SPEC 0.3: A User Manual

Alwen Tiu
Research School of Computer Science
The Australian National University

1 Overview

SPEC (for Spi-calculus Equivalence Checker) is an equivalence checker for a version of Abadi and Gordon's spi-calculus [AG99]. The spi-calculus is an extension of the π-calculus [MPW92], with operators encoding cryptographic primitives. In this implementation, we consider only a symmetric key encryption. As shown in [AG99], the spi-calculus can be used to encode security protocols, and via a notion of *observational equivalence*, security properties such as secrecy and authentication can be expressed and proved.

Intuitively, observational equivalence between two processes means that the (observable) actions of the processess cannot be distinguished in any execution environment (which may be hostile, e.g., if it represents an active attacker trying to compromise the protocol). The formal definition of observational equivalence [AG99] involves infinite quantification over all such execution environments and is therefore not an effective definition that can be implemented. Several approximations have been proposed [AG98, BNP02, BBN04, BN05, Tiu07, Tiu09]; the current implementation of SPEC is based on a notion of *open bisimulation* proposed in [Tiu07, Tiu09]. In particular, the decision procedure implemented here derives from [TD10]. It is important to note that the notion of observational equivalence as formalised in the spi-calculus assumes a model of intruder known as the Dolev-Yao model [DY83]. This means, among others, that we assume that the encryption function is perfect, in the sense that an attacker is not able to decrypt an encrypted message unless he/she knows the key.

The current version of SPEC is still at the alpha testing phase, and allows only modeling symmetric encryptions. It is also not yet optimised for performance. The tool can only reason about equivalence (or more precisely, bisimilarities) of finite processes, i.e., those without recursion. It cannot yet be realistically used to decide equivalence of protocols with unbounded sessions; this will require more sophisticated techniques. In future releases, it is hoped that more encryption operators will be supported and more functionalities will be added, in particular, facilities to do (symbolic) trace analyses, correspondence assertions, type checking and model checking.

The proof engine of SPEC is implemented in an improved version of the Bedwyr model checking system [BGM$^+$07], but the user interface is implemented directly in Ocaml, utilising a library of functions available from Bedwyr. The user, however, does not need to be aware of the underlying Bedwyr implementation and syntax in order to use the tool.

Readers who are interested only in using SPEC for checking process equivalence should read Section 2, Section 3, Section 4 and Section 5. Section 6 and Section 7 are intended for readers familiar with the Bedwyr system, and describe various extensions to the Bedwyr system that are implemented to support the proof engine of SPEC.

2 A quick start

2.1 Downloading and compiling SPEC

The latest version of SPEC can be downloaded from the project page:

> `http://users.cecs.anu.edu.au/~tiu/spec`.

The SPEC distribution includes a modified version of the Bedwyr prover. SPEC is implemented using the Ocaml language and has currently been tested only on the Linux operating system.

To compile SPEC, download the SPEC package from the website given above, and unpack it in a directory of your choice. For the purpose of this tutorial, we assume that the unpacked files are located in the `spec` directory inside the user's home directory. The structure of the distribution consits of the following directories:

- `src/` : Contains the source codes for both the modified Bedwyr and SPEC related files. The core engine of SPEC is located in the subdirectory `src/defs`: these are program files that will be run in the Bedwyr proof engine.
- `doc/` : Contains the manual of SPEC.
- `examples/` : Contains some examples of processess and protocols.

To compile the distribution, run the following commands:

```
# cd $HOME/spec
# ./configure
# make
```

This will create two executables: `spec` and `bedwyr`, in the subdirectory `src/`. If you are not interested in tweaking the Bedwyr codes, the program `spec` is all you need to be aware of.

Just as Bedwyr, by default, SPEC is built using the native-code Ocaml compiler `ocamlopt`, since it is much faster. If for some reason this feature is not desired, it can be disabled using `./configure --disable-native-code`.

2.2 Running SPEC

To run the SPEC user interface, type

```
# src/spec
```

This will bring up the command line interface

```
SPEC: An equivalence checker for the spi-calculus.

Version 0.2

This software is under GNU Public License.
Copyright (c) 2011-2013 Alwen Tiu

SPEC>
```

At the command-line interface, we can define processes to be checked for equivalence. The precise syntax will be given in Section 3. Here is an example process just as an illustration.

```
SPEC> P := nu (m, k). a< enc(m, k) >. [m = a] a<m>;
Reading spi definition
```

Here the symbol to the left of `:=` is a process identifier, and the expression to the right is the process itself. In this case, `P` is a process that creates two fresh names, m and k, and output the encrypted message `enc(m,k)` (representing a message that is obtained by encrypting m with key k) on channel `a`. It then performs a (useless) check on whether m is equal to a (which is always false, because m needs to be a fresh name), before outputting m on channel a.

Every definition (or any other statement) in the command prompt must ends with a semicolon.

By default, SPEC treats free names in a process as public names, i.e., they are known to the intruder. In this case, the only public name is `a`.

Let us define another process:

```
SPEC> Q := nu (k). a< enc(a, k) > ;
```

This process generates a fresh key `k` and encrypts the name `a` with `k`, and sends it off on channel `a`.

Having defined the two processes, we can query SPEC to check whether they are equivalent:

```
SPEC> bisim(P, Q);
Checking bisimilarity for:
nu(n1,n2).a<enc(n1,n2)>.[n1 = a]a<n1>.0
 and
nu(n1).a<enc(a,n1)>.0
```

```
..
The two processes are bisimilar.
Size of bisimulation set: 2.  Use #show_bisim to show the set.
Running time: + 0s
```

Note that the two processes are actually observationally equivalent, because an attacker cannot distinguish the message output by `P` and that output by `Q`, as the attacker does not have access to the key `k` (which is freshly generated). Moreover, the guard $[m = a]$ in P makes sure that the output action $a\langle m\rangle$ can never be executed, so it is essentially equivalent to 0.

