theories, i.e. there exist terms M and N such that $M \neq_\Sigma N$.

Destructors In [9], the rewrite rules describing the behaviour of destructors follow the usual definition of a rewrite rule. However, as previously mentioned, we want to introduce tests directly into terms and more specifically into the definition of destructors. Hence, we introduce *formulas* on messages in order to express these tests. We consider formulas ϕ of the form $\bigwedge_{i=1}^{n} \forall \tilde{x}_i.M_i \neq_\Sigma N_i$, where $\tilde{x}$ stands for a sequence of variables $x_1, \ldots, x_k$. We denote by $\top$ and $\bot$ the *true* and *false* formulas, respectively corresponding to an empty conjunction ($n = 0$) and to $x \neq_\Sigma x$, for instance. Formulas will be used as side conditions for destructors. We denote by $fv(\phi)$ the free variables of ϕ, *i.e.* the variables that are not universally quantified. Let σ be a substitution mapping variables to ground terms. We define $\sigma \vDash \phi$ as follows: $\sigma \vDash \bigwedge_{i=1}^{n} \forall \tilde{x}_i.M_i \neq_\Sigma N_i$ if and only if for $i = 1, \ldots n$, for all σ_i of domain $\tilde{x}_i$, $\sigma\sigma_i M_i \neq_\Sigma \sigma\sigma_i N_i$.

In [9], destructors are partial functions defined by rewrite rules; when no rewrite rule can be applied, we say that the destructor fails. However, this formalism does not allow

destructors to succeed when one of their arguments fails. We shall need this feature in order to include as many tests as possible in terms. Therefore, we extend the definition of destructors by defining *may-fail messages*, denoted by U, which can be messages M, the special value fail, or a variable u. We separate fail from ordinary messages M so that the equational theory does not apply to fail. May-fail messages represent the possible arguments and result of a destructor. We differentiate variables for may-fail messages, denoted u, v, w from variables for messages, denoted x, y, z. A may-fail variable u can be instantiated by a may-fail term while a message variable x can only be instantiated by a message, and so cannot be instantiated by fail.

For two ground may-fail messages U_1 and U_2, we say that $U_1 =_\Sigma U_2$ if and only if $U_1 = U_2 = \mathsf{fail}$ or U_1, U_2 are both messages, denoted M_1, M_2, and $M_1 =_\Sigma M_2$. Given a signature Σ, a destructor g of arity n is defined by a finite set of rewrite rules $\mathsf{g}(U_1, \ldots, U_n) \to U \mid\mid \phi$ where $U_1, \ldots, U_n, U$ are may-fail messages that do not contain any name, ϕ is a formula as defined above that does not contain any name, and the variables of U and $fv(\phi)$ are bound in $U_1, \ldots, U_n$. Note that all variables in $fv(\phi)$ are necessarily message variables. Variables are subject to renaming. We omit the formula ϕ when it is $\top$. We denote $\mathrm{def}_\Sigma(\mathsf{g})$ the set of rewrite rules describing g in the signature Σ.

Example 1. Consider a symmetric encryption scheme where the decryption function either properly decrypts a ciphertext using the correct private key, or fails. To model this encryption scheme, we consider, in a signature Σ, the constructor senc for encryption, the destructor sdec for decryption and the following rewrite rules:

- $\mathsf{sdec}(\mathsf{senc}(x, y), y) \to x$ (decryption succeeds)
- $\mathsf{sdec}(x, y) \to \mathsf{fail} \mid\mid \forall z.x \neq_\Sigma \mathsf{senc}(z, y)$ (decryption fails, because x is not a ciphertext under the correct key)
- $\mathsf{sdec}(\mathsf{fail}, u) \to \mathsf{fail}$, $\mathsf{sdec}(u, \mathsf{fail}) \to \mathsf{fail}$ (the arguments failed, the decryption also fails)

Consider $U_1, \ldots, U_n$ may-fail messages and consider g a destructor of arity n. We say that g rewrites $U_1, \ldots, U_n$ into U, denoted $\mathsf{g}(U_1, \ldots, U_n) \to U$, if there exist $\mathsf{g}(U'_1, \ldots, U'_n) \to U' \mid\mid \phi$ in $\mathrm{def}_\Sigma(\mathsf{g})$, and a substitution σ such that $\sigma U'_i =_\Sigma U_i$ for all $i = 1 \ldots n$, $\sigma U' = U$ and $\sigma \vDash \phi$. At last, we ask that given a signature Σ, for all destructors g of arity n, $\mathrm{def}_\Sigma(\mathsf{g})$ satisfies the following properties:

P1. For all ground may-fail messages $U_1, \ldots, U_n$, there exists a may-fail message U such that $\mathsf{g}(U_1, \ldots, U_n) \to U$.

P2. For all ground may-fail messages $U_1, \ldots, U_n, V_1, V_2$, if $\mathsf{g}(U_1, \ldots, U_n) \to V_1$ and $\mathsf{g}(U_1, \ldots, U_n) \to V_2$ then $V_1 =_\Sigma V_2$.

Property P1 expresses that all destructors are total while Property P2 expresses that all destructors are deterministic (modulo the equational theory). Note that thanks to Property P2, a destructor cannot reduce to fail and a message at the same time.

In Example 1, the destructor sdec follows the classical definition of the symmetric decryption. However, thanks to the formulas and the fact that the arguments of a destructor can fail, we can describe the behaviour of new primitives.

Example 2. We define a destructor that tests equality and returns a boolean as follows:

$$\mathsf{eq}(x,x) \to \mathsf{true} \qquad \mathsf{eq}(x,y) \to \mathsf{false} \mid\mid x \neq_\Sigma y$$
$$\mathsf{eq}(\mathsf{fail},u) \to \mathsf{fail} \qquad \mathsf{eq}(u,\mathsf{fail}) \to \mathsf{fail}$$

This destructor fails when one of its arguments fails. Such a destructor could not be defined in PROVERIF without our extension, because one could not test $x \neq_\Sigma y$.

From Usual Destructors to our Extension More generally, from a destructor defined, as in [9], by rewrite rules $\mathsf{g}(M_1,\ldots,M_n) \to M$ without side conditions and such that the destructor is considered to fail when no rewrite rule applies, we can build a destructor in our formalism. The algorithm is given in Lemma 1 below (proof in Appendix A).

Lemma 1. *Consider a signature Σ. Let* g *be a destructor of arity n described by the set of rewrite rules $\mathcal{S} = \{\mathsf{g}(M_1^i,\ldots,M_n^i) \to M^i \mid i = 1,\ldots,m\}$. Assume that* g *is deterministic,* i.e. *$\mathcal{S}$ satisfies Property P2. The following set $\mathrm{def}_\Sigma(\mathsf{g})$ satisfies Properties P1 and P2:*

$$\begin{aligned}\mathrm{def}_\Sigma(\mathsf{g}) = \mathcal{S} \cup \{&\mathsf{g}(x_1,\ldots,x_n) \to \mathsf{fail} \mid\mid \phi\}\\ \cup\{&\mathsf{g}(u_1,\ldots,u_{k-1},\mathsf{fail},u_{k+1},\ldots,u_n) \to \mathsf{fail} \mid k = 1,\ldots,n\}\end{aligned}$$

where $\phi = \bigwedge_{i=1}^m \forall \tilde{y}_i.(x_1,\ldots,x_n) \neq_\Sigma (M_1^i,\ldots,M_n^i)$ and $\tilde{y}_i$ are the variables of $(M_1^i,\ldots,M_n^i)$, and $x_1,\ldots,x_n$ are message variables.

The users can therefore continue defining destructors as before in PROVERIF; the tool checks that the destructors are deterministic and automatically completes the definition following Lemma 1.

Generation of Deterministic and Total Destructors With our extension, we want the users to be able to define destructors with side conditions. However, these destructors must satisfy Properties P1 and P2. Instead of having to verify these properties a posteriori, we use a method that allows the user to provide precisely the destructors that satisfy P1 and P2: the user inputs a sequence of rewrite rules $\mathsf{g}(U_1^1,\ldots,U_n^1) \to V^1$ otherwise $\ldots$ otherwise $\mathsf{g}(U_n^m,\ldots,U_n^m) \to V^m$ where U_k^i, V^i are may-fail messages, for all i,k. Intuitively, this sequence indicates that when reducing terms by the destructor g, we try to apply the rewrite rules in the order of the sequence, and if no rule is applicable then the destructors fails. To model the case where no rule is applicable, we add the rewrite rule $\mathsf{g}(u_1,\ldots,u_n) \to \mathsf{fail}$ where $u_1,\ldots,u_n$ are distinct may-fail variables, at the end of the previous sequence of rules. Then, the obtained sequence is translated into a set $\mathcal{S}$ of rewrite rules with side conditions as follows

$$\mathcal{S} \stackrel{def}{=} \left\{\mathsf{g}(U_1^i,\ldots,U_n^i) \to V^i \mid\mid \textstyle\bigwedge_{j<i} \forall \tilde{u}^j.(U_1^i,\ldots,U_n^i) \neq_\Sigma (U_1^j,\ldots,U_n^j)\right\}_{i=1..m+1}$$

where $\tilde{u}^j$ are the variables of $U_1^j,\ldots,U_n^j$. We use side-conditions to make sure that rule i is not applied if rule j for $j < i$ can be applied. Notice that in the set $\mathcal{S}$ defined above, the formulas may contain may-fail variables or the constant fail. In order to match our formalism, we instantiate these variables by either a message variable or fail, and then we simplify the formulas.

Term Evaluation A *term evaluation* represents the evaluation of a series of constructors and destructors. The term evaluation $\mathrm{eval}\ \mathsf{h}(D_1, \ldots, D_n)$ indicates that the function symbol h will be evaluated. While all destructors must be preceded by eval, some constructors might also be preceded by eval in a term evaluation. In fact, the reader may ignore the prefix eval since $\mathrm{eval}\ \mathsf{h}$ and h have the same semantics with the initial definition of constructors with equations. However, eval becomes useful when we convert equations into rewrite rules (see Section 4.1). Typically, eval is used to indicate when a term has been evaluated or not. Even though we allow may-fail messages in term evaluations, since no construct binds may-fail variables in processes, only messages M and fail may in fact occur. In order to avoid distinguishing constructors and destructors in the definition of term evaluation, for f a constructor of arity n, we let $\mathrm{def}_\Sigma(\mathsf{f}) = \{\mathsf{f}(x_1, \ldots, x_n) \to \mathsf{f}(x_1, \ldots, x_n)\} \cup \{\mathsf{f}(u_1, \ldots, u_{i-1}, \mathsf{fail}, u_{i+1}, \ldots, u_n) \to \mathsf{fail} \mid i = 1, \ldots, n\}$. The second part of the union corresponds to the failure cases: the constructor fails if and only if one of its arguments fails.

Processes At last, the syntax of processes corresponds exactly to [9]. A trailing 0 can be omitted after an input or an output. An else branch can be omitted when it is $\mathsf{else}\ 0$.

Even if the condition $\mathsf{if}\ M = N\ \mathsf{then}\ P\ \mathsf{else}\ Q$ is not included in our calculus, it can be defined as $\mathsf{let}\ x = \mathsf{equals}(M, N)\ \mathsf{in}\ P\ \mathsf{else}\ Q$, where x is a fresh variable and equals is a binary destructor with the rewrite rules $\{\mathsf{equals}(x, x) \to x, \mathsf{equals}(x, y) \to \mathsf{fail} \mid\mid x \neq_\Sigma y, \mathsf{equals}(\mathsf{fail}, u) \to \mathsf{fail}, \mathsf{equals}(u, \mathsf{fail}) \to \mathsf{fail}\}$. The destructor equals succeeds if and only if its two arguments are equal messages modulo the equational theory and different from fail. We always include this destructor in the signature Σ. An evaluation context C is a closed context built from $[\,]$, $C \mid P$, $P \mid C$, and $(\nu a)C$.

Example 3. We consider a slightly simplified version of the private authentication protocol given in [2]. In this protocol, a participant A is willing to engage in communication and reveal its identity to a participant B, without revealing it to other participants. The cryptographic primitives used in this protocol are the asymmetric encryption and pairing. Expressed in ProVerif syntax, the participants A and B proceed as follows:

$$A(sk_a, sk_b) \stackrel{def}{=} (\nu n_a)\overline{c}\langle \mathsf{aenc}(\langle n_a, \mathsf{pk}(sk_a)\rangle, \mathsf{pk}(sk_b))\rangle.c(x).0$$

$$\begin{aligned} B(sk_b, sk_a) \stackrel{def}{=}\ & (\nu n_b)c(y).\mathsf{let}\ x = \mathsf{adec}(y, sk_b)\ \mathsf{in} \\ & \quad \mathsf{let}\ xn_a = \mathsf{proj}_1(x)\ \mathsf{in} \\ & \quad\quad \mathsf{let}\ z = \mathsf{equals}(\mathsf{proj}_2(x), \mathsf{pk}(sk_a))\ \mathsf{in} \\ & \quad\quad\quad \overline{c}\langle \mathsf{aenc}(\langle xn_a, \langle n_b, \mathsf{pk}(sk_b)\rangle\rangle, \mathsf{pk}(sk_a))\rangle.0 \\ & \quad\quad \mathsf{else}\ \overline{c}\langle \mathsf{aenc}(n_b, \mathsf{pk}(sk_b))\rangle.0 \\ & \quad \mathsf{else}\ \overline{c}\langle \mathsf{aenc}(n_b, \mathsf{pk}(sk_b))\rangle.0 \\ & \mathsf{else}\ \overline{c}\langle \mathsf{senc}(n_b, \mathsf{pk}(sk_b))\rangle.0 \end{aligned}$$

$$System(sk_a, sk_b) \stackrel{def}{=} A(sk_a, sk_b) \mid B(sk_b, sk_a)$$

where sk_a and sk_b represent the respective private keys of A and B, proj_1 and proj_2 are the two projections of a pairing denoted by $\langle\ ,\ \rangle$, aenc and adec represent the asymmetric encryption and decryption, and pk represents the public key associated to a private key.

$U \Downarrow_\Sigma U$
eval $\mathsf{h}(D_1, \ldots, D_n) \Downarrow_\Sigma \sigma U$
if $\mathsf{h}(U_1, \ldots, U_n) \to U \mid\mid \phi$ in $\mathrm{def}_\Sigma(\mathsf{h})$, and σ is such that for all i, $D_i \Downarrow_\Sigma V_i$, $V_i =_\Sigma \sigma U_i$ and $\sigma \vDash \phi$

$$\overline{N}\langle M\rangle.Q \mid N'(x).P \;\to_\Sigma\; Q \mid P\{^M/_x\} \qquad \text{if } N =_\Sigma N' \quad \text{(Red I/O)}$$

$$\mathsf{let}\ x = D\ \mathsf{in}\ P\ \mathsf{else}\ Q \to_\Sigma P\{^M/_x\} \qquad \text{if } D \Downarrow_\Sigma M \quad \text{(Red Fun 1)}$$

$$\mathsf{let}\ x = D\ \mathsf{in}\ P\ \mathsf{else}\ Q \to_\Sigma Q \qquad \text{if } D \Downarrow_\Sigma \mathsf{fail} \quad \text{(Red Fun 2)}$$

$$!P \;\to_\Sigma\; P \mid !P \qquad \text{(Red Repl)}$$

$$P \;\to_\Sigma\; Q \;\Rightarrow\; P \mid R \;\to_\Sigma\; Q \mid R \qquad \text{(Red Par)}$$

$$P \;\to_\Sigma\; Q \;\Rightarrow\; \nu a.P \;\to_\Sigma\; \nu a.Q \qquad \text{(Red Res)}$$

$$P' \equiv P,\ P \;\to_\Sigma\; Q,\ Q \equiv Q' \;\Rightarrow\; P' \;\to_\Sigma\; Q' \qquad \text{(Red } \equiv\text{)}$$

Fig. 2. Semantics of terms and processes

In other words, A first sends to B a nonce n_a and its own public key $\mathsf{pk}(sk_a)$ encrypted with the public key of B, $\mathsf{pk}(sk_b)$. Then, after receiving the message, B first checks that the message is of the correct form and that it indeed contains the public key of A. If so, then B sends back to A the "correct" message composed of the nonce n_a he received, n_b a freshly generated nonce and his own public key ($\mathsf{pk}(sk_b)$), all this encrypted with the public key of A. Otherwise, B sends back a "dummy" message, $\mathsf{aenc}(n_b, \mathsf{pk}(sk_b))$. From the point of view of the attacker, this dummy message is indistinguishable from the "correct" one since the private keys sk_a and sk_b are unknown to the attacker, so the attacker should not be able to tell whether A or another is talking to B. This is what we are going to prove formally.

