Correctness-Preserving Translation from Spi Calculus to Java, Revision 2

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 protocols specification and verification. In this paper, a type system for the Spi Calculus and a translation function are formally defined, in order to formalize 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. A verified implementation of part of the custom library is further presented.

Keywords: *Model-based software development, Correctness preserving code generation, Code verification, 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. [13]) or a theorem prover (e.g. [11]), 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 [12]) attacker.

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 in 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, 18, 14, 7] and code generation [24, 22, 27]. 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 and that this relation preserves security properties.

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 [24, 22], 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. Moreover, a verified implementation of part of the library is presented in this paper.

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 the Spi Calculus specification it has been generated from, and shows a possible verified implementation of part of the custom library used by the generated code. 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 [24, 22] and [27]. However, none of these works provides rigorous formal proofs for the generated code.

The AGC-C# tool [17] 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 [15], 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, and, like in the previous case, 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 [20] protocol, written in F#. This implementation can be a starting point in future complete protocol implementations. 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 [19], by adding a fixed set of cryptographic primitives to the language, namely symmetric and asymmetric encryptions, and hash functions, thus enabling the description of 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,

$L, M, N ::=$	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
$P\|Q$	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

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 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 Dolev-Yao 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

$$\begin{array}{rcl} \overline{M}\langle N \rangle .P & \xrightarrow{M!N} & P \\ M(x).P & \xrightarrow{M?N} & P[^N/_x] \\ [M\ is\ M]\,P & \xrightarrow{\tau} & P \\ let\ (x,y) = (M,N)\ in\ P & \xrightarrow{\tau} & P[^M/_x][^N/_y] \\ case\ 0\ of\ 0 : P\ suc(x) : Q & \xrightarrow{\tau} & P \\ case\ suc(M)\ of\ 0 : P\ suc(x) : Q & \xrightarrow{\tau} & Q[^M/_x] \\ case\ \{M\}_N\ of\ \{x\}_N\ in\ P & \xrightarrow{\tau} & P[^M/_x] \\ case\ \{[M]\}_{N^+}\ of\ \{[x]\}_{N^-}\ in\ P & \xrightarrow{\tau} & P[^M/_x] \\ case\ [\{M\}]_{N^-}\ of\ [\{x\}]_{N^+}\ in\ P & \xrightarrow{\tau} & P[^M/_x] \end{array}$$

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

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 [16], 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 [16] 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 [21]. 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 $Name$

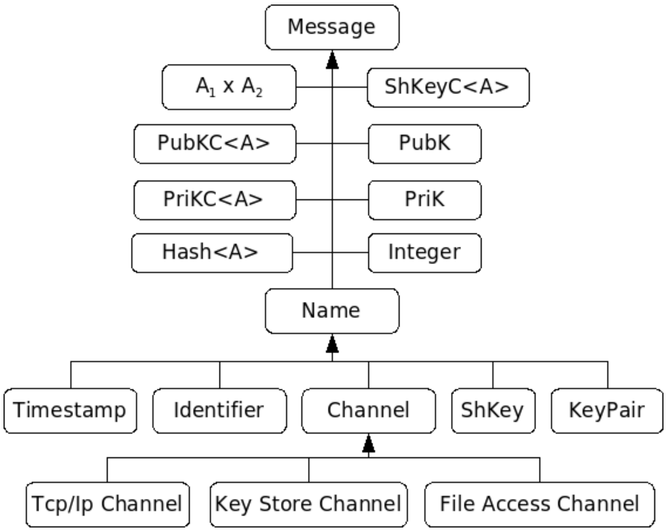

Figure 1. Type hierarchy.

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 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. As it will be clear later on, 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. Note that this constraint does not alter the Dolev-Yao attacker model. Indeed, during formal verification of a (well formed) process, the attacker is still modeled as the (possibly non well formed) environment.

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 the process that matches M and N, and then behaves like P, is well formed. 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, though probably too rigid. 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 fresh 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,

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

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

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

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

$$\frac{\Gamma \vdash M : Integer \qquad \Gamma \vdash P \qquad \Gamma, x : A \vdash Q\ \ A <: Integer}{\Gamma \vdash case\ M\ of\ 0 : P\ suc(x) : Q}\ \text{(P-IntCase)} \qquad \frac{}{\Gamma \vdash 0 : Integer}\ \text{(T-Zero)} \qquad \frac{\Gamma \vdash M : Integer}{\Gamma \vdash suc(M) : Integer}\ \text{(T-Succ)}$$

$$\frac{\Gamma \vdash M : KeyPair}{\Gamma \vdash M^+ : PubK}\ \text{(T-PubK)} \qquad \frac{\Gamma \vdash M : KeyPair}{\Gamma \vdash M^- : PriK}\ \text{(T-PriK)} \qquad \frac{\Gamma \vdash M : A}{\Gamma \vdash H(M) : Hash\,\langle A \rangle}\ \text{(T-Hash)}$$

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash K : PubK}{\Gamma \vdash \{[M]\}_K : PubKC\,\langle A \rangle}\ \text{(T-PubKC)} \qquad \frac{\Gamma \vdash M : A \qquad \Gamma \vdash K : PriK}{\Gamma \vdash [\{M\}]_K : PriKC\,\langle A \rangle}\ \text{(T-PriKC)}$$

$$\frac{\Gamma \vdash L : PriKC\,\langle A \rangle \qquad \Gamma \vdash N : PubK \qquad \Gamma, x : A \vdash P}{\Gamma \vdash case\ L\ of\ [\{x\}]_N\ in\ P}\ \text{(P-PriKD)} \qquad \frac{\Gamma \vdash L : PubKC\,\langle A \rangle \qquad \Gamma \vdash N : PriK \qquad \Gamma, x : A \vdash P}{\Gamma \vdash case\ L\ of\ \{[x]\}_N\ in\ P}\ \text{(P-PubKD)}$$

Figure 2. Typing rules.

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.

Note that this rather standard type system shares some common properties with other well known type systems [21]. 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 dealing with 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

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

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

$tr_t : SpiTerm, 2^{SpiTerm}, 2^{SpiTerm} \rightarrow Java$ 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 implements the operations that can be performed on the corresponding 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 enables the user to define her 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 [23], if some static checks on the user-written code are performed, such possible flaws are avoided.

The tr_t function is formally defined in figure 3. Some interesting cases are explained in details. 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 [22].

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.

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, and the (possibly empty) code needed to return the pair. Note that it is only needed to explicitly call ret on the pair, and not on its components, because the recursive invocations of tr_t on the components already ensure they are added, if needed, to the `_return` map as soon as they are built.

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.