A 'yes' or 'no' answer may not be particularly convincing, especially if the processes are large and complicated. SPEC provides a facility to produce an evidence of the claimed equivalence, in the form of a *bisimulation.* To show such a proof, use the following command:

```
SPEC> #show_bisim;
1.
Bitrace: []
First process: nu(n1,n2).a<enc(n1,n2)>.[n1 = a]a<n1>.0
Second process: nu(n1).a<enc(a,n1)>.0
2.
Bitrace: [(enc(n1,n2), enc(a,n2))^o.]
First process: [n1 = a]a<n1>.0
Second process: 0

SPEC>
```

In this case it is particularly simple. We'll see the meaning of the output in Section 5. There are also a couple of other forms of output, as we'll see in Section 5.

2.3 Available commands

There are some meta-commands available in SPEC, which can be queried by using the `#help` command. The list of commands is as follows.

- `#help`: Display the help message.
- `#exit`: Exit the program.
- `#equivariant`: Turning the equivariant tabling on/off. See Section 7.1. By default, equivariant is always set to on.
- `#reflexive`: Turning reflexivity checking on/off.
- `#load [file]`: Load a process definition file.
- `#reset`: Clears the current session. This removes all process definitions defined in the current session.

- `#show_bisim`: Displays the bisimulation set of the most recent bisimulation query.
- `#save_bisim [file]`: Save the current bisimulation set to a file.
- `#save_bisim_latex [file]` : Save the current bisimulation set to a file in the LaTeX format.
- `#save_bisim_raw [file]`: Save the current bisimulation set in the internal Bedwyr syntax.
- `#show_def [name]`: Show the definition for an agent.
- `#show_defs`: Show all the definitions.
- `#time [on/off]`: Show/hide the execution time of a query.
- `#trace [on/off]`: Enable/disable trace printing.

3 Syntax of processes

The spi-calculus generalises the π-calculus by allowing arbitrary terms to be output, instead of just simple names. To define the language of processes, we first need to define the set of terms (or messages). The set of terms allowed is defined by the following grammar:

$$M, N ::= a \mid x \mid \langle M, N \rangle \mid \mathrm{enc}(M, N)$$

where a denotes a name and x denotes a variable. Names are considered to be constants, i.e., they cannot be instantiated by other terms, wherease variables can be instantiated. The need for variables arises from the need to symbolically represent input values, which can range over an infinite set of messages, when unfolding the transitions of a process. The message $\langle M, N \rangle$ represents a pair of messages M and N, and $\mathrm{enc}(M, N)$ represents a message M encrypted with key N using a symmetric encryption function. Note that the key here can itself be any term.

We assume a set of *process identifiers*, ranged over by capital letters such as A,B,etc. The purpose of a process identifier is to simplify the writing of a process, by acting as a sort of ‘macros’ for processes. A process identifier may be assigned an arity, representing the number of arguments it accepts. The language of processes is given by the following grammar:

$$\begin{array}{rl} P ::= & A\{a_1, \dots, a_n\} \mid 0 \mid a(x).P \mid a\langle M \rangle \mid [M = N]P \mid \nu(x_1, \dots, x_m).P \mid \\ & (P \mid P) \mid \; !P \mid \mathrm{let}\ \langle x, y \rangle = M \ \mathrm{in}\ P \mid \mathrm{case}\ M\ \mathrm{of}\ \mathrm{enc}(x, N)\ \mathrm{in}\ P. \end{array}$$

The intuitive meaning of each of the process construct (except for process identifier, which is explained later) is as follow:

- 0 is a deadlocked process. It cannot perform any action.

- $a(x).P$ is an input-prefixed process. The variable x is a binder whose scope is P. The process accepts a value on channel a, binds it to the variable x and evolves into P.
- $a\langle M\rangle.P$ is an output-prefixed process. It outputs a term M on channel a and evolves into P.
- $[M = N]P$ is a process which behaves like P when M is syntactically equal to N, but is otherwise a deadlocked process.
- $\nu(x_1, \ldots, x_m).P$ is a process that creates m new names and behaves like P. The ν operator (also called the restriction operator) binds the name $x_1, \ldots, x_m$ in P.
- $P \mid Q$ is a paralel composition of P and Q.
- $!P$ is a replicated process; it represents an infinite paralel copies of P.
- let $\langle x, y\rangle = M$ in P. The variables x and y are binders whose scope is P. This process checks that M decomposes to a pair of messages, and binds those messages to x and y, respectively.
- case M of $\mathrm{enc}(x, N)$ in P. The variable x here is a binder whose scope is P. This process checks that M is a message encrypted with key N, decrypts the encrypted message and binds it to x.

A *process definition* is a statement of the form:

$$A(x_1, \ldots, x_n) := P$$

where A is a process identifier, of arity n, and P is a process. We call $A(x_1, \ldots, x_n)$ the *head* of the definition and P its *body*. The variables $x_1, \ldots, x_n$ are the *parameters* of the definition. A process identifier that occurs in a process P must be applied to arguments which are names. For example, if A is defined as:

$$A(x, y, z) := [z = y]z\langle x\rangle.0$$

then it can be used in a process such as this:

$$a(x).a(y).a(z).A\{x, y, z\}.$$

This process is equivalent to

$$a(x).a(y).a(z).[z = y]z\langle x\rangle.0$$

where the definition of A is expanded. SPEC does not allow recursive definitions, so in particular, in the definition of a process identifer A, the same identifier cannot occur in the body of the definition.