2.2 Semantics

The semantics for processes and term evaluations is summarised in Fig. 2. The formula $D \Downarrow_\Sigma U$ means that D evaluates to U. When the term evaluation corresponds to a function h preceded by eval, the evaluation proceeds recursively by evaluating the arguments of the function and then by applying the rewrite rules of h in $\mathrm{def}_\Sigma(\mathsf{h})$ to compute U, taking into account the side-conditions in ϕ.

The semantics for processes in ProVerif is defined by a *structural equivalence*, denoted $\equiv$ and some *internal reductions*. The structural equivalence $\equiv$ is the smallest equivalence relation on extended processes that is closed under α-conversion of names and variables, by application of evaluation contexts, and satisfying some further basic structural rules such as $P \mid 0 \equiv P$, associativity and commutativity of $\mid$ and scope extrusion. However, this structural equivalence does not substitute terms equal modulo the equational theory and does not model the replication. Both properties are in fact modelled as internal reduction rules for processes. This semantics is different from [9] by the rule (Red Fun 2) which previously corresponded to the case where the evaluation term D could not be reduced whereas D is reduced to fail in our semantics.

Both relations $\equiv$ and $\to_\Sigma$ are defined only on closed processes. Furthermore, we denote $\to_\Sigma^*$ the reflexive and transitive closure of $\to_\Sigma$. At last, we denote $\to_\Sigma^* \equiv$ for its union with $\equiv$. When Σ is clear from the context, we abbreviate $\to_\Sigma$ to $\to$ and $\Downarrow_\Sigma$ to $\Downarrow$.

$\overline{N}\langle M\rangle.Q \mid N'(x).P \;\rightarrow\; Q \mid P\{^{M}/_{x}\}$ (Red I/O)
if $\mathsf{fst}(N) =_\Sigma \mathsf{fst}(N')$ and $\mathsf{snd}(N) =_\Sigma \mathsf{snd}(N')$

$\mathsf{let}\ x = D\ \mathsf{in}\ P\ \mathsf{else}\ Q \rightarrow P\{^{\mathsf{diff}[M_1,M_2]}/_{x}\}$ (Red Fun 1)
if $\mathsf{fst}(D) \Downarrow_\Sigma M_1$ and $\mathsf{snd}(D) \Downarrow_\Sigma M_2$

$\mathsf{let}\ x = D\ \mathsf{in}\ P\ \mathsf{else}\ Q \rightarrow Q$ (Red Fun 2)
if $\mathsf{fst}(D) \Downarrow_\Sigma \mathsf{fail}$ and $\mathsf{snd}(D) \Downarrow_\Sigma \mathsf{fail}$

Fig. 3. Generalized rules for biprocesses

3 Using Biprocesses to Prove Observational Equivalence

In this section, we recall the notions of observational equivalence and biprocesses introduced in [9].

Definition 1. *The process P* emits on *M ($P\downarrow_M$) if and only if $P \rightarrow^*_\Sigma \equiv C[\overline{M'}\langle N\rangle.R]$ for some evaluation context C that does not bind $fn(M)$ and $M =_\Sigma M'$.*

The observational equivalence*, denoted $\approx$ is the largest symmetric relation $\mathcal{R}$ between closed processes with the same domain such that $P\ \mathcal{R}\ Q$ implies:*

1. *if $P\downarrow_M$, then $Q\downarrow_M$;*
2. *if $P \rightarrow^*_\Sigma P'$, then $Q \rightarrow^*_\Sigma Q'$ and $P'\ \mathcal{R}\ Q'$ for some Q';*
3. *$C[P]\ \mathcal{R}\ C[Q]$ for all closed evaluation contexts C.*

Intuitively, an evaluation context may represent an adversary, and two processes are observationally equivalent when no adversary can distinguish them. One of the most difficult parts of deciding the observational equivalence between two processes directly comes from the second item of Definition 1. Indeed, this condition indicates that each reduction of a process has to be matched in the second process. However, we consider a process algebra with replication, hence there are usually an infinite number of candidates for this mapping.

To solve this problem, [9] introduces a calculus that represents pairs of processes, called *biprocesses*, that have the same structure and differ only by the terms and term evaluations that they contain. The grammar of the calculus is a simple extension of the grammar of Figure 1 with additional cases so that $\mathsf{diff}[M, M']$ is a term and $\mathsf{diff}[D, D']$ is a term evaluation.

Given a biprocess P, we define two processes $\mathsf{fst}(P)$ and $\mathsf{snd}(P)$, as follows: $\mathsf{fst}(P)$ is obtained by replacing all occurrences of $\mathsf{diff}[M, M']$ with M and $\mathsf{diff}[D, D']$ with D in P, and similarly, $\mathsf{snd}(P)$ is obtained by replacing $\mathsf{diff}[M, M']$ with M' and $\mathsf{diff}[D, D']$ with D' in P. We define $\mathsf{fst}(D)$, $\mathsf{fst}(M)$, $\mathsf{snd}(D)$, and $\mathsf{snd}(M)$ similarly. A process or context is said to be *plain* when it does not contain diff.

Definition 2. *Let P be a closed biprocess. We say that P* satisfies observational equivalence *when $\mathsf{fst}(P) \approx \mathsf{snd}(P)$.*

The semantics of biprocesses is defined as in Figure 2 with generalized rules (Red I/O), (Red Fun 1), and (Red Fun 2) given in Figure 3.

The semantics of biprocesses is such that a biprocess reduces if and only if both sides of the biprocess reduce in the same way: a communication succeeds on both sides;

a term evaluation succeeds on both sides or fails on both sides. When the two sides of the biprocess reduce in different ways, the biprocess blocks. The following lemma shows that, when both sides of a biprocess always reduce in the same way, then that biprocess satisfies observational equivalence (proof in Appendix B).

Lemma 2. *Let P_0 be a closed biprocess. Suppose that, for all plain evaluation contexts C, all evaluation contexts C', and all reductions $C[P_0] \rightarrow^* P$,*

1. *if $P \equiv C'[\overline{N}\langle M\rangle.Q \mid N'(x).R]$ then $\mathsf{fst}(N) =_\Sigma \mathsf{fst}(N')$ if and only if $\mathsf{snd}(N) =_\Sigma \mathsf{snd}(N')$;*
2. *if $P \equiv C'[\mathsf{let}\ x = D\ \mathsf{in}\ Q\ \mathsf{else}\ R]$ then $\mathsf{fst}(D) \Downarrow_\Sigma \mathsf{fail}$ if and only if $\mathsf{snd}(D) \Downarrow_\Sigma \mathsf{fail}$.*

Then P_0 satisfies observational equivalence.

Intuitively, the semantics for biprocesses forces that each reduction of a process has to be matched by the same reduction in the second process. Hence, verifying the second item of Definition 1 becomes less problematic since we reduce to one the number of possible candidates Q'.

Example 4. Coming back to the private authentication protocol detailed in Example 3, we want to verify the anonymity of the participant A. Intuitively, this protocol preserves anonymity if an attacker cannot distinguish whether B is talking to A or to another participant A', assuming that A, A', and B are honest participants and furthermore assuming that the intruder knows the public keys of A, A', and B. Hence, the anonymity property is modelled by an observational equivalence between two instances of the protocol: one where B is talking to A and the other where B is talking to A', which is modelled as follows:

$$(\nu sk_a)(\nu sk'_a)(\nu sk_b)\overline{c}\langle \mathsf{pk}(sk_a), \mathsf{pk}(sk'_a), \mathsf{pk}(sk_b)\rangle.System(sk_a, sk_b)$$
$$\approx$$
$$(\nu sk_a)(\nu sk'_a)(\nu sk_b)\overline{c}\langle \mathsf{pk}(sk_a), \mathsf{pk}(sk'_a), \mathsf{pk}(sk_b)\rangle.System(sk'_a, sk_b)$$

Thanks to the fact that the "dummy" message and the "correct" one are indistinguishable from the point of view of the attacker, this equivalence holds. To prove this equivalence using ProVerif, we first have to transform this equivalence into a biprocess. This is easily done since only the private keys ska and ska' change between the two processes. Hence, we define the biprocess P_0 as follows:

$$(\nu sk_a)(\nu sk'_a)(\nu sk_b)\overline{c}\langle \mathsf{pk}(sk_a), \mathsf{pk}(sk'_a), \mathsf{pk}(sk_b)\rangle.System(\mathsf{diff}[sk_a, sk'_a], sk_b)$$

Note that $\mathsf{fst}(P_0)$ and $\mathsf{snd}(P_0)$ correspond to the two protocols of the equivalence. For simplicity, we only consider in this example two sessions but our results also apply to the anonymity for an unbounded number of sessions (for the definition of [5]).

4 Clause Generation

In [9], observational equivalence is verified by translating the considered biprocess into a set of Horn clauses, and using a resolution algorithm on these clauses. We adapt this translation to our new destructors.

4.1 From Equational Theories to Rewrite Rules

Equational theories are a very powerful tool for modeling cryptographic primitives. However, for a practical algorithm, it is easier to work with rewrite rules rather than with equational theories. Hence in [9], a signature Σ with an equational theory is transformed into a signature Σ' with rewrite rules that models Σ, when Σ has the finite variant property [13]. These rewrite rules may rewrite a term M into several irreducible forms (the variants), which are all equal modulo Σ, and such that, when M and M' are equal modulo Σ, M and M' rewrite to at least one common irreducible form. We reuse the algorithm from [9] for generating Σ', adapting it to our formalism by just completing the rewrite rules of constructors with rewrite rules that reduce to fail when an argument is fail (see Appendix C for more details).

4.2 Patterns and Facts

In the clauses, the messages are represented by patterns, with the following grammar:

$p ::=$	pattern	$mp ::=$	may-fail pattern
x, y, z, i	variables	p	pattern
$f(p_1, \ldots, p_n)$	constructor application	u, v	may-fail variables
$a[p_1, \ldots, p_n]$	name	fail	failure

The patterns p are the same as in [9]. The variable i represents a session identifier for each replication of a process. A pattern $a[p_1, \ldots, p_n]$ is assigned to each name of a process P. The arguments $p_1, \ldots, p_n$ allow one to model that a fresh name a is created at execution of $!\,(\nu a)$. For example, in the process $!\,c'(x).(\nu a)P$, each name created by νa is represented by $a[i, x]$ where i is the session identifier for the replication and x is the message received as input in $c'(x)$. Hence, the name a is represented as a function of i and x. In two different sessions, (i, x) takes two different values, so the two created instances of a ($a[i, x]$) are different.

Since our formalism introduced may-fail messages to describe possible failure of a destructor, we also define *may-fail patterns* to represent the failure in clauses. Similarly to messages and may-fail messages, a may-fail variable u can be instantiated by a pattern or fail, whereas a variable x cannot be instantiated by fail.

Clauses are built from the following predicates:

$F ::=$	facts
$\mathrm{att}'(mp, mp')$	attacker knowledge
$\mathrm{msg}'(p_1, p_2, p_1', p_2')$	output message p_2 on p_1 (resp. p_2' on p_1')
$\mathrm{input}'(p, p')$	input on p (resp. p')
$\mathrm{formula}(\bigwedge_i \forall \tilde{z}_i.p_i \neq_\Sigma p_i')$	formula
bad	bad

Intuitively, $\mathrm{att}'(mp, mp')$ means that the attacker may obtain mp in $\mathsf{fst}(P)$ and mp' in $\mathsf{snd}(P)$ by the same operations; the fact $\mathrm{msg}'(p_1, p_2, p_1', p_2')$ means that message p_2 may be output on channel p_1 by the process $\mathsf{fst}(P)$ while p_2' may be output on channel p_1' by the process $\mathsf{snd}(P)$ after the same reductions; $\mathrm{input}'(p, p')$ means that an input is possible on channel p in $\mathsf{fst}(P)$ and on channel p' in $\mathsf{snd}(P)$. Note that both facts

msg' and input' contain only patterns and not may-fail patterns. Hence channels and sent terms are necessarily messages and so cannot be fail. The fact $\mathrm{formula}(\phi)$ means that ϕ has to be satisfied. At last, bad serves in detecting violations of observational equivalence: when bad is not derivable, we have observational equivalence.

4.3 Clauses for the Attacker

The capabilities of the attacker are represented by clauses such as the ones below. They are directly inspired from the ones in [9] and correspond to the same semantics adapted to our formalism. The complete set of clauses is given in Appendix D.

$$\mathrm{att}'(\mathsf{fail}, \mathsf{fail}) \tag{Rfail}$$

For each function h, for each pair of rewrite rules
$\mathsf{h}(U_1, \ldots, U_n) \to U \mid\mid \phi$ and $\mathsf{h}(U'_1, \ldots, U'_n) \to U' \mid\mid \phi'$
in $\mathrm{def}_{\Sigma'}(\mathsf{h})$ (after renaming of variables),
$\mathrm{att}'(U_1, U'_1) \wedge \ldots \wedge \mathrm{att}'(U_n, U'_n) \wedge \mathrm{formula}(\phi \wedge \phi') \to \mathrm{att}'(U, U')$ (Rf)

$$\mathrm{input}'(x, x') \wedge \mathrm{msg}'(x, z, y', z') \wedge \mathrm{formula}(x' \neq_\Sigma y') \to \mathrm{bad} \tag{Rcom}$$

$$\mathrm{att}'(x, \mathsf{fail}) \to \mathrm{bad} \tag{Rfailure}$$

plus symmetric clauses (Rcom′) and (Rfailure′) obtained from (Rcom) and (Rfailure) by swapping the first and second arguments of att' and input', and the first and third arguments of msg'.

Clauses (Rf) apply a constructor or a destructor on the attacker's knowledge, given the definition of the destructor in $\mathrm{def}_{\Sigma'}(\mathsf{h})$. Since our destructors may return fail, by combining (Rf) with (Rfailure) or (Rfailure′), we can detect when a destructor succeeds in one variant of the biprocess and not in the other. We stress that, in clauses (Rfailure) and (Rcom), x, x', y, y' are message variables and so they cannot be instantiated by fail. (The messages sent on the network and the channels are never fail.)

4.4 Clauses for the Protocol

To translate the protocol into clauses, we first need to define evaluation on open terms, as a relation $D \Downarrow'_{\Sigma'} (U, \sigma, \phi)$, where σ collects instantiations of D obtained by unification and ϕ collects the side conditions of destructor applications:

$$U \Downarrow'_{\Sigma'} (U, \emptyset, \top)$$

$$\mathrm{eval}\ \mathsf{h}(D_1, \ldots, D_n) \Downarrow'_{\Sigma'} (\sigma_u V, \sigma_u \sigma', \sigma_u \phi' \wedge \sigma_u \phi)$$
if $(D_1, \ldots, D_n) \Downarrow'_{\Sigma'} ((U_1, \ldots, U_n), \sigma', \phi')$,
$\mathsf{h}(V_1, \ldots, V_n) \to V \mid\mid \phi \in \mathrm{def}_{\Sigma'}(\mathsf{h})$ and
σ_u is a most general unifier of $(U_1, V_1), \ldots, (U_n, V_n)$

$$(D_1, \ldots, D_n) \Downarrow'_{\Sigma'} ((\sigma_n U_1, \ldots, \sigma_n U_{n-1}, U_n), \sigma_n \sigma, \sigma_n \phi \wedge \phi_n)$$
if $(D_1, \ldots, D_{n-1}) \Downarrow'_{\Sigma'} ((U_1, \ldots, U_{n-1}), \sigma, \phi)$ and $\sigma D_n \Downarrow' (U_n, \sigma_n, \phi_n)$

The most general unifier of may-fail messages is computed similarly to the most general unifier of messages, even though specific cases hold due to may-fail variables and message variables: there is no unifier of M and fail, for any message M (including variables

x, because these variables can be instantiated only by messages); the most general unifier of u and U is $\{^U/_u\}$; the most general unifier of fail and fail is the identity; finally, the most general unifier of M and M' is computed as usual.

The translation $[\![P]\!]\rho s H$ of a biprocess P is a set of clauses, where ρ is an environment that associates a pair of patterns with each name and variable, s is a sequence of patterns, and H is a sequence of facts. The empty sequence is written $\emptyset$; the concatenation of a pattern p to the sequence s is written s, p; the concatenation of a fact F to the sequence H is written $H \wedge F$. Intuitively, H represents the hypothesis of the clauses, ρ represents the names and variables that are already associated with a pattern, and s represents the current values of session identifiers and inputs.

When ρ associates a pair of patterns with each name and variable, and f is a constructor, we extend ρ as a substitution by $\rho(\mathsf{f}(M_1, \ldots, M_n)) = (\mathsf{f}(p_1, \ldots, p_n), \mathsf{f}(p'_1, \ldots, p'_n))$ where $\rho(M_i) = (p_i, p'_i)$ for all $i \in \{1, \ldots, n\}$. We denote by $\rho(M)_1$ and $\rho(M)_2$ the components of the pair $\rho(M)$. We let $\rho(\mathsf{diff}[M, M']) = (\rho(M)_1, \rho(M')_2)$.

