Newcastle University

COMPUTING SCIENCE

Efficient Java Code Generation of Security Protocols Specified in AnB/AnBx

Paolo Modesti

TECHNICAL REPORT SERIES

No. CS-TR-1422 May 2014

TECHNICAL REPORT SERIES

No. CS-TR-1422 May, 2014

Efficient Java Code Generation of Security Protocols Specified in AnB/AnBx

P. Modesti

Abstract

The implementation of security protocols is challenging and error-prone, as experience has proved that even widely used and heavily tested protocols like TLS and SSH need to be patched every year due to low-level implementation bugs. A model-driven development approach allows automatic generation of an application, from a simpler and abstract model that can be formally verified. In this work we present the AnBx compiler, a tool for automatic generation of Java code of security protocols specified in the popular Alice & Bob notation, suitable for agile prototyping. In contrast with the existing tools, the AnBx compiler uses a simpler specification language and computes the consistency checks that agents has to perform on reception of messages. This is an important feature for robust implementations. Moreover, the tool applies various optimization strategies to achieve efficiency both at compile time and at run time. A support library interfaces the Java Cryptographic Architecture allowing for easy customization of the application.

© 2014 Newcastle University.
Printed and published by Newcastle University,
Computing Science, Claremont Tower, Claremont Road,
Newcastle upon Tyne, NE1 7RU, England.

Bibliographical details

MODESTI, P.

Efficient Java Code Generation of Security Protocols specified in AnB/AnBx
[By] P. Modesti
Newcastle upon Tyne: Newcastle University: Computing Science, 2014.

(Newcastle University, Computing Science, Technical Report Series, No. CS-TR-1422)

Added entries

NEWCASTLE UNIVERSITY
Computing Science. Technical Report Series. CS-TR-1422

Abstract

The implementation of security protocols is challenging and error-prone, as experience has proved that even widely used and heavily tested protocols like TLS and SSH need to be patched every year due to low-level implementation bugs. A model-driven development approach allows automatic generation of an application, from a simpler and abstract model that can be formally verified. In this work we present the AnBx compiler, a tool for automatic generation of Java code of security protocols specified in the popular Alice & Bob notation, suitable for agile prototyping. In contrast with the existing tools, the AnBx compiler uses a simpler specification language and computes the consistency checks that agents has to perform on reception of messages. This is an important feature for robust implementations. Moreover, the tool applies various optimization strategies to achieve efficiency both at compile time and at run time. A support library interfaces the Java Cryptographic Architecture allowing for easy customization of the application.

About the authors

Paolo Modesti is a Research Associate at the Newcastle University, UK. His main research interests are languages and tools for security, applied formal methods for security, and automatic code generation. He has worked as a senior IT officer for the Italian Revenue Agency, as an IT Security Adviser for the EU-Cafao mission in Bosnia and Herzegovina, and as a Computer Science teacher for several high schools in Italy. Paolo holds a Ph.D. in Computer Science from Ca' Foscari University Venice, Italy (2012), a Master in Software Engineering (1999) from Serc (USA)-Tecnopadova (Italy) and a M.Sc. in Computer Science from University of Udine, Italy (1990).

Suggested keywords

SECURITY PROTOCOLS
JAVA CODE GENERATION
APPLIED FORMAL METHODS

Efficient Java Code Generation of Security Protocols specified in *AnB/AnBx*

Paolo Modesti

School of Computing Science, Newcastle University, UK
paolo.modesti@newcastle.ac.uk

Abstract. The implementation of security protocols is challenging and error-prone, as experience has shown that even widely used and heavily tested protocols like TLS and SSH need to be patched every year due to low-level implementation bugs. A model-driven development approach allows the automatic generation of an application, from a simpler and abstract model that can be formally verified. In this work we present the AnBx compiler, a tool for automatic generation of Java code of security protocols specified in the popular Alice & Bob notation, suitable for agile prototyping. In contrast with existing tools, the AnBx compiler uses a simpler specification language and computes the consistency checks that agents have to perform on reception of messages. This is an important feature for robust implementation because it enforces the run-time behavior of the application. Moreover, the tool applies various optimization strategies to achieve efficiency both at compile time and at run time. A support library interfaces the Java Cryptographic Architecture allowing for easy customization of the application.

1 Introduction

The implementation of security protocols is challenging and error-prone, as experience has shown [1,2,3,4] that even widely used and heavily tested protocols like TLS and SSH need to be patched every year due to low-level implementation bugs. Moreover, the recent *"Heartbleed"* bug of OpenSSL [5] and the "*goto fail*" bug of the Apple TLS implementation [6] have shown that missing (or untested) checks, hidden deep in the code, may have a severe impact. In general, manual implementation of security protocols is prone to errors and one may easily raise the question about how many unknown bugs are just waiting to be discovered and exploited by malicious users in such harmful code. The critical aspect is that the high-level security properties of a protocol must be hard-coded explicitly, in terms of low-level cryptographic operations and checks of well-formedness.

To counter this problem, it may be useful to consider a model-driven development approach that allows automatic generation of an application, from a simpler and abstract model that can be formally verified. In this work we present the *AnBx Compiler and Code Generator*, a tool for automatic generation of Java code of security protocols specified in the popular Alice & Bob notation, suitable for agile prototyping.