3.1 Concrete syntax

The concrete syntax accepted by the parser of SPEC is very similar to the abstract one described previously. We follow the following conventions with respect to names and identifiers:

- Names and variables are represented by any alpha-numeric strings that start with a lower case letter, e.g., `a`, `b`, etc.
- Process identifiers are represented by any alpha-numeric strings that start with a capital letter, e.g., `A`, `B`. We also allow the underscore '_' letter, e.g., as in `Agent_A`.

The concrete syntax for each construct follows closely the abstract syntax given previously, except for the ν operator which is represented by the keyword `nu`. For example, the following processes

$$\begin{array}{llll} 0 \quad A\{a,b,c\} & a(x).[x = \langle b,c\rangle]0 & (A\{a,b,c\} \mid a\langle \mathrm{enc}(b,k)\rangle.0) \\ \nu(x,y).a\langle x\rangle.a\langle y\rangle.0 & !(\nu(x).a\langle x\rangle.0) & \end{array}$$

are respectively written in the concrete syntax as:

```
0    A{a,b,c}   a(x).[x = <b,c>]0     (A{a,b,c} | a<enc(b,k)>.0)
nu (x,y). a<x>.a<y>.0   !(nu (x).a<x>.0)
```

3.2 Operator precedence, associativity and simplication

To ease writing down (and reading) processes, we adopt the following conventions:

- All the unary operators, e.g., input/output prefix, the match operator, the replication and the restriction operator, have higher precedence than the parallel composition. For example, the expression

  $$\nu(x).a\langle x\rangle.a(y).[x = y]0 \mid a(y).0$$

  represents the process

  $$(\nu(x).a\langle x\rangle.a(y).[x = y]0) \mid a(y).0.$$

- The paralel composition associates to the right. Thus $(P \mid Q \mid R)$ is the same as $(P \mid (Q \mid R))$.
- We can omit the trailing 0 in input/ouput prefixed processes. For example, we write $a\langle x\rangle.a(y)$ instead of $a\langle x\rangle.a(y).0$.

3.3 Inputing process definitions

As mentioned in Section 2, the definition of a process identifier can be entered at the SPEC command prompt, by ending it with a semicolon. SPEC checks whether all free names in the body of the definition are in the parameters of the definition. If there is a name in the body which does not occur as a parameter, SPEC gives a warning.

Process definitions can also be imported from a file. To do so, simply write down the definitions in a file, with each definition ended with a semicolon. To load the definition, use `#load`. For example, the following loads the file `wideMouthedFrog.spi` in the `examples` subdirectory.

```
SPEC> #load "examples/wideMouthedFrog.spi";
8 process definition(s) read. Use #show_defs to show all definitions
SPEC>
```

The definition of a particular process identifier can be queried using the command `#show_def`, followed by the identifier. For example:

```
SPEC> #show_def P;
P := nu(n1,n2).a<enc(n1,n2)>.[n1 = a]a<n1>.0
SPEC>
```

The command `#show_defs` will show all the definitions entered so far.

4 Some theoretical backgrounds

This section is intended to provide a very brief account of the notion of bisimulation used in SPEC in order to understand its output. For a more detailed account, the reader is refered to [Tiu07, Tiu09].

The procedure to prove observational equivalence implemented by SPEC is based on a notion *strong open bisimulation* for the spi-calculus developed in [Tiu07]. Open bisimulation is sound with respect to observational equivalence, in the sense that when two processes are open bisimilar, they are also observationally equivalence. However, it is not complete, as there are observationally equivalent processes that cannot be proved using open bisimulation. This is not specific to our notion of bisimulation, rather it is a feature of strong bisimulation which do not abstract away from internal (unobservable) transitions. For example, the process $\nu(x).(x\langle a\rangle \mid x(y).a\langle b\rangle)$ and the process $a\langle b\rangle$ are observationally equivalent, but are not strongly bisimilar.

A bisimulation is essentially a set that satisfies some closure conditions. We shall not explain in details what those conditions are; these can be found in [Tiu09, TD10]. Instead, we shall only illustrate what the members of such a bisimulation look like. Our notion of bisimulation is parametric on a structure which we call *bitraces*. A bitrace represents a history of the (input/output) actions of the pair of processes being checked for bisimilarity. A bitrace is essentially a list of *i/o pairs*. An *i/o pair* is either an *input pair*, written $(M, N)^i$, where M and N are messages, or an *output pair*, written $(M, N)^o$. We write a bitrace by separating the i/o pairs with a '.' ('dot'). For example, here's a bitrace:

$$(a, b)^o.(x, x)^i.(\text{enc}(x, a), \text{enc}(x, b))^o.$$

Recall our convention that a and b are names, whereas x is a variable, representing unknown values. Implicitly, the ordering of elements of the bitrace represents a 'timestamp' of the message pairs. In this case, the pair of processes first output, respectively, a and b, and then input (an unspecified) x, before outputing, respectively, $\text{enc}(\text{x}, \text{a})$ and $\text{enc}(\text{x}, \text{b})$. The empty bitrace is denoted by [ ].

A bitrace can be seen as a pair of symbolic traces, where the input values can be left unspecified. Bitraces are subject to certain well-formedness criteria, which can be found in [Tiu09]. The essence of those criteria is that the two symbolic traces that the bitrace represent are *observationally indistinguishable* from the view point of an intruder. The technical definition of observationally indistinguishability of traces is a bit complicated, but here is a simplified account. When an intruder observes a (symbolic) trace, what he is observing is not the identity of a particular message or messages, but rather what sorts of operations one can do with them. For example, in a sequence of messages (or a trace) such as:

$$k.\text{enc}(m, k).m$$

(following bitrace notations, we use dots to separate individual messages), what the intruder observes is that he can use the first message in the sequence to decrypt the second message and obtain a message that is identical to the third message in the sequence. Here the particular names used in the message is unimportant, as long as he could do those two operations of decryption and message comparison. So from this perspective, the intruder would consider the following sequence of messages observationally equivalent to the previous one:

$$l.\text{enc}(n, l).n$$

despite the fact that we use different names. However, the intruder will be able to detect a difference with the following trace (assuming m and n are distinct names):

$$l.\text{enc}(n, l).m$$

as the intruder notices that the result of decrypting the second message with the first message as the key produces a different message than the third one.