The definition of $[\![P]\!]\rho s H$ is directly inspired from [9]. We only present below the case $[\![\mathsf{let}\ x = D\ \mathsf{in}\ P\ \mathsf{else}\ Q]\!]\rho s H$ but the full definition of $[\![P]\!]\rho s H$ is given in Appendix D.

$$
\begin{aligned}
&[\![\mathsf{let}\ x = D\ \mathsf{in}\ P\ \mathsf{else}\ Q]\!]\rho s H = \\
&\quad \bigcup \{ [\![P]\!]((\sigma\rho)[x \mapsto (p, p')])(\sigma s, p, p')(\sigma H \wedge \mathrm{formula}(\phi)) \\
&\qquad \mid (\rho(D)_1, \rho(D)_2) \Downarrow' ((p, p'), \sigma, \phi) \} \\
&\quad \cup \bigcup \{ [\![Q]\!](\sigma\rho)(\sigma s)(\sigma H \wedge \mathrm{formula}(\phi)) \mid (\rho(D)_1, \rho(D)_2) \Downarrow' ((\mathsf{fail}, \mathsf{fail}), \sigma, \phi) \} \\
&\quad \cup \{ \sigma H \wedge \mathrm{formula}(\phi) \to \mathrm{bad} \mid (\rho(D)_1, \rho(D)_2) \Downarrow' ((p, \mathsf{fail}), \sigma, \phi) \} \\
&\quad \cup \{ \sigma H \wedge \mathrm{formula}(\phi) \to \mathrm{bad} \mid (\rho(D)_1, \rho(D)_2) \Downarrow' ((\mathsf{fail}, p'), \sigma, \phi) \}
\end{aligned}
$$

Comparing with the clauses described in [9], we can recognise the same kind of generated clauses: when both $\rho(D)_1$ and $\rho(D)_2$ succeed, the process P is translated, instantiating terms with the substitution σ and taking into account the side-condition ϕ, to make sure that $\rho(D)_1$ and $\rho(D)_2$ succeed ; when both fail, the process Q is translated ; and at last when one of $\rho(D)_1$, $\rho(D)_2$ succeeds and the other fails, clauses deriving bad are generated. Since may-fail variables do not occur in D, we can show by induction on the computation of $\Downarrow'$ that, when $(\rho(D)_1, \rho(D)_2) \Downarrow' ((mp_1, mp_2), \sigma, \phi)$, mp_1 and mp_2 are either fail or a pattern, but cannot be a may-fail variable, so our definition of $[\![\mathsf{let}\ x = D\ \mathsf{in}\ P\ \mathsf{else}\ Q]\!]\rho s H$ handles all cases.

4.5 Proving Equivalences

Let $\rho_0 = \{a \mapsto (a[\,], a[\,]) \mid a \in \mathit{fn}(P_0)\}$. We define the set of clauses that corresponds to biprocess P_0 as:

$$\mathcal{R}_{P_0} = [\![P_0]\!]\rho_0\emptyset\emptyset \cup \{(\mathrm{Rfail}), \ldots, (\mathrm{Rfailure}')\}$$

The following theorem enables us to prove equivalences from these clauses.

Theorem 1. *If* bad *is not a logical consequence of* $\mathcal{R}_{P_0}$*, then* P_0 *satisfies observational equivalence.*

This theorem shows the soundness of the translation. The proof of this theorem is adapted from the proof of Theorem 3 of [9]. Furthermore, since we use almost the same patterns and facts as in [9], we also use the algorithm proposed in [9] to automatically check if bad is a logical consequence of $\mathcal{R}_{P_0}$, with the only change that we use the unification algorithm for may-fail patterns.

5 Automatic Modification of the Protocol

In this section, we first present the kind of false attack that we want to avoid and then propose an algorithm to automatically generate, from a biprocess P, equivalent biprocesses on which ProVerif will avoid this kind of false attack.

5.1 Targeted False Attacks

We present a false attack on the anonymity of the *private authentication* protocol due to structural conditional branching.

Example 5. Coming back to the *private authentication* protocol (see Example 4), we obtained a biprocess P_0 on which we would ask ProVerif to check the equivalence. Unfortunately, ProVerif is unable to prove the equivalence of P_0 and yields a false attack. Indeed, consider the evaluation context C defined as follows:

$$C \stackrel{def}{=} \_ \mid (\nu n_i)c(x_{sk_a}).c(x_{sk_{a'}}).c(x_{sk_b}).\overline{c}\langle \mathsf{aenc}(\langle n_i, x_{sk_a}\rangle, x_{sk_b})\rangle$$

The biprocess $C[P_0]$ can be derived as follows:

$$\begin{aligned} C[P_0] \rightarrow^*_\Sigma\ & (\nu n_i)(\nu sk_a)(\nu sk_{a'})(\nu sk_b) \\ & \big(\overline{c}\langle \mathsf{aenc}(\langle n_i, \mathsf{pk}(sk_a)\rangle, \mathsf{pk}(sk_b))\rangle \mid System(\mathsf{diff}[sk_a, sk_{a'}], sk_b)\big) \\ \rightarrow^*_\Sigma\ & (\nu n_i)(\nu sk_a)(\nu sk_{a'})(\nu sk_b)(A(\mathsf{diff}[sk_a, sk_{a'}], sk_b) \mid \\ & \quad \mathsf{let}\ z = \mathsf{equals}(\mathsf{proj}_2(\langle n_i, \mathsf{pk}(sk_a)\rangle)), \mathsf{pk}(\mathsf{diff}[sk_a, sk_{a'}]))\ \mathsf{in} \\ & \quad\ \overline{c}\langle \mathsf{aenc}(\langle n_i, \langle n_b, \mathsf{pk}(sk_b)\rangle\rangle, \mathsf{pk}(\mathsf{diff}[sk_a, sk_{a'}]))\rangle \\ & \quad \mathsf{else}\ \overline{c}\langle \mathsf{aenc}(n_b, \mathsf{pk}(sk_b))\rangle) \end{aligned}$$

However from this point, the biprocess gets stucked, *i.e.* no internal reduction rule is applicable. More specifically, neither the internal rule (Red Fun 1) nor (Red Fun 2) is applicable. Indeed, if we denote $D = \mathsf{equals}(\mathsf{proj}_2(\langle n_i, sk_a\rangle)), \mathsf{pk}(\mathsf{diff}[sk_a, sk_{a'}]))$, we have that $\mathsf{snd}(D) \Downarrow_\Sigma \mathsf{fail}$ and $\mathsf{fst}(D) \Downarrow_\Sigma \mathsf{pk}(sk_a)$, which contradicts the item 2 of Lemma 2. So ProVerif cannot prove the equivalence. But, although a different branch of the let is taken, the process outputs the message $\mathsf{aenc}(\langle n_b, \langle n_a, \mathsf{pk}(sk_b)\rangle\rangle, \mathsf{pk}(sk_a))$ in the first variant (in branch of the let) and the message $\mathsf{aenc}(n_b, \mathsf{pk}(sk_b))$ in the second variant (else branch of the let). Intuitively, these two messages are indistinguishable, so in fact the attacker will not be able to determine which branch of the let is taken, and observational equivalence still holds.

In order to avoid the false attacks similar to Example 5, we transform term evaluations $\mathsf{let}\ x = D\ \mathsf{in}\ \overline{c}\langle M_1\rangle\ \mathsf{else}\ \overline{c}\langle M_2\rangle$ into a computation that always succeeds

$\mathsf{let}\ x = D'\ \mathsf{in}\ \mathsf{let}\ m = D''\ \mathsf{in}\ \overline{c}\langle m\rangle$. The term evaluation D' will correspond to the value of the evaluation of D when the latter succeeds and a new constant when D fails. Thus we ensure that D' never fails. Moreover, the term evaluation D'' computes either M_1 or M_2 depending on the value of D', *i.e.* depending on whether D succeeds or not. The omitted else 0 branches are never taken. Since the same branch is always taken, the false attack disappears. To do that, we introduce three new destructors catchfail, letin, notfail and a constant $\mathsf{c_{fail}}$, which rely on the side conditions that we have added to destructors. These new destructors are defined as follows:

$$\begin{array}{lll}
\mathrm{def}_\Sigma(\mathsf{catchfail}) = & \mathrm{def}_\Sigma(\mathsf{letin}) = & \mathrm{def}_\Sigma(\mathsf{notfail}) = \\
\quad \mathsf{catchfail}(x) \to x & \quad \mathsf{letin}(x,u,v) \to u \mid\mid x \neq_\Sigma \mathsf{c_{fail}} & \quad \mathsf{notfail}(x) \to \mathsf{fail} \\
\quad \mathsf{catchfail}(\mathsf{fail}) \to \mathsf{c_{fail}} & \quad \mathsf{letin}(\mathsf{c_{fail}},u,v) \to v & \quad \mathsf{notfail}(\mathsf{fail}) \to \mathsf{c_{fail}} \\
 & \quad \mathsf{letin}(\mathsf{fail},u,v) \to \mathsf{fail} &
\end{array}$$

One can easily check that $\mathrm{def}_\Sigma(\mathsf{catchfail})$, $\mathrm{def}_\Sigma(\mathsf{letin})$ and $\mathrm{def}_\Sigma(\mathsf{notfail})$ satisfy Properties P1 and P2. Intuitively, the destructor catchfail evaluates its argument and returns either the result of this evaluation when it did not fail or else returns the new constant $\mathsf{c_{fail}}$ instead of the failure constant fail. The destructor letin will get the result of catchfail as first argument and return its third argument if catchfail returned $\mathsf{c_{fail}}$, and its second argument otherwise. Importantly, catchfail never fails: it returns $\mathsf{c_{fail}}$ instead of fail. Hence, $\mathsf{let}\ x = D\ \mathsf{in}\ \overline{c}\langle M_1\rangle\ \mathsf{else}\ \overline{c}\langle M_2\rangle$ can be transformed into $\mathsf{let}\ x = \mathrm{eval}\ \mathsf{catchfail}(D)\ \mathsf{in}\ \mathsf{let}\ m = \mathrm{eval}\ \mathsf{letin}(x, M_1, M_2)\ \mathsf{in}\ \overline{c}\langle m\rangle$: if D succeeds, x has the same value as before, and $x \neq \mathsf{c_{fail}}$, so $\mathsf{letin}(x, M_1, M_2)$ returns M_1; if D fails, $x = \mathsf{c_{fail}}$ and $\mathsf{letin}(x, M_1, M_2)$ returns M_2. The destructor notfail inverses the status of a term evaluation: it fails if and only if its argument does not fail. This destructor will be used in the next section.

Example 6. Coming back to Example 5, the false attack occurs due to the following term evaluation:

$$\begin{array}{l}
\mathsf{let}\ z = \mathsf{equals}(\mathsf{proj}_2(x), \mathsf{pk}(\mathsf{diff}[ska, ska'])) \ \mathsf{in} \\
\quad \overline{c}\langle \mathsf{aenc}(\langle n_i, \langle n_b, \mathsf{pk}(sk_b)\rangle\rangle, \mathsf{pk}(\mathsf{diff}[sk_a, sk_{a'}]))\rangle \\
\mathsf{else}\ \overline{c}\langle \mathsf{aenc}(n_b, \mathsf{pk}(sk_b))\rangle
\end{array}$$

We transform this term evaluation as explained above:

$$\mathsf{let}\ z = \mathsf{letin}(\mathsf{catchfail}(\mathsf{equals}(\mathsf{proj}_2(x), \mathsf{pk}(\mathsf{diff}[ska, ska']))), M, M')\ \mathsf{in}\ \overline{c}\langle z\rangle$$

where $M = \mathsf{aenc}(\langle n_i, \langle n_b, \mathsf{pk}(sk_b)\rangle\rangle, \mathsf{pk}(\mathsf{diff}[sk_a, sk_{a'}]))$, $M' = \mathsf{aenc}(n_b, \mathsf{pk}(sk_b))$. Note that with $x = \langle n_i, \mathsf{pk}(sk_a)\rangle$ (see Example 5), if D is the term evaluation $D = \mathsf{letin}(\mathsf{catchfail}(\mathsf{equals}(\mathsf{proj}_2(x), \mathsf{pk}(\mathsf{diff}[ska, ska']))), M, M')$, we obtain that:

- $\mathsf{fst}(D) \Downarrow \mathsf{aenc}(\langle n_i, \langle n_b, \mathsf{pk}(sk_b)\rangle\rangle, \mathsf{pk}(sk_a))$
- $\mathsf{snd}(D) \Downarrow \mathsf{aenc}(n_b, \mathsf{pk}(sk_b))$

which corresponds to what $\mathsf{fst}(P_0)$ and $\mathsf{snd}(P_0)$ respectively output. Thanks to this, if we denote by P_0' our new biprocess, we obtain that $\mathsf{fst}(P_0) \approx \mathsf{fst}(P_0')$ and $\mathsf{snd}(P_0) \approx \mathsf{snd}(P_0')$. Furthermore, PROVERIF will be able to prove that the biprocess P_0' satisfies equivalence, i.e. $\mathsf{fst}(P_0') \approx \mathsf{snd}(P_0')$ and so $\mathsf{fst}(P_0) \approx \mathsf{snd}(P_0)$.

The transformation proposed in the previous example can be generalised to term evaluations that perform other actions than just a single output. However, it is possible only if the success branch and the failure branch of the term evaluation both input and output the same number of terms. For example, the biprocess $\mathsf{let}\ x = D\ \mathsf{in}\ \overline{c}\langle M\rangle.\overline{c}\langle M'\rangle$ $\mathsf{else}\ \overline{c}\langle N\rangle$ cannot be modified into a biprocess without else branch even with our new destructors. On the other hand, the success or failure of D can really be observed by the adversary, by tracking the number of outputs on the channel c, so the failure of the proof of equivalence corresponds to a real attack in this case.

5.2 Merging and Simplifying Biprocesses

To automatically detect and apply this transformation, we define two functions, denoted $merge$ and $simpl$, defined respectively in Fig. 4 and 5. The function $merge$ is partial whereas $simpl$ is total. Intuitively, the function $merge$ takes two biprocesses as arguments and detects if those two biprocesses can be merged into one biprocess. Furthermore, if the merging is possible, it returns the merged biprocess. This merged biprocess is expressed using a new operator diff', similar to diff, in such a way that $\mathsf{diff}'[D, D']$ is a term evaluation. We introduce the functions fst' and snd': $\mathsf{fst}'(P)$ (resp. $\mathsf{snd}'(P)$) replaces each $\mathsf{diff}'[D, D']$ with D (resp. D') in P.

Case (Mout) detects that both biprocesses output a message while case (Min) detects that both biprocesses input a message. We introduce a let for the channels and messages so that they can later be replaced by a term evaluation. Case (Mpar) uses the commutativity and associativity of parallel composition to increase the chances of success of $merge$. Cases (Mres) and (Mres$'$) use $Q \approx (\nu a)Q$ when $a \notin fn(Q)$ to allow merging processes even when a restriction occurs only on one side. Case (Mrepl2) is the basic merging of replicated processes, while Case (Mrepl1) allows merging $!^{!}P$ with $!P'$ (case $n = 0$) because $!P \approx !^{!}P$, and furthermore allows restrictions between the two replications, using $Q \approx (\nu a)Q$. Case (Mlet1) merges two processes that both contain term evaluations, by merging their success branches together and their failure branches together. On the other hand, Cases (Mlet2), (Mlet2$'$) also merge two processes that contain term evaluations, by merging the success branch of one process with the failure branch of the other process. Cases (Mlet3), (Mlet3$'$), (Mlet4), (Mlet4$'$) allow merging a term evaluation with another process P', by merging P' with either the success branch or the failure branch of the term evaluation. This merging is useful when ProVerif can prove that the resulting process satisfies equivalence, hence when both sides of the obtained let succeed simultaneously. Therefore, rule (Mlet3) is helpful when the term evaluation D always succeeds, and rule (Mlet4) when D always fails. When no such case applies, merging fails.

The function $simpl$ takes one biprocess as argument and tries to simplify the subprocesses $\mathsf{let}\ x = D\ \mathsf{in}\ P\ \mathsf{else}\ Q$ when P and Q can be merged. The only interesting case is (Smerge), which performs the transformation of term evaluations outlined above, when we can merge the success and failure branches. $Q'\{^{\mathrm{eval}\ \mathsf{letin}(x,D_1,D_2)}/_{\mathsf{diff}'[D_1,D_2]}\}$ means that we replace in Q' every instance of $\mathsf{diff}'[D_1, D_2]$, for some D_1, D_2, with $\mathrm{eval}\ \mathsf{letin}(x, D_1, D_2)$.

