Formally Sound Refinement of Spi Calculus Protocol Specifications into Java Code

Alfredo Pironti and Riccardo Sisto
Politecnico di Torino
Dip. di Automatica e Informatica
c.so Duca degli Abruzzi 24, I-10129 Torino (Italy)
e-mail: {alfredo.pironti,riccardo.sisto}@polito.it

Abstract

Spi Calculus is an untyped high level modeling language for security protocols, used for formal protocol specification and verification. In this paper, a type system for the Spi Calculus and a translation function are formally defined, in order to enable the refinement of a Spi Calculus specification into a Java implementation. Since the generated Java implementation uses a custom Java library, formal conditions on the custom Java library are also stated, so that, if the library implementation code satisfies such conditions, then the generated Java implementation correctly simulates the Spi Calculus specification.

Keywords: *Model-based software development, Correctness preserving code generation, Formal methods, Security protocols*

1. Introduction

The Spi Calculus [3] is a formal domain-specific language that can be used to abstractly specify security protocols, that is communication protocols that use cryptographic primitives in order to protect some assets. A Spi Calculus specification can then be passed to an automatic tool, usually a model checker (e.g. [12]) or a theorem prover (e.g. [10]), that verifies some security properties on it [2, 1], i.e. it verifies that the protocol actually works as it is intended by the designer, and is resilient to attacks performed by a (Dolev-Yao [11]) intruder.

However, real implementations of security protocols, implemented in a programming language, may significantly differ from the verified formal specification expressed in Spi Calculus, so that the real behavior of the protocol differs from the verified one, possibly enabling attacks that are not present by looking at the formal specification. For instance, a protocol implementation may miss to perform a check on a received nonce, thus enabling replay attacks.

For these reasons, and by taking into account that security protocols are usually applied in safety or mission critical environments, assessing the correctness of a protocol implementation with respect to its formally verified specification is a great challenge for the software engineering community. By testing the protocol implementation, only few scenarios are taken into account, and this approach may not give enough confidence about implementation correctness, due to two facts: the distributed and concurrent nature of security protocols, that generates a large (usually unbounded) number of possible application scenarios, and the presence of an active attacker, who can behave in the worst way, and is not under the software developer control.

In principle, in order to ensure that the formal model is correctly refined by the implementation, two development methodologies can be used, namely model extraction [9, 17, 13, 7] and code generation [23, 21, 25]. By using the first methodology, one starts by manually developing a full blown implementation of a security protocol, and automatically extracts a formal model from it. The extracted formal model is then verified, in order to check the desired security properties. By using the second methodology, one starts from a formal specification of a security protocol, and automatically generates the code that implements it, filling user-provided implementation details that are not caught by the formal model. In order for each methodology to be useful, it must be possible to formally show that a refinement relation between the implementation code and the abstract formal model exists.

The work presented in this paper aims to improve the correctness assurance that can be achieved by using an automatic code generation approach, by formally defining what was informally described in [23, 21], that is a translation function from the Spi Calculus specification language, to the Java implementation language. The translation function relies on a Java library which essentially wraps the Java Cryptography Architecture library calls that implement cryptographic primitives in the Java environment. In this paper it is formally shown that, if it is assumed that the implementation of the library satisfies some conditions that we formally express, then the generated Java code correctly refines the abstract model.

The remainder of this paper is organized as follows. Section 2 describes some works related to the one presented here. Section 3 briefly describes the Spi Calculus and presents a type inference system for the Spi Calculus and a formal translation from the Spi Calculus to Java. Section 4 presents the main correctness property of the translation, i.e. that the generated Java implementation simulates a given Spi Calculus specification. Finally, section 5 concludes and gives some hints for future work.

2. Related Work

Up to our knowledge, there are two independent works dealing with generation of Java code, starting from Spi Calculus specifications, namely [23, 21] and [25]. However, none of these works provides rigorous formal proofs for the generated code.

The AGC-C# tool [16] automatically generates C# code from a verified Casper script. Since in that work a formal translation function is not provided, it is not possible to obtain soundness proofs. Moreover, the generated code is obtained from a script that is manually modified w.r.t. the verified Casper script. These manual changes may introduce errors, and the generated code starts from a model that is not the verified one, thus possibly invalidating a soundness assertion.

In [14], a manual refinement of CSP protocol specifications into JML constraints is described. However, no formal translation rules from CSP processes to JML constraints are provided. Again, the lack of automatic tools makes the manual refinement process error prone.

In web services, security properties are expressed at a higher level, as policy assertions [6, 5]. Rather than specifying *how* security is achieved, through the coordination of cryptographic primitives inside the protocol specification, a policy assertion specifies a property that must hold for a specific set of SOAP messages.

The tool described in [8], checks user given policy assertions, in order to find common flaws. However, it does not provide any formal proof about the correctness of the user given policy assertions w.r.t. a specification. Moreover, there is still the need to verify that the policy assertion implementations are correct. The work presented in [7] gives a verified reference implementation of the WS-Security [19] protocol, written in F#. This implementation can be used in future complete protocol implementations as a starting point. However, no tool that verifies an user given implementation of policies, nor that generates a correct implementation from user given policies, is, to the best of our knowledge, currently available for web services policy assertions.

By looking at the model extraction approach, the work in [9] is, up to our knowledge, the only one providing a formally proven correct abstraction from a subset of F# code, to applied π-calculus. In [9], like in this paper, correctness of some low level cryptographic libraries is assumed, that is, the concrete low level libraries are assumed to behave like the abstract symbolic counterparts. Although promising, this approach currently uses a starting language that is not very common in the programming practice, and the constraints on the selected subset of F# currently allow only *ad hoc* written code to be verified.

3. Formalizing the Translation

3.1. The Spi Calculus

The Spi Calculus extends the π-calculus [18], by adding a fixed set of cryptographic primitives to the language, namely symmetric and asymmetric encryptions, and hash functions, thus enabling the description of all the main security protocols. Adding more custom cryptographic primitives to the language, so that more security protocols can be described, is rather straightforward.

A Spi Calculus specification is a system of concurrent processes that operates on untyped data, called terms. Terms can be exchanged between processes by means of input/output operations. Table 1 contains the terms defined by the Spi Calculus, while table 2 shows the processes. A name n is an atomic value, and a pair (M, N) is a compound term, composed of the

$L, M, N ::=$	terms
n	name
(M, N)	pair
0	zero
$suc(M)$	successor
x	variable
$\{M\}_N$	shared-key encryption
$H(M)$	hashing
M^+	public part
M^-	private part
$\{[M]\}_N$	public-key encryption
$[\{M\}]_N$	private-key signature

Table 1. Spi Calculus terms.

$P, Q, R ::=$	processes
$\overline{M}\langle N \rangle .P$	output
$M(x).P$	input
$P \vert Q$	composition
$!P$	replication
$(\nu n) P$	restriction
$[M \; is \; N] P$	match
$\mathbf{0}$	nil
$let \; (x, y) = M \; in \; P$	pair splitting
$case \; M \; of \; 0 : P \; suc(x) : Q$	integer case
$case \; L \; of \; \{x\}_N \; in \; P$	shared-key decryption
$case \; L \; of \; \{[x]\}_N \; in \; P$	decryption
$case \; L \; of \; [\{x\}]_N \; in \; P$	signature check

Table 2. Spi Calculus processes

terms M and N. The 0 and $suc(M)$ terms represent the value of zero and the logical successor of some term M, respectively. A variable x represents any term, and it can be bound once to the value of another term. A variable that is not bound is free. Names can be regarded as special kinds of variables that can take only atomic terms as values. Then, free variables include free names. The term $\{M\}_N$ represents the encryption of the plaintext M with the symmetric key N, while $H(M)$ represents the result of hashing M. The M^+ and M^- terms represent the public and private part of the keypair M respectively, while $\{[M]\}_N$ and $[\{M\}]_N$ represent public key and private key asymmetric encryptions respectively. Informally, the $\overline{M}\langle N \rangle .P$ process sends message N on channel M, and then behaves like P, while the $M(x).P$ process receives a message from channel M, and then behaves like P, with x bound to the received value in P. A process P can perform an input or output operation iff there is a reacting process Q that is ready to perform the dual output or input operation. Note, however, that processes run within an environment (the attacker) that is always ready to perform input or output operations. Composition $P|Q$ means parallel execution of processes P and Q, while replication $!P$ means an unbounded number of instances of P ran in parallel. The restriction process $(\nu n)P$ indicates that n is a fresh name (i.e. not previously used, and unknown to the attacker) in P. The match process executes like P, if M equals N, otherwise is stuck. The nil process does nothing. The pair splitting process binds the variables x and y to the components of the pair M, otherwise, if M is not a pair, the process is stuck. The integer case process executes like P if M is 0, else it executes like Q if M is $suc(N)$ and x is bound to N, otherwise the process is stuck. If L is $\{M\}_N$, then the shared-key decryption process executes like P, with x bound to M, else it is stuck, and analogous reasoning holds for the decryption and signature check processes.