More specifically, the observational capabilities of an intruder, as formalised in [AG98], can be summarised as follows:

- Decryption: the ability to decrypt an encrypted message given the right key.
- Projection: the ability to extract components of a pair.
- Equality testing: the ability to compare two messages for syntactic equality.

For the purpose of simplifying the implementation of SPEC, we assume an additional capability: we assume that the intruder has knowledge of a set of "public" names (as in the notion of *frames* in Abadi and Gordon's *frame bisimulation* [AG98]). That is, we shall designate a certain set of names to be public names. We shall simply call these names *constants*. Hence the intruder also possesses the following capability:

- Constant testing: the ability to recognise a constant.

In the user interface of SPEC, any free names entered by the user are considered constants. The purpose of introducing constants as public names is to reduce the size of bitraces when checking for equivalence, as these constants need not be included in the bitraces.

A *consistent bitrace* is then a bitrace where all possible observations applying to its first projection are also valid observations on its second projection, and vice versa. This is related to the notion of *static equivalence* in the applied-pi calculus [AF01], where a bitrace can be seen as a pair of statically equivalent *frames* [AF01].

A *bisimulation* is a set of triples of the form (h, P, Q) where h is a consistent bitrace, and P and Q are processes, satisfying some closure conditions (see [Tiu09]). Two processes P and Q are observationally equivalent (or, more precisely, bisimilar) if we can exhibit a bisimulation set containing the triple $([\,], P, Q)$.

5 Checking process equivalence and understanding its outputs

Let us start by defining a couple of simple processes.

```
SPEC> P := nu (x,y). a< enc(x,y) >.a<b> ;
SPEC> Q := nu (x,y). a< enc(b,y) >.a<b> ;
```

This defines two processes P and Q. We can use the command `#show_defs` to show the process definitions we entered so far:

```
SPEC> #show_defs;
P := nu(n1,n2).a<enc(n1,n2)>.a<b>.0
Q := nu(n1,n2).a<enc(b,n2)>.a<b>.0
SPEC>
```

Notice that SPEC automatically renames the bound variables x and y to its internally generated fresh names ($n1$ and $n2$ in this case). In general, SPEC will prefix any bound names or names freshly generated with the letter 'n'. The engine also performs α-conversion as neccessary to avoid name clashes.

As discussed in the previous section, free names in a process entered by the user are automatically considered as constants by SPEC, with the consequence that they are known publicly to the intruder.

Recall from Section 2 that the command `bisim` allows use to check for bisimilarity:

```
SPEC> bisim(P,Q);
Checking bisimilarity for:
nu(n1,n2).a<enc(n1,n2)>.a<b>.0
 and
nu(n1,n2).a<enc(b,n2)>.a<b>.0
..
The two processes are bisimilar.
Size of bisimulation set: 2.  Use #show_bisim to show the set.
Running time: + 0s
```

What the implementation of bisimulation in SPEC does is basically unfold the transitions of P and Q in locked steps, and at each step, the messages input/output by the processes are added to the current bitrace (initially, it is the empty bitrace). At each step, SPEC also checks that the bitrace produced in that step is consistent. The bisimulation set produced by SPEC can be displayed using `#show_bisim`:

```
SPEC> #show_bisim;
1.
Bitrace: []
First process: nu(n1,n2).a<enc(n1,n2)>.a<b>.0
Second process: nu(n1,n2).a<enc(b,n2)>.a<b>.0
2.
Bitrace: [(enc(n1,n2), enc(b,n2))^o.]
First process: a<b>.0
Second process: a<b>.0
SPEC>
```

The original processes are listed in the first item, which get unfolded into the second item. In the second item, we notice that the first process is actually equal to the second process. SPEC recognises this automatically and stops the unfolding at this point, as bisimilarity is reflexive. The resulting bitrace at each step is also given. As we said earlier, input or output messages are appended to the current bitrace at each step.

5.1 Separation of names in bisimulation

An important thing to note in reading a bisimulation triple (h, P, Q) is that the names (other than constants) used in the first projection of the bitrace h and process P are *unrelated* to the names used in the second projection of h and Q. This is because names represent entities that may be generated internally by the processes (via scope extrusion) and may be unknown to the intruder, hence their identities are not something which is generally observable. See [AG98] for an explaination of this.

5.2 Equivariance of bisimilarity

Another important thing to keep in mind when reading the output bisimulation set produced by SPEC is that each triple in the set represents an *equivalence class of triples modulo renaming* of variables and names (but excluding constants). Thus, a triple such as

$$([(x, x)^i.(\mathrm{enc}(n, m_1), \mathrm{enc}(b, m_2))^o], P, Q)$$

where x is a variable, n and m are non-constant names and b is a constant, represents the set containing

$$([(x\theta, x\theta)^i.(\mathrm{enc}(n\theta, m_1\theta), \mathrm{enc}(b, m_2\theta))^o], P\theta, Q\theta)$$

where θ is an renaming substitution.

5.3 Examples

We show here a simple but interesting example to illustrate features of the bisimulation output. The SPEC distribution contains a number of small and big examples in the directory `examples`. Some of these examples encode a number of security protocols. These examples may produce very large bisimulation sets (in the order of hundreds to more than a thousand elements) and may take quite a while to execute.