Note that both functions are non-deterministic; the implementation may try all possibilities. In the current implementation of ProVerif, we apply the rules (Mlet3)

$merge(0,0) \stackrel{def}{=} 0$ (Mnil)

$merge(\overline{M}\langle N\rangle.P, \overline{M'}\langle N'\rangle.P') \stackrel{def}{=}$
$\quad \mathsf{let}\ x = \mathsf{diff}'[M, M']\ \mathsf{in}\ \mathsf{let}\ x' = \mathsf{diff}'[N, N']\ \mathsf{in}\ \overline{x}\langle x'\rangle.merge(P, P')$
$\quad$ where x and x' are fresh variables (Mout)

$merge(M(x).P, M'(x').P') \stackrel{def}{=}$
$\quad \mathsf{let}\ y = \mathsf{diff}'[M, M']\ \mathsf{in}\ y(y').merge(P\{^{y'}/_{x}\}, P'\{^{y'}/_{x'}\})$
$\quad$ where y and y' are fresh variables (Min)

$merge(P_1 \mid \ldots \mid P_n, P'_1, \mid \ldots \mid P'_n) \stackrel{def}{=} Q_1 \mid \ldots \mid Q'_n$
$\quad$ if $(i_1, \ldots, i_n)$ is a permutation of $(1, \ldots, n)$
$\quad$ and for all $k \in \{1, \ldots, n\}$, $Q_k = merge(P_k, P'_{i_k})$ (Mpar)

$merge(\nu a.P, Q) \stackrel{def}{=} \nu a.merge(P, Q)$
$\quad$ after renaming a such that $a \notin fn(Q)$ (Mres)

$merge(!\,(\nu a_1) \ldots (\nu a_n)!\,P, !\,P') \stackrel{def}{=} !\,(\nu a_1) \ldots (\nu a_n) merge(!\,P, !\,P')$
$\quad$ after renaming $a_1, \ldots, a_n$ such that $a_1, \ldots, a_n \notin fn(P')$ (Mrepl1)

$merge(!\,P, !\,P') \stackrel{def}{=} !\,merge(P, P')$
$\quad$ if there is no P_1 such that $P = !\,P_1$ and no P'_1 such that $P' = !\,P'_1$ (Mrepl2)

$merge(\mathsf{let}\ x = D\ \mathsf{in}\ P_1\ \mathsf{else}\ P_2, \mathsf{let}\ x' = D'\ \mathsf{in}\ P'_1\ \mathsf{else}\ P'_2) \stackrel{def}{=}$
$\quad \mathsf{let}\ y = \mathsf{diff}'[D, D']\ \mathsf{in}\ Q_1\ \mathsf{else}\ Q_2$ if y is a fresh variable,
$\quad Q_1 = merge(P_1\{^{y}/_{x}\}, P'_1\{^{y}/_{x'}\})$, and $Q_2 = merge(P_2, P'_2)$ (Mlet1)

$merge(\mathsf{let}\ x = D\ \mathsf{in}\ P_1\ \mathsf{else}\ P_2, \mathsf{let}\ x' = D'\ \mathsf{in}\ P'_1\ \mathsf{else}\ P'_2) \stackrel{def}{=}$
$\quad \mathsf{let}\ y = \mathsf{diff}'[D, \mathsf{notfail}(D')]\ \mathsf{in}\ Q_1\ \mathsf{else}\ Q_2$ if y is a fresh variable,
$\quad x' \notin fv(P'_1)$, $Q_1 = merge(P_1\{^{y}/_{x}\}, P'_2)$, and $Q_2 = merge(P_2, P'_1)$ (Mlet2)

$merge(\mathsf{let}\ x = D\ \mathsf{in}\ P_1\ \mathsf{else}\ P_2, P') \stackrel{def}{=} \mathsf{let}\ y = \mathsf{diff}'[D, \mathsf{c_{fail}}]\ \mathsf{in}\ Q\ \mathsf{else}\ P_2$
$\quad$ if y is a fresh variable and $Q = merge(P_1\{^{y}/_{x}\}, P')$ (Mlet3)

$merge(\mathsf{let}\ x = D\ \mathsf{in}\ P_1\ \mathsf{else}\ P_2, P') \stackrel{def}{=} \mathsf{let}\ y = \mathsf{diff}'[D, \mathsf{fail}]\ \mathsf{in}\ P_1\{^{y}/_{x}\}\ \mathsf{else}\ Q$
$\quad$ if y is a fresh variable and $Q = merge(P_2, P')$ (Mlet4)

plus symmetric cases (Mres′), (Mrepl1′), (Mlet2′), (Mlet3′), and (Mlet4′) obtained from (Mres), (Mrepl1), (Mlet2), (Mlet3), and (Mlet4) by swapping the first and second arguments of $merge$ and diff'.

Fig. 4. Definition of the function $merge$

$$simpl(0) \stackrel{def}{=} 0 \quad \text{(Snil)}$$

$$simpl(\overline{M}\langle N\rangle.P) \stackrel{def}{=} \overline{M}\langle N\rangle.simpl(P) \quad \text{(Sout)}$$

$$simpl(M(x).P) \stackrel{def}{=} M(x).simpl(P) \quad \text{(Sin)}$$

$$simpl(P \mid Q) \stackrel{def}{=} simpl(P) \mid simpl(Q) \quad \text{(Smid)}$$

$$simpl((\nu a)P) \stackrel{def}{=} \nu a.simpl(P) \quad \text{(Sres)}$$

$$simpl(!\,P) \stackrel{def}{=} !\,simpl(P) \quad \text{(Srepl)}$$

$$simpl(\mathsf{let}\ x = D\ \mathsf{in}\ P\ \mathsf{else}\ P') \stackrel{def}{=} \mathsf{let}\ x = \mathrm{eval}\ \mathsf{catchfail}(D)\ \mathsf{in}\ Q\ \mathsf{else}\ 0$$
$$\text{if } Q' = merge(simpl(P), simpl(P')) \text{ and} \quad \text{(Smerge)}$$
$$Q = Q'\{^{\mathrm{eval}\ \mathsf{letin}(x,D_1,D_2)}/_{\mathsf{diff}'[D_1,D_2]}\}$$

$$simpl(\mathsf{let}\ x = D\ \mathsf{in}\ P\ \mathsf{else}\ P') \stackrel{def}{=} \mathsf{let}\ x = D\ \mathsf{in}\ simpl(P)\ \mathsf{else}\ simpl(P') \quad \text{(Slet)}$$
$$\text{if there is no } Q \text{ such that } Q = merge(simpl(P), simpl(P'))$$

Fig. 5. Definition of the function $simpl$

and (Mlet4) only if the rules (Mlet1) and (Mlet2) are not applicable. Moreover, we never merge 0 with a process different from 0. This last restriction is crucial to reduce the number of biprocesses returned by the functions $merge$ and $simpl$. Typically, we avoid that 0 and $\mathsf{let}\ x = M\ \mathsf{in}\ P\ \mathsf{else}\ 0$ are merged by the rule (Mlet4).

5.3 Results

Lemmas 3 and 4 below show how the observational equivalence is preserved by the functions $merge$ and $simpl$. In these two lemmas, we consider biprocesses P and P' that are not necessarily closed. We say that a context C is closing for P when $C[P]$ is closed. Moreover, given two biprocesses P and Q, we say that $P \approx Q$ if, and only if, $\mathsf{fst}(P) \approx \mathsf{fst}(Q)$ and $\mathsf{snd}(P) \approx \mathsf{snd}(Q)$. All results of this section are proved in Appendix E.

Lemma 3 (merge). *Let P and P' be two biprocesses. If $merge(P, P') = Q$, then:*

- *for all contexts C closing for P, $C[P] \approx C[\mathsf{fst}'(Q)]$;*
- *for all contexts C closing for P', $C[P'] \approx C[\mathsf{snd}'(Q)]$.*

Lemma 4 (simplify). *Let P be a biprocess. For all contexts C closing for P,*

$$C[P] \approx C[simpl(P)]$$

From the two previous lemmas, we can derive the two main results of this section.

Theorem 2. *Let P be a closed biprocess. If $simpl(P)$ satisfies observational equivalence then $\mathsf{fst}(P) \approx \mathsf{snd}(P)$.*

From Theorem 2, we can extract our new algorithm. Given a biprocess P as input, we first apply the function $simpl$ on P. Since the function $simpl$ is total but non-deterministic, we may have several biprocesses as result for $simpl(P)$. If ProVerif is able to prove equivalence on at least one of them, then we conclude that $\mathsf{fst}(P) \approx \mathsf{snd}(P)$.

Theorem 3. *Let P and P' be two closed processes that do no contain terms with* diff. *Let $Q = merge(simpl(P), simpl(P'))$. If the biprocess $Q\{^{\mathsf{diff}[D,D']}/_{\mathsf{diff}'[D,D']}\}$ satisfies observational equivalence, then $P \approx P'$.*

The previous version ProVerif can only take a biprocess as input. However, transforming two processes into a biprocess is usually not as easy as in the private authentication protocol example. Thanks to Theorem 3, we are able to automate this transformation.

6 Conclusion

In the present paper, we have extended the framework of ProVerif by allowing the destructors to be modelled by rewrite rules with inequalities as side-conditions. We have proposed a procedure relying on these new rewrite rules to automatically transform a biprocess into equivalent biprocesses on which ProVerif avoids the possible false attacks due to conditional branchings.

A beta version of ProVerif including our extension is available at `http://proverif.inria.fr`. Experimentation showed that the automatic transformation of a biprocess is efficient and returns few biprocesses in practice. In particular, our extension can automatically prove the anonymity of the private authentication protocol for an unbounded number of sessions (following the definition of anonymity of [5]).

However, ProVerif still unable to prove the unlinkability of the English e-passport protocol [5] even though we managed to avoid some previously existing false attacks. This a consequence of the matching by ProVerif of traces with the same scheduling in the two variants of the biprocesses. Thus we would like to relax a little bit the matching of traces, *e.g.* by modifying the replication identifiers on the left and right parts of biprocesses. This would allow us to prove even more equivalences with ProVerif and in particular the e-passport protocol.

Another direction for future research would be to define equations with inequalities as side-conditions. It may be possible to convert such equations into rewrite rules with side-conditions, like we convert equations into rewrite rules.

Acknowledgments This work has been partially supported by the ANR project PROSE (decision ANR 2010-VERS-004) and JCJC VIP n° 11 JS02 006 01, as well as the grant DIGITEO API from Région Île-de-France. It was partly done while the authors were at Ecole Normale Supérieure, Paris.

References

1. Abadi, M., Fournet, C.: Mobile values, new names, and secure communication. In: POPL'01. pp. 104–115. ACM, New York (2001)

2. Abadi, M., Fournet, C.: Private authentication. Theoretical Computer Science 322(3), 427–476 (2004)
3. Abadi, M., Gordon, A.D.: A calculus for cryptographic protocols: The spi calculus. Information and Computation 148(1), 1–70 (1999)
4. Arapinis, M., Cheval, V., Delaune, S.: Verifying privacy-type properties in a modular way. In: CSF'12. pp. 95–109. IEEE Computer Society Press, Los Alamitos (2012)
5. Arapinis, M., Chothia, T., Ritter, E., Ryan, M.: Analysing unlinkability and anonymity using the applied pi calculus. In: CSF'10. pp. 107–121. IEEE Computer Society Press (2010)
6. Armando, A., et al.: The AVISPA tool for automated validation of Internet security protocols and applications. In: Etessami, K., Rajamani, S.K. (eds.) CAV'05. LNCS, vol. 3576, pp. 281–285. Springer, Heidelberg (2005)
7. Baudet, M.: Sécurité des protocoles cryptographiques: aspects logiques et calculatoires. Ph.D. thesis, Ecole Normale Supérieure de Cachan (2007)
8. Blanchet, B.: An efficient cryptographic protocol verifier based on Prolog rules. In: CSFW-14. pp. 82–96. IEEE, Los Alamitos (2001)
9. Blanchet, B., Abadi, M., Fournet, C.: Automated verification of selected equivalences for security protocols. Journal of Logic and Algebraic Programming 75(1), 3–51 (2008)
10. Borgström, J., Briais, S., Nestmann, U.: Symbolic bisimulation in the spi calculus. In: Gardner, P., Yoshida, N. (eds.) CONCUR'04. LNCS, vol. 3170, pp. 161–176. Springer, Heidelberg (2004)
11. Cheval, V., Comon-Lundh, H., Delaune, S.: Trace equivalence decision: Negative tests and non-determinism. In: CCS'11. pp. 321–330. ACM, New York (2011)
12. Ciobâcă, Ş.: Automated Verification of Security Protocols with Applications to Electronic Voting. Ph.D. thesis, Laboratoire Spécification et Vérification, ENS Cachan, France (2011)
13. Comon-Lundh, H., Delaune, S.: The finite variant property: How to get rid of some algebraic properties. In: Giesl, J. (ed.) RTA'05. LNCS, vol. 3467, pp. 294–307. Springer, Heidelberg (2005)
14. Cremers, C.J.F.: Scyther - Semantics and Verification of Security Protocols. Ph.D. thesis, Eindhoven University of Technology (2006)
15. Delaune, S., Kremer, S., Ryan, M.D.: Symbolic bisimulation for the applied pi-calculus. In: Arvind, V., Prasad, S. (eds.) FSTTCS'07. LNCS, vol. 4855, pp. 133–145. Springer, Heidelberg (2007)
16. Durante, L., Sisto, R., Valenzano, A.: Automatic testing equivalence verification of spi calculus specifications. ACM TOSEM 12(2), 222–284 (2003)
17. Hüttel, H.: Deciding framed bisimilarity. In: INFINITY'02. pp. 1–20 (2002)
18. Liu, J., Lin, H.: A complete symbolic bisimulation for full applied pi calculus. In: van Leeuwen, J., Muscholl, A., Peleg, D., Pokorný, J., Rumpe, B. (eds.) SOFSEM'10. LNCS, vol. 5901, pp. 552–563. Springer, Heidelberg (2010)
19. Nicola, R.D., Hennessy, M.: Testing equivalences for processes. Theoretical Computer Science 34, 83–133 (1984)
20. Ramanujam, R., Suresh, S.: Tagging makes secrecy decidable with unbounded nonces as well. In: Pandya, P., Radhakrishnan, J. (eds.) FSTTCS'03. LNCS, vol. 2914, pp. 363–374. Springer, Heidelberg (2003)
21. Rusinowitch, M., Turuani, M.: Protocol insecurity with finite number of sessions is NP-complete. Theoretical Computer Science 299(1–3), 451–475 (2003)
22. Ryan, P., Schneider, S., Goldsmith, M., Lowe, G., Roscoe, B.: The Modelling and Analysis of Security Protocols. Addison Wesley (2000)
23. Tiu, A., Dawson, J.E.: Automating open bisimulation checking for the spi calculus. In: CSF'10. pp. 307–321. IEEE Computer Society Press (2010)

Appendix

A Properties of Destructors

Lemma 1. *Consider a signature Σ. Let* g *be a destructor of arity n described by the set of rewrite rules $\mathcal{S} = \{\mathsf{g}(M_1^i, \ldots, M_n^i) \to M^i \mid i = 1, \ldots, m\}$. Assume that* g *is deterministic,* i.e. *$\mathcal{S}$ satisfies Property P2. The following set $\mathrm{def}_\Sigma(\mathsf{g})$ satisfies Properties P1 and P2:*

$$\begin{aligned}\mathrm{def}_\Sigma(\mathsf{g}) = \mathcal{S} \cup \{&\mathsf{g}(x_1, \ldots, x_n) \to \mathsf{fail} \mid\mid \phi\} \\ &\cup \{\mathsf{g}(u_1, \ldots, u_{k-1}, \mathsf{fail}, u_{k+1}, \ldots, u_n) \to \mathsf{fail} \mid k = 1, \ldots, n\}\end{aligned}$$

where $\phi = \bigwedge_{i=1}^{m} \forall \tilde{y}_i.(x_1, \ldots, x_n) \neq_\Sigma (M_1^i, \ldots, M_n^i)$ and $\tilde{y}_i$ are the variables of $(M_1^i, \ldots, M_n^i)$, and $x_1, \ldots, x_n$ are message variables.