Note that, when defining the translation function, we focus on the sequential part of the Spi Calculus only, i.e. we do not deal with the translation of composition and replication, and we do not deal with recursive processes. This is not an important limitation, and it does not prevent us from dealing with most security protocols, because each protocol role normally consists of finite sequential behavior, made up of a short sequence of message exchanges, with checks on the received data. This sequential behavior can then be instantiated in several concurrent runs, in the various hosts of the distributed system. Thus, the composition operator is typically used, in protocol specifications, only to instantiate different protocol roles that run concurrently, and the replication operator is typically used only to describe the possibility to run an unbounded number of protocol sessions. Recursion could be useful to describe protocol runs of unbounded length, but this feature is normally not allowed by verification tools.

The semantics of the Spi Calculus has been originally expressed for closed processes, by means of a reaction relation $P \rightarrow P'$ and a reduction relation $P > P'$ [3]. $P \rightarrow P'$ means that P can evolve into P' after a message exchange between two parallel components of P, while $P > P'$ means that P can evolve into P' by performing some other (internal) operation. In this paper, instead, we need expressing the semantics of open processes, i.e. the possible evolutions of an open process P, regardless of its environment. For this purpose, we introduce a classical labelled transition system (LTS). In this system, a τ transition $P \xrightarrow{\tau} P'$ means $P \rightarrow P'$ or $P > P'$, i.e. P can evolve into P' without interaction with its environment. Instead, $P \xrightarrow{M!N} P'$ and $P \xrightarrow{M?N} P'$ mean that P can interact with its environment by respectively sending or receiving N on channel M. Formally,

$$\begin{array}{lll} P \xrightarrow{M!N} P' & \text{means} & \exists\overline{y}|\forall Q.\ (P|M(x).Q) \rightarrow \\ & & \qquad (\nu\overline{y})(P'|Q[N/x]) \\ P \xrightarrow{M?N} P' & \text{means} & \exists\overline{y}|\forall Q.\ (P|(\nu\overline{y})\overline{M}\,\langle N\rangle\,.Q) \rightarrow \\ & & \qquad (\nu\overline{y})(P'|Q) \end{array}$$

where $\overline{y}$ is a possibly empty list of names.

According to these definitions, it can be shown that the semantics of Spi Calculus sequential processes can be expressed by the rules

$$\begin{array}{rcl} \overline{M}\,\langle N\rangle\,.P & \xrightarrow{M!N} & P \\ M(x).P & \xrightarrow{M?N} & P[^N/_x] \\ [M\ is\ M]\,P & \xrightarrow{\tau} & P \\ let\ (x,y) = (M,N)\ in\ P & \xrightarrow{\tau} & P[^M/_x][^N/_y] \\ case\{M\}_N\ of\ \{x\}_N\ in\ P & \xrightarrow{\tau} & P[^M/_x] \end{array}$$

$$\frac{P \xrightarrow{\mathcal{L}} P'}{(\nu n)P \xrightarrow{\mathcal{L}} (\nu n)P'}$$

where $\mathcal{L}$ ranges over labels.

3.2. The Type System

Spi Calculus terms are untyped, which enables, during formal verification, to find type flaw attacks, i.e. attacks that are based on type confusion. However, since Java is statically typed, in order to enable the former language to be translated to the latter, it is necessary to assign a static type to every term used in a Spi Calculus specification. Types can be inferred automatically to some extent, so that the user work is minimized. However, unfortunately, it is not possible to reuse existing type systems for the Spi Calculus, because they have different purposes. For example, in [15], a generic type system is developed in order to describe some process behavior properties, such as deadlock-freedom or race-freedom (which, by the way, are not so related to some common security properties, such as secrecy or authentication). It turns out that, for our purposes, the type system in [15] is even too much expressive about program behavior, but it does not assign static types to terms, thus being useless for our purposes.

The type system and associated type inference algorithm developed in this paper recall some standard type systems for the λ-calculus, such as the one in [20]. Essentially, the type system allows the type of a term to be inferred by looking at the context where that term is used. The type system relies on a set of known, hierarchically related (by the subtype relation $<:$) types, depicted in figure 1. $Message$ is the top type, representing any message, so that every term has type $Message$. The types that directly descend from $Message$ correspond to different forms of Spi Calculus terms, while the subtypes of $Name$ correspond to different usages of names. The user is allowed to extend the given type hierarchy, by adding more specialized types.

In order to formalize the type system, a typing context Γ is defined as a set containing type assignments of the form $x : A$, where x ranges over variables and A over types. The judgment $\Gamma \vdash M : A$ means that, within the typing context Γ, which must contain type assignments for all the free variables of M, M is well formed and must have type A (it can still have substypes of A). The judgment $\Gamma \vdash P$ means that process P is well formed in the typing context Γ. Note that the

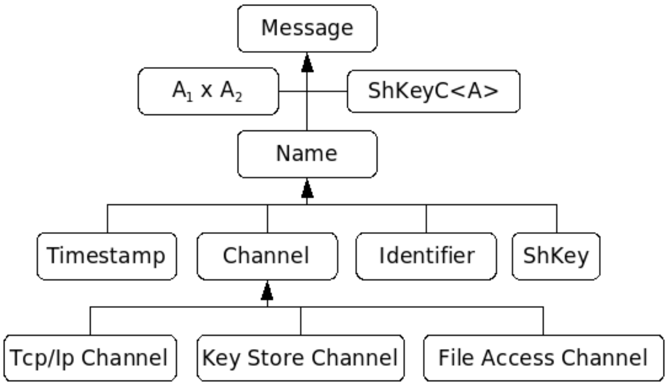

Figure 1. Type hierarchy for the selected Spi Calculus subset.

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A}\ \text{(T-Var)} \qquad \frac{\Gamma \vdash M_1 : A_1 \qquad \Gamma \vdash M_2 : A_2}{\Gamma \vdash (M_1, M_2) : A_1 \times A_2}\ \text{(T-Pair)} \qquad \frac{\Gamma \vdash M : A \qquad \Gamma \vdash K : ShKey}{\Gamma \vdash \{M\}_K : ShKeyC\langle A\rangle}\ \text{(T-ShKC)}$$

$$\frac{\Gamma \vdash M : Channel \qquad \Gamma \vdash N : Message \qquad \Gamma \vdash P}{\Gamma \vdash \overline{M}\,\langle N\rangle\,.P}\ \text{(P-Out)} \qquad \frac{\Gamma \vdash M : Channel \qquad \Gamma, x : A \vdash P}{\Gamma \vdash M(x).P}\ \text{(P-In)}$$

$$\frac{\Gamma \vdash M : Message \qquad \Gamma \vdash N : Message \qquad \Gamma \vdash P}{\Gamma \vdash [M\ is\ N]\ P}\ \text{(P-Match)} \qquad \frac{}{\Gamma \vdash \mathbf{0}}\ \text{(P-Nil)} \qquad \frac{\Gamma, n : A \vdash P \qquad A <: Name}{\Gamma \vdash (\nu n)P}\ \text{(P-Restr)}$$

$$\frac{\Gamma \vdash M : ShKeyC\langle A\rangle \qquad \Gamma \vdash K : ShKey \qquad \Gamma, x : A \vdash P}{\Gamma \vdash case\ M\ of\ \{x\}_K\ in\ P}\ \text{(P-ShKD)} \qquad \frac{\Gamma \vdash M : A_1 \times A_2 \qquad \Gamma, x : A_1, y : A_2 \vdash P}{\Gamma \vdash let\ (x, y) = M\ in\ P}\ \text{(P-PSplit)}$$

Figure 2. Typing rules for the selected Spi Calculus subset.

proposed type system does not assign types to processes, but only to terms. Indeed, in order to enable translation into Java, it is sufficient to assign a static type to each term, because terms are translated into Java typed data, while this is not needed for processes, which are translated into sequences of Java statements. For P to be well formed within Γ, it is necessary (but not sufficient) that all of its free variables appear in Γ. Given a generic Spi Calculus process P, it may not be possible to find Γ such that P is well formed. Since our translation function only translates well formed processes, it turns out that only the set Spi of well formed processes, which is a subset of all Spi Calculus processes, is translated by our function.