$tr_t(M, built, return) =$ "", 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` $Ts(\{M\}_N)$`(`$J(M)$`,` $J(N)$`,` $Param(Ts(\{M\}_N))$`));` $ret(\{M\}_N, return)$"
$tr_t(0, built, return) =$ "$T(0)\ J(0) =$ `new` $Ts(0)$`(0,` $Param(Ts(0))$`);` $ret(0, return)$"
$tr_t(suc(M), built, return) =$ "$tr_t(M, built, return)$
$T(suc(M))\ J(suc(M)) =$ `new` $Ts(suc(M))$`(`$J(M)$`,` $Param(Ts(suc(M)))$`));` $ret(suc(M), return)$"
$tr_t(H(M), built, return) =$ "$tr_t(M, built, return)$
$T(H(M))\ J(H(M)) =$ `new` $Ts(H(M))$`(`$J(M)$`,` $Param(Ts(H(M)))$`));` $ret(H(M), return)$"
$tr_t(K^+, built, return) =$ "$tr_t(K, built, return)$
$T(K^+)\ J(K^+) = J(K)$`.getPublic(`$Ts(K^+)$`.class,` $Param(Ts(K^+))$`);` $ret(K^+, return)$"
$tr_t(K^-, built, return) =$ "$tr_t(K, built, return)$
$T(K^-)\ J(K^-) = J(K)$`.getPrivate(`$Ts(K^-)$`.class,` $Param(Ts(K^-))$`);` $ret(K^-, return)$"
$tr_t(\{[M]\}_N, built, return) =$ "$tr_t(M, built, return)\ tr_t(N, ub(M, built), return)$
$T(\{[M]\}_N)\ J(\{[M]\}_N) =$ `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.

Finally, in the public key K^+ case, the `T getPublic(Class<T>,...)` method is invoked, that encapsulates the public part of the keypair K, in the given class. Same reasoning applies to the private key K^- case.

Note that no case is available for variables. Indeed, variables cannot be "built", rather, they are declared and assigned 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. Some interesting cases are explained in details. 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 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; when n is built, a constructor is called, which generates the Java implementation of a new fresh name of the expected type. A note must be made on creation of restricted channels: since data transmitted on restricted channels shall not be available to the attacker, it is only allowed to create restricted channels that communicate over a secure medium (such as IPSec or TLS) or over the local filesystem (such as the file access channel); it is not allowed instead to create restricted channels that communicate over insecure mediums (such as TCP/IP). This is enforced by tagging channel types that use an insecure medium, so that translation is aborted if a restricted channel of a tagged type is going to be built. 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

$tr_p(\mathbf{0}, built, return) =$ “”
$tr_p(\overline{M}\,\langle N\rangle\,.P, built, return) =$ “$tr_t(M, built, return)$ $tr_t(N, ub(M, built), return)$
$J(M)$`.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(`$J(N)$`,`$Param(Ts(x))$`);` $ret(x, return)$
$tr_p(P, ub(L, built) \cup ub(N, built) \cup ub(x, built), return)$
$tr_p(case\ M\ of\ 0 : P\ suc(x) : Q, built, return) =$ “$tr_t(M, built, return)$
`if (`$J(M)$`.isSpiZero()) {`
$tr_p(P, ub(M, built), return)$
`} else {`
$T(x)$ $J(x)$ `=` $J(M)$`.getPrevious(`$Ts(x)$`.class,`$Param(Ts(x))$`);` $ret(x, return)$
$tr_p(Q, ub(M, built) \cup ub(x, built), return)$
`}`”
$tr_p(case\ L\ of\ \{[x]\}_N\ in\ P, built, return) =$ “$tr_t(L, built, return)$ $tr_t(N, ub(L, built), return)$
$T(x)$ $J(x)$ `=` $J(L)$`.decrypt(`$J(N)$`,`$Param(Ts(x))$`);` $ret(x, return)$
$tr_p(P, ub(L, built) \cup ub(N, built) \cup ub(x, built), return)$”
$tr_p(case\ L\ of\ [\{x\}]_N\ in\ P, built, return) =$ “$tr_t(L, built, return)$ $tr_t(N, ub(L, built), return)$
$T(x)$ $J(x)$ `=` $J(L)$`.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.

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. In the integer case, three cases are possible: if M is 0 then P is executed, if M is $suc(N)$, then $Q[^N/_x]$ is executed, else the process is stuck. However, note that the type system enforces M to be typed as $Integer$, so, by the canonical form lemma, M must be either 0 or $suc(N)$, where N is typed as $Integer$ too. For this reason, the third case, where the process is stuck, cannot happen at runtime, and only the first two cases are handled by the `if-else` Java statement.