Example 1 Let

$$P1 := a(x).\nu(k).a\langle \mathrm{enc}(x, k)\rangle.\nu(m).a\langle \mathrm{enc}(m, enc(a, k))\rangle.m\langle a\rangle$$

$$Q1 := a(x).\nu(k).a\langle \mathrm{enc}(x, k)\rangle.\nu(m).a\langle \mathrm{enc}(m, enc(a, k))\rangle.[x = a]m\langle a\rangle$$

The process $P1$ inputs a term via channel a, binds it to x and output an encrypted message $\mathrm{enc}(x, k)$. It then generates a new channel m, sends it off encrypted with the key $\mathrm{enc}(a, k)$. Here the name a is a constant, so it is known to the intruder. The process then sends a message on the newly generated channel m. Although the channel m is a secret generated by $P1$, and it is not explicitly extruded, the intruder can still interact via m if it feeds the name a to $P1$ (hence binds x to a). As a result, the (symbolic) output $\mathrm{enc}(x, k)$ can be ‘concretized’ to $\mathrm{enc}(a, k)$, which can be used to decrypt $\mathrm{enc}(m, \mathrm{enc}(a, k))$ to obtain m.

The process $Q1$ is very similar, except that it puts a ‘guard’ on the possibility of interacting on m by insisting that $x = a$. The above informal reasoning about the behavior of $P1$ shows that it should be observationally equivalent to $Q1$.

Before we try proving the bisimilarity of $P1$ and $Q1$, first execute the following command:

```
SPEC>#reflexive off;
Reflexivity checking is disabled.
```

This instructs SPEC to disable an optimisation, i.e., the reflexivity checking. Reflexivity checking allows SPEC to conclude immediately that identical pairs of processes are bisimilar. Disabling this allows us to see a bit more clearly how the bitraces in the bisimulation set in the following example change from one entry to another.

After disabling reflexivity checking and running the command `bisim(P1,Q1)`, we get that the two processes are indeed bisimilar. The bisimulation set produced can be displayed using the `#show_bisim` command, but here we shall try to output it in a LaTeX format:

1. Bi-trace: [ ]

   First process:

   $\mathrm{a}(n1).$
   $\nu\ (n2).$
   $\overline{\mathrm{a}}\ \langle\ \mathrm{enc}(n1, n2)\ \rangle.$
   $\nu\ (n3).$
   $\overline{\mathrm{a}}\ \langle\ \mathrm{enc}(n3, \mathrm{enc}(\mathrm{a}, n2))\ \rangle.$
   $\overline{n3}\ \langle\ \mathrm{a}\ \rangle.$
   0

   Second process:

   $\mathrm{a}(n1).$
   $\nu\ (n2).$
   $\overline{\mathrm{a}}\ \langle\ \mathrm{enc}(n1, n2)\ \rangle.$
   $\nu\ (n3).$
   $\overline{\mathrm{a}}\ \langle\ \mathrm{enc}(n3, \mathrm{enc}(\mathrm{a}, n2))\ \rangle.$
   $[n1 = \mathrm{a}]$
   $\overline{n3}\ \langle\ \mathrm{a}\ \rangle.$
   0

2. Bi-trace:

   $$(\mathrm{enc}(\mathrm{a}, n1)\ ,\ \mathrm{enc}(\mathrm{a}, n1))^o.$$
   $$(\mathrm{enc}(n2, \mathrm{enc}(\mathrm{a}, n1))\ ,\ \mathrm{enc}(n2, \mathrm{enc}(\mathrm{a}, n1)))^o.$$

   First process:

   0

   Second process:

   0

3. Bi-trace:

$$(?n1\ ,\ ?n1)^i.$$

First process:

$\nu\ (n2).$
$\overline{\text{a}}\ \langle\ \text{enc}(?n1, n2)\ \rangle.$
$\nu\ (n3).$
$\overline{\text{a}}\ \langle\ \text{enc}(n3, \text{enc}(\text{a}, n2))\ \rangle.$
$\overline{n3}\ \langle\ \text{a}\ \rangle.$
0

Second process:

$\nu\ (n2).$
$\overline{\text{a}}\ \langle\ \text{enc}(?n1, n2)\ \rangle.$
$\nu\ (n3).$
$\overline{\text{a}}\ \langle\ \text{enc}(n3, \text{enc}(\text{a}, n2))\ \rangle.$
$[?n1 = \text{a}]$
$\overline{n3}\ \langle\ \text{a}\ \rangle.$
0

4. Bi-trace:

$$(?n1\ ,\ ?n1)^i.$$
$$(\text{enc}(?n1, n2)\ ,\ \text{enc}(?n1, n2))^o.$$

First process:

$\nu\ (n3).$
$\overline{\text{a}}\ \langle\ \text{enc}(n3, \text{enc}(\text{a}, n2))\ \rangle.$
$\overline{n3}\ \langle\ \text{a}\ \rangle.$
0

Second process:

$\nu\ (n3).$
$\overline{\text{a}}\ \langle\ \text{enc}(n3, \text{enc}(\text{a}, n2))\ \rangle.$
$[?n1 = \text{a}]$
$\overline{n3}\ \langle\ \text{a}\ \rangle.$
0

5. Bi-trace:

   $$(?n1\ ,\ ?n1)^i.$$
   $$(\mathrm{enc}(?n1, n2)\ ,\ \mathrm{enc}(?n1, n2))^o.$$
   $$(\mathrm{enc}(n3, \mathrm{enc}(\mathrm{a}, n2))\ ,\ \mathrm{enc}(n3, \mathrm{enc}(\mathrm{a}, n2)))^o.$$

   First process:

   $\overline{n3}\ \langle\ \mathrm{a}\ \rangle.$
   0

   Second process:

   $[?n1 = \mathrm{a}]$
   $\overline{n3}\ \langle\ \mathrm{a}\ \rangle.$
   0

A few notes on the output produced by SPEC:

- *Typesetting of names and variables*: Variables are typeset by prepending the variables with a question mark '?'. Bound names and non-constant names are typeset using italised fonts, while constants are un-italicised.