The typing rules for the Spi Calculus subset considered in this work are reported in figure 2; this type system uses the standard subsumption rule (not shown). For brevity, only some significant rules are commented. The (T-Pair) rule states that if M_1 has type A_1 and M_2 has type A_2, then it is possible to state that the pair (M_1, M_2) has type $A_1 \times A_2$. Note that this formalization keeps, for each pair, information on the types of the contained items. A possible Java implementation of this feature can be obtained by using Java generic types, introduced in Java 5. Otherwise, a single $Pair$ type could be defined. The latter formalization would require some run-time downcasts, as discussed later. The (P-Match) rule states that if M and N are two well formed messages, and P is a well formed process, then it is possible to try to match M and N, and then to behave like P. Note that the (P-Match) rule does not require M and N to have the same type: we consider well formed a process where M and N are matched, even if they have different, incompatible types (e.g. M is typed as $Name$ and N as $Name \times Name$). This is not an issue, because when the desired security properties are checked against the untyped Spi Calculus, type flaw attacks are taken into account, so even an erroneous Java implementation that would consider equal two terms with incompatible types, would have been considered in the formal verification step. By the way, adding to (P-Match) the constraint stating that the type of M must equal the type of N would also be possible. In the (P-ShKD) rule, note that the third premise requires the variable x to be added to the typing context, because x appears free in P, thus satisfying the necessary condition on Γ. The same reasoning holds for the first premise of (P-Restr). In the latter rule, however, it is also required that $A <: Name$ holds, because the restriction process generates a fresh *name*, and not a generic term.

We point out that using Java generic types can save run-time downcasts, when the type of a term is statically known; if the type is to be discovered at run-time, then a downcast will be necessary anyway, and it may fail. If generic types are not used, then more run-time downcasts will be necessary; however, it is fairly easy to formally show that all downcasts that could be replaced by using generic types cannot fail at run-time. So, using generic types or not, is more a matter of run-time performances (the negligible time required to perform a type check in Java), rather than a matter of type system.

Note that this rather standard type system shares some common properties with other well known type systems [20]. In particular, the canonical form lemma can be proven, stating that if a term M has a particular type A, then it can only assume the particular values of A. Moreover, it can be shown that a type inference algorithm terminates finding the principal type for every term, if it is possible to find one.

Finally, the user can manually refine the type of a particular term. A user-provided type refinement for a given term c can be represented by adding a custom downcast rule, only valid for term c, to the type system. This is needed, because some types (e.g. the subtypes of $Channel$) cannot be inferred automatically only on the basis of their usage. For example, if the user wants to specify that c is a *Tcp/Ip Channel*, then the following rule is added.

$$\frac{\Gamma \vdash c : Channel \qquad Tcp/Ip\,Channel <: Channel}{\Gamma \vdash c : Tcp/Ip\,Channel}$$

Note that the properties of the type system are still preserved even when custom downcasts are added, thanks to the premises of the downcast rules. Indeed, the first premise ensures that the downcast is performed only if the type inference algorithm can infer that c must have type $Channel$; the second rule ensures that the downcast required by the user is coherent with the type hierarchy.

3.3. The Translation Function

The translation from Spi Calculus to Java is formalized by a set of functions, each of which represents a particular aspect of the translation. All of these functions operate on well formed Spi Calculus processes and terms, that is processes and terms for which a type derivation tree can be found.

Each sequential Spi Calculus process, typically representing one of the protocol roles, is translated into a sequence of Java statements implementing the Spi Calculus process. These statements are embedded into a `try` block, followed by `catch` and `finally` blocks, which are in turn embedded into a method, that is invoked when a protocol run is requested. All the Java code surrounding the generated statements is called here the "context". The generated method will have one input parameter for each free variable of the Spi Calculus process; for example, the $\overline{c}\,\langle M\rangle\,.c(x).\mathbf{0}$ process has two free variables, namely the channel c and the message M, so the generated Java method will have two input parameters, because it is assumed that the user will provide sensible values for these two variables.

Let Spi be the aforecited set of well formed Spi Calculus processes, $SpiTerm$ the set of well formed terms, and $Java$ the set of strings representing sequences of Java statements. Then, the function $tr_p : Spi, 2^{SpiTerm}, 2^{SpiTerm} \rightarrow Java$ generates the Java statements for the Spi Calculus process given as its first parameter. In Java, all variables must be declared and initialized before they can be used. The second parameter of tr_p, let us call it $built \in 2^{SpiTerm}$, traces the well formed terms that have already been declared and initialized in the Java code. Moreover, some value should be returned after a successful protocol run (e.g. a negotiated shared secret, or a secure session id). The third parameter of tr_p, let us call it $return \in 2^{SpiTerm}$, contains the well formed terms that must be returned if a protocol run ends successfully. In order to return the desired values to the user, a Java object declared as `Map<String,Message> _return` is maintained, that maps the Java name of every term to be returned onto its value (i.e. the Java object that implements it); the map is filled as long as the values to be returned become available.

Before showing the definition of tr_p, some auxiliary functions are introduced. $ub : SpiTerm, 2^{SpiTerm} \rightarrow 2^{SpiTerm}$ updates the $built$ set taken as second parameter, by adding to it a term M, taken as first parameter, and its subterms. Formally, ub is defined as

$$ub(M, built) = built \cup subterms(M)$$

where $subterms(M)$ is the set containing M and all its subterms.

$ret : SpiTerm, 2^{SpiTerm} \rightarrow Java$ generates the Java code that fills the `_return` map. It puts the reference to the Java object that represents the term that is passed as its first parameter into the map, if it is in the set of terms that must be returned when the protocol run ends. This set is the second parameter. For every function that returns Java code, the following typographical conventions are used: the returned text is quoted by double quotes; inside the quoted text, $italic$ is used for functions that return text, while `courier` is used for verbatim returned text. The ret function is formally defined as

$$ret(M, return) = \text{“”, if } M \notin return$$
$$\text{“}\texttt{\_return.put("}J(M)\texttt{",}\,J(M)\texttt{);}\text{”, otherwise}$$

where $J(M)$ is a bijection that gives the name of the Java variable for term M, by mangling it.

$tr_t(M, built, return) =$ "", if $M \in built$
$tr_t(n, built, return) =$
  "$T(n)$ $J(n)$ `= new` $Ts(n)$`(`$Param(Ts(n))$`);`
  $ret(n, return)$"
$tr_t((M, N), built, return) =$
  "$tr_t(M, built, return)$ $tr_t(N, ub(M, built), return)$
  $T((M, N))$ $J((M, N))$ `= new`
  $Ts((M, N))$`(`$J(M)$`,` $J(N)$`,`
  $Param(Ts((M, N)))$`);` $ret((M, N), return)$"
$tr_t(\{M\}_N, built, return) =$
  "$tr_t(M, built, return)$ $tr_t(N, ub(M, built), return)$
  $T(\{M\}_N)$ $J(\{M\}_N)$ `= new` $Ts(\{M\}_N)$`(`$J(M)$`,`
  $J(N)$`,` $Param(Ts(\{M\}_N))$`);` $ret(\{M\}_N, return)$"

Figure 3. Definition of the tr_t function.

$tr_t : SpiTerm, 2^{SpiTerm}, 2^{SpiTerm} \rightarrow Java$ takes a term M, the $built$ set and the $return$ set, and generates the Java code that "builds", that is declares and initializes, the Java variable $J(M)$ for the given Spi Calculus term M, if it has not yet been built. Note that our translation relies on a Java library, called *SpiWrapper*, that offers an abstract Java class for each type depicted in figure 1. Each abstract class represents a type, and implements the operations that can be performed on that type. For instance, the abstract class `Pair<A,B>` offers the methods `A getLeft();` and `B getRight();`, that retrieve the first and second items of the pair, allowing the pair splitting process to be implemented. Every SpiWrapper class is abstract, because only the internal data representation is handled, while the marshalling functions, encoding the internal representation into the external one that is sent over the network and vice versa, are declared, but not implemented. This gives the user the ability to define his own marshalling layer, by extending the abstract class, and implementing the marshalling and unmarshalling functions. It could be argued that letting the user implement the marshalling functions could introduce security flaws that were not present in the abstract model. However, as stated in [22], if some static checks on the user-written code are performed, it is possible to avoid such flaws.

The tr_t function is formally defined in figure 3. All declared Java variables are actually also marked as `final`, which however is omitted here for brevity. For every term M, if M is already in the $built$ set, then no code is generated, because the Java variable has already been declared and initialized. The $T(M)$ function returns the inferred type for the term M, which corresponds to one of the SpiWrapper abstract classes. The $Ts(M)$ function returns instead the name of the concrete user-provided Java class implementing the marshalling functions and extending the class returned by $T(M)$. Finally the $Param(Ts(M))$ function returns some user-defined parameters needed to make the protocol interoperable; such parameters depend on the type of the term. For example, if the type of term M is $ShKey$ (a shared key), then the parameters will be the key length, the key type and the desired JCA provider. The reader should not be distracted by interoperability details, though; more details on interoperability can be found in [21]. In the name n case, the code emitted by the ret auxiliary function is appended to the generated code, so that, if n is to be returned, it is added to the `_return` map as soon as n has been declared and initialized.