From the design perspective, working on a simplified abstract model has proven very effective. It not only allows reasoning about the high-level security property, abstracting from the low-level details of the cryptographic implementation, but it also helps to reduce the problem to a size that can be handled efficiently by automatic verification tools [7,8,9,10].

Although formal languages and calculi, such as the SPI calculus [11], have been created to model security protocols, their use among code developers remains negligible due to their complexity. Instead protocol narrations, like the *Alice & Bob* notation [12], are still a very popular way to describe security protocols as a sequence of message exchanges among different principals. However, despite being intuitive, this specification technique is semi-formal because it contains a lot of implicit concepts. In particular, it does not say explicitly which (defensive) consistency checks on the received data need to be performed to verify that the protocol is running according to the specification. This fact cannot be neglected when translating narrations to real programming languages, because otherwise it would lead to incomplete and intrinsically weak implementations.

It is important to recognize that while some checks on reception are trivially derived from the narrations (verification of a digital signature, comparison of agent's identities), others are more complex. For example in the 3KP e-commerce protocol [13] the acquirer A needs to perform a lot of different checks to verify the consistency of the information provided by the customer C and the merchant M, before

authorizing a credit card transaction. These checks require the comparison of hash and MAC values computed on complex data structures on which cryptographic operators are applied. For example:

$$dec_{sk(C)}(\pi_{10}(\mathtt{R})) = hash(\pi_2(y), hash(\pi_1(z), \pi_1(\mathtt{R}), \pi_2(\mathtt{R}), \pi_3(\mathtt{R}), hmac_K(\pi_3(z), \pi_7(\mathtt{R}), \pi_5(\mathtt{R}), \pi_6(\mathtt{R})))$$

where R is the variable binding the message received by A originating from M, $y = dec_{sk(M)}(\pi_9(\mathtt{R}))$, $z = dec_{inv(pk(A))}(\pi_2(\mathtt{R}))$, $K = \pi_4(\pi_2(z))$, π_i is the projector operator, *dec* is the decryption function, $sk(C)$ and $sk(M)$ are the public keys of customer and merchant used by A to verify their digital signatures and $inv(pk(A))$ is the private key the acquirer uses to decrypt.

Such checks cannot be easily derived by hand, given that in theory in 3KP there are more than 10,000 possible different equality checks, as computed by the algorithm described in [14], that can be done by the acquirer. The good news is that they can be reduced to just a couple of dozen with an appropriate modelling of the cryptographic primitives, reordering and variable substitutions, with the added benefit of improving the efficiency of the application. Therefore, an intelligent organization of checks has a strong impact, but managing such complexity is a challenging task even for an expert programmer.

Indeed, the automatic generation of code from an abstract model is a strategy that has been already applied by different tools [15,16,17,18,19]. However, we found two aspects unsatisfactory: all the tools require the manual coding of the checks on reception and with a few exceptions (but still requiring annotations to generate running code), they do not use an *Alice & Bob* style specification language. Our *AnBx Compiler and Code Generator* is, to the best of our knowledge, the first available tool that combines these features:

- it automatically and explicitly computes the checks on reception.
- it uses a simple specification language like *AnB* [12] that is suitable for a large audience of developers and can be easily used for both formal verification with automatic tools and agile prototyping of the real implementation. The tool also supports the *AnBx* language [20], an extension of *AnB* to be employed for a purely declarative modelling of distributed protocols. These abstractions provide a compact specification of the high-level security guarantees they convey, and help to shield the designer from the details of the underlying cryptographic infrastructure.
- it applies optimization techniques to efficiently generate the implementation of industrial-size protocols, for instance SET [21] and iKP [13].
- it applies optimization techniques, such as common subexpression elimination (CSE), to improve the execution speed of the generated application, avoiding the repetition of the same cryptographic operations which are computationally expensive.
- it generates runnable Java code which interfaces, by means of a support library, the Java Cryptographic Architecture (JCA) [22,23] , directly from the *AnB* specification. We do not use annotations but only a few simple naming conventions that are used to map the *AnB* types to concrete Java types.
- it allows for flexible customization (by means of application templates and configuration files) of the generated application.
- it works for an entire protocol suite (more than 40) and not just on few examples. The suite includes complex protocols like SET, iKP, Kerberos, ISO H530, Google SSO and others, many of them derived from the *AnB* examples included in the OFMC model-checker [8].

Beyond the main contribution of an end-to-end *AnBx* to Java compiler, we have a number of further contributions:

- there is an improved way to compute the checks on reception with respect to a previous solution proposed by Briais and Nestmann [14]. This allows reducing the compilation time (in one case even from days to seconds), preventing space state explosion problems in the optimization phase, and increasing the execution speed. No previous research has been conducted on this issue.
- there is an effective compiling strategy that tries to maintain the pure and simple representation of the abstract protocol longer through the compilation process. This helps to control the complexity [24] and hide the implementation details until the code emission. One advantage of this compiling strategy is the possibility of using serialization as a method to get standard compliant and interoperable wire-format as in [18].