- The original process pair is given in item 1. The unfolding sequence proceeds as follows: 1 → 3 → 4 → 5 → 1. Notice that in moving from 5 to 1, the input pair disappears from the bitrace. This is because the variable $?n1$ gets instantiated to a, which is a constant (hence, public knowledge), and it is removed by the simplification step of SPEC.

- *Equivariance of bisimulation*: Notice that in proceeding from 5 to 1 there is an implicit renaming performed by SPEC.[1] Without renaming, the bitrace in item 1 would have been the following:

  $$(\mathrm{enc}(\mathrm{a}, n2)\ ,\ \mathrm{enc}(\mathrm{a}, n2))^o.$$
  $$(\mathrm{enc}(n3, \mathrm{enc}(\mathrm{a}, n2))\ ,\ \mathrm{enc}(n3, \mathrm{enc}(\mathrm{a}, n2)))^o.$$

6 Interacting at the Bedwyr level

This section is intended only for readers familiar with the Bedwyr system. Please refer to the Bedwyr website (`http://slimmer.gforge.inria.fr/bedwyr/`) for an introduction to the Bedwyr system.

The SPEC proof engine is implemented on top of a modified version of Bedwyr. The Bedwyr codes for SPEC are contained in the subdirectory `src/defs`. Here is a brief description of the important definition files in that directory:

[1] This automatic renaming is a built-in feature of Bedwyr, so it is not possible to alter it without modifiying the search engine of Bedwyr. It is a by-product of *equivariance tabling* implemented in Bedwyr. See Section 7.

- `bisim.def`: This is the main module that implements the bisimulation checking.
- `bitrace.def`: This module implements procedures related to bitraces, in particular, the decision procedure for checking bitrace consistency.
- `intruder.def`: This module implements a decision procedure for solving deducibility constraints, under the Dolev-Yao intruder model. It is used in the decision procedure for bitrace consistency.
- `spi.def`: This module implements the operational semantics of the spi-calculus.

6.1 Intermediate syntax of processes

The theoretical foundation of Bedwyr is based on a variant of Church's simple theory of types. All expressions in Bedwyr are encoded as simply typed λ-terms. For the implementation of SPEC, we introduce (among others) the following types for the syntactic categories of terms and processes:

- `name`: the set of 'unsorted' names. Spi-calculus names, constants and variables are encoded using additonal unary operators applied to these unsorted names.
- `tm`: the set of terms (i.e., messages). Constructors for terms are as follows:
  - `ct` : `name` → `tm`. This is used ot encode constant.
  - `var` : `name` → `tm`. This is used to encode variables.
  - `nm` : `name` → `tm`. This is used to encode names (other than constants).
  - `pr` : `tm` → `tm` → `tm`. This encodes pairing.
  - `enc` : `tm` → `tm` → `tm`. This encodes symmetric encryption.
- `proc`: this is the type of processes. Constructors for processes are as follows:
  - `zero` : `proc`. The 0 process.
  - `par` : `proc` → `proc` → `proc`. Parallel composition.
  - `nu` : (`tm` → `proc`) → `proc`. Restriction.
  - `match` : `tm` → `tm` → `proc` → `proc`. Matching.
  - `inp` : `tm` → (`tm` → `proc`) → `proc`. Input prefix.
  - `outp` : `tm` → `tm` → `proc` → `proc`. Output prefix.
  - `let` : `tm` → (`tm` → `tm` → `proc`) → `proc`. Let-operator.
  - `case` : `tm` → `tm` → (`tm` → `proc`) → `proc`. Case-operator.
  - `bang` : `proc` → `proc`. Replication.

For example, the process $\nu(y).(a(x).[x = y]0 \mid a\langle y\rangle.0)$ will be represented as the simply-typed term (assuming we have a term a of type `name`):

$$\texttt{nu}\ \lambda y.(\texttt{par}\ (\texttt{inp}\ (\texttt{ct}\ a)\ (\lambda x.\texttt{match}\ x\ y\ \texttt{zero}))\ \ (\texttt{outp}\ (\texttt{ct}\ a)\ y\ \texttt{zero})).$$

Note that the `nu` operator binds only one name at a time, so to encode multiple binders such as $\nu(x, y).[x = y]0$, we iterate the `nu` operator twice, i.e.,

$$\texttt{nu}\ \lambda x.(\texttt{nu}\ \lambda y.\texttt{match}\ x\ y\ \texttt{zero}).$$

6.2 Running the bisimulation checker

To main procedure for checking bisimilarity is located in the file `src/defs/bisim.def`. We first invoke Bedwyr and load this file:

```
# src/bedwyr src/defs/bisim.def
Bedwyr 1.4-alpha2 (revision 861) welcomes you.

This software is under GNU Public License.
Copyright (C) 2005-2012 Slimmer project.

For a little help, type "#help."

?=
```

The predicate for bisimulation checking is called `bisim`, and it takes three arguments: the first one is for bitrace, and the second and the third are processes we want to check equivalence for.

A bitrace is represented as a list of i/o pairs in Bedwyr. The type `bt_pair` denotes the set of i/o pairs, and it has the following constructors:

- `in` : `tm` → `tm` → `bt_pair`, used to encode input pairs, and
- `out` : `tm` → `tm` → `bt_pair`, used to encode output pairs.