In the pair (M, N) case, first tr_t is invoked on M and N, to ensure they are built. Note that, by invoking $tr_t(N, ub(M, built), return)$, N is built by taking into account that M and all its subterms have already been built, so they are not built twice. For example, if $M = (a, b)$ and $N = (b, c)$, then b is built when M is built, and it must not be built again when N is built. Once M and N are built, tr_t appends the code that actually builds the pair, possibly appending the code needed to return the pair. Note that it is only needed to explicitly call ret on the pair, and not on the components, because each invocation of tr_t on the components will ensure they are added, if needed, to the `_return` map as soon as they are built.

Finally, in the shared key ciphered $\{M\}_N$ case, first M and N are built, then the shared key ciphered object is instantiated and assigned to the variable named $J(\{M\}_N)$, and the ret function is invoked.

Note that no case is available for variables. Indeed, variables cannot be "built", rather, they are declared and assigned to Java objects by the code implementing Spi Calculus processes that bind variables.

Now that all the auxiliary functions have been defined, the formal definition of tr_p is given in figure 4. Like for tr_t, all declared variables are also marked as `final`, though not shown here. When translating the output process, first M and N are built, then the `send` method is invoked on the channel referenced by $J(M)$, passing $J(N)$ as first argument, so that it

$tr_p(\mathbf{0}, built, return) =$ “”
$tr_p(\overline{M}\langle N\rangle.P, built, return) =$
“$tr_t(M, built, return)$ $tr_t(N, ub(M, built), return)$
$J(M)$`.send(`$J(N)$`,`$Param(Ts(N))$`);`
$tr_p(P, ub(M, built) \cup ub(N, built), return)$”
$tr_p(M(x).P, built, return) =$
“$tr_t(M, built, return)$
$T(x)$ $J(x)$ `=` $J(M)$`.receive(`$Ts(x)$`.class,`
$Param(Ts(x))$`);` $ret(x, return)$
$tr_p(P, ub(M, built) \cup ub(x, built), return)$”
$tr_p((\nu n)P, built, return) =$
“$tr_t(n, built, return)$ $tr_p(P, ub(n, built), return)$”
$tr_p([M\ is\ N]P, built, return) =$
“$tr_t(M, built, return)$ $tr_t(N, ub(M, built), return)$
`if (!`$J(M)$`.equals(`$J(N)$`))`
`{ throw new MatchException(); }`
$tr_p(P, ub(M, built) \cup ub(N, built), return)$”
$tr_p(let\ (x, y) = M\ in\ P, built, return) =$
“$tr_t(M, built, return)$
$T(x)$ $J(x)$ `=` $J(M)$`.getLeft();` $ret(x, return)$
$T(y)$ $J(y)$ `=` $J(M)$`.getRight();` $ret(y, return)$
$tr_p(P, ub(M, built) \cup ub(x, built) \cup ub(y, built),$
$return)$”
$tr_p(case\ L\ of\ \{x\}_N\ in\ P, built, return) =$
“$tr_t(L, built, return)$ $tr_t(N, ub(L, built), return)$
$T(x)$ $J(x)$ `=` $J(L)$`.decrypt(`$J(N)$`,`
$Param(Ts(x))$`);` $ret(x, return)$ $tr_p(P,$
$ub(L, built) \cup ub(N, built) \cup ub(x, built), return)$

Figure 4. Definition of the tr_p function.

is sent over the channel. For the input process, after the channel M has been built, the `T receive(Class<T>,...);` method is invoked. Its arguments allow this method to create an instance of $Ts(x)$, fill it with the received data, and return its reference, that is assigned to the Java variable $J(x)$. In the decryption process the decryption key is passed as argument. If `ShKC<T>` is the type of $J(L)$, the decrypt method returns a newly created object of type `T` containing the decrypted data. In the restriction process, the name n is built, then the rest of the process is translated. Note that, when n is built, a constructor is called, which generates the Java implementation of a new fresh name of the expected type. With the pair splitting process, first M is built, then the x and y variables are declared and assigned, which corresponds to the Spi Calculus binding of a variable. Note that when the following P process will then be translated, M, x and y will all have already been built; indeed, x and y are not built by the `new` operator, rather, being variables, they are assigned the value (Java reference) of another term. When translating the match case, after having built M and N, if they are not equal, execution is stopped by throwing an exception, which is handled by the context, else the match is successful, and execution can continue with the translation of the P process, where both M and N are marked as built. The context handles the exception by setting the `_return` map to `null`, thus simulating a stuck Spi Calculus process (a successful run of a protocol that does not need to return any value, still returns an empty map, and not `null`). Note that when a message is received from a channel, or a plaintext is reconstructed from a ciphertext, a new SpiWrapper object holding the obtained data must be created, even if the corresponding Spi Calculus term is already instantiated in another Java object. For example, the Java code implementing the Spi Calculus process $\overline{c}\langle M\rangle.c(x).\mathbf{0}$, will store one object for the M term, and one different object for the received x term. It may happen, however, that x is assigned the same value of M (simulating that the Spi Calculus process receives exactly the M term back), although they are two different objects. For this reason, equality of objects cannot be checked by means of reference equality, but the `equals` method must check if the value of the two objects is the same. Using singleton instances to represent Spi Calculus terms, and thus letting the match case check for reference equivalence, would also be possible, but it would not be better. Indeed, in the `receive` (`decrypt`) method, it would be necessary to check the content of received (decrypted) data, to decide if their representing singleton is already instantiated or not.

Handling channels (e.g. TCP/IP channels), and in general all Java resources that need to be allocated before usage and disposed after usage, requires a little more effort. For scope reasons, all the channels are declared before the `try` block. For each channel, opening it is performed at the same time when it is built, while closing it is done, if it has been initialized and must not be returned, in the `finally` block in the context, either if protocol ends successfully or it gets stuck by throwing an exception (accessing the channels from the `try` and `finally` blocks is why it is needed to declare them before the `try` block). Of course, the formal translation functions are modified so that the channels are just assigned, and not declared, when they are built, which simply amounts to omit their type when assigning them. So, the skeleton of the generated method looks like

```
Map<String,Message>
  generatedSpi(@InputParams@) {
 Map<String,Message> _return =
  new TreeMap<String,Message>();
 @ChannelsDeclaration@
 try { @GeneratedSpiImpl@ }
 catch { _return = null; }
 finally { @CloseChannels@ }
 return _return;
}
```

4. Soundness

The formal definitions of the translation functions tr_p, tr_t, ub and ret allow some properties about the generated code to be stated. Given a well formed Spi Calculus process P, let $fv(P) \in 2^{SpiTerm}$ be the set of the free variables in P, and $t(P) \in 2^{SpiTerm}$ be the set of all the terms in P. Moreover, given a well formed term M, let $atoms(M) \subseteq subterms(M)$ be the set of all names and variables in M.

Theorem 1. *If $\Gamma \vdash P$ and $return \subseteq t(P)$, then $tr_p(P, fv(P), return)$ is well formed.*

By "well formed" we mean a sequence of Java statements that, put in the context, forms a Java program that compiles, i.e. a Java program that is correct from a syntactic and static semantic point of view. Note that the terms in $fv(P)$ are the protocol input parameters, so their corresponding Java variables are already declared in the context, as method input parameters.

In order to prove theorem 1, a lemma is first introduced. From now on, we make the following reasonable assumptions. We assume that, for any term M, the relation $Ts(M) <: T(M)$ holds, which can be enforced by checking that the user-provided implementation of the marshalling functions is done by extending the appropriate SpiWrapper abstract classes. Moreover, for every constructor c offered by the SpiWrapper class $T(M)$, it is assumed that a constructor c' exists in the user-provided class $Ts(M)$ that extends $T(M)$. The c' constructor is assumed to have the same parameters of c and to be implemented only by a call to the `super` method. Unfortunately, Java does not allow constructor inheritance, so the $Ts(M)$ class is not sintactically forced to have such constructor, and the existence of this constructor must be checked by other means. Finally, we assume that $Param(Ts(M))$ returns the correct number and type of user-provided interoperability parameters for the constructor $Ts(M)$`(...)`.

Lemma 1. *Let $\Gamma \vdash M : A$, $built \supseteq atoms(M)$ and $return \subseteq SpiTerm$. If built is closed under the $subterms$ function, if $\mathcal{C}[\cdot]$ is a Java context that compiles, and that includes the declaration and initialization of all and only Java variables that correspond to the terms in built, then $\mathcal{C}[tr_t(M, built, return), \cdot]$ is a context that compiles, and that includes the declaration and initialization of all and only Java variables that correspond to the terms in $ub(M, built)$.*

Proof. If $M \in built$, then tr_t always returns the empty string and, by hypotheses, $\mathcal{C}[\cdot]$ compiles and declares and initializes all terms in $ub(M, built) = built$, so the case is proven. The interesting case is when $M \notin built$.