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;
}
```

where `@InputParams@` gets substituted by the free variables of the translated Spi Calculus process, and `@GeneratedSpiImpl@` by the generated Java code.

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. In order to enable the formal proofs of these properties, some reasonable assumptions are explicited below.

- For any term M, it is assumed that 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.
- 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.
- It is assumed that $Param(Ts(M))$ returns the correct number and type of user-provided interoperability parameters.

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. Under the assumptions made, the following theorem can be proven.

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.

Formally, the context is denoted by $\mathcal{C}[\cdot]$, where the central dot '$\cdot$' represents the hole in the context; by stating that a context $\mathcal{C}[\cdot]$ compiles, we mean that the Java program $\mathcal{C}[``"]$, obtained by filling the hole in the context with an empty string, compiles.

In order to prove theorem 1, a lemma is first introduced.

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, $\mathcal{C}[``"\ \cdot] = \mathcal{C}[\cdot]$ 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 (it is an instance of the $M \in built$ case). Some interesting 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 $ub(N, built)$. Similarly, since $ub(N, built) \supseteq atoms(N')$, the inductive hypotheses apply to the code that follows 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 us to satisfy the inductive hypotheses and to avoid duplicate variable declarations 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 (where `A_1` and `A_2` are the ASCII representations of the A_1 and A_2 types respectively). 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 and 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.

$\square$

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 that includes the declaration and initialization of all and only Java variables that correspond to the terms in $built$, the $\mathcal{C}[tr_p(P, built, return)]$ Java program compiles too. 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 **0**, is trivial, because the empty string is well formed by hypoteses.

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 the 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

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

is a Java program that compiles, and the whole generated code is well formed in $\mathcal{C}[\cdot]$.

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

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

is a Java program that compiles, and the whole generated code is well formed in $\mathcal{C}[\cdot]$.

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 the empty string, 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,

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

is a Java program that compiles, and the whole generated code is well formed in $\mathcal{C}[\cdot]$.

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 in $\mathcal{C}[\cdot]$. □

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 [26].

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

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

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 defined 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 j be the Java code that is going to be executed, $JavaVar$ the set of identifiers that can be used as variables in Java programs, and $JavaObj$ the set of object identifiers. Then a generic state (j, Val, Res) is defined by the code j that is going to be executed, plus a partial function $Val : JavaObj \to SpiTerm$, mapping each Java object that has been created by previously executed code to the Spi Calculus term the object is implementing, and a partial function $Res : JavaVar \to JavaObj$ mapping each Java variable in the scope of j to the referenced Java object. For example, $Val(o) = \{M\}_N$ means that the Java object o implements the $\{M\}_N$ Spi Calculus term; $Res(\texttt{var}) = o$ means that the Java variable `var` references the object o. The intended invariant that should hold is $Val(Res(J(M))) = M\sigma$, where σ is the variable substitution in the corresponding Spi Calculus process. That is, the object referenced by the Java variable $J(M)$ must implement the $M\sigma$ term, which is the run-time value of the M Spi Calculus term. A Java state (j, Val, Res) is defined as external iff $j = tr_p(P, dom(Val \circ Res \circ J), return)$ for some Spi Calculus process P that does not begin with a restriction and for some return set $return$. Note that, since $Val \circ Res \circ J$ is a composition of partial functions, the domain of J is properly restricted such that its codomain is the domain of Res; in turn this is restricted so that its codomain is the domain of Val. The transitions of the form

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

take from one generic state to another, following an abstract operational semantics for the Java language. In this work, we formally define an operational semantics that, if implemented by the SpiWrapper classes, makes it possible to have a weak simulation relation between the Spi Calculus process and the generated Java code. The formal semantics for 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; variable assignment evolves into the $unit$ value, but its side effect is to map the variable to the assigned object, formally

$$T(x)\ J(x) = o, Val, Res \xrightarrow{\tau} unit, Val, \{(J(x), o)\} \cup Res$$

Generic types are not taken into account into the definition of the dynamic semantics, because in Java they are implemented by erasure, that is they are checked at compile time, and then discarded in the compiled bytecode. For this reason the dynamic semantics can be defined without them. 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. Formally, when an exception is thrown from any state (j, Val, Res), the following semantics is assumed

$$j, Val, Res \xrightarrow{\tau} \bot$$

where $\bot$ is a special non-external state that represents all the states where the control has passed to the context, due to a 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

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

for any Spi Calculus process P that does not begin with a restriction, and any Val, Res such that $dom(Val \circ Res \circ J)$ is closed under the $subterms$ function. Informally, a Spi Calculus state $((\nu\overline{n})P)\sigma$ and a Java state (j, Val, Res) are S-related, iff the Java state is external, and the invariant $M\sigma = Val(Res(J(M)))$ holds. Note that it is required that the domain of $Val \circ Res \circ J$ contains all the free variables in P; however some compound terms may not (yet) be stored in Java memory (because they will be built by the code generated by the $tr_t(\cdot)$ function): it is enough to require that the invariant holds for the already built terms, which are stored in Java memory.

$\texttt{new TMarsh(params)}, Val, Res$	$\xrightarrow{\tau}^{*}$	$\mathcal{o}, \{(\mathcal{o}, n)\} \cup Val, Res \ \wedge\ n \notin codom(Val)$
$\mathcal{c}\texttt{.send(}\mathcal{M}\texttt{)}, \{(\mathcal{c}, c), (\mathcal{M}, M)\} \cup Val, Res$	$\xrightarrow{\tau}^{*}\xrightarrow{c!M}\xrightarrow{\tau}^{*}$	$unit, \{(\mathcal{c}, c), (\mathcal{M}, M)\} \cup Val, Res$
$\mathcal{c}\texttt{.receive(Ts.class,params)}, \{(\mathcal{c}, c)\} \cup Val, Res$	$\xrightarrow{\tau}^{*}\xrightarrow{c?N}\xrightarrow{\tau}^{*}$	$\mathcal{N}, \{(\mathcal{c}, c), (\mathcal{N}, N)\} \cup Val, Res$
$M = N \Rightarrow \mathcal{a}\texttt{.equals(}\mathcal{b}\texttt{)}, \{(\mathcal{a}, M), (\mathcal{b}, N)\} \cup Val, Res$	$\xrightarrow{\tau}^{*}$	$\texttt{true}, \{(\mathcal{a}, M), (\mathcal{b}, N)\} \cup Val, Res$