Lists are encoded using the standard polymorphic constructors: `nil` for the empty list and `cons` for constructing a new list given an element and another list.

What the SPEC interface does when checking bisimilarity is basically calling the `bisim` predicate with the empty list as the bitrace. For example, for checking bisimilarity of $\nu(m, k).a\langle \mathrm{enc}(m, k)\rangle.0$ and $\nu(k).a\langle \mathrm{enc}(a, k)\rangle.0$, one would type the following query:

```
?= nabla a\ bisim nil (nu m\ nu k\ outp (ct a) (en m k) zero)
             (nu k\ outp (ct a) (en (ct a) k) zero).
..Yes.
More [y] ?
```

The ∇ quantifier provides a new name. Here we assume that a is a constant.

Working directly in Bedwyr allows us to enter a more flexible query, such as attaching a non-trivial bitrace to the bisimulation pair. For this, we shall utilise the predicate `toplevel_bisim`. This predicate works as `bisim`, except it also checks the bitrace for consistency, whereas `bisim` does not check for consistency. For example, instead of assuming a as constant, we could assume it is a (non-constant) name, using the constructor `nm`. We will need to add this name explicitly in the bitrace, otherwise the bisimulation checker will reject it.

```
?= nabla a\ toplevel_bisim
              (cons (out (nm a) (nm a)) nil)
              (nu m\ nu k\ outp (nm a) (en m k) zero)
              (nu k\ outp (nm a) (en (nm a) k) zero).
..Yes.
More [y] ?
```

6.3 Bisimulation output in Bedwyr syntax

The predicate `bisim` is declared as a coinductive predicate in `bisim.def`. This means that Bedwyr automatically saves successful or failed proof search associated with the `bisim` predicate in a table. The set of bisimulation associated with the current bisimulation query can be displayed using the `#show_table` command. For example, continuing the previous example, one can display the entries of the table as follows.

```
?= #show_table bisim.
Table for bisim contains (P=Proved, D=Disproved):
 [P] (nabla (x1\
             nabla (x2\
                    nabla (x3\
                           bisim (cons (out (nm x3) (nm x3))
                                    (cons (out (en (nm x2) (nm x1))
                                             (en (nm x3) (nm x2)))
                                       nil))
                             zero zero))))
 [P] (nabla (x1\
             bisim (cons (out (nm x1) (nm x1)) nil)
               (nu (x2\ nu (x3\ outp (nm x1) (en x2 x3) zero)))
               (nu (x2\ outp (nm x1) (en (nm x1) x2) zero))))
 [D] (nabla (x1\
             bisim (cons (out (nm x1) (nm x1)) nil)
               (nu (x2\ nu (x3\ outp (ct x1) (en x2 x3) zero)))
               (nu (x2\ outp (nm x1) (en (ct x1) x2) zero))))
?=
```

The bisimulation set consists of those entries marked with `[P]`. Notice that the names in the entries of the table are ∇-quantified. This reflects the fact that the bisimulation set is closed under renaming.

Note that in the SPEC interface, one can also output the bisimulation set in the Bedwyr syntax using the command `#save_bisim_raw`.

7 Extensions to Bedwyr

This section is intended only for readers familiar with the Bedwyr system. Please refer to the Bedwyr website (http://slimmer.gforge.inria.fr/bedwyr/) for an introduction to the Bedwyr system.

The Bedwyr version used to implement SPEC is a modification of the official release of Bedwyr. The reason for this modification is mainly because that the decision procedure for checking bitrace consistency needs to manipulate substitutions of the object-level (i.e., spi-calculus) variables explicitly and perform unification on them. The implementation encodes object-level variables and names as rigid terms in Bedwyr, but dynamically convert these to logic variables and back in order to access the built-in unification procedure of Bedwyr. As a consequence, some of the modifications to Bedwyr introduce non-logical features that allow abstracting away logic variables. The logical interpretations of these additional features are currently unclear, so the implementation of SPEC in this modified Bedwyr does not inherit the meta-logical properties of the logic underlying Bedwyr [BGM^{+}07]. It is therefore more appropriate to view this modified Bedwyr as a programming framework rather than a proper logical framework like it was designed to be. However, as the bisimulation checker in SPEC can actually output a bisimulation set as a witness of observational equivalence, unsoundness of Bedwyr is not a real concern as that witness can be checked for correctness independently of the Bedwyr engine.

The list of non-logical predicates added to Bedwyr is given below. All these are prefixed with an underscore (_).

- `_not` : `o` → `o`. This is the standard negation-as-failure as in prolog.
- `_if`: $o \to o \to o \to o$. This is an if-then-else operator. The query (`_if` P Q R) is basically equivalent to $(P \wedge Q) \vee$ (`_not`$(P) \wedge Q)$. The slight difference is that the second disjunct will not be tried if the query P succeeds.
- `_distinct`: $o \to o$. Calling (`_distinct` P) directs bedwyr to produce only distinct answer substitutions. For example, if a predicate `p` is defined as follows:

  ```
  p a.
  p a.
  ```

  then querying `p X` in Bedwyr will answer with the same answer substitution twice.

  ```
  ?= p X.
  ```

```
Solution found:
 X = a
More [y] ? y
Solution found:
 X = a
More [y] ? y
No more solutions.
?=
```

If we apply the _distinct operator, only one answer substitution is given.

```
?= _distinct (p X).
Solution found:
 X = a
More [y] ? y
No more solutions.
?=
```

- _abstract : $\alpha \to (\beta \to \alpha) \to \alpha \to o$. Here α and β can be any types. A query like (_abstract T Abs T') abstracts the logic variables in T of type β, and apply the constructor Abs to each abstraction, and unify the result with T'. For example:

  ```
  ?= _abstract (pr X Y) abs T.
  Solution found:
  X = X
  Y = Y
  T = (abs (x1\ abs (x2\ pr x1 x2)))
  More [y] ? y
  No more solutions.
  ```

  Note that because _abstract can abstract any logic variables, and because the underlying proof search engine of Bedwyr is untyped, the abstraction produced by _abstract may not always respect the type of the constructor abs. For example, consider the above example. If pr is of type $\alpha \to \beta \to \alpha$, for some distinct types α and β, and abs is of type $\alpha \to \alpha$, then the above query will still succeed despite the fact that abs is applied to terms of both types. Hence type checking at the user level does not guarantee runtime type soundness ('well typed programs don't go wrong'). In the future, we plan to integrate type checking directly into the Bedwyr engine, so that it only abstracts variables of the correct types.