Proof. Let $U_1, \ldots, U_n$ be ground may-fail messages. Consider first the case where there exists $i \in \{1, \ldots, n\}$ such that $U_i =_\Sigma \mathsf{fail}$. Since $x_1, \ldots, x_n$ are message variables and for all $i \in \{1, \ldots, m\}$, for all $j \in \{1, \ldots, n\}$, M_j^i is a message, the only rules that can rewrite $U_1, \ldots, U_n$ are $\{\mathsf{g}(u_1, \ldots, u_{k-1}, \mathsf{fail}, u_{k+1}, \ldots, u_n) \to \mathsf{fail} \mid k = 1, \ldots, n\}$. Moreover, let σ be the substitution such that $\sigma u_j = U_j$ for all $j \neq i$. We trivially have that $\sigma u_j =_\Sigma U_j$ and by hypothesis $U_i =_\Sigma \mathsf{fail}$. Thus, $\mathsf{g}(U_1, \ldots, U_n) \to \mathsf{fail}$ and there is no message M such that $\mathsf{g}(U_1, \ldots, U_n) \to M$.

Consider now the case where $U_1, \ldots, U_n$ are all messages, that we rename $M_1, \ldots, M_n$. Thus, $M_1, \ldots, M_n$ can only be rewriten by the rules in $\mathcal{S} \cup \{\mathsf{g}(x_1, \ldots, x_n) \to \mathsf{fail} \mid\mid \phi\}$. Let $i \in \{1, \ldots, m\}$. By definition, we have that $M_1, \ldots, M_n$ is rewritten by the rule $\mathsf{g}(M_1^i, \ldots, M_n^i) \to M^i$ if, and only if, there exists a substitution σ such that $\sigma \vDash \exists \tilde{y}_i.(x_1, \ldots, x_n) =_\Sigma (M_1^i, \ldots, M_n^i)$ and $\sigma x_j = M_j$ for $j = 1 \ldots n$ where $\tilde{y}_i = fv(M_1^i, \ldots, M_n^i)$. Hence, $M_1, \ldots, M_n$ can be rewritten by one of the rules in $\mathcal{S}$ if, and only if, there exists a substitution σ such that $\sigma \vDash \neg\phi$. On the other hand, $M_1, \ldots, M_n$ can be rewritten by $\mathsf{g}(x_1, \ldots, x_n) \to \mathsf{fail} \mid\mid \phi$ if, and only if, $\sigma \vDash \phi$. Hence we deduce that $\mathrm{def}_\Sigma(\mathsf{g})$ satisfies Property P1. Moreover, since M^i is a message for all $i = 1 \ldots m$, and there is no substitution σ such that $\sigma \vDash \phi \wedge \neg\phi$, we deduce that $\mathrm{def}_\Sigma(\mathsf{g})$ satisfies Property P2. □

B Equivalence Proofs

This section is devoted to the proof of Lemma 2 which is the equivalent in our framework of [9, Corollary 1]. We first show a preliminary result on term evaluations:

Lemma 5. *Consider a signature Σ. For all ground term evaluations D,*

- *there exists a ground may-fail message U such that $D \Downarrow_\Sigma U$.*
- *for all ground may-fail messages U_1, U_2, if $D \Downarrow_\Sigma U_1$ and $D \Downarrow_\Sigma U_2$ then $U_1 =_\Sigma U_2$.*

Proof. We prove this result by induction on D:

Case $D = U$: In such a case, we have that $D \Downarrow_\Sigma U$ hence the result trivially holds.

Case $D = \text{eval } \mathsf{g}(D_1, \ldots, D_n)$*:* By inductive hypothesis, there exist $U_1, \ldots, U_n$ ground may-fail messages such that $D_i \Downarrow_\Sigma U_i$ for all $i = 1 \ldots n$. By Property P1, we know that there exists $\mathsf{g}(U'_1, \ldots, U'_n) \to U' \mid\mid \phi$ in $\mathrm{def}_\Sigma(\mathsf{g})$ and a substitution σ such that $\sigma U'_i =_\Sigma U_i$ for $i = 1 \ldots n$ and $\sigma \vDash \phi$ hence $D \Downarrow_\Sigma U'\sigma$. Moreover, since $fv(U') \subseteq fv(U'_1, \ldots, U'_n)$, we deduce that $\sigma U'$ is ground. Thus the first item is satisfied.

Let V_1, V_2 two may-fail messages such that $D \Downarrow_\Sigma V_1$ and $D \Downarrow_\Sigma V_2$. $D \Downarrow_\Sigma V_1$ implies that there exist $U_1, \ldots, U_n$ ground may-fail messages such that $D_i \Downarrow_\Sigma U_i$ for all $i = 1 \ldots n$ and $\mathsf{g}(U_1, \ldots, U_n) \to V_1$. Similarly, $D \Downarrow_\Sigma V_2$ implies that there exist $W_1, \ldots, W_n$ ground fail-messages such that $D_i \Downarrow_\Sigma W_i$ for all $i = 1 \ldots n$ and $\mathsf{g}(W_1, \ldots, W_n) \to V_2$. By our inductive hypothesis, we deduce that $W_i =_\Sigma U_i$ for all $i = 1 \ldots n$. Hence, by definition of the reduction of may-fail messages by a destructor, we deduce that $\mathsf{g}(U_1, \ldots, U_n) \to V_2$. But $\mathrm{def}_\Sigma(\mathsf{g})$ satisfies Property P2, thus we conclude that $V_1 =_\Sigma V_2$. □

The previous lemma indicates that the properties P1 and P2 of destructors can be extended to term evaluation.

We say that a biprocess P is uniform when $\mathsf{fst}(P) \to Q_1$ implies that $P \to Q$ for some biprocess Q with $\mathsf{fst}(Q) \equiv Q_1$, and symmetrically for $\mathsf{snd}(P) \to Q_2$. We can now prove Lemma 2, relying on [9, Theorem 1].

Lemma 2. *Let P_0 be a closed biprocess. Suppose that, for all plain evaluation contexts C, all evaluation contexts C', and all reductions $C[P_0] \to^* P$,*

1. *if $P \equiv C'[\overline{N}\langle M\rangle.Q \mid N'(x).R]$ then $\mathsf{fst}(N) =_\Sigma \mathsf{fst}(N')$ if and only if $\mathsf{snd}(N) =_\Sigma \mathsf{snd}(N')$;*
2. *if $P \equiv C'[\mathsf{let}\ x = D\ \mathsf{in}\ Q\ \mathsf{else}\ R]$ then $\mathsf{fst}(D) \Downarrow_\Sigma \mathsf{fail}$ if and only if $\mathsf{snd}(D) \Downarrow_\Sigma \mathsf{fail}$.*

Then P_0 satisfies observational equivalence.

Proof. We show that P is uniform, then we conclude by [9, Theorem 1]. Let us show that, if $\mathsf{fst}(P) \to P'_1$ then there exists a biprocess P' such that $P \to P'$ and $\mathsf{fst}(P') \equiv P'_1$. The case for $\mathsf{snd}(P) \to P'_2$ is symmetric.

By induction on the derivation of $\mathsf{fst}(P) \to P'_1$, we first show that there exist C, Q, and Q'_1 such that $P \equiv C[Q]$, $P'_1 \equiv \mathsf{fst}(C)[Q'_1]$, and $\mathsf{fst}(Q) \to Q'_1$ using one of the four process rules (Red I/O), (Red Fun 1), (Red Fun 2), or (Red Repl): every step in this derivation trivially commutes with fst, except for structural steps that involve a parallel composition and a restriction, in case $a \in fn(P)$ but $a \notin fn(\mathsf{fst}(P))$. In that case, we use a preliminary renaming from a to some fresh $a' \notin fn(P)$.

For each of these four rules, relying on a hypothesis of Lemma 2, we find Q' such that $\mathsf{fst}(Q') = Q'_1$ and $Q \to Q'$ using the corresponding biprocess rule:

(Red I/O): We have $Q = \overline{N}\langle M\rangle.R \mid N'(x).R'$ with $\vdash \mathsf{fst}(N) =_\Sigma \mathsf{fst}(N')$ and $Q'_1 = \mathsf{fst}(R) \mid \mathsf{fst}(R')\{\mathsf{fst}(M)/x\}$. For $Q' = R \mid R'\{M/x\}$, we have $\mathsf{fst}(Q') = Q'_1$ and, by hypothesis 1, $\mathsf{snd}(N) =_\Sigma \mathsf{snd}(N')$, hence $Q \to Q'$.

(Red Fun 1): We have $Q = \mathsf{let}\ x = D\ \mathsf{in}\ R\ \mathsf{else}\ R'$ with $\mathsf{fst}(D) \Downarrow_\Sigma M_1$ and $Q'_1 = \mathsf{fst}(R)\{M_1/x\}$. By hypothesis 2 and Lemma 5, $\mathsf{snd}(D) \Downarrow_\Sigma M_2$ for some M_2. We take $Q' = R\{\mathsf{diff}[M_1, M_2]/x\}$, so that $\mathsf{fst}(Q') = Q'_1$ and $Q \to Q'$.

(Red Fun 2): We have $Q = \mathsf{let}\ x = D\ \mathsf{in}\ R\ \mathsf{else}\ R'$ with $\mathsf{fst}(D) \Downarrow_\Sigma \mathsf{fail}$ and $Q'_1 = \mathsf{fst}(R')$. By hypothesis 2, $\mathsf{snd}(D) \Downarrow_\Sigma \mathsf{fail}$. We obtain $Q \to Q'$ for $Q' = R'$.
(Red Repl): We have $Q =!\, R$ and $Q'_1 = \mathsf{fst}(R) \mid !\, \mathsf{fst}(R)$. We take $Q' = R \mid !\, R$, so that $\mathsf{fst}(Q') = Q'_1$ and $Q \to Q'$.

To conclude, we take the biprocess $P' = C[Q']$ and the reduction $P \to P'$. □

C From Equational Theories to Rewrite Rules

In this section, we explain in detail our we adapted the algorithm from [9] for generating a signature Σ' with the empty equational theory that *models* a signature Σ with an equational theory

We consider an auxiliary rewriting system on terms, $\mathcal{S}$, that defines partial normal forms. The rules of $\mathcal{S}$ do not contain names and do not have a single variable on the left-hand side. We say that a term is irreducible by $\mathcal{S}$ when none of the rewrite rules of $\mathcal{S}$ applies to it; we say that the set of terms $\mathcal{M}$ is in normal form relatively to $\mathcal{S}$ and Σ, and write $\mathrm{nf}_{\mathcal{S},\Sigma}(\mathcal{M})$, if and only if all terms of $\mathcal{M}$ are irreducible by $\mathcal{S}$ and, for all subterms N_1 and N_2 of terms of $\mathcal{M}$, if $N_1 =_\Sigma N_2$ then $N_1 = N_2$. Typically, while a term might have several partial normal forms, any equal (sub)terms in $\mathcal{M}$ modulo the equational theory are in the same normal form in $\mathrm{nf}_{\mathcal{S},\Sigma}(\mathcal{M})$. We extend the definition of $\mathrm{nf}_{\mathcal{S},\Sigma}(\cdot)$ to sets of processes: $\mathrm{nf}_{\mathcal{S},\Sigma}(\mathcal{P})$ if and only if the set of terms that appear in processes in $\mathcal{P}$ is in normal form. At last, we define $\mathrm{addeval}(U_1, \ldots, U_n)$ as the tuple of term evaluation obtained by adding eval before each function symbol of $U_1, \ldots, U_n$.

Definition 3. *Let Σ and Σ' be two signature on the same function symbols. We say that Σ' models Σ if and only if*

1. *The equational theory of Σ' is syntactic equality: $M =_{\Sigma'} N$ if and only if $M = N$.*
2. *The constructors of Σ' are the constructors of Σ; their definition $\mathrm{def}_{\Sigma'}(f)$ contains the rule $f(x_1, \ldots, x_n) \to f(x_1, \ldots, x_n)$, the rewrite rules corresponding to possible failure,* i.e. *for all $i \in \{1 \ldots n\}$, $\mathsf{f}(u_1, \ldots, u_{i-1}, \mathsf{fail}, u_{i+1}, \ldots, u_n) \to \mathsf{fail}$, and perhaps other rules such that there exists a rewriting system $\mathcal{S}$ on terms that satisfies the following properties:*
   - *S1. If $M \to N$ is in $\mathcal{S}$, then $M =_\Sigma N$.*
   - *S2. If $\mathrm{nf}_{\mathcal{S},\Sigma}(\mathcal{M})$, then for any term M there exists M' such that $M' =_\Sigma M$ and $\mathrm{nf}_{\mathcal{S},\Sigma}(\mathcal{M} \cup \{M'\})$.*
   - *S3. If $f(N_1, \ldots, N_n) \to N \mid\mid \phi$ is in $\mathrm{def}_{\Sigma'}(f)$, then $f(N_1, \ldots, N_n) =_\Sigma N$ and $\phi = \top$*
   - *S4. If $f(M_1, \ldots, M_n) =_\Sigma M$ and $\mathrm{nf}_{\mathcal{S},\Sigma}(\{M_1, \ldots, M_n, M\})$, then there exist σ and $f(N_1, \ldots, N_n) \to N$ in $\mathrm{def}_{\Sigma'}(f)$ such that $M = \sigma N$ and $M_i = \sigma N_i$ for all $i \in \{1, \ldots, n\}$.*
3. *The destructors of Σ' are the destructors of Σ, with a rule $\mathsf{g}(U'_1, \ldots, U'_n) \to U' \mid\mid \sigma\phi \wedge \phi'$ in $\mathrm{def}_{\Sigma'}(g)$ for each $\mathsf{g}(U_1, \ldots, U_n) \to U \mid\mid \phi$ in $\mathrm{def}_\Sigma(g)$ and each $\mathrm{addeval}(U_1, \ldots, U_n, U) \Downarrow'_{\Sigma'} ((U'_1, \ldots, U'_n, U'), \sigma, \phi')$.*

Stress that, in Item 3, since $U_1, \ldots, U_n$ do not contain any destructor and the side conditions of constructors are always $\top$, ϕ' is necessarily $\top$.

Note that, in Σ', the side conditions of destructors still rely on the equational theory of Σ. Moreover, even if the semantics of the function symbols of Σ' are all defined by rewrite rules, we still consider the constructors of Σ as the constructors in Σ'.

Example 7. Consider the signature Σ that has the constructors senc and sdec with the equations

$$\mathsf{sdec}(\mathsf{senc}(x, y), y) = x \qquad \mathsf{senc}(\mathsf{sdec}(x, y), y) = x$$

In Σ', we adopt the rewrite rules:

$$\begin{array}{ll} \mathsf{sdec}(x, y) \to \mathsf{sdec}(x, y) & \mathsf{senc}(x, y) \to \mathsf{senc}(x, y) \\ \mathsf{sdec}(\mathsf{senc}(x, y), y) \to x & \mathsf{senc}(\mathsf{sdec}(x, y), y) \to x \\ \mathsf{sdec}(\mathsf{fail}, u) \to \mathsf{fail} & \mathsf{senc}(\mathsf{fail}, u) \to \mathsf{fail} \\ \mathsf{sdec}(u, \mathsf{fail}) \to \mathsf{fail} & \mathsf{senc}(u, \mathsf{fail}) \to \mathsf{fail} \end{array}$$

We have that Σ' models Σ for the rewriting system $\mathcal{S}$ with rules $\mathsf{sdec}(\mathsf{senc}(x, y), y) \to x$ and $\mathsf{senc}(\mathsf{sdec}(x, y), y) \to x$, and a single normal form for every term.

D Clauses for the Protocol and the Attacker

This section describes the complete set of clauses for the protocol and the attacker. In particular, the following clauses represent the capabilities of the attacker:

$$\begin{array}{ll} \text{For each } a \in \mathit{fn}(P_0), \mathrm{att}'(a[\,], a[\,]) & \text{(Rinit)} \\ \text{For some } b \text{ that does not occur in } P_0, \mathrm{att}'(b[x], b[x]) & \text{(Rn)} \\ \mathrm{att}'(\mathsf{fail}, \mathsf{fail}) & \text{(Rfail)} \\ \text{For each function } \mathsf{h}, \text{ for each pair of rewrite rules} & \\ \quad \mathsf{h}(U_1, \ldots, U_n) \to U \mid\mid \phi \text{ and } \mathsf{h}(U'_1, \ldots, U'_n) \to U' \mid\mid \phi' & \\ \quad \text{in } \mathrm{def}_{\Sigma'}(\mathsf{h}) \text{ (after renaming of variables),} & \text{(Rf)} \\ \quad \mathrm{att}'(U_1, U'_1) \wedge \ldots \wedge \mathrm{att}'(U_n, U'_n) \wedge \mathrm{formula}(\phi \wedge \phi') \to \mathrm{att}'(U, U') & \\ \mathrm{msg}'(x, y, x', y') \wedge \mathrm{att}'(x, x') \to \mathrm{att}'(y, y') & \text{(Rl)} \\ \mathrm{att}'(x, x') \wedge \mathrm{att}'(y, y') \to \mathrm{msg}'(x, y, x', y') & \text{(Rs)} \\ \mathrm{att}'(x, x') \wedge \to \mathrm{input}'(x, x') & \text{(Ri)} \\ \mathrm{input}'(x, x') \wedge \mathrm{msg}'(x, z, y', z') \wedge \mathrm{formula}(x' \neq_\Sigma y') \to \mathrm{bad} & \text{(Rcom)} \\ \mathrm{att}'(x, \mathsf{fail}) \to \mathrm{bad} & \text{(Rfailure)} \end{array}$$

plus symmetric clauses (Rcom′) and (Rfailure′) obtained from (Rcom) and (Rfailure) by swapping the first and second arguments of att' and input', and the first and third arguments of msg'.