$M \neq N \Rightarrow \mathcal{a}\texttt{.equals(}\mathcal{b}\texttt{)}, \{(\mathcal{a}, M), (\mathcal{b}, N)\} \cup Val, Res$	$\xrightarrow{\tau}^{*}$	$\texttt{false}, \{(\mathcal{a}, M), (\mathcal{b}, N)\} \cup Val, Res$
$\texttt{new PairMarsh(}\mathcal{A}\texttt{,}\mathcal{B}\texttt{,params)},$ $\{(\mathcal{A}, A), (\mathcal{B}, B)\} \cup Val, Res$	$\xrightarrow{\tau}^{*}$	$\mathcal{o}, \{(\mathcal{A}, A), (\mathcal{B}, B), (\mathcal{o}, (A, B))\} \cup Val, Res$
$\mathcal{o}\texttt{.getLeft()}, \{(\mathcal{o}, (M, N)), (\mathcal{M}, M)\} \cup Val, Res$	$\xrightarrow{\tau}^{*}$	$\mathcal{M}, \{(\mathcal{o}, (M, N)), (\mathcal{M}, M)\} \cup Val, Res$
$\mathcal{o}\texttt{.getRight()}, \{(\mathcal{o}, (M, N)), (\mathcal{N}, N)\} \cup Val, Res$	$\xrightarrow{\tau}^{*}$	$\mathcal{N}, \{(\mathcal{o}, (M, N)), (\mathcal{N}, N)\} \cup Val, Res$
$\texttt{new ShKCMarsh(}\mathcal{M}\texttt{,}\mathcal{K}\texttt{,params)},$ $\{(\mathcal{M}, M), (\mathcal{K}, K)\} \cup Val, Res$	$\xrightarrow{\tau}^{*}$	$\mathcal{o}, \{(\mathcal{M}, M), (\mathcal{K}, K), (\mathcal{o}, \{M\}_K)\} \cup Val, Res$
$\mathcal{o}\texttt{.decrypt(}\mathcal{K}\texttt{,params)},$ $\{(\mathcal{K}, K), (\mathcal{o}, \{M\}_K)\} \cup Val, Res$	$\xrightarrow{\tau}^{*}$	$\mathcal{M}, \{(\mathcal{M}, M), (\mathcal{K}, K), (\mathcal{o}, \{M\}_K)\} \cup Val, Res$
$\texttt{new IntMarsh(0,params)}, Val, Res$	$\xrightarrow{\tau}^{*}$	$\mathcal{Z}, \{(\mathcal{Z}, 0)\} \cup Val, Res$
$\texttt{new IntMarsh(}\mathcal{M}\texttt{,params)}, \{(\mathcal{M}, M)\} \cup Val, Res$	$\xrightarrow{\tau}^{*}$	$\mathcal{N}, \{(\mathcal{M}, M), (\mathcal{N}, suc(M))\} \cup Val, Res$
$\texttt{new HashMarsh(}\mathcal{M}\texttt{,params)}, \{(\mathcal{M}, M)\} \cup Val, Res$	$\xrightarrow{\tau}^{*}$	$\mathcal{N}, \{(\mathcal{M}, M), (\mathcal{N}, H(M))\} \cup Val, Res$
$\mathcal{K}\texttt{.getPublic(Ts.class,params)},$ $\{(\mathcal{K}, K)\} \cup Val, Res$	$\xrightarrow{\tau}^{*}$	$\mathcal{L}, \{(\mathcal{K}, K), (\mathcal{L}, K^+)\} \cup Val, Res$
$\mathcal{K}\texttt{.getPrivate(Ts.class,params)},$ $\{(\mathcal{K}, K)\} \cup Val, Res$	$\xrightarrow{\tau}^{*}$	$\mathcal{L}, \{(\mathcal{K}, K), (\mathcal{L}, K^-)\} \cup Val, Res$
$\texttt{new PubKCMarsh<T>(}\mathcal{M}\texttt{,}\mathcal{K}\texttt{,params)},$ $\{(\mathcal{M}, M), (\mathcal{K}, K)\} \cup Val, Res$	$\xrightarrow{\tau}^{*}$	$\mathcal{N}, \{(\mathcal{M}, M), (\mathcal{K}, K), (\mathcal{N}, \{[M]\}_K)\} \cup Val, Res$
$\texttt{new PriKCMarsh<T>(}\mathcal{M}\texttt{,}\mathcal{K}\texttt{,params)},$ $\{(\mathcal{M}, M), (\mathcal{K}, K)\} \cup Val, Res$	$\xrightarrow{\tau}^{*}$	$\mathcal{N}, \{(\mathcal{M}, M), (\mathcal{K}, K), (\mathcal{N}, [\{M\}]_K)\} \cup Val, Res$
$\mathcal{Z}\texttt{.isSpiZero()}, \{(\mathcal{Z}, 0)\} \cup Val, Res$	$\xrightarrow{\tau}^{*}$	$\texttt{true}, \{(\mathcal{Z}, 0)\} \cup Val, Res$
$\mathcal{N}\texttt{.isSpiZero()}, \{(\mathcal{N}, suc(M))\} \cup Val, Res$	$\xrightarrow{\tau}^{*}$	$\texttt{false}, \{(\mathcal{N}, suc(M))\} \cup Val, Res$
$\mathcal{N}\texttt{.getPrevious(Ts.class,params)},$ $\{(\mathcal{N}, suc(M))\} \cup Val, Res$	$\xrightarrow{\tau}^{*}$	$\mathcal{M}, \{(\mathcal{N}, suc(M)), (\mathcal{M}, M)\} \cup Val, Res$
$\mathcal{N}\texttt{.decrypt(}\mathcal{K}\texttt{,params)},$ $\{(\mathcal{N}, \{[M]\}_{K^+}), (\mathcal{K}, K^-)\} \cup Val, Res$	$\xrightarrow{\tau}^{*}$	$\mathcal{M}, \{(\mathcal{N}, \{[M]\}_{K^+}), (\mathcal{K}, K^-), (\mathcal{M}, M)\} \cup Val, Res$
$\mathcal{N}\texttt{.decrypt(}\mathcal{K}\texttt{,params)},$ $\{(\mathcal{N}, [\{M\}]_{K^-}), (\mathcal{K}, K^+)\} \cup Val, Res$	$\xrightarrow{\tau}^{*}$	$\mathcal{M}, \{(\mathcal{N}, [\{M\}]_{K^-}), (\mathcal{K}, K^+), (\mathcal{M}, M)\} \cup Val, Res$

Table 3. Formal semantics of the SpiWrapper library.

Theorem 2. *If the SpiWrapper library behaves as specified in table 3, then, for any external state* (j', Val', Res')

$$\begin{gathered} S((\nu\overline{n})P\sigma, (j, Val, Res)) \wedge \\ j, Val, Res \xrightarrow{\tau}^{*}\xrightarrow{\mathcal{L}}\xrightarrow{\tau}^{*} j', Val', Res' \Rightarrow \\ P\sigma \xrightarrow{\mathcal{L}} (\nu\overline{m})P'\sigma' \ \wedge\ S((\nu\overline{n})(\nu\overline{m})P'\sigma', (j', Val', Res')) \end{gathered}$$

Theorem 2 formally expresses the simulation relation between a Spi Calculus process and its corresponding generated Java program. If the simulation relation holds between a state of a Spi Calculus process and a state of its corresponding Java program, and if the Java program can evolve into a new external state, then the Spi Calculus process can evolve into a new external state too, and the new external states are still related by the simulation relation S.

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, Res *such that* $dom(Val \circ Res \circ J)$ *is closed under the* *subterms function, and for any substitution σ, there exists Val', Res' such that*

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

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

Informally, this lemma asseses that, after execution, the Java code generated to build M, actually creates a Java object representing M and a variable $J(M)$ storing the reference to that object. More precisely, this lemma states that if the Java program memory correctly represents the terms in the corresponding Spi Calculus process, and the generated Java code building M can correctly execute, then in the final state, the Java memory is representing M, and all of its subterms.

Proof. If $M \in dom(Val \circ Res \circ J)$, by hypotheses, $M\sigma = Val(Res(J(M)))$. Moreover, tr_t returns the empty string, letting the final state equal to the initial state, so this case is proven.

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

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

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

So the inductive hypotheses hold, and

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

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

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

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

In particular, $Val''(Res''(J(N))) = N\sigma$ and $Val''(Res''(J(N'))) = N'\sigma$ so the appended code executes as

$$\begin{gathered}T((N, N'))\ J((N, N'))\ \texttt{ = new }\ Ts((N, N'))\texttt{(}J(N)\texttt{, }J(N')\texttt{, }Param(Ts((N, N')))\texttt{)}, Val'', Res'' \xrightarrow{\tau}{}^{*} \\ T((N, N'))\ J((N, N'))\ \texttt{ = }\ o, Val''', Res'' \xrightarrow{\tau} \\ unit, Val''', Res'''\end{gathered}$$

where $Val''' = \{(o, (N, N')\sigma)\} \cup Val''$ and $Res''' = \{(J((N, N')), o)\} \cup Res''$

Finally, by noting that $dom(Val''' \circ Res''' \circ J)$ 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 does not begin with a replication, Val, Res such that $dom(Val \circ Res \circ J)$ is closed under the subterms function*

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

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 $m\sigma = m$, 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

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

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

$$T(m)\ J(m)\ \texttt{=}\ \texttt{new}\ Ts(m)\texttt{(}Param(Ts(m))\texttt{)}\,\texttt{,}\,Val, Res \xrightarrow{\tau}{}^{*}$$
$$T(m)\ J(m)\ \texttt{=}\ o, Val', Res \xrightarrow{\tau}$$
$$unit, Val', Res'$$

where $Val' = \{(o, m)\} \cup Val$ and $Res' = \{(J(m), o)\} \cup Res$. 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.

The inductive hypotheses hold, because $dom(Val' \circ Res' \circ J) = ub(m, dom(Val \circ Res \circ J))$, so

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

can execute, and $S(((\nu\overline{m'})P)\sigma, (tr_p(P, dom(Val'' \circ Res'' \circ J), return), Val'', Ret''))$ holds. Since $m\sigma = m = Val''(Res''(J(m)))$, it follows that $S(((\nu m)(\nu\overline{m'})P)\sigma, (tr_p(P, dom(Val'' \circ Res'' \circ J), return), Val'', Res''))$ 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 $\mathbf{0}$ 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