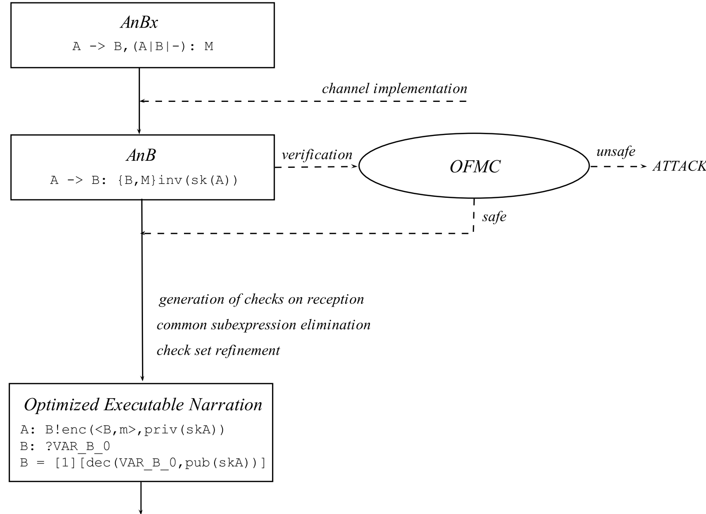

Fig. 1. Compiler front-end: pre-processing, verification, optimization of the *ExecNarr*

– there is an efficient albeit lightweight solution to check whether the formats of the incoming messages are correct, using a type system together with the native mechanisms of Java, namely casting and serialization.
– there is evidence of effectiveness of the *AnBx* language in coding revised versions of complex protocols such as iKP and SET.

In summary we present an end-to-end *AnBx* to Java compiler, which computes efficiently the checks on receptions and applies optimization strategies to generate fast code.

Plan of the paper In §2 we present the architecture of the *AnBx Compiler and Code Generator*. In §3 and §4 the translation into the intermediate format (*executable narration*) is described and the experimental results of some optimization strategies are discussed. In §5 we show how the *executable narration* is translated into Java code and its automatic generation. In §6 we illustrate the security library which wraps the JCA. In §7 we conclude and discuss related and future work.

2 Architecture of the *AnBx* compiler

The automatic Java code generation of security protocols specified in *AnBx* or *AnB* comprises several phases. A global view of the tool, which is developed in Haskell, is shown in Figures 1 and 2 and can be summarized as follows:

Pre-Processing and Verification *AnBx* → *AnB* → (*verification*)
The *AnBx* protocol is lexed, parsed and then compiled to *AnB* [12], a format suitable for verification with the external tool OFMC [8], a state of the art model checker which is part of the AVISPA [25] and AVANTSSAR [9] platforms. The compiler can also read protocols directly in *AnB*. *AnBx* and its translation to *AnB* have already been described in other works [20,26,27,28] and we refer the reader to them. We just point out that translation from *AnBx* to *AnB* can be parametrized using different channel implementations which realize the security properties, specified at the channel level, by means of different cryptographic operations.

Front-end *AnB* → *ExecNarr* → *Opt-ExecNarr*
After the verification, if the protocol is deemed safe, the *AnB* specification can be compiled into an *executable narration* (*ExecNarr*), a set of action that gives an interpretation on how the protocol participants are supposed to execute the protocol. The core of this phase (§3) is the automatic generation of the consistency checks derived from the static information of protocol narrations. The underlying theory is

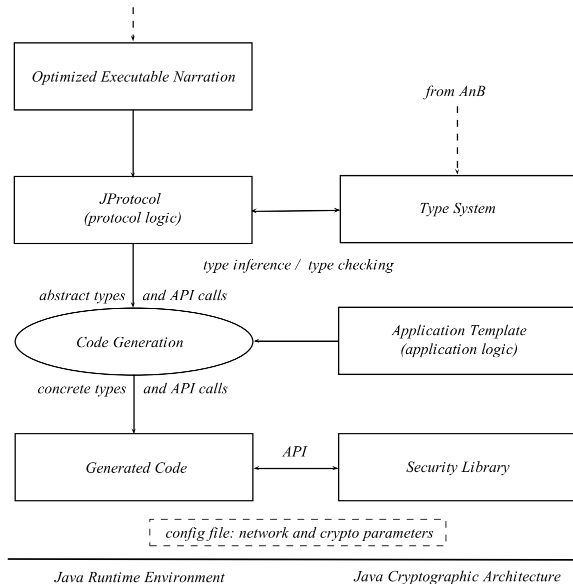

Fig. 2. Compiler back-end (Type System and Code generator) and run-time support

a formal operational semantics for protocol narrations, proposed by Briais and Nestmann [14].Since the language features considered in [14] are insufficient to model many real protocols (and even less expressive than those available in *AnB*) we extended the executable narration semantics to support MACs, key agreements (i.e Diffie-Hellmann), tuples and user-defined functions. In this way it is possible to model a wider and more realistic range of applications. The checks are expressed by means of consistency formulas, and given the high number of generated formulas, the tool applies some simplification strategies which offer good results in practice. In the original proposal [14] some simplification was already implemented but this turned out to be insufficient to deal with some large protocols. We changed the rule of analysis and synthesis applied to messages and expressions in order to improve the computation of checks and prune the checks that were inconsistent with the behavior of the cryptographic functions used in the implementation.