The clauses modelling the behaviour of the biprocess are generated by the translation $[\![P]\!]\rho s H$ of a biprocess P, where ρ is an environment that associates a pair of

patterns with each name and variable, s is a sequence of patterns, and H is a sequence of facts. $[\![P]\!]\rho sH$ is defined as follows:

$$
\begin{aligned}
&[\![0]\!]\rho sH = \emptyset\\
&[\![!\,P]\!]\rho sH = [\![P]\!]\rho(s,i)H, \text{ where } i \text{ is a fresh variable}\\
&[\![P \mid Q]\!]\rho sH = [\![P]\!]\rho sH \cup [\![Q]\!]\rho sH\\
&[\![(\nu a)P]\!]\rho sH = [\![P]\!](\rho[a \mapsto (a[s], a[s])])sH\\
&[\![M(x).P]\!]\rho sH =\\
&\qquad [\![P]\!](\rho[x \mapsto (x', x'')])(s, x', x'')(H \wedge \mathrm{msg}'(\rho(M)_1, x', \rho(M)_2, x''))\\
&\quad \cup \{H \to \mathrm{input}'(\rho(M)_1, \rho(M)_2)\}\\
&\quad \text{where } x' \text{ and } x'' \text{ are fresh variables}\\
&[\![\overline{M}\langle N\rangle.P]\!]\rho sH = [\![P]\!]\rho sH \cup \{H \to \mathrm{msg}'(\rho(M)_1, \rho(N)_1, \rho(M)_2, \rho(N)_2)\}\\
&[\![\mathsf{let}\ x = D\ \mathsf{in}\ P\ \mathsf{else}\ Q]\!]\rho sH =\\
&\quad \bigcup\{[\![P]\!]((\sigma\rho)[x \mapsto (p, p')])(\sigma s, p, p')(\sigma H \wedge \mathrm{formula}(\phi))\\
&\qquad \mid (\rho(D)_1, \rho(D)_2) \Downarrow' ((p, p'), \sigma, \phi)\}\\
&\quad \cup \bigcup\{[\![Q]\!](\sigma\rho)(\sigma s)(\sigma H \wedge \mathrm{formula}(\phi)) \mid (\rho(D)_1, \rho(D)_2) \Downarrow' ((\mathsf{fail}, \mathsf{fail}), \sigma, \phi)\}\\
&\quad \cup \{\sigma H \wedge \mathrm{formula}(\phi) \to \mathrm{bad} \mid (\rho(D)_1, \rho(D)_2) \Downarrow' ((p, \mathsf{fail}), \sigma, \phi)\}\\
&\quad \cup \{\sigma H \wedge \mathrm{formula}(\phi) \to \mathrm{bad} \mid (\rho(D)_1, \rho(D)_2) \Downarrow' ((\mathsf{fail}, p'), \sigma, \phi)\}
\end{aligned}
$$

E Proof of the Automatic Simplification

This appendix provides the proofs of the results announced in Section 5.

E.1 Preliminary Lemmas

Lemma 6. *Let P, Q be processes. Suppose that, for all substitutions σ closing for P and Q, $\sigma P \approx \sigma Q$. Then, for all contexts C closing for P and Q, $C[P] \approx C[Q]$.*

Proof. This lemma is fairly standard in process calculi. For example, [3, Appendix C.3] shows a similar result for the spi calculus. We give a proof for our calculus.

We rely on Definition 1, and use a relation $\mathcal{R}$ defined by $P_0 \mathcal{R} P_0'$ if and only if

$$P_0 \approx C'[P_1, \ldots, P_n] \text{ and } P_0' \approx C'[P_1', \ldots, P_n']$$

for some context C' such that no hole of C' is in evaluation position and for $i = 1, \ldots, n$, for all substitutions σ closing for P_i and P_i', $\sigma P_i \approx \sigma P_i'$.

We have that, for contexts C closing for P and Q, $C[P] \mathcal{R} C[Q]$. Indeed, if the hole of C is not in evaluation position, $C[P] \approx C[P]$, $C[Q] \approx C[Q]$, and for all substitutions σ closing for P and Q, $\sigma P \approx \sigma Q$. If the hole of C is in evaluation position, P and Q are closed, so $P \approx Q$ by hypothesis, hence $C[P] \approx C[Q]$, so letting $P_0 = C[P]$ and

$P'_0 = C' = C[Q]$ (C' is a context with no hole), we have $P_0 \approx C'$, $P'_0 \approx C'$, hence $C[P] \mathcal{R} C[Q]$.

We show that $\mathcal{R}$ satisfies the three conditions of Definition 1. Moreover $\mathcal{R}$ is symmetric, so we can then conclude that $\mathcal{R} \subseteq \approx$, which implies the desired equivalence.

Condition 3 of Definition 1: Suppose $P_0 \mathcal{R} P'_0$, and let C be an evaluation context. $P_0 \mathcal{R} P'_0$ implies that there exists a context C', some processes $P_1, \ldots, P_n, P'_1, \ldots, P'_n$ such that $P_0 \approx C'[P_1, \ldots, P_n]$ and $P'_0 \approx C'[P'_1, \ldots, P'_n]$. Hence, we have $C[P_0] \approx C[C'[P_1, \ldots, P_n]]$ and $C[P'_0] \approx C[C'[P'_1, \ldots, P'_n]]$ by Condition 3 of Definition 1. Since no hole of C' is in evaluation position, then no hole of $C[C']$ is also in evaluation position. Hence, we deduce that $C[P_0] \mathcal{R} C[P'_0]$.

Condition 2 of Definition 1: We first show that, if $C'[P_1, \ldots, P_n] \equiv Q_0$ where C' is any context such that no hole of C' is in evaluation position, then $C'[P'_1, \ldots, P'_n] \equiv C''[P'_1, \ldots, P'_n]$ and $Q_0 = C''[P_1, \ldots, P_n]$ for some context C'' such that no hole of C'' is in evaluation position. The proof is done by induction on the derivation of $C'[P_1, \ldots, P_n] \equiv Q_0$.

Next, we show that, if $C'[P_1, \ldots, P_n] \to Q_0$ where C' is any context such that no hole of C' is in evaluation position, and for $i = 1, \ldots, n$, for all substitutions σ closing for P_i and P'_i, $\sigma P_i \approx \sigma P'_i$, then $C'[P'_1, \ldots, P'_n] \to Q'_0 \approx C''[Q'_1, \ldots, Q'_{n'}]$ and $Q_0 = C''[Q_1, \ldots, Q_{n'}]$ for some context C'' and processes Q'_0, Q'_1, …, $Q'_{n'}$ such that no hole of C'' is in evaluation position and for $i = 1, \ldots, n'$, for all substitutions σ closing for Q_i and Q'_i, $\sigma Q_i \approx \sigma Q'_i$. The proof is done by induction on the derivation of $C'[P_1, \ldots, P_n] \to Q_0$.

- Case (Red I/O): We have $C'[P_1, \ldots, P_n] = \overline{N}\langle M \rangle.Q \mid N'(x).P \to Q_0 = Q \mid P\{^M/_x\}$ with $N =_\Sigma N'$. Since no hole of C' is in evaluation position, we have $Q = C_1[P_i(i \in S)]$, $P = C_2[P_i(i \notin S)]$ for some contexts C_1, C_2 and some subset S of $\{1, \ldots, n\}$, and $C'[P'_1, \ldots, P'_n] = \overline{N}\langle M \rangle.C_1[P'_i(i \in S)] \mid N'(x).C_2[P_i(i \notin S)] \to C_1[P'_i(i \in S)] \mid C_2[P'_i(i \notin S)]\{^M/_x\}$. We let Q_i and Q'_i be the processes P_i and P'_i for $i \in S$ and $P_i\{^M/_x\}$ and $P'_i\{^M/_x\}$ for $i \notin S$, such that the corresponding hole of C_1 or C_2 is not in evaluation position. We have for all substitutions σ closing for Q_i and Q'_i, $\sigma Q_i \approx \sigma Q'_i$, since that property is preserved by instantiation. We let $C'' = C_1[P_i(i \in S$, in evaluation position)$][\,] \mid C_2\{^M/_x\}[P_i\{^M/_x\}(i \notin S$, in evaluation position)$][\,]$ where only the holes not in evaluation position remain. We let $Q'_0 = C_1[P'_i(i \in S)] \mid C_2[P'_i(i \notin S)]\{^M/_x\}$. We have $Q_0 = C''[Q_1, \ldots, Q_{n'}]$ and $C'[P'_1, \ldots, P'_n] \to Q'_0 \approx C''[Q'_1, \ldots, Q'_{n'}]$.
- Case (Red Fun 1): As in the case (Red I/O), the holes in the in branch are instantiated. The holes in the $else$ branch disappear. The holes that become in evaluation position are handled as in (Red I/O).
- Case (Red Fun 2): The holes in the in branch disappear. The holes that become in evaluation position are handled as in (Red I/O).
- Case (Red Repl): The holes are duplicated.
- Cases (Red Par) and (Red Res): these cases follow immediately from the induction hypothesis.
- Case (Red $\equiv$): we use the property shown above for $\equiv$ and the induction hypothesis.

Suppose that $P_0 \rightarrow^* Q_0$ and $P_0 \ \mathcal{R} \ P'_0$. We have $P_0 \approx C'[P_1, \ldots, P_n]$ and $P'_0 \approx C'[P'_1, \ldots, P'_n]$ for some context C' and processes $P_1, \ldots, P_n, P'_1, \ldots, P'_n$ such that no hole of C' is in evaluation position and for $i = 1, \ldots, n$, for all substitutions σ closing for P_i and P'_i, $\sigma P_i \approx \sigma P'_i$. By Condition 2 of Definition 1, $C'[P_1, \ldots, P_n] \rightarrow^* Q''_0$ and $Q_0 \approx Q''_0$ for some Q''_0. By the property above, $C'[P'_1, \ldots, P'_n](\rightarrow\approx)^* C''[Q'_1, \ldots, Q'_{n'}]$ and $Q''_0 = C''[Q_1, \ldots, Q_{n'}]$ for some context C'' and processes $Q'_1, \ldots, Q'_{n'}$ such that no hole of C'' is in evaluation position and for $i = 1, \ldots, n'$, for all substitutions σ closing for Q_i and Q'_i, $\sigma Q_i \approx \sigma Q'_i$. By Condition 2 of Definition 1 again, $P'_0 \rightarrow^* Q'_0$ and $C''[Q'_1, \ldots, Q'_{n'}] \approx Q'_0$ for some Q'_0. Hence $Q_0 \approx C''[Q_1, \ldots, Q_{n'}]$ and $Q'_0 \approx C''[Q'_1, \ldots, Q'_{n'}]$, so $Q_0 \ \mathcal{R} \ Q'_0$.

Condition 1 of Definition 1: Let $P \downarrow^0_M$ if and only if $P = C[\overline{M'}\langle N\rangle.R]$ for some evaluation context C that does not bind $fn(M)$ and $M =_\Sigma M'$. We first notice that, if $C'[P_1, \ldots, P_n]\downarrow^0_M$ where no hole of C' is in evaluation position, then we have $C'[P'_1, \ldots, P'_n]\downarrow^0_M$, since the difference between $C'[P_1, \ldots, P_n]$ and $C'[P'_1, \ldots, P'_n]$ is only is non-evaluation positions.

Suppose that $P_0 \downarrow_M$ and $P_0 \ \mathcal{R} \ P'_0$. We have $P_0 \approx C'[P_1, \ldots, P_n]$ and $P'_0 \approx C'[P'_1, \ldots, P'_n]$ for some context C' and processes $P_1, \ldots, P_n, P'_1, \ldots, P'_n$ such that no hole of C' is in evaluation position and for $i = 1, \ldots, n$, for all substitutions σ closing for P_i and P'_i, $\sigma P_i \approx \sigma P'_i$. By Condition 1 of Definition 1, $C'[P_1, \ldots, P_n]\downarrow_M$, that is, $C'[P_1, \ldots, P_n] \rightarrow^* \downarrow^0_M$. By the properties proved regarding Condition 2, $C'[P'_1, \ldots, P'_n](\rightarrow\approx)^* C''[Q'_1, \ldots, Q'_{n'}]$ and $C''[Q_1, \ldots, Q_{n'}]\downarrow^0_M$ for some context C'' and processes $Q'_1, \ldots, Q'_{n'}$ such that no hole of C'' is in evaluation position. Hence $C''[Q'_1, \ldots, Q'_{n'}]\downarrow^0_M$, so by Conditions 1 and 2 of Definition 1, $C'[P'_1, \ldots, P'_n]\downarrow_M$. Since $C'[P'_1, \ldots, P'_n] \approx P'_0$, we conclude that $P'_0\downarrow_M$. □

Relying on the previous lemma, we now show that the transformations on the processes, applied by the functions $merge$ and $simpl$, preserve observational equivalence.

Lemma 7. *Let P, Q, R be processes. Let a be a name, x a variable, M a term, and D a term evaluation. We have that:*

1. *If $a \notin fn(P)$, then for all contexts C closing for P, $C[P] \approx C[(\nu a)P]$.*
2. *For all contexts C closing for $!\,P$, $C[!\,P] \approx C[!\,!\,P]$.*
3. *For all contexts C closing for $P\{^M/_x\}$, $C[P\{^M/_x\}] \approx C[\mathsf{let}\ x = M\ \mathsf{in}\ P\ \mathsf{else}\ Q]$.*
4. *For all contexts C closing for P, $C[P] \approx C[\mathsf{let}\ x = \mathsf{fail}\ \mathsf{in}\ Q\ \mathsf{else}\ P]$.*
5. *For all contexts C closing for $P_0 = \mathsf{let}\ x = D\ \mathsf{in}\ \mathsf{fst}'(P)\ \mathsf{else}\ \mathsf{snd}'(P)$, $C[P_0] \approx C[\mathsf{let}\ x = \mathrm{eval}\ \mathsf{catchfail}(D)\ \mathsf{in}\ P\{^{\mathrm{eval}\ \mathsf{letin}(x, D_1, D_2)}/_{\mathsf{diff}'[D_1, D_2]}\}\ \mathsf{else}\ 0]$.*
6. *For all contexts C closing for $P \mid Q$, $C[P \mid Q] \approx C[Q \mid P]$.*
   For all contexts C closing for $(P \mid Q) \mid R$, $C[(P \mid Q) \mid R] \approx C[P \mid (Q \mid R)]$.
7. *If $x \notin fv(P) \cup fv(Q)$ then for all contexts C closing for $\mathsf{let}\ x = D\ \mathsf{in}\ P\ \mathsf{else}\ Q$, $C[\mathsf{let}\ x = D\ \mathsf{in}\ P\ \mathsf{else}\ Q] \approx C[\mathsf{let}\ x = \mathsf{notfail}(D)\ \mathsf{in}\ Q\ \mathsf{else}\ P]$.*

Proof. We first prove this result for processes that do not contain diff. By applying the obtained result twice, once for the component fst and once for the component snd, we obtain the same equivalence for processes that may contain diff. By Lemma 6, it is enough to show:

1. If $a \notin fn(P)$, then $P \approx (\nu a)P$.
2. $!\,P \approx !\,!\,P$.
3. $P\{^{M}/_{x}\} \approx \mathsf{let}\ x = M\ \mathsf{in}\ P\ \mathsf{else}\ Q$.
4. $P \approx \mathsf{let}\ x = \mathsf{fail}\ \mathsf{in}\ Q\ \mathsf{else}\ P$.
5. $\mathsf{let}\ x = D\ \mathsf{in}\ \mathsf{fst}'(P)\ \mathsf{else}\ \mathsf{snd}'(P) \approx$
   $\mathsf{let}\ x = \mathrm{eval}\ \mathsf{catchfail}(D)\ \mathsf{in}\ P\{^{\mathrm{eval}\ \mathsf{letin}(x,D_1,D_2)}/_{\mathsf{diff}'[D_1,D_2]}\}\ \mathsf{else}\ 0$.
6. $P \mid Q \approx Q \mid P$ and $(P \mid Q) \mid R \approx P \mid (Q \mid R)$.
7. $\mathsf{let}\ x = D\ \mathsf{in}\ P\ \mathsf{else}\ Q \approx \mathsf{let}\ x = \mathsf{equals}(D, \mathsf{fail})\ \mathsf{in}\ Q\ \mathsf{else}\ P$

where P, Q, R are closed processes, M and D are ground terms and term evaluations, except for P in Item 3 in which $fv(P) \subseteq \{x\}$, for Q in Item 4 in which $fv(Q) \subseteq \{x\}$, and for P in Item 5 in which $fv(\mathsf{fst}'(P)) \subseteq \{x\}$ and $fv(\mathsf{snd}'(P)) = \emptyset$. Item 6 is obvious since $\equiv \subseteq \approx$. Note that in Item 7, since P and Q are closed, $x \notin fv(P) \cup fv(Q)$. Let us prove the other cases by relying on Definition 1.