The proof is carried out by structural induction over M. The base case, where M is a name or variable n, is trivial: $atoms(n) = \{n\} \subseteq built$, so this case cannot happen, and it is proven as an instance of the $M \in built$ case. The inductive cases are now analyzed.

case $M = (N, N')$ By definition, $atoms((N, N')) = atoms(N) \cup atoms(N')$. In particular, $built \supseteq atoms(N)$, so the inductive hypotheses apply, and

$$\mathcal{C}'[\cdot] = \mathcal{C}[tr_t(N, built, return), \cdot]$$

compiles and builds all and only terms in $ub(N, built)$. Since $ub(N, built) \supseteq atoms(N')$, the inductive hypotheses apply for the appended code too, so

$$\mathcal{C}''[\cdot] = \mathcal{C}'[tr_t(N', ub(N, built), return), \cdot]$$

compiles and builds all and only terms in $ub(N', ub(N, built)) = ub(N, built) \cup ub(N', built)$. Note that properly updating the $built$ set allows to satisfy the inductive hypotheses and to avoid duplicate declarations of variables when invoking tr_t on N' too.

By the canonical forms lemma, it follows that (N, N') must have the $A_1 \times A_2$ type, whose corresponding SpiWrapper class offers the $T((M, N))$`(A_1 left, A_2 right, ...)` constructor. By the stated assumptions, $Ts((M, N)) <: T((M, N))$, so the constructor $Ts((M, N))$`(A_1 left, A_2 right, ...)` exists. Moreover, since (N, N') is well formed, and the (T-Pair) is the only rule that could be applied, it follows that N must have type A_1 and N' must have type A_2, so it is verified that the generated code is well formed in $\mathcal{C}''[\cdot]$, and all only the terms in $ub((N, N'), built)$ are built.

case $M = \{L\}_N$ By definition, $atoms(\{L\}_N) = atoms(L) \cup atoms(N)$. So, like in the pair case, the inductive hypotheses apply, and

$$\mathcal{C}'[\cdot] = \mathcal{C}[tr_t(L, built, return)\ tr_t(N, ub(L, built), return), \cdot]$$

compiles and builds all and only the variables in $ub(L, built) \cup ub(N, built)$. By the canonical form lemma, $\{L\}_N$ must have type $ShKC\,\langle A \rangle$, whose SpiWrapper class offers the $T(\{L\}_N)$`(A plaintext, ShKey key,...)` constructor. Again, it is assumed that the corresponding constructor in $Ts(\{L\}_N)$ exists. Moreover, by the well formedness of $\{L\}_N$, it follows that L must have type A and N must have type $ShKey$, so the generated code is well formed in $\mathcal{C}'[\cdot]$, and all and only the terms in $ub(\{L\}_N, built)$ are built.

□

Proof of theorem 1. The following will be proven, which implies the theorem. Let $built$ be closed under the $subterms$ function, and $built \supseteq fv(P)$. For any context $\mathcal{C}[\cdot]$ that compiles and includes declaration and initialization of all and only Java variables that correspond to the terms in $built$, $\mathcal{C}[tr_p(P, built, return)]$ compiles. Note that the main theorem is implied, because it is a particular instance of this predicate: $built = fv(P)$, and all free variables are declared and assigned in the context.

The proof is carried out by structural induction over the process P. The base case, where P is $\mathbf{0}$, is trivial, because the empty string is well formed.

Some significant inductive cases are now reported.

case P **is** $\overline{M}\,\langle N \rangle\,.Q$ By definition, $fv(P) = atoms(M) \cup atoms(N) \cup fv(Q)$, in particular $built \supseteq atoms(M)$, and lemma 1 can be applied. So

$$\mathcal{C}'[\cdot] = \mathcal{C}[tr_t(M, built, return), \cdot]$$

compiles and builds all and only terms in $ub(M, built)$. Again, lemma 1 can be applied for

$$\mathcal{C}''[\cdot] = \mathcal{C}'[tr_t(N, ub(M, built), return), \cdot]$$

too, which compiles and builds all and only terms in $ub(M, built) \cup ub(N, built)$. By the well formedness of P, being (P-Out) the only applicable rule, it follows that M must have type $Channel$ and N must have type $Message$. In particular, the SpiWrapper class implementing the $Channel$ type offers the `void send(Message toSend,...)` method, that can indeed be invoked. So

$$\mathcal{C}'''[\cdot] = \mathcal{C}''[\texttt{M.send(N,...);}, \cdot]$$

compiles and does not declare any other Java variable.

Since

$$ub(M, built) \cup ub(N, built) \supseteq built \supseteq fv(P) \supseteq fv(Q)$$

it follows that the inductive hypotheses hold, so

$$\mathcal{C}'''[tr_p(Q, ub(M, built) \cup ub(N, built), return)]$$

compiles, and the whole generated code is well formed.

case P **is** $M(x).Q$ By definition, $fv(P) = atoms(M) \cup fv(Q) \setminus \{x\}$ because x is bound by the input process. For the same reason, $x \notin fv(M)$ must hold too. So, $built \supseteq atoms(M)$, and lemma 1 can be applied, letting

$$\mathcal{C}'[\cdot] = \mathcal{C}[tr_t(M, built, return), \cdot]$$

compile and build all and only terms in $ub(M, built)$. Again, by well formedness of P, it follows that M must have type $Channel$, whose corresponding SpiWrapper class offers the `T receive(Class<T> x,...)` method, that can indeed be invoked. Moreover, whatever is the type inferred for x, by the assumption $Ts(x) <: T(x)$, the assignment of variable x is well formed. So

$$\mathcal{C}''[\cdot] = \mathcal{C}'[\texttt{T x = M.receive(Ts.class,...);}, \cdot]$$

compiles. Also note that the generated code declares and assigns `x`, so in $\mathcal{C}''[\cdot]$ the declared and assigned variables are $ub(M, built) \cup ub(x, built) \supseteq fv(Q)$. So the inductive hypotheses hold, and

$$\mathcal{C}''[tr_p(Q, ub(M, built) \cup ub(x, built), return)]$$

compiles and the whole generated code is well formed.

case P **is** $(\nu n)Q$ By definition, $fv(P) = fv(Q) \setminus \{n\}$, which does not ensure that $built \supseteq atoms(n) = \{n\}$, so, in general, lemma 1 cannot be applied. Nevertheless, if $n \in built$, then $tr_t(n, built, return)$ generates no code, and n is already built in $\mathcal{C}[\cdot]$ by hypotheses, so $\mathcal{C}'[\cdot] = \mathcal{C}[\text{“”}, \cdot]$ compiles and builds all and only terms in $ub(n, built) = built$. Conversely, if $n \notin built$, by inspection of tr_t, it follows that

$$\mathcal{C}'[\cdot] = \mathcal{C}[\texttt{T n = new Ts(...);}, \cdot]$$

compiles and all and only terms in $ub(n, built)$ are built in $\mathcal{C}'[\cdot]$.

By the hypothesis $built \supseteq fv(P)$, it follows that $ub(n, built) \supseteq fv(Q)$, so by inductive hypotheses,

$$\mathcal{C}'[tr_p(Q, ub(n, built), return)]$$

compiles, and the whole generated code is well formed.

The proof structure is very similar for the remaining cases. In general, it is shown that

$$\mathcal{C}'[\cdot] = \mathcal{C}[tr_t(\ldots), \cdot]$$

compiles and ensures that all needed variables are built. Then, it is shown that the method `m` called on object `o` is available, and all its parameters match, because of the well formedness of P. Finally, it is shown that the inductive hypotheses apply, so that the translation of the remaining process is well formed, letting the whole generated code be well formed. □

Now, let us go for the semantic properties of the generated Java code. The main goal is to show that, under some assumptions on the behavior of the SpiWrapper classes, the generated Java code simulates the corresponding Spi Calculus process. More precisely, a weak simulation relation between the generated Java code and the corresponding Spi Calculus is shown. Note that security properties, such as secrecy and authentication, are safety properties of the global concurrent system, so their preservation in the refinement from Spi Calculus to Java is implied by the simulation relation. Briefly, a weak simulation relation binds the transitions between external states of an abstract process to the transitions between external states of a concrete process, but each process is still allowed to perform any internal step in between two external states. More details about the weak refinement used here can be found, for example, in [24].