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

It follows that lemma 2 can be applied

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

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

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

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

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

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

(The steps where the Java variables $J(M)$ and $J(N)$ are substituted by their referenced values, $Res''(J(M))$ and $Res''(J(N))$ respectively, are included within the $\stackrel{\tau}{\rightarrow}^*$ transitions.)

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

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

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

On the Spi Calculus side, the process can evolve like

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

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, getting to the final result that the Q' process is proven to be S-related with its Java translation, which proves 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 $\sigma_{|atoms(M)} = Val \circ Res \circ J_{|atoms(M)}$, and lemma 2 can be applied, letting

$$tr_t(M, dom(Val \circ Res \circ J), return), Val, Res \stackrel{\tau}{\rightarrow}^* unit, Val', Res'$$

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

By the assumption on the behavior of the SpiWrapper library,

$$T(x)\ J(x) = J(M)\texttt{.receive(}Ts(x)\texttt{.class,}Param(Ts(x))\texttt{)}, Val', Res' \stackrel{\tau}{\rightarrow}^* \stackrel{M\sigma?N}{\rightarrow} \stackrel{\tau}{\rightarrow}^*$$
$$T(x)\ J(x) = \mathcal{N}, Val'', Res' \stackrel{\tau}{\rightarrow} unit, Val'', Res''$$

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

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

The Java state is (j'', Val'', Res'').

On the Spi Calculus side, the process can evolve like

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

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 its Java translation, thus proving the case.

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

$$tr_t(M, dom(Val \circ Res \circ J), return)\ tr_t(N, ub(M, dom(Val \circ Res \circ J)), return), Val, Res \stackrel{\tau}{\rightarrow}^* unit, Val', Res'$$

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

Two cases must be considered, either $M\sigma \neq N\sigma$ or $M\sigma = N\sigma$. If $M\sigma \neq N\sigma$, 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

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

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 $M\sigma = N\sigma$, 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

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

Let

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

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

On the Spi Calculus side,

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

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 its Java translation, thus proving the case.

□

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

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

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.

4.1 Verification of a SpiWrapper Implementation

As stated by theorem 2, the generated Java code simulates the Spi Calculus process from which it has been generated, by relying on the custom SpiWrapper library, whose behavior is formally specified. As a step further, an implementation of part of the SpiWrapper library is now presented, that is shown to be correct with respect to its specification reported in table 3. In order to reason on executions of the Java code implementing the library, the Middleweight Java (MJ) framework [10, 25] is used. Essentially, MJ specifies a small-step operational semantics for a rich subset of sequential Java. States are called *configurations*; a configuration is a four-tuple made of (H, VS, CF, FS), where

H is the heap, mapping object identifiers (oids) to their type and to a field function. A field function is a map from field names to values.

VS is the variable stack, mapping variable names to their type and to the referred oid or to their value.

CF is a closed frame of Java code to be evaluated.

FS is a frame stack, that is the program context in which CF is being evaluated.

In the original MJ, transitions are not labeled, because all side effects are captured by the subsequent configuration. In this work, transitions that produce input or output of message M on channel c, are labelled with $c!M$ and $c?M$, while all other transitions are labelled with τ.

A relation between Java states, defined as (j, Val, Res), and MJ configurations is defined, so that the two frameworks can be related. Indeed, states and MJ configurations are very similar, with MJ configurations storing more information, which is unnedeed for SpiWrapper behavior specification. In fact, j corresponds to the MJ CF, that is the code to be evaluated; Res serves the same purpose as VS, although VS also store some other information about variable scopes and types. Since we are interested in single method behaviors, rather than in full program behaviors, the FS context is set empty (denoted by $[]$) before the execution of the method, and it will be empty after method executed. A note must be made on the relation between Val and MJ H, which completes the two frameworks relation. On one hand, Val maps oids to the implemented Spi Calculus terms; on the other hand, H maps oids to their internal state, which is the value of their fields. So, $Val(o) = M$ (read "object o implements Spi Calculus term M") *means* that $H(o) = v$ (read "object o has fields set as described by v") for some v. For example

$$\begin{array}{rcl} Val(v_p) = (M, N) & \Leftrightarrow & H(v_p) = \left(PairS, \begin{array}{l} \texttt{left} \rightarrow v_{\mathcal{M}} \\ \texttt{right} \rightarrow v_{\mathcal{N}} \end{array} \right) \wedge \\ & & PairS <: Pair \wedge \\ & & Val(v_{\mathcal{M}}) = M \wedge Val(v_{\mathcal{N}}) = N \end{array} \tag{1}$$

states that "object v_p implements the pair (M, N)" means that object v_p has a subtype of the $Pair$ type, and it has exactly two fields, namely `left` and `right`, pointing to two objects $v_{\mathcal{M}}$ and $v_{\mathcal{N}}$, implementing the M and N terms respectively. Note that the relation between Val and H is implementation dependent.

MJ does not support generic types, and there is no need to add such support, because, as explained before, generic types are implemented by erasure, so they can be disregarded when analyzing run time behavior.

The `Pair` class is now verified. In the Spi2Java framework, the `Pair` class is abstract, and extended by another class implementing marshalling functions. For simplicity, since marshalling functions are irrelevant in this context, and adding them would not add value to this example, the `Pair` class is considered to be concrete, and marshalling functions are neglected.

Figure 5 shows a possible implementation of the `Pair` class that fits in the MJ framework.

Proof of correctness of the constructor. By looking up table 3, the initial state of the `Pair` constructor invocation is

$$\texttt{new Pair(}\mathcal{A}\texttt{,}\mathcal{B}\texttt{)}, \{(\mathcal{A}, A), (\mathcal{B}, B)\}, \emptyset$$

In the presented implementation, no additional marshalling parameters are required, so the '`,params`' argument can be neglected.

This state corresponds to the initial MJ configuration

$$(\underbrace{\{(\mathcal{A}, (T_A, \mathbb{F}_A)), (\mathcal{B}, (T_B, \mathbb{F}_B))\}}_{H_0}, \underbrace{[]}_{VS_0}, \underbrace{\texttt{new Pair(}\mathcal{A}\texttt{,}\mathcal{B}\texttt{);}}_{CF_0}, \underbrace{[]}_{FS_0})$$