Items 3 and 4: we use a relation $\mathcal{R}$ defined by $P_0\ \mathcal{R}\ P_0'$ if and only if

$$P_0 = C[P_1] \text{ and } P_0' = C[P_1']$$

or

$$P_0 = C[P_1'] \text{ and } P_0' = C[P_1]$$

or

$$P_0 = P_0'$$

for some evaluation context C and processes P_1, P_1' such that $P_1\ \mathcal{R}_1\ P_1'$ where

- in Item 3, $\mathcal{R}_1$ is defined by $P\{^{M}/_{x}\}\ \mathcal{R}_1\ \mathsf{let}\ x = M\ \mathsf{in}\ P\ \mathsf{else}\ Q$ for all P, Q, x, M;
- in Item 4, $\mathcal{R}_1$ is defined by $P\ \mathcal{R}_1\ \mathsf{let}\ x = \mathsf{fail}\ \mathsf{in}\ Q\ \mathsf{else}\ P$ for all P, Q, x.

We show that $\mathcal{R}$ satisfies the three conditions of Definition 1. Moreover $\mathcal{R}$ is symmetric, so we can then conclude that $\mathcal{R} \subseteq \approx$, which implies the desired equivalences.

- $\mathcal{R}$ obviously satisfies Condition 3 of Definition 1.
- To show Condition 2 of Definition 1, we show that, if $P_0 \to Q_0$ and $P_0\ \mathcal{R}\ P_0'$, then there exists Q_0' such that $Q_0\ \mathcal{R}\ Q_0'$ and $P_0' \to^* Q_0'$.
  If $P_0 = C[P_1]$, $P_0' = C[P_1']$, and $P_1\ \mathcal{R}_1\ P_1'$, then we first reduce P_1' into P_1, transforming $\mathsf{let}\ x = M\ \mathsf{in}\ P\ \mathsf{else}\ Q$ into $P\{M/x\}$ (Item 3) and $\mathsf{let}\ x = \mathsf{fail}\ \mathsf{in}\ Q\ \mathsf{else}\ P$ into P (Item 4). In this case, we have $P_0' = C[P_1'] \to C[P_1] = P_0 \to Q_0$. Taking $Q_0' = Q_0$, we have $Q_0 \mathcal{R} Q_0'$ and $P_0' \to^* Q_0'$.
  If $P_0 = C[P_1']$, $P_0' = C[P_1]$, and $P_1 \mathcal{R}_1 P_1'$, we show the following.
  1. If $C[P_1'] \equiv P'$, then $P' = C'[P_1']$ and $C[P_1] \equiv C'[P_1]$ for some evaluation context C', by induction on the derivation of $C[P_1'] \equiv P'$.
  2. If $C[P_1'] \to P'$, then either $P' = C'[P_1]$ and $C[P_1] \equiv C'[P_1]$, or $P' = C'[P_1']$ and $C[P_1] \to C'[P_1]$, for some evaluation context C'. The proof proceeds by induction of the derivation of $C[P_1'] \to P'$:
     Case (Red I/O) is impossible.
     Case (Red Fun 1) can be applied only in Item 3. In this case, $C[P_1'] = P_1' \to P_1$ and we take $C' = [\,]$.

Case (Red Fun 2) can be applied only in Item 4. In this case, $C[P_1'] = P_1' \to P_1$ and we take $C' = [\,]$.
In case (Red Par), we have $C[P_1'] = P \mid R \to Q \mid R = P'$ with $P \to Q$. First case: $R = C''[P_1']$, $C = P \mid C''$. Then $P' = Q \mid C''[P_1']$. Let $C' = Q \mid C''$. Then $P' = C'[P_1']$, and $C[P_1] = P \mid C''[P_1] \to Q \mid C''[P_1] = C'[P_1]$. Second case: $P = C''[P_1']$, $C = C'' \mid R$. Then $C''[P_1'] \to Q$. By induction hypothesis, either $Q = C'''[P_1]$ and $C''[P_1] \equiv C'''[P_1]$, or $Q = C'''[P_1']$ and $C''[P_1] \to C'''[P_1]$, for some evaluation context C'. Let $C' = C''' \mid R$. Either $P' = Q \mid R = C'''[P_1] \mid R = C'[P_1]$ and $C[P_1] = C''[P_1] \mid R \equiv C'''[P_1] \mid R = C'[P_1]$, or $P' = Q \mid R = C'''[P_1'] \mid R = C'[P_1']$ and $C[P_1] = C''[P_1] \mid R \to C'''[P_1] \mid R = C'[P_1]$.
Case (Red Res) follows by induction hypothesis, similarly to the second case of (Red Par).
Case (Red $\equiv$) follows using the property above for $\equiv$ and the induction hypothesis.

We have $C[P_1'] = P_0 \to Q_0$, so either $Q_0 = C'[P_1]$ and $C[P_1] \equiv C'[P_1]$, or $Q_0 = C'[P_1']$ and $C[P_1] \to C'[P_1]$, for some evaluation context C'. We let $Q_0' = C'[P_1]$. In the first case, we have $Q_0 = Q_0'$ so $Q_0 \mathcal{R} Q_0'$ and $P_0' = C[P_1] \to C'[P_1] = Q_0'$. In the second case, we have $Q_0 = C'[P_1']$ and $Q_0' = C'[P_1]$ so $Q_0 \mathcal{R} Q_0'$ and $P_0' = C[P_1] \to C'[P_1] = Q_0'$.
If $P_0 = P_0'$, let $Q_0' = Q_0$. We have $Q_0 \mathcal{R} Q_0'$ and $P_0' = P_0 \to Q_0 = Q_0'$.

– To show Condition 1, using Condition 2, it is enough to show that, if $P_0 \mathcal{R} P_0'$ and $P_0 \equiv C'[\overline{M'}\langle N\rangle.R]$ for some evaluation context C' that does not bind $fn(M)$ and $M =_\Sigma M'$, then $P_0' \downarrow_M$.
If $P_0 = C[P_1]$, $P_0' = C[P_1']$, and $P_1 \mathcal{R}_1 P_1'$, we have $P_0' = C[P_1'] \to C[P_1] = P_0$ hence $P_0' \downarrow_M$.
If $P_0 = C[P_1']$, $P_0' = C[P_1]$, and $P_1 \mathcal{R}_1 P_1'$, we have $P_0 \equiv C'[\overline{M'}\langle N\rangle.R]$. By Property 1 shown above for $\equiv$, $C'[\overline{M'}\langle N\rangle.R] = C''[P_1']$ and $C[P_1] \equiv C''[P_1]$ for some C''. Hence there is an evaluation context C''' that does not bind $fn(M)$ such that $C'[\overline{M'}\langle N\rangle.R] = C''[P_1'] = C'''[P_1', \overline{M'}\langle N\rangle.R]$, thus we deduce that $P_0' \equiv C'''[P_1, \overline{M'}\langle N\rangle.R]$ and so $P_0' \downarrow_M$.
If $P_0 = P_0'$, the result is obvious.

Item 5: we define the relation $\mathcal{R}$ by $P_0 \mathcal{R} P_0'$ if and only if

$P_0 = C[\mathsf{let}\ x = D\ \mathsf{in}\ \mathsf{fst}'(P)\ \mathsf{else}\ \mathsf{snd}'(P)]$ and

$P_0' = C[\mathsf{let}\ x = \mathrm{eval}\ \mathsf{catchfail}(D)\ \mathsf{in}\ P\{^{\mathrm{eval}\ \mathsf{letin}(x,D_1,D_2)}/_{\mathsf{diff}'[D_1,D_2]}\}\ \mathsf{else}\ 0]$

for some evaluation context C, variable x, term evaluation D, and process P

or

$P_0 = \mathsf{fst}'(P)\{^M/_x\}$ and $P_0' = P\{^{\mathrm{eval}\ \mathsf{letin}(M,D_1,D_2)}/_{\mathsf{diff}'[D_1,D_2]}\}$ for some P and M

or

$P_0 = \mathsf{snd}'(P)$ and $P_0' = P\{^{\mathrm{eval}\ \mathsf{letin}(\mathsf{c_{fail}},D_1,D_2)}/_{\mathsf{diff}'[D_1,D_2]}\}$ for some P

or the symmetric obtained by swapping P_0 and P_0'. We show $\mathsf{let}\ x = D\ \mathsf{in}\ \mathsf{fst}'(P)\ \mathsf{else}\ \mathsf{snd}'(P) \approx \mathsf{let}\ x = \mathrm{eval}\ \mathsf{catchfail}(D)\ \mathsf{in}\ P\{^{\mathrm{eval}\ \mathsf{letin}(x,D_1,D_2)}/_{\mathsf{diff}'[D_1,D_2]}\}\ \mathsf{else}\ 0$, by

proving that $\mathcal{R}$ satisfies the three conditions of Definition 1, similarly to the proof we have done for Items 3 and 4.

Item 1: we define the relation $\mathcal{R}$ by $P_0 \mathcal{R} P_0'$ if and only if $(\nu a)P_0 \equiv P_0'$ or $(\nu a)P_0' \equiv P_0$ for some $a \notin fn(P_0)$ or $a \notin fn(P_0')$ respectively. We show $P \approx (\nu a)P$, by proving that $\mathcal{R}$ satisfies the three conditions of Definition 1.

Item 2: we define the relation $\mathcal{R}$ by $P_0 \mathcal{R} P_0'$ if and only if

$$P_0 \equiv C[!\,P] \text{ and } P_0' \equiv C[!\,!\,P \mid !\,P \mid \ldots \mid !\,P]$$

or

$$P_0 \equiv C[!\,!\,P \mid !\,P \mid \ldots \mid !\,P] \text{ and } P_0' \equiv C[!\,P]$$

for some evaluation context C and process P. We show $!\,P \approx !\,!\,P$, by proving that $\mathcal{R}$ satisfies the three conditions of Definition 1.

The proof follows a strategy similar to the proof of Items 3 and 4. For Items 1 and 2, some details are however more complex, because the structural equivalence may modify $(\nu a)P$ and $!\,!\,P \mid !\,P \mid \ldots \mid !\,P$. For instance, to show Condition 2 of Definition 1 for Item 1, we need to show that, if $P_0 \to Q_0$ and $P_0 \equiv (\nu a)P_0'$, then there exists Q_0' such that $Q_0 \equiv (\nu a)Q_0'$ and $P_0' \to Q_0'$. Such a result is fairly standard in process calculi and can be proved by showing a series of lemmas decomposing reduction of $(\nu a)P$ and $P \mid Q$, using a labeled semantics.

Item 7: we define the relation $\mathcal{R}$ by $P_0 \mathcal{R} P_0'$ if and only if

$P_0 = C[\mathsf{let}\ x = D\ \mathsf{in}\ P\ \mathsf{else}\ Q]$ and
$P_0' = C[\mathsf{let}\ x = \mathsf{notfail}(D)\ \mathsf{in}\ Q\ \mathsf{else}\ P]$
for some evaluation context C, variable x, term evaluation D, and processes P, Q such that $x \notin fv(P) \cup fv(Q)$

or

$$P_0 = P_0'$$

or the symmetric obtained by swapping P_0 and P_0'. We show $\mathsf{let}\ x = D\ \mathsf{in}\ P\ \mathsf{else}\ Q \approx \mathsf{let}\ x = \mathsf{notfail}(D)\ \mathsf{in}\ Q\ \mathsf{else}\ P$, by proving that $\mathcal{R}$ satisfies the three conditions of Definition 1, similarly to the proof we have done for Items 3 and 4. It relies on the fact that if $D \Downarrow_\Sigma M$ then $\mathsf{notfail}(D) \Downarrow_\Sigma \mathsf{fail}$, and if $D \Downarrow_\Sigma \mathsf{fail}$ then $\mathsf{notfail}(D) \Downarrow_\Sigma \mathsf{c_{fail}}$ which is a message. □

E.2 Proof of Lemma 3

Lemma 3 (merge). *Let P and P' be two biprocesses. If $merge(P, P') = Q$, then:*

- *for all contexts C closing for P, $C[P] \approx C[\mathsf{fst}'(Q)]$;*
- *for all contexts C closing for P', $C[P'] \approx C[\mathsf{snd}'(Q)]$.*

Proof. Let P_0, P'_0 be two biprocesses. We prove the result by induction on P_0 and P'_0. We do a case analysis on the rules of Figure 4:

Case (Mnil)*:* Trivial.

Case (Mout)*:* We have $P_0 = \overline{M}\langle N\rangle.P$ and $P'_0 = \overline{M'}\langle N'\rangle.P'$. Let C be a context closing for P_0. We show that $C[P_0] \approx C[\mathsf{fst}'(merge(P_0, P'_0)]$, that is, $C[\overline{M}\langle N\rangle.P] \approx C[\mathsf{let}\ x = M\ \mathsf{in}\ \mathsf{let}\ x' = N\ \mathsf{in}\ \overline{x}\langle x'\rangle.\mathsf{fst}'(merge(P, P'))]$. Let $C_1 = C[\mathsf{let}\ x = M\ \mathsf{in}\ \mathsf{let}\ x' = N\ \mathsf{in}\ \overline{x}\langle x'\rangle.[\ ]]$. By induction hypothesis on $merge(P, P')$, $C_1[P] \approx C_1[\mathsf{fst}'(merge(P, P'))]$, so we just have to show that $C[\overline{M}\langle N\rangle.P] \approx C[\mathsf{let}\ x = M\ \mathsf{in}\ \mathsf{let}\ x' = N\ \mathsf{in}\ \overline{x}\langle x'\rangle.P]$. This follows by two applications of Lemma 7, Item 3.

By symmetry, for all contexts C closing for P'_0, $C[P'_0] \approx C[\mathsf{snd}'(merge(P_0, P'_0))]$.

Case (Min)*:* This case is similar to the case (Mout).

Case (Mpar)*:* We have $P_0 = P_1 \mid \ldots \mid P_n$ and $P'_0 = P'_1 \mid \ldots \mid P'_n$. Furthermore, there exists a permutation $(i_k)_{k=1..n}$ of $(1, \ldots, n)$ such that $merge(P_0, P'_0) = Q_1 \mid \ldots \mid Q_n$ with $Q_k = merge(P_k, P'_{i_k})$, for $k = 1..n$. Let C a context closing for P_0. Let $C_k = C[\mathsf{fst}'(Q_1) \mid \ldots \mid \mathsf{fst}'(Q_{k-1}) \mid [\ ] \mid P_{k+1} \mid \ldots \mid P_n]$. Since C is closing for P_0, we have that C_k is closing for P_k. By induction hypothesis on Q_k, we deduce that $C_k[P_k] \approx C_k[\mathsf{fst}'(Q_k)]$. Moreover, $C_k[\mathsf{fst}'(Q_k)] = C_{k+1}[P_{k+1}]$. Hence, with a simple induction on n, we deduce that $C[P_0] \approx C[\mathsf{fst}'(merge(P_0, P'_0))]$.

Using a similar proof and Lemma 7, Item 6 to permute the elements of the parallel composition, we obtain that $C[P'_0] \approx C[\mathsf{snd}'(merge(P_0, P'_0))]$, for all contexts C closing for P'_0.

Case (Mres)*:* We have $P_0 = (\nu a)P$ and $a \notin fn(P'_0)$. Let C a context closing for $(\nu a)P$. By induction hypothesis on $merge(P, P'_0)$, $C[(\nu a)P] \approx C_1[(\nu a)\mathsf{fst}'(merge(P, P'_0))]$. Hence, $C[P_0] \approx C[\mathsf{fst}'(merge(P_0, P'_0))]$.

Let C be a context closing for P'_0. Since $a \notin fn(P'_0)$, by Lemma 7, Item 1, $C[P'_0] \approx C[(\nu a)P'_0]$. By induction hypothesis on $merge(P, P'_0)$, we deduce that $C[(\nu a)P'_0] \approx C[(\nu a)\mathsf{snd}'(merge(P, P'_0))]$. Hence $C[P'_0] \approx C[\mathsf{snd}'(merge(P_0, P'_0))]$.