In order to state and prove the simulation relation, the semantics of the Spi Calculus must be written in a slightly different way. According to the notation introduced in section 3, a transition can be in general written as

$$P \xrightarrow{\mathcal{L}} P'\sigma'$$

$$
\begin{array}{rcl}
J(n) = \texttt{a} \quad \Rightarrow \quad \texttt{T a = new Ts(params)}, Val & \xrightarrow{\tau} & unit, \{(\texttt{a}, n)\} \cup Val \\
\texttt{c.send(M)}, \{(\texttt{c}, c), (\texttt{M}, M)\} \cup Val & \xrightarrow{c!M} & unit, \{(\texttt{c}, c), (\texttt{M}, M)\} \cup Val \\
\texttt{T x = c.receive(Ts.class)}, \{(\texttt{c}, c)\} \cup Val & \xrightarrow{c?N} & unit, \{(\texttt{c}, c), (\texttt{x}, N)\} \cup Val \\
M = N \quad \Rightarrow \quad \texttt{a.equals(b)}, \{(\texttt{a}, M), (\texttt{b}, N)\} \cup Val & \xrightarrow{\tau} & \texttt{true}, \{(\texttt{a}, M), (\texttt{b}, N)\} \cup Val \\
M \neq N \quad \Rightarrow \quad \texttt{a.equals(b)}, \{(\texttt{a}, M), (\texttt{b}, N)\} \cup Val & \xrightarrow{\tau} & \texttt{false}, \{(\texttt{a}, M), (\texttt{b}, N)\} \cup Val \\
\texttt{Pair<T1,T2> A\_B = new Pairs<T1,T2>(A,} & & \\
\texttt{B,params)}, \{(\texttt{A}, A), (\texttt{B}, B)\} \cup Val & \xrightarrow{\tau} & unit, \{(\texttt{A}, A), (\texttt{B}, B), (\texttt{A\_B}, (A, B))\} \cup Val \\
\texttt{T x = p.getLeft()}, \{(\texttt{p}, (M, N))\} \cup Val & \xrightarrow{\tau} & unit, \{(\texttt{p}, (M, N)), (\texttt{x}, M)\} \cup Val \\
\texttt{T y = p.getRight()}, \{(\texttt{p}, (M, N))\} \cup Val & \xrightarrow{\tau} & unit, \{(\texttt{p}, (M, N)), (\texttt{y}, N)\} \cup Val \\
\texttt{ShKC<T> s = new ShKCs<T>(M,K,params)}, & & \\
\{(\texttt{M}, M), (\texttt{K}, K)\} \cup Val & \xrightarrow{\tau} & unit, \{(\texttt{M}, M), (\texttt{K}, K), (\texttt{s}, \{M\}_K)\} \cup Val \\
\texttt{T x = p.decrypt(k,params)}, \{(\texttt{p}, \{M\}_K), (\texttt{k}, K)\} \cup Val & \xrightarrow{\tau} & unit, \{(\texttt{p}, \{M\}_K), (\texttt{k}, K), (\texttt{x}, M)\} \cup Val
\end{array}
$$

Table 3. Formal semantics for the selected SpiWrapper classes subset.

where $\mathcal{L}$ is the transition label, and σ' is a (possibly empty) substitution that binds variables. If we do not apply substitutions, but we leave them explicitly indicated as postfix operators, a generic state P of a system run will be written as $P_1\sigma$, where P_1 includes all the free variables of P, as well as all the bound variables that have been substituted in the past evolution that has led to P, while σ incorporates such substitutions. Using this representation for processes, a generic transition can be written as

$$P_1\sigma \xrightarrow{\mathcal{L}} P_1'\sigma\sigma'$$

and a state can be divided into two components: a process expression followed by a variable substitution.

In the LTS for the Spi Calculus, all states are marked as external.

An LTS for a Java sequential program obtained by our transformation function is now defined. In order to relate the Java behavior with the Spi Calculus behavior, the Java LTS uses the same abstract labels used for the Spi Calculus LTS. Let $JavaVar$ be the set of Java variables declared in the Java program. Then a generic state (j, Val) is defined by the code that is going to be executed, called j, plus a function $Val : JavaVar \rightarrow SpiTerm$, mapping each declared Java variable to the corresponding Spi Calculus term that the referenced Java object is implementing. For example, $Val(\texttt{var}) = \{M\}_N$ means that the Java variable `var` references a Java object that implements the $\{M\}_N$ Spi Calculus term. The intended invariant that should hold is $Val(J(M)) = \sigma(M)$, where σ is the substitution in the corresponding Spi Calculus process. A Java state (J, Val) is defined to be external iff $J = tr_p(P, dom(Val), return)$ for any Spi Calculus process P that is not a restriction and return set $return$. The transitions of the form

$$j, Val \xrightarrow{\mathcal{L}} j', Val'$$

bring from one generic state to another, following an abstract operational semantics for the Java language. In this work, we formally define the operational semantics that must be implemented by the SpiWrapper classes, so that a weak simulation relation between Spi Calculus and the generated Java code can be found. The formal semantics that must be implemented by the selected subset of the SpiWrapper classes presented in this work is reported in table 3. A `void` method returns the $unit$ value, while `true` and `false` are the boolean values. It is assumed that if a method cannot succeed (for example, a wrong key is passed to the `decrypt` method), it throws an exception, that simulates the stuck process, as explained above. Formally, when an exception is thrown from any state (j, Val), the following semantics is assumed

$$j, val \xrightarrow{\tau} \perp$$

where $\perp$ is a special state that can never be external, and indicates that the control has passed to the context, due to the thrown exception. Standard congruence and computation semantic rules are assumed for the sequential concatenation of statements, and for the other Java statements.

The simulation relation S, that relates external Spi Calculus states to external Java states, is formally defined as

$$S(((\nu\overline{n})P)\sigma, (J, Val)) \Leftrightarrow J = tr_p(P, dom(Val), return) \,\wedge\, \sigma_{|fv(P)} = Val \circ J_{|fv(P)} \,\wedge\, \sigma \supseteq Val \circ J$$

for any Spi Calculus process P that is not a restriction, and any Val such that $dom(Val \circ J)$ is closed under the $subterms$ function.

Theorem 2. *If the SpiWrapper library behaves as specified in table 3, then*

$$S(P\sigma, (J, Val)) \ \wedge \ J, Val \xrightarrow{\tau}^{*} \xrightarrow{\mathcal{L}} \xrightarrow{\tau}^{*} J', Val' \Rightarrow \\ P\sigma \xrightarrow{\mathcal{L}} P'\sigma' \ \wedge \ S(P'\sigma', (J', Val'))$$

In order to prove theorem 2, some lemmata are first introduced.

Lemma 2. *For any term M such that $\Gamma \vdash M : A$, for any Val such that $dom(Val \circ J)$ is closed under the $subterms$ function, and for any substitution σ, there exists Val' such that*

$$\sigma_{|atoms(M)} = Val \circ J_{|atoms(M)} \ \wedge \ \sigma \supseteq Val \circ J \ \wedge \ (tr_t(M, dom(Val), return), Val \xrightarrow{\tau}^{*} unit, Val') \\ \Rightarrow \sigma(M) = Val'(J(M)) \ \wedge \ \sigma \supseteq Val' \circ J$$

and $dom(Val' \circ J)$ is closed under the $subterms$ function.

Proof. If $J(M) \in dom(Val)$, by hypotheses, $\sigma(M) = Val(J(M))$. Moreover, no code is generated, letting the final state equal to the initial state, so this case is proven.

If $J(M) \notin dom(Val)$, the proof is carried out by induction over the structure of M.

base case: M is n By definition, $atoms(n) = n$, which implies $J(n) \in dom(Val)$, so this case cannot happen, and it is proven as a particular instance of the $J(M) \in dom(Val)$ case.

inductive case: M is (N, N') By definition, $atoms((N, N')) = atoms(N) \cup atoms(N')$, in particular, for any name or variable n

$$n \in atoms(N) \Rightarrow n \in atoms((N, N')) \Rightarrow J(n) \in dom(Val)$$

So the inductive hypotheses hold, and

$$tr_t(N, dom(Val), return), Val \xrightarrow{\tau}^{*} unit, Val'$$

where $\sigma(N) = Val'(J(N))$, $\sigma \supseteq Val' \circ J$ and $dom(Val' \circ J)$ is still closed under the $subterms$ function. Note that the inductive hypotheses hold for the appended code too, so

$$tr_t(N', dom(Val'), return), Val \xrightarrow{\tau}^{*} unit, Val''$$

where $\sigma(N') = Val''(J(N'))$, $\sigma \supseteq Val'' \circ J$ and $dom(Val'' \circ J)$ is closed under the $subterms$ function.

In particular, $\{(J(N), \sigma(N)), (J(N'), \sigma(N'))\} \subseteq Val''$, so, the appended code executes as

$$T((N, N'))\ J((N, N')) \ \texttt{ = new } Ts((N, N'))\texttt{(}J(N)\texttt{,}\ J(N')\texttt{,}\ Param(Ts((N, N')))\texttt{)}, Val'' \xrightarrow{\tau} \\ unit, \{(J((N, N')), \sigma((N, N')))\} \cup Val''$$

Finally, by noting that $dom((\{(J((N, N')), \sigma((N, N')))\} \cup Val'') \circ J)$ is still closed under the $subterms$ function, it is possible to state that this case is proven.