The *optimized executable narration* (*Opt-ExecNarr*) (§4) goes further in this direction and applies some optimization techniques, including common subexpression elimination (CSE), which in general are useful to generate efficient code. We identify the set of cryptographic operations, which in general are computationally expensive, and optimize the code to reduce the overall execution time, introducing variables storing partial results, and making a reordering with the purpose of minimizing the number of cryptographic operation performed. An additional option offered by the reordering made during the CSE, is the possibility to prune the check set. In fact, provided the success of a predecessor check that compares a variable with an expression, it is safe to substitute the occurrences of the expression with the variable in the subsequent checks, dropping the duplicated checks.

Back-end *Opt-ExecNarr* $\rightarrow$ (*protocol logic*) + (*application logic*) $\rightarrow$ *Java*

The final result of the compilation is the generation of the Java source code from the *Opt-ExecNarr*. The previous phases are fully language independent and do not require any adaptation in case another programming language is considered. But even in the back-end we postponed any language dependent decision in order to increase the compiler's portability and simplify re-targeting, as long as other object oriented and procedural programming languages are considered. Moreover, we designed a versatile tool that allows for a wide range of user customizations. The details of the compiler back-end are presented in Section 5. Here we summarize the main features and components:

Code generation strategy First of all, we make a distinction between the *protocol logic* and the *application logic*. The latter is implemented by means of parametrized application template files written in the target language. The templates are instantiated with the information (the *protocol logic*) derived from the optimized executable narration. We model the *protocol logic* by means of a language independent

```
Protocol: Fresh_From_A
Types:
  Agent A,B;
  Number Msg,Nonce;
  Function pk,sk,hash
Knowledge:
  A: A,B,pk,sk,inv(pk(A)),inv(sk(A));
  B: A,B,pk,sk
Actions:
  A -> B: A
  B -> A: {Nonce,B}pk(A)
  A -> B: {Nonce,B,Msg}inv(sk(A))
Goals:
  B authenticates A on Msg
```

Fig. 3. A challenge-response implementation in *AnB* of fresh authentic exchange

intermediate format called *JProtocol*, which is basically a typed representation of the *Opt-ExecNarr*. This is useful to parametrize the translation and to make easier to emit code in other programming languages.

Type System Building the *JProtocol* requires a type system modeling a typed abstract representation of the security related portion of a generic procedural language supporting a rich set of abstract cryptographic primitives. The type system is used to infer the type of expressions and variables and insures that the generated code is well-typed. It has the additional benefit to detect at run time if the structure of the incoming messages is coherent with the one specified by the narration. We delegate to the Java run-time support to check whether the incoming messages belong to the expected class and, in the case of detection of an error, a `ClassCastException` is raised. In our approach, we found this to be the proper level to handle this problem, avoiding to overload the abstract model with additional implementation details (i.e., annotations). The types, the typing rules, and the naming conventions mapping the *AnB* types to concrete Java types are shown in Section 5.2. The type checker also detects the correct use of the naming convention.

Security API The run-time support relies on the the cryptographic services offered by the Java Cryptographic Architecture (JCA) [22,23]. In order to connect to the JCA, we designed an API for security (Section 6) which wraps, in an abstract way, the JCA interface and implements the custom classes necessary to encode the generated programs in Java. The *AnBxJ* library offers a high degree of generality and customization, since the API does not commit to any specific cryptographic solution (algorithms, libraries, providers). Moreover, the library provides access in an abstract way to the communication primitives used to exchange messages in the standard TCP/IP network environment. Communication and cryptographic run-time errors are handled at this level, and exceptions are raised. The generated code comes along with a configuration file that allows the developer to fully customize the deployment of the application at the cryptographic (keystore location, aliases, cipher schemes, key lengths, etc) and network level (IP addresses, ports, etc) without requiring to regenerate the application.

Code emission The code emission is performed by instantiating the protocol templates, i.e., the skeleton of the application, with the information derived from the protocol logic (Section 5.4). It is worth noting that only at this final stage the language specific features and their API calls are actually bound to the protocol logic. To this end two mappings are required. One between the abstract and the concrete types; another one between the abstract actions and the concrete API calls. It is important to underline that the application templates are generic, i.e., independent from the specific protocol, and can be modified by the user in order to fit his application domain. An Ant [29] build file is generated to easily build and run the application, in addition to the Java classes and a configuration file. In summary, the tool allows for a one-click code generation of widely configurable and customizable ready-to-run Java applications from an *AnBx* or *AnB* specification.

3 Compiling *AnB*/*AnBx* into Executable Narrations

We now describe how protocols in *AnB* and *AnBx* can be compiled into *ExecNarr*. The goal is to obtain an operational description of the actions each agents has to perform, including the informative checks on reception of messages. This process involves modeling the agent's knowledge, mapping the

AnB actions to the *ExecNarr* format and computing the checks. The major difference between the two specifications is that *AnB* describes the ideal running of the protocols showing the messages exchanged by the agents from the point of view of an external observer, while the executable narration describes operationally the actions performed from the point of view of each agent. This is evident considering the different perspective of an *AnB* action like `B->A: {Nonce,B}pk(A)`, as in the protocol in Figure 3. From the operational point of view of the Agent `B`, this correspond to construct an expression with two components `{Nonce,B}`, then encrypt it with the public key of `A` and send the ciphertext to the agent `A`. On the other end of the communication link, from the point of view of `A`, this action corresponds to receiving a value which can be stored in a variable `R1`. Then `A` has to deconstruct `R1` performing the decryption with her private key and verifying whether the second component of the plaintext correspond to the identity of `B` that she has stored in her knowledge.