```
package it.polito.spi2java.spiWrapper;

public class Pair extends Message {
  protected Message left;
  protected Message right;

  public Pair(Message left,
              Message right) {
    super();
    this.left = left;
    this.right = right;
  }

  public Message getLeft() {
    return this.left;
  }

  public Message getRight() {
    return this.right;
  }

  public boolean equals(Object obj) {
    boolean result = false;
    if (obj instanceof Pair) {
      Pair otherPair = (Pair) obj;
      boolean leftOK = this.getLeft()
                 .equals(otherPair.getLeft());
      result = leftOK && this.getRight()
                 .equals(otherPair.getRight());
    }
    return result;
  }
}
```

Figure 5. A possible implementation of the `Pair` class.

for some types $T_A, T_B <: Message$ and mapping functions $\mathbb{F}_A, \mathbb{F}_B$ such that the $\mathcal{A}$ and $\mathcal{B}$ objects implement the A and B Spi Calculus terms respectively.

The following steps lead to the final configuration. Note that all transitions are labelled with τ (which is omitted for brevity), because no input or output operations occur. In order to make the evaluation steps more readable, each transition is marked with the labels specified in [10].

$$
\begin{aligned}
& (H_0, VS_0, CF_0, FS_0) \\
\xrightarrow{\text{E-New}}\ & (\underbrace{H_0 \cup \{(o, \left(Pair, \begin{array}{l}\texttt{left} \to \texttt{null}\\ \texttt{right} \to \texttt{null}\end{array}\right))\}}_{H_1}, \underbrace{\left(\left\{\begin{array}{l}\texttt{this} \to (o, Pair)\\ \texttt{left} \to (\mathcal{A}, T_A)\\ \texttt{right} \to (\mathcal{B}, T_B)\end{array}\right\} \circ []\right) \circ VS_0}_{VS_1}, \\
& \texttt{super(); ...}, (\texttt{return } o\texttt{;}) \circ FS_0) \\
\xrightarrow{\text{EC-Seq}}\ & (H_1, VS_1, \texttt{super();}, \underbrace{(\texttt{this.left = left; ...}\ ) \circ (\texttt{return } o\texttt{;}) \circ FS_0}_{FS_1}) \\
\xrightarrow{\text{EC-ExpState}}\ & (H_1, VS_1, \texttt{super()}, FS_1) \\
\xrightarrow{\text{E-Super}}\ & (H_1, \{\texttt{this} \to (o_s, Message)\} \circ [] \circ VS_1, \{\}, (\texttt{return } o_s\texttt{;}) \circ FS_1)
\end{aligned}
$$

For brevity, the constructor of $Message$ is considered empty, instead of recursively invoking the constructor of $Object$, which is actually empty. In practice, considering the constructor of $Object$ would lead to the same evaluation steps, modulo one stack level.

$$
\begin{aligned}
\xrightarrow{\text{E-BlockElim}}\ & (H_1, [] \circ VS_1, \texttt{;}, (\texttt{return } o_s\texttt{;}) \circ FS_1) \\
\xrightarrow{\text{E-Skip}}\ & (H_1, [] \circ VS_1, \texttt{return } o_s\texttt{;}, FS_1) \\
\xrightarrow{\text{EC-ExpState}}\ & (H_1, [] \circ VS_1, \texttt{return } o_s, FS_1) \\
\xrightarrow{\text{E-Return}}\ & (H_1, VS_1, o_s, FS_1) \\
\xrightarrow{\text{E-Sub}}\ & (H_1, VS_1, \texttt{this.left = left; ...}\ , (\texttt{return } o\texttt{;}) \circ FS_0) \\
\xrightarrow{\text{EC-Seq}}\ & (H_1, VS_1, \texttt{this.left = left;}, \underbrace{(\texttt{this.right = right;}) \circ (\texttt{return } o\texttt{;}) \circ FS_0}_{FS_2}) \\
\xrightarrow{\text{EC-FieldWrite1}}\ & (H_1, VS_1, \texttt{this}, (\cdot\texttt{.left = left;}) \circ FS_2) \\
\xrightarrow{\text{E-VarAccess}}\ & (H_1, VS_1, o, (\cdot\texttt{.left = left;}) \circ FS_2) \\
\xrightarrow{\text{E-Sub}}\ & (H_1, VS_1, o\texttt{.left = left;}, FS_2) \\
\xrightarrow{\text{EC-FieldWrite2}}\ & (H_1, VS_1, \texttt{left}, (o\texttt{.left = }\cdot) \circ FS_2) \\
\xrightarrow{\text{E-VarAccess}}\ & (H_1, VS_1, \mathcal{A}, (o\texttt{.left = }\cdot) \circ FS_2) \\
\xrightarrow{\text{E-Sub}}\ & (H_1, VS_1, o\texttt{.left = }\mathcal{A}, FS_2) \\
\xrightarrow{\text{E-FieldWrite}}\ & (H_0 \cup \{(o, \left(Pair, \begin{array}{l}\texttt{left} \to \mathcal{A}\\ \texttt{right} \to \texttt{null}\end{array}\right))\}, VS_1, \texttt{;}, FS_2)
\end{aligned}
$$

By performing the same steps, the `right` field is assigned the $\mathcal{B}$ value, leading to the state

$$
\begin{aligned}
& (\underbrace{H_0 \cup \{(o, \left(Pair, \begin{array}{l}\texttt{left} \to \mathcal{A}\\ \texttt{right} \to \mathcal{B}\end{array}\right))\}}_{H_2}, VS_1, \texttt{;}, (\texttt{return } o\texttt{;}) \circ FS_0) \\
\xrightarrow{\text{E-Skip}}\ & (H_2, VS_1, \texttt{return } o\texttt{;}, FS_0) \\
\xrightarrow{\text{E-Return}}\ & (H_2, VS_0, o, FS_0)
\end{aligned}
$$

Since in the final configuration the right side of (1) is true, this configuration corresponds to the final state

$$o, \{(\mathcal{A}, A), (\mathcal{B}, B), (o, (A, B))\}, \emptyset$$

□

Proof of correctness of the `getLeft()` *method.* By looking up table 3, the initial state for `getLeft()` method invocation is

$$o\texttt{.getLeft();}, \{(o, (M, N)), (\mathcal{M}, M), (\mathcal{N}, N)\}, \emptyset$$

From (1),

$$Val(o) = (M, N) \Rightarrow H(o) = \left(Pair, \begin{array}{l} \texttt{left} \ \rightarrow \mathcal{M} \\ \texttt{right} \rightarrow \mathcal{N} \end{array} \right)$$

So, the initial MJ configuration for the `getLeft()` method invocation is

$$(\underbrace{\{(o, \left(Pair, \begin{array}{l} \texttt{left} \ \rightarrow \mathcal{M} \\ \texttt{right} \rightarrow \mathcal{N} \end{array} \right)), (\mathcal{M}, (T_M, \mathbb{F}_M))\} \cup H_0}_{H_1}, \underbrace{[]}_{VS_0}, o.\texttt{getLeft();}, \underbrace{[]}_{FS_0})$$

for some type $T_M <: Message$ and some mapping function $\mathbb{F}_M$, such that the $\mathcal{M}$ object implements the M Spi Calculus term, and where H_0 contains information on $\mathcal{N}$ implementing N, which is irrelevant for this method execution.

The following steps lead to the final state.

$$\begin{array}{ll}
 & (H_1, VS_0, o.\texttt{getLeft();}, FS_0) \\
\stackrel{\text{E-Method}}{\rightarrow} & (H_1, \underbrace{(\{\texttt{this} \rightarrow (o, Pair)\} \circ []) \circ VS_0}_{VS_1}, \texttt{return this.left;}, FS_0) \\
\stackrel{\text{EC-Return}}{\rightarrow} & (H_1, VS_1, \texttt{this.left;}, (\texttt{return} \ \cdot) \circ FS_0) \\
\stackrel{\text{EC-FieldAccess}}{\rightarrow} & (H_1, VS_1, \texttt{this}, (\cdot\texttt{.left}) \circ (\texttt{return} \ \cdot) \circ FS_0) \\