Case (Mrepl1)*:* We have $P_0 = !\,(\nu a_1)\ldots(\nu a_n)!\,P$ and $P'_0 = !\,P'$. Thanks to our induction hypothesis on $merge(!\,P, P'_0)$, we have that, for all contexts C closing for P_0, $C[P_0] \approx C[\mathsf{fst}'(merge(P_0, P'_0))]$.

Let C be a context closing for P'_0. We have $C[P'_0] = C[!\,P'] \approx C[!\,!\,P']$ by Lemma 7, Item 2, so $C[P'_0] \approx C[!\,(\nu a_1)\ldots(\nu a_n)!\,P']$ by n applications of Lemma 7, Item 1. Let $C_1 = C[!\,(\nu a_1)\ldots(\nu a_n)[\ ]]$. By induction hypothesis on $merge(!\,P, !\,P')$, we deduce that $C_1[!\,P'] \approx C_1[\mathsf{snd}'(merge(!\,P, !\,P'))]$, so $C[P'_0] \approx C[!\,(\nu a_1)\ldots(\nu a_n)\ \mathsf{snd}'(merge(!\,P, !\,P'))] = C[\mathsf{snd}'(merge(P_0, P'_0))]$.

Case (Mrepl2)*:* This case follows immediately by induction hypothesis.

Case (Mlet1)*:* In this case, we have $P_0 = \mathsf{let}\ x = D\ \mathsf{in}\ P_1\ \mathsf{else}\ P_2$ and $P'_0 = \mathsf{let}\ x' = D'\ \mathsf{in}\ P'_1\ \mathsf{else}\ P'_2$. Let C be a context closing for P_0. Since y is a fresh variable, we have that $C[P_0] \approx C[\mathsf{let}\ y = D\ \mathsf{in}\ P_1\{^y/_x\}\ \mathsf{else}\ P_2]$. By induction hypothesis on both $Q_1 = merge(P_1\{^y/_x\}, P'_1\{^y/_{x'}\})$ and $Q_2 = merge(P_2, P'_2)$, we obtain that $C[\mathsf{let}\ y = D\ \mathsf{in}\ P_1\{^y/_x\}\ \mathsf{else}\ P_2] \approx C[\mathsf{let}\ y = D\ \mathsf{in}\ \mathsf{fst}'(Q_1)\ \mathsf{else}\ P_2] \approx C[\mathsf{let}\ y = D\ \mathsf{in}\ \mathsf{fst}'(Q_1)\ \mathsf{else}\ \mathsf{fst}'(Q_2)]$. Since $\mathsf{fst}'(\mathsf{diff}'[D, D']) = D$, we obtain

that $C[P_0] \approx C[\mathsf{fst}'(merge(P_0, P_0'))]$. By symmetry, we obtain that, for all contexts C closing for P_0', $C[P_0'] \approx C[\mathsf{snd}'(merge(P_0, P_0'))]$.

Case (Mlet2)*:* We have $P_0 = \mathsf{let}\ x = D\ \mathsf{in}\ P_1\ \mathsf{else}\ P_2$ and $P_0' = \mathsf{let}\ x' = D'\ \mathsf{in}\ P_1'\ \mathsf{else}\ P_2'$. Let C be a context closing for P_0. Since y is a fresh variable, we have that $C[P_0] \approx C[\mathsf{let}\ y = D\ \mathsf{in}\ P_1\{^y/_x\}\ \mathsf{else}\ P_2]$. By induction hypothesis on $merge(P_1\{^y/_x\}, P_2') = Q_1$ and $merge(P_2, P_1') = Q_2$, we obtain that $C[\mathsf{let}\ y = D\ \mathsf{in}\ P_1\{^y/_x\}\ \mathsf{else}\ P_2] \approx C[\mathsf{let}\ y = D\ \mathsf{in}\ \mathsf{fst}'(Q_1)\ \mathsf{else}\ P_2] \approx C[\mathsf{let}\ y = D\ \mathsf{in}\ \mathsf{fst}'(Q_1)\ \mathsf{else}\ \mathsf{fst}'(Q_2)]$. Since $\mathsf{fst}'(\mathsf{diff}'[D, \mathsf{notfail}(D')]) = D$, we obtain that $C[P_0] \approx C[\mathsf{fst}'(merge(P_0, P_0'))]$.

Consider C a context closing for P_0'. Thanks to Lemma 7, Item 7, we have that $C[P_0'] \approx C[\mathsf{let}\ x' = \mathsf{notfail}(D')\ \mathsf{in}\ P_2'\ \mathsf{else}\ P_1']$. By inductive hypothesis on both $Q_1 = merge(P_1\{^y/_x\}, P_2')$ and $Q_2 = merge(P_2, P_1')$, we have $C[\mathsf{let}\ x' = \mathsf{notfail}(D')\ \mathsf{in}\ P_2'\ \mathsf{else}\ P_1'] \approx C[\mathsf{let}\ x' = \mathsf{notfail}(D')\ \mathsf{in}\ \mathsf{snd}'(Q_1)\ \mathsf{else}\ P_1'] \approx C[\mathsf{let}\ x' = \mathsf{notfail}(D')\ \mathsf{in}\ \mathsf{snd}'(Q_1)\ \mathsf{else}\ \mathsf{snd}'(Q_2)]$. Since $\mathsf{snd}'(\mathsf{diff}'[D, \mathsf{notfail}(D')]) = \mathsf{notfail}(D')$, we obtain that $C[P_0'] \approx C[\mathsf{snd}'(merge(P_0, P_0'))]$.

Case (Mlet3)*:* We have $P_0 = \mathsf{let}\ x = D\ \mathsf{in}\ P_1\ \mathsf{else}\ P_2$. Let C be a closing context for P_0. Since $\approx$ is closed under renaming and y is a fresh variable, we have that $C[P_0] \approx C[\mathsf{let}\ y = D\ \mathsf{in}\ P_1\{^y/_x\}\ \mathsf{else}\ P_2]$. By induction hypothesis on $Q = merge(P_1\{^y/_x\}, P_0')$ with the context $C[\mathsf{let}\ y = D\ \mathsf{in}\ [\,]\ \mathsf{else}\ P_2]$, we obtain that $C[P_0] \approx C[\mathsf{let}\ y = D\ \mathsf{in}\ \mathsf{fst}'(Q)\ \mathsf{else}\ P_2]$. Since $C[\mathsf{let}\ y = D\ \mathsf{in}\ \mathsf{fst}'(Q)\ \mathsf{else}\ P_2] = C[\mathsf{fst}'(\mathsf{let}\ y = \mathsf{diff}'[D, \mathsf{c_{fail}}]\ \mathsf{in}\ Q\ \mathsf{else}\ P_2)]$, we obtain $C[P_0] \approx C[\mathsf{fst}'(merge(P_0, P_0'))]$.

Let C be a context closing for P_0'. Since y is a fresh variable and so not a variable of P_0', by Lemma 7, Item 3, $C[P_0'] \approx C[\mathsf{let}\ y = \mathsf{c_{fail}}\ \mathsf{in}\ P_0'\ \mathsf{else}\ P_2]$. By induction hypothesis on $Q = merge(P\{^y/_x\}, P_0')$ with the context $C[\mathsf{let}\ y = \mathsf{c_{fail}}\ \mathsf{in}\ [\,]\ \mathsf{else}\ P_2]$, we obtain that $C[\mathsf{let}\ y = \mathsf{c_{fail}}\ \mathsf{in}\ P_0'\ \mathsf{else}\ P_2] \approx C[\mathsf{let}\ y = \mathsf{c_{fail}}\ \mathsf{in}\ \mathsf{snd}'(Q)\ \mathsf{else}\ P_2]$. Since $C[\mathsf{let}\ y = \mathsf{c_{fail}}\ \mathsf{in}\ \mathsf{snd}'(Q)\ \mathsf{else}\ P_2] = C[\mathsf{snd}'(\mathsf{let}\ y = \mathsf{diff}'[D, \mathsf{c_{fail}}]\ \mathsf{in}\ Q\ \mathsf{else}\ P_2)]$, we conclude that $C[P_0'] \approx C[\mathsf{snd}'(merge(P_0, P_0'))]$.

Case (Mlet4)*:* We have $P_0 = \mathsf{let}\ x = D\ \mathsf{in}\ P_1\ \mathsf{else}\ P_2$. Let C be a closing context for P_0. Since $\approx$ is closed under renaming and y is a fresh variable, we have that $C[P_0] \approx C[\mathsf{let}\ y = D\ \mathsf{in}\ P_1\{^y/_x\}\ \mathsf{else}\ P_2]$. By induction hypothesis on $Q = merge(P_2, P_0')$ with the context $C[\mathsf{let}\ y = D\ \mathsf{in}\ P_1\{^y/_x\}\ \mathsf{else}\ [\,]]$, we obtain that $C[P_0] \approx C[\mathsf{let}\ y = D\ \mathsf{in}\ P_1\{^y/_x\}\ \mathsf{else}\ \mathsf{fst}'(Q)] = C[\mathsf{fst}'(\mathsf{let}\ y = \mathsf{diff}'[D, \mathsf{fail}]\ \mathsf{in}\ P_1\{^y/_x\}\ \mathsf{else}\ Q)]$, we obtain the desired result: $C[P_0] \approx C[\mathsf{fst}'(merge(P_0, P_0'))]$.

Let C be a context closing for P_0'. Thanks to Lemma 7, Item 4, we deduce that $C[P_0'] \approx C[\mathsf{let}\ y = \mathsf{fail}\ \mathsf{in}\ P_1\{^y/_x\}\ \mathsf{else}\ P_0']$. By induction hypothesis on $merge(P_2, P_0') = Q$ with the context $C[\mathsf{let}\ y = \mathsf{fail}\ \mathsf{in}\ P_1\{^y/_x\}\ \mathsf{else}\ [\,]]$, we obtain that $C[\mathsf{let}\ y = \mathsf{fail}\ \mathsf{in}\ P_1\{^y/_x\}\ \mathsf{else}\ P_0'] \approx C[\mathsf{let}\ y = \mathsf{fail}\ \mathsf{in}\ P_1\{^y/_x\}\ \mathsf{else}\ \mathsf{snd}'(Q)] = C[\mathsf{snd}'(\mathsf{let}\ y = \mathsf{diff}'[D, \mathsf{fail}]\ \mathsf{in}\ P_1\{^y/_x\}\ \mathsf{else}\ Q)]$. Hence $C[P_0'] \approx C[\mathsf{snd}'(merge(P_0, P_0'))]$. □

E.3 Proof of Lemma 4

Lemma 4 (simplify). *Let P be a biprocess. For all contexts C closing for P,*

$$C[P] \approx C[simpl(P)]$$

Proof. Let P_0 be a biprocess. We prove by induction on P_0 that, for all contexts C closing for P_0, $C[P_0] \approx C[simpl[P_0]$. We do a case analysis on the rules of Figure 5.

Case (Snil)*:* Trivial.

Case (Sout)*:* We have $P_0 = \overline{M}\langle N\rangle.P$ and $simpl(P_0) = \overline{M}\langle N\rangle.simpl(P)$. Let C be a context closing for P_0. Let $C_1 = C[\overline{M}\langle N\rangle.[\,]]$. We have that $C[P_0] = C_1[P]$ with C_1 closing for P. By induction hypothesis on $simpl(P)$, we can deduce that $C_1[P] \approx C_1[simpl(P))] = C[\overline{M}\langle N\rangle.simpl(P)] = C[simpl(P_0)]$.

Cases (Sin)*,* (Sres) *and* (Srepl)*:* Proof similar to case (Sout).

Case (Smid)*:* We have $P_0 = P \mid Q$. Let C be a context closing for P_0. Let $C_1 = [\,] \mid Q$ and $C_2 = simpl(P) \mid [\,]$. By induction hypothesis on $simpl(P)$, we obtain that $C_1[P] \approx C_1[simpl(P)]$. Moreover, $C_1[simpl(P)] = C_2[Q]$. By induction hypothesis on $simpl(Q)$, we have that $C_2[Q] \approx C_2[simpl(Q)]$. With $C_2[simpl(Q)] = C[simpl(P_0)]$, we conclude that $C[P_0] \approx C[simpl(P_0)]$.

Case (Slet)*:* Proof similar to case (Smid).

Case (Smerge)*:* We have $P_0 = \mathsf{let}\ x = D\ \mathsf{in}\ P\ \mathsf{else}\ P'$. Let $Q' = merge(simpl(P), simpl(P'))$ and $Q = Q'\{^{\mathrm{eval}\ \mathsf{letin}(x,D_1,D_2)}/_{\mathsf{diff}'[D_1,D_2]}\}$. Thanks to Lemma 7, Item 5, we deduce that $C[\mathsf{let}\ x = D\ \mathsf{in}\ \mathsf{fst}'(Q')\ \mathsf{else}\ \mathsf{snd}'(Q')] \approx C[\mathsf{let}\ x = \mathrm{eval}\ \mathsf{catchfail}(D)\ \mathsf{in}\ Q\ \mathsf{else}\ 0]$.

Furthermore, by induction hypothesis on P and Lemma 3 with the context $C_1 = C[\mathsf{let}\ x = D\ \mathsf{in}\ [\,]\ \mathsf{else}\ P']$, we obtain $C_1[\mathsf{fst}'(Q')] \approx C_1[simpl(P)] \approx C_1[P]$. Hence $C[\mathsf{let}\ x = D\ \mathsf{in}\ \mathsf{fst}'(Q')\ \mathsf{else}\ P'] \approx C[\mathsf{let}\ x = D\ \mathsf{in}\ P\ \mathsf{else}\ P'] = C[P_0]$. By induction hypothesis on P' and Lemma 3 with $C_2 = C[\mathsf{let}\ x = D\ \mathsf{in}\ \mathsf{fst}'(Q')\ \mathsf{else}\ [\,]]$, we obtain similarly $C_2[\mathsf{snd}'(Q')] \approx C_2[simpl(P')] \approx C_2[P']$, that is, $C[\mathsf{let}\ x = D\ \mathsf{in}\ \mathsf{fst}'(Q')\ \mathsf{else}\ \mathsf{snd}'(Q')] \approx C[\mathsf{let}\ x = D\ \mathsf{in}\ \mathsf{fst}'(Q')\ \mathsf{else}\ P']$, so we deduce that $C[P_0] \approx C[\mathsf{let}\ x = D\ \mathsf{in}\ \mathsf{fst}'(Q')\ \mathsf{else}\ \mathsf{snd}'(Q')]]$.

By combining the two equivalences, we conclude that $C[P_0] \approx C[simpl(P_0)]$

E.4 Proofs of the Main Results

Theorem 2. *Let P be a closed biprocess. If $simpl(P)$ satisfies observational equivalence then $\mathsf{fst}(P) \approx \mathsf{snd}(P)$.*

Proof. We know, by definition 2, $simpl(P)$ satisfies observational equivalence implies that $\mathsf{fst}(simpl(P)) \approx \mathsf{snd}(simpl(P))$. Since P is closed, we can apply Lemma 4 with the empty context C. Hence $\mathsf{fst}(P) \approx \mathsf{fst}(simpl(P))$ and $\mathsf{snd}(P) \approx \mathsf{snd}(simpl(P))$. Therefore, we conclude that $\mathsf{fst}(P) \approx \mathsf{snd}(P)$. □

Theorem 3. *Let P and P' be two closed processes that do no contain terms with* diff*. Let $Q = merge(simpl(P), simpl(P'))$. If the biprocess $Q\{^{\mathsf{diff}[D,D']}/_{\mathsf{diff}'[D,D']}\}$ satisfies observational equivalence, then $P \approx P'$.*

Proof. According to Fig. 4 and 5, if P and P' do not contain any diff then so do $simpl(P)$, $simpl(P')$, and $merge(P, P')$. Hence, Q does not contain any diff. Hence, the biprocess $Q' = Q\{^{\mathsf{diff}[D,D']}/_{\mathsf{diff}'[D,D']}\}$ satisfies $\mathsf{fst}(Q') = \mathsf{fst}'(Q)$ and $\mathsf{snd}(Q') =$

$\mathsf{snd}'(Q)$. Since Q' satisfies observational equivalence, we have that $\mathsf{fst}(Q') \approx \mathsf{snd}(Q')$ thus $\mathsf{fst}'(Q) \approx \mathsf{snd}'(Q)$. Since P and P' are closed processes, by Lemma 4, we have $P \approx simpl(P)$ and $P' \approx simpl(P')$. Furthermore, by Lemma 3, we deduce that $simpl(P) \approx \mathsf{fst}'(Q)$ and $simpl(P') \approx \mathsf{snd}'(Q)$. We conclude that $P \approx P'$.