3.1 The *AnB* language

An example of a simple *AnB* protocol is given in Figure 3. Its goal is to allow a fresh authentic exchange of the message `Msg` between two parties using challange-response. The language description and the formal semantics is available in [12]. For simplicity, here we skip some details but before proceeding it is useful to recall that a protocol specification in *AnB* comprises of four mandatory sections:

- `Types`: describes the principals (`Agent`) involved in the protocol, along with protocol data (`Number`) and operators on them (`Function`), including the cryptographic functions for signing (`sk`) and encrypting (`pk`) using public key cryptography and an hash function (`hash`);
- `Knowledge`: specifies the initial knowledge of each principal. The original specification of *AnB* permits inclusion of agents names, constants, functions, and their combination, but not freshly generated values because they are created during the protocol execution. This poses a problem in protocols where the agents have to use a symmetric pre-shared key or, like in the e-commerce protocols iKP [30,13] and SET [31], where the description of the payment process, assumes that the price and the order details have been already agreed between the customer C and the merchant M before the protocol execution. This schema is supported by *AnBx* by means of a new construct in section `Knowledge` allowing the specification of values shared among a set of agents as in the statement `C,M share Price,Desc`, which models the aforementioned scenario;
- `Actions`: specifies the sequence of statements that constitute the ideal run of the protocol;
- `Goals`: specifies the goals that the protocol is meant to convey. These are useful for the verification of the protocol but are not used to generate the executable narration.

Now we show how to translate an *AnB* protocol to an executable narration.

3.2 The executable narration syntax

In the target format (the syntax is shown in Table 1), the protocol is composed of two sections, a *declaration,* a header of the actual *narration,* which includes the initial knowledge of each agent, the names generated by them and the names that are assumed to be initially known only by a subset of agents. The latter is similar to the `share` construct we introduced in *AnBx*.

The *agents* are taken from set of agent names $\mathbf{A}$ and the *messages* are built upon set of names $\mathbf{N}$. It is assumed that $\mathbf{A} \bigcap \mathbf{N} = \oslash$.

To handle asymmetric cryptography, the inverse key $inv(M)$ of a message M is defined as follows:

$$inv(M) = \begin{cases} pub(M') & if\ M = priv(M') \\ priv(M') & if\ M = pub(M') \\ \perp & otherwise \end{cases}$$

The agents can verify if two messages M_1 and M_2 are inverse keys one of each other, trying to decrypt with M_2 a message encrypted with M_1 and conversely.

The statement **private** k means that k is a name which is initially only available for the agents involved in the protocol. For instance, this is useful to simulate that an agent A and a server S initially share a secret key kAS:

expressions		
$E, F ::=$	a	*name*
	$\mid\ A$	*agent name*
	$\mid\ x$	*variable*
	$\mid\ hash(E)$	*hashing*
	$\mid\ pub(E)$	*public key*
	$\mid\ priv(E)$	*private key*
	$\mid\ (E_1, ..., E_n)$	*tuple* *
	$\mid\ \pi_i(E)$	$i - th\ projection$ *
	$\mid\ enc(E, F)$	*asymmetric encryption*
	$encS(F, F)$	*symmetric encryption* *
	$\mid\ dec(E, F)$	*decryption*
	$\mid\ hmac(E, F)$	*hmac* *
	$\mid\ kap(E, F)$	$key\ agreement\ half\ key$ *
	$\mid\ kas(E, F)$	$key\ agreement\ full\ key$ *
	$\mid\ E(F)$	*function* *
formulae		
$\phi ::=$	$[E = F]$	*matching*
	$\mid\ [E : \mathbf{M}]$	$well - formedness\ test$
	$\mid\ inv(E, F)$	*inversion test*
simple actions		(A, B *are agent names*)
$I ::=$	$A : \mathbf{new}\ k$	*fresh name generation*
	$\mid\ A : send(B, E)$	*message emission*
	$\mid\ A : receive(x)$	*message reception*
	$\mid\ A : x := E$	*assignment*
	$\mid\ A : \phi$	*check*
narrations		
$L ::=$	ϵ	*empty narration*
	$\mid\ I; L$	*non empty narration*
declarations		
$D ::=$	$A\ \mathbf{knows}\ M$	*initial knowledge* (M *is a ground expression*)
	$\mid\ A\ \mathbf{generates}\ n$	*fresh name generation*
	$\mid\ \mathbf{private}\ k$	*private name*
protocol narrations		
$P ::=$	$D; P \mid L$	*declarations + narration*

Table 1. Syntax of the executable narrations (Extensions with respect to [14] are marked with *. However, when computing checks, as in [14], pairs are used instead of tuples $\star$ for performance reasons.)

private kAS
A **knows** kAS
S **knows** kAS

A **knows** m denotes that, initially, the agent A knows the message m. The statement A **generates** n implies that the agent A generates a fresh name n (a nonce or a freshly generated key). All fresh names must be declared explicitly. Deriving the declaration section from the *AnB* agent's knowledge is straightforward (see [26]).