\stackrel{\text{E-VarAccess}}{\rightarrow} & (H_1, VS_1, o, (\cdot\texttt{.left}) \circ (\texttt{return} \ \cdot) \circ FS_0) \\
\stackrel{\text{E-Sub}}{\rightarrow} & (H_1, VS_1, o.\texttt{left}, (\texttt{return} \ \cdot) \circ FS_0) \\
\stackrel{\text{E-FieldAccess}}{\rightarrow} & (H_1, VS_1, \mathcal{M}, (\texttt{return} \ \cdot) \circ FS_0) \\
\stackrel{\text{E-Sub}}{\rightarrow} & (H_1, VS_1, \texttt{return} \ \mathcal{M}, FS_0) \\
\stackrel{\text{E-Return}}{\rightarrow} & (H_1, VS_0, \mathcal{M}, FS_0)
\end{array}$$

The final MJ configuration corresponds to the

$$\mathcal{M}, \{(o, (M, N)), (\mathcal{M}, M), (\mathcal{N}, N)\}, \emptyset$$

state, which concludes the proof. □

The `getRight()` method verification is the same as the one for the `getLeft()` method.

Proof of correctness of the `equals()` *method.* In order to assess correctness of the `equals()` method, all the three possible cases must be covered:

1. the two objects are equal, and the method returns `true`;
2. the two objects have the same type, but their contents differ, and the method returns `false`;
3. the two objects have different types, and the method returns `false`.

For brevity, only the first case is reported here, which we believe is significant enough. Moreover, only significant configurations are explicitly written. Finally, since some extensions introduced in [25] are required, transitions will be marked with the labels specified in that work, instead of those specified in [10].

By looking up table 3, the initial state is

$$a.\texttt{equals(}b\texttt{);}, \{(a, (M, N)), (b, (M, N))\} \cup Val, Res$$

By (1), there exist $\mathcal{M}_a, \mathcal{M}_b, \mathcal{N}_a, \mathcal{N}_b$ such that

$$\begin{array}{c} Val(a) = (M, N) \Rightarrow \\ H(a) = \left(Pair, \begin{array}{l} \texttt{left} \ \rightarrow \mathcal{M}_a \\ \texttt{right} \rightarrow \mathcal{N}_a \end{array} \right) \wedge \\ Val(\mathcal{M}_a) = M \ \wedge \ Val(\mathcal{N}_a) = N \end{array}$$

and

$$
\begin{gathered}
Val(\mathscr{b}) = (M, N) \Rightarrow \\
H(\mathscr{b}) = \left(Pair,\ \begin{array}{l} \texttt{left} \;\; \to \mathcal{M}_{\mathscr{b}} \\ \texttt{right} \to \mathcal{N}_{\mathscr{b}} \end{array} \right) \wedge \\
Val(\mathcal{M}_{\mathscr{b}}) = M \ \wedge\ Val(\mathcal{N}_{\mathscr{b}}) = N
\end{gathered}
$$

So, the initial MJ configuration is

$$
(\underbrace{\{(\mathscr{a}, \left(Pair,\ \begin{array}{l} \texttt{left} \;\; \to \mathcal{M}_{\mathscr{a}} \\ \texttt{right} \to \mathcal{N}_{\mathscr{a}} \end{array} \right)), (\mathscr{b}, \left(Pair,\ \begin{array}{l} \texttt{left} \;\; \to \mathcal{M}_{\mathscr{b}} \\ \texttt{right} \to \mathcal{N}_{\mathscr{b}} \end{array} \right))\} \cup H_0}_{H_1}, \underbrace{[]}_{VS_0}, \mathscr{a}\texttt{.equals(}\mathscr{b}\texttt{);}, \underbrace{[]}_{FS_0})
$$

where H_0 contains information on $\mathcal{M}_{\mathscr{a}}, \mathcal{M}_{\mathscr{b}}$ implementing M and $\mathcal{N}_{\mathscr{a}}, \mathcal{N}_{\mathscr{b}}$ implementing N.

The following steps lead to the final configuration.

$$(H_1, VS_0, a\texttt{.equals(}b\texttt{);}, FS_0)$$

$$\xrightarrow{\text{Call}}$$

$$(H_1, (\{\texttt{this} \to (a, Pair), \texttt{obj} \to (b, Pair)\} \circ []) \circ VS_0,$$
$$\texttt{boolean result = false; if (...) \{...\} return result;}, FS_0)$$

$$\xrightarrow{\text{Sequence}}\xrightarrow{\text{VarDecl}}\xrightarrow{\text{No-op}}$$

$$(H_1, \underbrace{(\{\texttt{this} \to (a, Pair), \texttt{obj} \to (b, Pair), \texttt{result} \to (\texttt{false}, boolean)\} \circ []) \circ VS_0}_{VS_1},$$
$$\texttt{if (obj instanceof Pair) \{...\} return result;}, FS_0)$$

$$\xrightarrow{\text{Sequence}}\xrightarrow{\text{FrmOpen}}\xrightarrow{\text{VarRead}}\xrightarrow{\text{FrmClose}}$$

$$(H_1, VS_1, \texttt{if (}b\texttt{ instanceof Pair) \{...\}}, (\texttt{return result;}) \circ FS_0)$$

$$\xrightarrow{\text{FrmOpen}}\xrightarrow{\text{IofTrue}}\xrightarrow{\text{FrmClose}}$$

$$(H_1, VS_1, \texttt{if (true) \{ Pair otherPair = (Pair) obj; ... \}}, (\texttt{return result;}) \circ FS_0)$$

$$\xrightarrow{\text{IfTrue}}\xrightarrow{\text{BlkBegin}}$$

$$(H_1, \{\} \circ VS_1, \texttt{Pair otherPair = (Pair) obj; ...}, \{\} \circ (\texttt{return result;}) \circ FS_0)$$

$$\xrightarrow{\text{Sequence}}\xrightarrow{\text{FrmOpen}}\xrightarrow{\text{FrmOpen}}\xrightarrow{\text{VarRead}}\xrightarrow{\text{FrmClose}}$$

$$(H_1, \{\} \circ VS_1, \texttt{(Pair) }b, (\texttt{Pair otherPair = }\cdot) \circ (\texttt{boolean leftOK = ...}) \circ \{\} \circ (\texttt{return result;}) \circ FS_0)$$

$$\xrightarrow{\text{Cast}}\xrightarrow{\text{FrmClose}}$$

$$(H_1, \{\} \circ VS_1, \texttt{Pair otherPair = }b, (\texttt{boolean leftOK = ...}) \circ \{\} \circ (\texttt{return result;}) \circ FS_0)$$

$$\xrightarrow{\text{VarDecl}}\xrightarrow{\text{No-op}}\xrightarrow{\text{Sequence}}$$

$$(H_1, \underbrace{\{\texttt{otherPair} \to (b, Pair)\} \circ VS_1}_{VS_2}, \texttt{boolean leftOK = this.getLeft().equals(otherPair.getLeft());},$$
$$\underbrace{(\texttt{result = ...; }) \circ \{\} \circ (\texttt{return result;}) \circ FS_0}_{FS_1})$$

$$\xrightarrow{\text{FrmOpen}}\xrightarrow{\text{FrmOpen}}\xrightarrow{\text{FrmOpen}}\xrightarrow{\text{VarRead}}\xrightarrow{\text{FrmClose}}$$

$$(H_1, VS_2, a\texttt{.getLeft()},$$
$$(\cdot\texttt{.equals(...)}) \circ (\texttt{boolean leftOK = }\cdot) \circ FS_1)$$

$$\xrightarrow{\tau}{}^{*}$$

$$(H_1, VS_2, \mathcal{M}_a$$
$$(\cdot\texttt{.equals(otherPair.getLeft())}) \circ (\texttt{boolean leftOK = }\cdot) \circ FS_1)$$

$$\xrightarrow{\text{FrmClose}}$$

$$(H_1, VS_2, \mathcal{M}_a\texttt{.equals(otherPair.getLeft())}$$
$$(\texttt{boolean leftOK = }\cdot) \circ FS_1)$$

$$\xrightarrow{\text{FrmOpen}}\xrightarrow{\text{FrmOpen}}\xrightarrow{\text{VarRead}}\xrightarrow{\text{FrmClose}}$$

$$(H_1, VS_2, b\texttt{.getLeft()},$$
$$(\mathcal{M}_a\texttt{.equals(}\cdot\texttt{)}) \circ (\texttt{boolean leftOK = }\cdot) \circ FS_1)$$

$$\xrightarrow{\tau}{}^{*}$$

$$(H_1, VS_2, \mathcal{M}_b,$$