Same reasoning applies to the remaining inductive cases. ☐

Lemma 3. *For any Spi Calculus process $((\nu\overline{n})P)\sigma$ such that P is not a replication, Val such that $dom(Val \circ J)$ is closed under the $subterms$ function*

$$\sigma_{|fv(P)} = Val \circ J_{|fv(P)} \ \wedge \ \sigma \supseteq Val \circ J \ \wedge \\ tr_p((\nu\overline{n})P, dom(Val), return), Val \xrightarrow{\tau}^{*} tr_p(P, dom(Val'), return), Val' \\ \Rightarrow S(((\nu\overline{n})P)\sigma, (tr_p(P, dom(Val'), return), Val'))$$

This lemma handles the restriction process, showing that its translation corresponds to the translation of the process that follows, provided the memory has been correctly set up.

Proof. The proof is carried out by induction over the number of restricted names.

case $(\nu\overline{n})$ **is empty** If there is no restricted name, then the initial state is equal to the final state and, by hypotheses, the simulation relation S holds in the final state.

case $(\nu\overline{n}) = (\nu m)(\nu\overline{m'})$ Note that $\sigma(m) = m$, because

$$((\nu m)(\nu\overline{m'})P)\sigma = (\nu m)(\nu\overline{m'})(P\sigma)$$

and m is not a free variable in $((\nu m)(\nu\overline{m'})P)$, so σ acts as the identity on m.

By the definition of tr_p, the

$$tr_t(m, dom(Val), return)\ tr_p((\nu\overline{m'})P, ub(m, dom(Val)), return)$$

functions are invoked. If $J(m) \in dom(Val)$, then by lemma 2, tr_t evolves in $unit$ and $\sigma(m) = Val(J(m))$. If $J(m) \notin dom(Val)$, then, by the definition of tr_t and by the assumed semantics of the SpiWrapper library

$$T(m)\ J(m)\ \texttt{ = new }\ Ts(m)\texttt{(}Param(Ts(m))\texttt{)}, Val \rightarrow unit, \{(J(m), \sigma(m))\} \cup Val$$

Note that the code appended by the ret function does not alter the state of the program, as far as the Spi Calculus simulation is concerned.

Let $Val' = \{(J(m), \sigma(m))\} \cup Val$, then $dom(Val') = ub(m, dom(Val))$ and the inductive hypotheses hold, so

$$tr_p((\nu\overline{m'})P, dom(Val'), return), Val' \xrightarrow{\tau}{}^{*}\ tr_p(P, dom(Val''), return), Val''$$

executes, and $S(((\nu\overline{m'})P)\sigma, (tr_p(P, dom(Val''), return), Val''))$ holds. Since $\sigma(m) = m = Val(J(m))$, it follows that $S(((\nu m)(\nu\overline{m'})P)\sigma, (tr_p(P, dom(Val''), return), Val''))$ holds too, thus proving the case.

□

Proof of theorem 2. The proof is carried out by inspection on the form of Spi Calculus processes. Note that the case when P is **0** can be neglected, because the generated Java code, which is the empty string, cannot evolve in any other state, so the theorem cannot be applied.

The most interesting cases are now reported. Since S holds in the initial state, it is not possible that the translated process P is a restriction. Restriction processes are indeed handled by lemma 3.

case P **is** $\overline{M}\,\langle N\rangle\,.Q$ By definition, $fv(P) = atoms(M) \cup atoms(N) \cup fv(Q)$, so

$$\forall\, m\ \cdot\ m \in atoms(M) \Rightarrow \sigma(m) = Val(J(m))$$

It follows that lemma 2 can be applied

$$tr_t(M, dom(Val), return), Val \xrightarrow{\tau}{}^{*}\ unit, Val'$$

where $\sigma(M) = Val'(J(M))$ and $dom(Val' \circ J)$ is still closed under the $subterms$ function. Inductive hypotheses hold for the appended code too, so

$$tr_t(N, dom(Val'), return), Val \xrightarrow{\tau}{}^{*}\ unit, Val''$$

where $\sigma(N) = Val''(J(N))$ and $dom(Val'' \circ J)$ is still closed under the $subterms$ function.

In particular, $\sigma(M) = Val(J(M))$ and $\sigma(N) = Val(J(N))$. Then, by the assumption on the behavior of the SpiWrapper library, the code executes as

$$J(M)\texttt{.send(}J(N)\texttt{,}\ Param(Ts(N))\texttt{)}, Val'' \xrightarrow{\sigma(M)!\sigma(N)} unit, Val''$$

As explained above, the ret function only adds an irrelevant side effect w.r.t. the Spi Calculus simulation. Let

$$J'' = tr_p(Q, ub(M, dom(Val)) \cup ub(N, dom(Val)), return) = tr_p(Q, dom(Val''), return)$$

The state of the Java program is then (J'', Val'').

On the Spi Calculus side, the process can evolve like

$$P\sigma = (\overline{M}\langle N\rangle .Q)\sigma \overset{\sigma(M)!\sigma(N)}{\rightarrow} Q\sigma$$

In general, Q can be a restriction $(\nu\overline{n})Q'$ (with zero or more restricted names). Since $fv(P) \supseteq fv(Q)$, lemma 3 can be applied, and the Java translation of the Q' process is proven to be S-related with the Q process, thus proving this case.

case P **is** $M(x).Q$ By definition, $fv(P) = atoms(M) \cup fv(Q) \setminus \{x\}$, and $x \notin atoms(M)$. It follows that $\sigma_{|atoms(M)} = Val \circ J_{|atoms(M)}$, and lemma 2 can be applied, letting

$$tr_t(M, dom(Val), return), Val \xrightarrow{\tau}^* unit, Val'$$

where $\sigma(M) = Val'(J(M))$ and $dom(Val' \circ J)$ is still closed under the $subterms$ function.

By the assumption on the behavior of the SpiWrapper library,

$$T(x)\ J(x) = J(M)\texttt{.receive(}Ts(x)\texttt{.class,}Param(Ts(x))\texttt{)}, Val' \overset{\sigma(M)?N}{\rightarrow} unit, Val''$$

where $Val'' = \{(J(x), N)\} \cup Val'$. Again, the code inserted by the ret function does not alter the state. Let

$$J'' = tr_p(Q, ub(M, dom(Val)) \cup ub(x, dom(Val)), return) = tr_p(Q, dom(Val''), return)$$

The Java state is (J'', Val'').

On the Spi Calculus side, the process can evolve like

$$P\sigma = (M(x).Q)\sigma \overset{\sigma(M)?N}{\rightarrow} Q\sigma[^N/_x]$$

Like in the output case, Q can be a restriction process $(\nu\overline{m})Q'$; by lemma 3, the translation of Q' is shown to be S-related with the Q process, thus proving the case.

case P **is** $[M\ is\ N]Q$ Like in the output case, lemma 2 can be applied twice, so

$$tr_t(M, dom(Val), return)\ tr_t(N, ub(M, dom(Val)), return), Val \xrightarrow{\tau}^* unit, Val'$$

where $\sigma(M) = Val'(J(M))$, $\sigma(N) = Val'(J(N))$, and $Val' \circ J$ is still closed under the $subterms$ function.

Two cases must be considered, either $\sigma(M) \neq \sigma(N)$ or $\sigma(M) = \sigma(N)$. If $\sigma(M) \neq \sigma(N)$, by the assumption on the behavior of the SpiWrapper library, and by applying the standard congruence and evaluation rules for the `if` statement, we have

```
if (!J(M).equals(J(N)))
  { throw new MatchException(); }
  tr_p(Q, ub(M, dom(val)) ∪ ub(N, dom(val)), return), Val'
```

$$\xrightarrow{\tau}$$

```
if (!false)
  { throw new MatchException(); }
  tr_p(Q, ub(M, dom(val)) ∪ ub(N, dom(val)), return), Val'
```

$$\xrightarrow{\tau}$$

```
if (true)
  { throw new MatchException(); }
  tr_p(Q, ub(M, dom(val)) ∪ ub(N, dom(val)), return), Val'
```

$$\xrightarrow{\tau}$$

```
throw new MatchException(), Val'
```

$$\xrightarrow{\tau}$$

$$\perp$$

Since this case does not lead to an external state, no simulation must be shown w.r.t. the Spi Calculus, so it can be disregarded.