The meaning of actions in section *narration* is intuitive. $A \to B :\ M$ denotes that the agent A sends a message M to the agent B. A prerequisite here is to translate the *AnB* to a format (2) compatible with the definition of expressions in the executable narrations. This is because one of the goal of this phase is to derive the expression that map messages from the agent's point of view. Again the mapping of messages is quite straightforward and implies the definition of a function $\tau : \mathbf{M_{AnB}} \to \mathbf{M}$ that translate of *AnB* messages to their equivalent, where $\mathbf{M_{AnB}}$ and $\mathbf{M}$ are the sets of messages in the two formats (Table 3). To compute τ, we also need the type information available from the protocol header. Each *AnB* action on a plain channel $A \to B :\ M$ is simply translated to an equivalent action $A \to B :\ \tau\,(M)$.

messages $\mathbf{M}$		
$M, N ::=$	a	*name*
	$\mid A$	*agent name*
	$\mid hash(M)$	*hashing*
	$\mid pub(M)$	*public key*
	$\mid priv(M)$	*private key*
	$\mid (M.N)$	*pair*
	$\mid enc(M,N)$	*asymmetric encryption*
	$\mid encS(M,N)$	*symmetric encryption* *
	$\mid hmac(M,N)$	*hmac* *
	$\mid kap(M,N)$	*key agreement half key* *
	$\mid kas(M,N)$	*key agreement full key* *
	$\mid M(N)$	*function* *

Table 2. Syntax of Messages for the protocol narrations (plus extensions *)

$$\tau(\cdot) : \mathbf{M_{AnB}} \rightarrow \{\bot\} \cup \mathbf{M}$$

$\tau(a)$	$:= toUpper(a)$	$if\ a \in \mathbf{Agent}$
$\tau(a)$	$:= toLower(a)$	$if\ a \in \mathbf{Ident}\ \wedge\ a \notin \mathbf{Agent}$
$\tau\left(\{m\}_k\right)$	$:= enc(\tau(m), \tau(k))$	
$\tau\left(\{\lvert m\rvert\}_k\right)$	$:= encS(\tau(m), \tau(k))$	
$\tau(op(a))$	$:= pub(op + \tau(a))$	$if\ a \in \mathbf{Agent}\ \wedge\ op \in \mathbf{Function}$ $\wedge\ op \in \{pk, sk\}$
$\tau(inv(op(a)))$	$:= priv(op + \tau(a))$	$if\ a \in \mathbf{Agent}\ \wedge\ op \in \mathbf{Function}$ $\wedge\ op \in \{pk, sk\}$
$\tau(exp(g,x))$	$:= kap(\tau(g), \tau(x))$	$if\ g, x \in \mathbf{Number}$
$\tau(exp(exp(g,x), y)$	$:= kas(kap(\tau(g), \tau(x)), \tau(y))$	$if\ g, x, y \in \mathbf{Number}$
$\tau(hmac_k(m))$	$:= hmac(\tau(m), \tau(k))$	$if\ k \in \mathbf{Symmetric_key}$
$\tau(hash(m))$	$:= hash(\tau(m))$	
$\tau(f(x))$	$:= F(\tau(x))$	$if\ \tau(f) = F \in \mathbf{M}\ \wedge\ f \in \mathbf{Function}$
$\tau(E)$	$:= \bot$	*otherwise*

Table 3. Translation of *AnB* messages to *executable narrations* (+ is the concatenation of names)

In contrast to the original work [14], we found more convenient, for efficiency reasons we are going to detail later, to distinguish between symmetric and asymmetric encryption. Moreover, we introduced the support of operators like *hmac*, *kap*, *kas* and user defined functions. *kap* and *kas* are used to model the basic operations on keys which are available in key agreement protocols like Diffie-Hellman. These functions are borrowed from [32] (we explicit here the parameter g because it is necessary in the next steps of the compilation):

- $kap(g, x)$ is the half key computed from secret x;
- $kas(k, y)$ is the full key computed from an half-key k and a secret y.

They satisfy the algebraic property $kas(kap(g, x), y) \approx kas(kap(g, y), x)$, given the pre-shared parameter g.

3.3 Compiling Protocol Narrations

Having completed the previous steps it is now possible to compute the checks on reception applying the ideas proposed by Briais and Nestmann [14]. In this section we summarize their approach. Then *AnB* actions are decomposed making the behavior of every single agent more explicit. Atomic exchanges of the form $A \rightarrow B :\ M$ can hence be compiled to more specific basic actions:

emission $A :$ $\texttt{send}(B, E)$ of a message expression E (evaluating to M);
reception $B :$ $x := \texttt{receive}()$ of a message and its binding to a variable x;