- _rigid : $\alpha \to o$. This is a meta-level assertion predicate. The query (_rigid X) will throw an exception (hence causes the prover to abort) if X is not a ground term. This predicate is mainly used as an assertion in SPEC prover to ensure that no (logic) variable leaks occur when converting an object variable to a logic variable and vice versa.

- `_trace`: *o*. This nullary predicate returns the value of the trace flag set via the `#trace` command (see below). It can be used by programs to toggle on/off printing of traces or other debugging information.
- `_abort`: *o*. This predicate aborts the proof search and returns to the toplevel query (if in interactive mode).

Two new system-level commands have been introduced:

- `#trace <on/off>`: This system-level command sets the trace flag. Note: the system only sets the flag to true/false. The use of this flag is up to the program. Its value can be read off by a program via the predicate `_trace`.
- `#save_table <predicate> <filename>`: This will save the table for a predicate to a definition file. A proved entry will become the argument of a predicate called 'proved'. Similarly, an unproved entry will be the argument of an 'unproved' predicate.

7.1 Modifications to the tabling mechanism

The tabling mechanism now allows for matching of table entries up to renaming of ∇-variables. As an example, consider this very simple program:

```
inductive q X Y := println "Proved without using table".
```

and consider this query:

```
?= nabla x\ nabla y\ q x y.
Proved without using table
Yes.
More [y] ? n
```

Bedwyr executes the body of the definition clause for `q`, which prints a message, and stores the goal in the table. Now, the second time the query is repeated:

```
?= nabla x\ nabla y\ q x y.
Yes.
More [y] ?
```

we see that the print statement is not executed, because Bedwyr proves the query by a lookup in the table of previously proved queries and does not execute the body of the clause. To see the effect of automatic renaming, consider this query:

```
?= nabla x\ nabla y\ q y x.
Yes.
More [y] ?
```

where the roles of `x` and `y` are exchanged. This query is also executed successfully by a table lookup, as the goal can be matched to the table entry by renaming `y` to `x`. We see that adding vacuous ∇-quantification does not affect this equivariant matching, as in the following query:

```
?= nabla x\ nabla y\ nabla z\ q x z.
Yes.
More [y] ?
```

What the tabling lookup checks is that `q` has two distinct ∇-variables as arguments. So the following query will not match the table (as a consequence, the message in the body of the clause will again be printed):

```
?= nabla x\ q x x.
Proved without using table
Yes.
More [y] ?
```

Acknowledgment

This work is supported by the ARC Discovery grants DP0880549 and DP110103173. The author would like to thank David Baelde for many useful correspondences regarding the implementation of the Bedwyr system, and Quentin Heath for many improvements to the Bedwyr proof engine used to implement SPEC.

References

[AF01] Martín Abadi and Cédric Fournet. Mobile values, new names, and secure communication. In *POPL*, pages 104–115, 2001.

[AG98] Martín Abadi and Andrew D. Gordon. A bisimulation method for cryptographic protocols. *Nord. J. Comput.*, 5(4):267–303, 1998.

[AG99] Martin Abadi and Andrew D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, 148(1):1–70, 1999.

[BBN04] Johannes Borgström, Sébastien Briais, and Uwe Nestmann. Symbolic bisimulation in the spi calculus. In Philippa Gardner and Nobuko Yoshida, editors, *CONCUR*, volume 3170 of *Lecture Notes in Computer Science*, pages 161–176. Springer, 2004.

[BGM$^+$07] David Baelde, Andrew Gacek, Dale Miller, Gopalan Nadathur, and Alwen Tiu. The Bedwyr system for model checking over syntactic expressions. In Frank Pfenning, editor, *21th Conference on Automated Deduction*, number 4603 in LNAI, pages 391–397. Springer, 2007.

[BN05] Johannes Borgström and Uwe Nestmann. On bisimulations for the spi calculus. *Mathematical Structures in Computer Science*, 15(3):487–552, 2005.

[BNP02] Michele Boreale, Rocco De Nicola, and Rosario Pugliese. Proof techniques for cryptographic processes. *SIAM Journal of Computing*, 31(3):947–986, 2002.

[DY83] D. Dolev and A. Yao. On the security of public-key protocols. *IEEE Transactions on Information Theory*, 2(29), 1983.

[MPW92] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, Part II. *Information and Computation*, pages 41–77, 1992.

[TD10] Alwen Tiu and Jeremy E. Dawson. Automating open bisimulation checking for the spi calculus. In *Proceedings of the 23rd IEEE Computer Security Foundations Symposium (CSF 2010)*, pages 307–321. IEEE Computer Society, 2010.

[Tiu07] Alwen Tiu. A trace based bisimulation for the spi calculus: An extended abstract. In *APLAS*, volume 4807 of *Lecture Notes in Computer Science*, pages 367–382. Springer, 2007.

[Tiu09] Alwen Tiu. A trace based bisimulation for the spi calculus. *CoRR*, abs/0901.2166, 2009.