$$(\mathcal{M}_a\texttt{.equals(}\cdot\texttt{)}) \circ (\texttt{boolean leftOK = }\cdot) \circ FS_1)$$

$$\xrightarrow{\text{FrmClose}}$$

$$(H_1, VS_2, \mathcal{M}_a\texttt{.equals(}\mathcal{M}_b\texttt{)},$$
$$(\texttt{boolean leftOK = }\cdot) \circ FS_1)$$

$$\xrightarrow{\tau}{}^{*}$$

$$(H_1, VS_2, \texttt{true}, (\texttt{boolean leftOK = }\cdot) \circ FS_1)$$

Note that the $\mathcal{M}_a\texttt{.equals(}\mathcal{M}_b\texttt{)}$ method invocation returns `true` because we assume that both $\mathcal{M}_a$ and $\mathcal{M}_b$ are implementing the M Spi Calculus term, and, by induction, we can assume correctness of the method implementation.

$$\overset{\text{FrmClose}}{\longrightarrow}$$

$$(H_1, VS_2, \texttt{boolean leftOK = true}, FS_1)$$

$$\overset{\text{VarDecl}}{\longrightarrow}\overset{\text{No-op}}{\longrightarrow}$$

$$(H_1, \underbrace{\{\texttt{otherPair} \to (b, Pair), \texttt{leftOK} \to (\texttt{true}, boolean)\}}_{VS_3} \circ VS_1, \texttt{result = leftOK \&\& ...;},$$

$$\underbrace{\{\} \circ (\texttt{return result;}) \circ FS_0}_{FS_2})$$

$$\overset{\text{FrmOpen}}{\longrightarrow}\overset{\text{FrmOpen}}{\longrightarrow}\overset{\text{VarRead}}{\longrightarrow}\overset{\text{FrmClose}}{\longrightarrow}$$

$$(H_1, VS_3 \circ VS_1, \texttt{true \&\& this.getRight().equals(otherPair.getRight());}, (\texttt{result = }\cdot) \circ FS_2)$$

$$\overset{\text{FrmOpen}}{\longrightarrow}$$

$$(H_1, VS_3 \circ VS_1, \texttt{this.getRight().equals(otherPair.getRight())}, (\texttt{true \&\& }\cdot) \circ (\texttt{result = }\cdot) \circ FS_2)$$

By using the same reasoning as for the `this.getLeft().equals(otherPair.getLeft())` method invocation, this statement evolves to `true`.

$$(H_1, VS_3 \circ VS_1, \texttt{true}, (\texttt{true \&\& }\cdot) \circ (\texttt{result = }\cdot) \circ FS_2)$$

$$\overset{\text{FrmClose}}{\longrightarrow}$$

$$(H_1, VS_3 \circ VS_1, \texttt{true \&\& true}, (\texttt{result = }\cdot) \circ FS_2)$$

$$\overset{\text{AND}}{\longrightarrow}\overset{\text{FrmClose}}{\longrightarrow}$$

$$(H_1, VS_3 \circ VS_1, \texttt{result = true}, FS_2)$$

$$\overset{\text{VarWrite}}{\longrightarrow}\overset{\text{No-op}}{\longrightarrow}$$

$$(H_1, VS_3 \circ \underbrace{(\{\texttt{this} \to (a, Pair), \texttt{obj} \to (b, Pair), \texttt{result} \to (\texttt{true}, boolean)\} \circ []) \circ VS_0}_{VS_4}, \{\}, (\texttt{return result;}) \circ FS_0)$$

$$\overset{\text{BlkEnd}}{\longrightarrow}\overset{\text{No-op}}{\longrightarrow}$$

$$(H_1, VS_4, \texttt{return result;}, FS_0)$$

$$\overset{\text{FrmOpen}}{\longrightarrow}\overset{\text{VarRead}}{\longrightarrow}\overset{\text{FrmClose}}{\longrightarrow}$$

$$(H_1, VS_4, \texttt{return true;}, FS_0)$$

$$\overset{\text{Return}}{\longrightarrow}$$

$$(H_1, VS_0, \texttt{true}, FS_0)$$

The final configuration corresponds to the final Java state

$$\texttt{true}, \{(a, (M, N)), (b, (M, N))\} \cup Val, Res$$

which completes the proof. □

5. Conclusions

This paper shows how a provably correct refinement from Spi Calculus specifications into Java code implementations can be defined, thus enabling automatic generation of the implementations, while preserving the security properties verified on the specifications. This refinement relation has been 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, which is achieved by letting the user implement the marshalling functions, are also taken into account by the translation function.

As a further step, we proved 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 to 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.

Finally, it has been shown how an implementation of a class belonging to the SpiWrapper library can be verified correct with respect to the intended behavior. Correctness is proven by evaluating the Java code implementing the class within the MJ framework, and by relating that framework with the one presented here. This result increases confidence about the correctness of the whole system, because only correctness of standard libraries, like the JCA, is assumed, while the custom SpiWrapper library can be proven correct.

The translation function that has been formalized in this paper has been implemented in the spi2java tool, which has been used to successfully generate interoperable implementations of real cryptographic protocols. For example, in [22], a description of using spi2java for the implementation of the SSH transport protocol can be found.

There are still open issues that would complete and improve this work. For instance, more classes belonging to the SpiWrapper library could be proven correct by using the same approach. 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] G. Bierman, M. Parkinson, and A. Pitts. MJ: An imperative core calculus for Java and Java with effects. Technical Report 563, Cambridge University Computer Laboratory, 2003.

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

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

[13] 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.

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

[15] 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.

[16] 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.

[17] 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.

[18] 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.

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

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

[21] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.
[22] 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.
[23] 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.
[24] 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.
[25] U. Sammapun, R. Sharykin, M. DeLap, M. Kim, and S. Zdancewic. Formalizing Java-MaC. *Electronic Notes in Theoretical Computer Science*, 89(2):171–190, 2003.
[26] G. Schellhorn. ASM refinement and generalizations of forward simulation in data refinement: a comparison. *Theoretical Computer Science*, 336(2-3):403–435, 2005.
[27] 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.