$$[\![\cdot]\!] : E \rightarrow \{\bot\} \cup \mathbf{M}$$

$$\begin{array}{rll}
[\![E]\!] & := E & \mathit{if}\ E \in \mathbf{N} \cup \mathbf{A} \\
[\![(E.F)]\!] & := ([\![E]\!].[\![F]\!]) & \\
[\![\pi_1(E)]\!] & := M & \mathit{if}\ [\![E]\!] = (M.N) \in \mathbf{M} \\
[\![\pi_2(E)]\!] & := N & \mathit{if}\ [\![E]\!] = (M.N) \in \mathbf{M} \\
[\![enc(E,F)]\!] & := enc([\![E]\!],[\![F]\!]) & \\
[\![dec(E,F)]\!] & := M & \mathit{if}\ [\![E]\!] = enc(M,N) \in \mathbf{M}\ \wedge\ [\![F]\!] = N \in \mathbf{M} \\
[\![encS(E,F)]\!] & := encS([\![E]\!],[\![F]\!]) & \\
[\![dec(E,F)]\!] & := M & \mathit{if}\ [\![E]\!] = encS(M,N) \in \mathbf{M}\ \wedge\ [\![F]\!] = N \in \mathbf{M} \\
[\![op(E)]\!] & := op([\![E]\!]) & \mathit{if}\ op \in \{pub, priv, hash\} \\
[\![op(E,F)]\!] & := op([\![E]\!],[\![F]\!]) & \mathit{if}\ op \in \{hmac, kas, kap\} \\
[\![E(F)]\!] & := M([\![F]\!]) & \mathit{if}\ [\![E]\!] = M \in \mathbf{M} \\
[\![E]\!] & := \bot & \mathit{otherwise}
\end{array}$$

$$[\![\cdot]\!] : F \rightarrow \{\mathbf{true}, \mathbf{false}\}$$

$$\begin{array}{rll}
[\![tt]\!] & := \mathbf{true} & \\
[\![\phi \wedge \psi]\!] & := \mathbf{true} & \mathit{if}\ [\![\phi]\!] = [\![\psi]\!] = true \\
[\![[E = F]]\!] & := \mathbf{true} & \mathit{if}\ [\![E]\!] = [\![F]\!] = M \in \mathbf{M} \\
[\![[E : \mathbf{M}]]\!] & := \mathbf{true} & \mathit{if}\ [\![E]\!] = M \in \mathbf{M} \\
[\![inv(E,F)]\!] & := \mathbf{true} & \mathit{if}\ [\![E]\!] = M \in \mathbf{M}\ \wedge\ [\![F]\!] = inv(N) \in \mathbf{M} \\
[\![\phi]\!] & := \mathbf{false} & \mathit{otherwise}
\end{array}$$

Table 4. Definition of the evaluation of expressions and formulas

check $B : \phi$ for the validity of the formula ϕ from the point of view of agent B.

In addition, the following actions are useful to express additional actions during the protocol execution:

scoping $A :$ `new` k, represents the creation and scope of private names;
assignment $A : x := E$ the variable x assume the value of the expression E (this became handy only when performing the CSE optimization)

When an agent receives a message, he binds that message to a fresh variable for reference in the subsequent processing. For this purpose, a set $x, y, z, ...$ of variables $\mathbf{V}$ is introduced. Such set is assumed to be disjoint from $\mathbf{N} \cup \mathbf{A}$.

Since an agent does not only handle messages but also variables, the notion of *message expressions* ($\mathbf{E}$) is introduced, along with the operations needed to construct and deconstruct messages. Messages received during the protocol execution, and stored in variables x, are closely related to the statically intended messages M described in the narration. For this reason, bindings $(M, x) \in \mathbf{M} \times \mathbf{E}$ are used.

The process of finding out whether some expression represents a particular message, is formalized by means of an evaluation function which is shown in Table 4. Note that if an expression contains variables the evaluation fails.

Formulas ϕ on received messages are described by a conjunctions of three kinds of checks:

equality $[E = F]$ on expressions denoting the comparison of two bit-streams of E and F;
well-formedness $[E : \mathbf{M}]$ denoting the verification of whether the projections and decryption contained in E are likely to succeed;
inversion $inv(E, F)$ denoting the verification that E and F evaluate to inverse messages.

The evaluation function is extended to formulas; it can be seen in [14] that, $[E : \mathbf{M}]$ is just a macro for $[E = E]$. Similarly, $inv(E, F)$ can be encoded (for example) as $[dec(enc((E.F), E), F) : \mathbf{M}]$.

Since consistency checks will have to operate on *(message,expression)* pairs, the representation of the agent's knowledge must be generalized to finite subsets of $\mathbf{M} \times \mathbf{E}$. The underlying idea is that a pair (M, E) denotes that an expression E is equivalent to the message M. For this reason is it necessary to introduce the notion of *knowledge sets*, and two operations on them:

– *synthesis* reflecting the closure of knowledge sets using message constructors;

$$\textsf{SYN-OP1}\ \frac{(M,E) \in \mathcal{S}(K)}{(op(M), op(E)) \in \mathcal{S}(K)}\ op \in \{pub, priv, hash\}$$

$$\textsf{SYN-OP2}\ \frac{(M,E) \in \mathcal{S}(K) \qquad (N,F) \in \mathcal{S}(K)}{(op(M,N), op(E,F)) \in \mathcal{S}(K)}\ op \in \{enc, encS, hmac, kas, kap\}$$

$$\textsf{SYN-PAIR}\ \frac{(M,E) \in \mathcal{S}(K) \qquad (N,F) \in \mathcal{S}(K)}{((M.N), (E.F)) \in \mathcal{S}(K)}$$

$$\textsf{SYN-FUN}\ \frac{(M,E) \in \mathcal{S}(K) \qquad (M,F) \in \mathcal{S}(K) \qquad M \in \mathbf{N}}{(M(N), (E(F)) \in \mathcal{S}(K)}$$

$$\textsf{SYN-KAP}\ \frac{(M,E) \in \mathcal{S}(K) \qquad (N,F) \in \mathcal{S}(K) \qquad M \in \mathbf{N}}{(kap(M,N), kap(E,F)) \in \mathcal{S}(K)}$$

$$\textsf{SYN-KA-EQ}\ \frac{(kas(kap(M,N),O), kas(kap(E,F),G) \in \mathcal{S}(K) \qquad M \in \mathbf{N}}{(kas(kap(M,O),N), kas(kap(E,G),F) \in \mathcal{S}(K)}$$

Table 5. Synthesis SYN-rules

- *analysis* reflecting the exhaustive recursive decomposition of knowledge pairs as enabled by the currently available knowledge.