If $\sigma(M) = \sigma(N)$, then by the assumption on the behavior of the SpiWrapper library, and by applying the standard congruence and evaluation rules for the `if` statement, we have

$$\begin{array}{l}
\texttt{if (!}J(M)\texttt{.equals(}J(N)\texttt{))} \\
\quad \{\ \texttt{throw new MatchException();}\ \} \\
\quad tr_p(Q, ub(M, dom(val)) \cup ub(N, dom(val)), return), Val' \\
\qquad\qquad\qquad\qquad\qquad \xrightarrow{\tau} \\
\texttt{if (!true)} \\
\quad \{\ \texttt{throw new MatchException();}\ \} \\
\quad tr_p(Q, ub(M, dom(val)) \cup ub(N, dom(val)), return), Val' \\
\qquad\qquad\qquad\qquad\qquad \xrightarrow{\tau} \\
\texttt{if (false)} \\
\quad \{\ \texttt{throw new MatchException();}\ \} \\
\quad tr_p(Q, ub(M, dom(val)) \cup ub(N, dom(val)), return), Val' \\
\qquad\qquad\qquad\qquad\qquad \xrightarrow{\tau} \\
tr_p(Q, ub(M, dom(val)) \cup ub(N, dom(val)), return), Val'
\end{array}$$

Let

$$J' = tr_p(Q, ub(M, dom(Val)) \cup ub(N, dom(Val)), return) = tr_p(Q, dom(Val'), return)$$

then (J', Val') is the Java state after the `if` statement.

On the Spi Calculus side,

$$P\sigma = [\sigma(M)\ is\ \sigma(N)](Q\sigma) \xrightarrow{\tau} Q\sigma$$

By noting that $fv(P) \supseteq fv(Q)$, and that Q can be a restriction process $(\nu\overline{n})Q'$, by lemma 3, the translation of Q' is shown to be S-related with Q, thus proving the case.

$\square$

Note that the initial state of a Java program could be an internal state, if the translated Spi Calculus process P is a restriction. Nevertheless, it is reasonale to assume that

$$\sigma_{|fv(P)} = \sigma = Val \circ J = Val \circ J_{|fv(P)}$$

because this means assuming that the user provided sensible values for all and only the free variables in P. So lemma 3 can be applied, enabling theorem 2, thus getting to the final result that the Java code simulates the Spi Calculus process from which it has been generated.

As a preliminary step towards developing a provably correct implementation of the SpiWrapper library, let us analyze for example the code for the abstract class that implements a pair:

```
abstract class Pair<A extends Message,
   B extends Message> extends Message {
 A left; B right;
 Pair(A left, B right) {
  super(); this.left = left;
  this.right = right; }
 A getLeft() { return this.left; }
 B getRight() { return this.right; }
 boolean equals(Object obj) {...}
}
```

It turns out that, for example, the implementation of the getLeft and getRight methods fits into the “Featherweight Java”, a very small subset of Java presented in [20], whose formal operational semantics is completely defined. Using this formalization, it is possible to simply show that this implementation is correct w.r.t. the requested semantics for the pair SpiWrapper type.

5. Conclusions

This paper formally defines a provably correct refinement from Spi Calculus specifications into Java code implementations, thus enabling automatic generation of the implementations, while preserving the security properties verified on the specifications. This refinement relation is obtained by defining a type system, that allows to assign static types to the untyped Spi Calculus terms, and then to use the same types for Java data representing the Spi Calculus terms. Moreover, a translation function from well-formed sequential Spi Calculus processes to Java code has been formally defined, so that it is possible to reason on the relation between the Spi Calculus source processes, and the generated Java code. The first result is that the translation of a well-formed Spi Calculus process always lead to the generation of well-formed Java code, that is code that compiles. Some features, like disposable resources, protocol return parameters, and interoperability of the generated application, that is needed to let the program adhere to existing standards, are also taken into account by the translation function.

As a step further, we prove that the generated Java code is a correct refinement of the Spi Calculus specification, if correctness of an underlying custom Java library, called SpiWrapper, is assumed. In order to achieve this result, the formal definition of the intended behavior of the SpiWrapper library has been formalized. Then, it has been shown that the intended behavior of this library can be related with the formal semantics of the Spi Calculus, so that the generated Java code, by properly using the SpiWrapper library, can simulate the Spi Calculus specification.

The translation function that has been formalized in this paper has been implemented in the spi2java tool, which has been used to successfully implement interoperable implementations of real cryptographic protocols. For example, in [21], a description of using spi2java for the implementation of the SSH transport protocol can be found.

There are still open challenges that would improve this work. One interesting result would be the development of a provably correct implementation of the SpiWrapper library, that is an implementation that behaves as defined in this paper. As a preliminary step towards this result, the correctness of part of a simple class belonging to the SpiWrapper library is shown, thought the most interesting and challenging cases are left for future work. Another possibility is to link this work to existing proposals [4] of cryptographic libraries that offer provably correct implementations of abstract cryptographic primitives like the ones used in the Spi Calculus.

References

[1] M. Abadi and B. Blanchet. Computer-Assisted Verification of a Protocol for Certified Email. *Science of Computer Programming*, 58(1–2):3–27, 2005.

[2] M. Abadi, B. Blanchet, and C. Fournet. Just fast keying in the pi calculus. *ACM Transactions on Information and System Security*, 10(3):1–59, 2007.

[3] M. Abadi and A. D. Gordon. A calculus for cryptographic protocols the spi calculus. Research Report 149, 1998.

[4] M. Backes, B. Pfitzmann, and M. Waidner. A composable cryptographic library with nested operations. In *ACM Conference on Computer and Communications Security*, pages 220–230, 2003.

[5] S. Bajaj et al. Web services policy 1.2 - attachment (WS-policyattachment). W3C Recommendation, 2006.

[6] S. Bajaj et al. Web services policy 1.2 - framework (WS-policy). W3C Recommendation, 2006.

[7] K. Bhargavan, C. Fournet, and A. D. Gordon. Verified reference implementations of WS-security protocols. In *Web Services and Formal Methods*, pages 88–106, 2006.

[8] K. Bhargavan, C. Fournet, A. D. Gordon, and G. O'Shea. An advisor for web services security policies. In *Workshop on Secure web services*, pages 1–9, 2005.

[9] K. Bhargavan, C. Fournet, A. D. Gordon, and S. Tse. Verified interoperable implementations of security protocols. In *Computer Security Foundations Workshop*, pages 139–152, 2006.

[10] B. Blanchet. An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In *IEEE Computer Security Foundations Workshop*, pages 82–96, 2001.

[11] D. Dolev and A. C.-C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–207, 1983.

[12] L. Durante, R. Sisto, and A. Valenzano. Automatic testing equivalence verification of spi calculus specifications. *ACM Transactions on Software Engineering and Methodology*, 12(2):222–284, 2003.

[13] J. Goubault-Larrecq and F. Parrennes. Cryptographic protocol analysis on real C code. In *Verification, Model Checking, and Abstract Interpretation*, pages 363–379, 2005.

[14] E. Hubbers, M. Oostdijk, and E. Poll. Implementing a formally verifiable security protocol in java card. In *Security in Pervasive Computing*, pages 213–226, 2003.

[15] A. Igarashi and N. Kobayashi. A generic type system for the pi-calculus. *Electronic Notes in Theoretical Computer Science*, 311(1-3):121–163, 2004.

[16] C.-W. Jeon, I.-G. Kim, and J.-Y. Choi. Automatic generation of the C# code for security protocols verified with casper/FDR. In *International Conference on Advanced Information Networking and Applications*, pages 507–510, 2005.

[17] J. Jürjens. Verification of low-level crypto-protocol implementations using automated theorem proving. In *Formal Methods and Models for Co-Design*, pages 89–98, 2005.

[18] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, parts I and II. *Information and Computation*, 100(1):1–77, 1992.

[19] A. Nadalin, C. Kaler, P. Hallam-Baker, and R. Monzillo. OASIS web services security: SOAP message security 1.1 (WS-security 2004), 2006.

[20] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[21] A. Pironti and R. Sisto. An experiment in interoperable cryptographic protocol implementation using automatic code generation. In *IEEE Symposium on Computers and Communications*, pages 839–844, 2007.

[22] A. Pironti and R. Sisto. Soundness conditions for message encoding abstractions in formal security protocol models. In *International Conference on Availability, Reliability and Security*, pages 72–79, 2008.

[23] D. Pozza, R. Sisto, and L. Durante. Spi2java: Automatic cryptographic protocol java code generation from spi calculus. In *International Conference on Advanced Information Networking and Applications*, pages 400–405, 2004.

[24] G. Schellhorn. ASM refinement and generalizations of forward simulation in data refinement: a comparison. *Theoretical Computer Science*, 336(2-3):403–435, 2005.

[25] B. Tobler and A. Hutchison. Generating network security protocol implementations from formal specifications. In *Certification and Security in Inter-Organizational E-Services*, Toulouse, France, 2004.