Formally these sets and operations are defined as follows [14]:

Definition 1 (Knowledge).

- *Knowledge sets* $K \in \mathbf{K}$ are finite subsets of $\mathbf{M} \times \mathbf{E}$.
- The *synthesis* $\mathcal{S}(K)$ of K is the smallest subset of $\mathbf{M} \times \mathbf{E}$ containing K and satisfying the SYN-rules in Table 5.
- The *analysis* $\mathcal{A}(K)$ of K is $\bigcup_{n \in \mathbb{N}} \mathcal{A}_n(K)$ where the sets $\mathcal{A}_i(K)$ are the smallest sets satisfying the ANA-rules in Table 6.

With respect to the original work of [14] we added the SYN-rules SYN-OP2, SYN-FUN, SYN-KAP, SYN-KA-EQ and the ANA-rules ANA-OP2, ANA-FUN in order to support a more expressive language like *AnB*. In addition, we find useful to distinguish between symmetric and asymmetric encryption which came handy in optimization to take advantage of the different properties of these primitives. These new rules are necessary to generalize the notion of synthesis and analysis with functions and operators defined in *AnB*/*AnBx*, and previously unavailable in the original work. It is worth noting that the SYN-KA-EQ rule is necessary to model the algebraic equivalence $(kas(kap(g,x),y) \approx (kas(kap(g,y),x)$.

The above knowledge representation allows generating the checks required on message reception. Assuming that agents have some initial knowledge (usually of the form (M, M) where M are the messages corresponding to the known facts as stated in the knowledge section of the protocol) the knowledge set is extended during the protocol execution, according to the information learned during the reception actions: the expected message and the corresponding expression. The checks must verify:

1. if the expectation of the recipient on the received message (as expressed statically in the narration) is matched by the recipient's current knowledge;
2. if the knowledge increase obtained while receiving the message is consistent with the previously acquired knowledge.

Some checks depend on the structure of messages: for instance, if an agent receives an encrypted message he should be able to decrypt the corresponding expression if he knows the correct key.

Other checks result instead from the fact that a message M may occur more than once in a protocol narration. In this case the same message M could be associated to two different expressions E_1 and E_2. Since the term M is used in the protocol specification to refer to the very same message, the current knowledge set can be considered consistent only if the two expressions evaluate to the same message. In

$$\text{ANA-INI}\ \frac{(M,E) \in K}{(M,E) \in \mathcal{A}_0(K)}$$

$$\text{ANA-OP1}\ \frac{(op(M),E) \in \mathcal{A}_n(K)}{(op(M),E) \in \mathcal{A}_{n+1}(K)}\ op \in \{pub, priv, hash\}$$

$$\text{ANA-OP2}\ \frac{(op(M,N),E) \in \mathcal{A}_n(K)}{(op(M,N),E) \in \mathcal{A}_{n+1}(K)}\ op \in \{hmac, kap, kas\}$$

$$\text{ANA-FUN}\ \frac{(M(N)),E) \in \mathcal{A}_n(K)}{((M(N)),E) \in \mathcal{A}_{n+1}(K)}$$

$$\text{ANA-FST}\ \frac{((M.N),E) \in \mathcal{A}_n(K)}{((M,\pi_1(E)) \in \mathcal{A}_{n+1}(K)}$$

$$\text{ANA-SND}\ \frac{((M.N),E) \in \mathcal{A}_n(K)}{((N,\pi_2(E)) \in \mathcal{A}_{n+1}(K)}$$

$$\text{ANA-DEC}\ \frac{(encop(M,N),E) \in \mathcal{A}_n(K) \qquad (inv(N),F) \in \mathcal{S}(\mathcal{A}_n(K))}{(M, dec(E,F) \in \mathcal{A}_{n+1}(K)}\ encop \in \{enc, encS\}$$

$$\text{ANA-DEC-REC}\ \frac{(encop(M,N),E) \in \mathcal{A}_n(K) \qquad (inv(N),F) \notin \mathcal{S}(\mathcal{A}_n(K))}{(encop(M,N),E) \in \mathcal{A}_{n+1}(K)}\ encop \in \{enc, encS\}$$

$$\text{ANA-NAM-REC}\ \frac{(M,E) \in \mathcal{A}_n(K) \qquad M \in \mathbf{N} \cup \mathbf{A}}{(M,E) \in \mathcal{A}_{n+1}(K)}$$

Table 6. Analysis ANA-rules

the case of asymmetric keys, it can also happen that, in some knowledge set, there is a combination of (M_1, E_1) and (M_2, E_2) where $M_1 = inv(M_2)$. In this case, $inv(E_1, E_2)$ should also be satisfied.

These requirements are formalized in the definition of consistency formula [14]:

Definition 2 (Consistency Checks).
Let K be a knowledge set. Its consistency formula $\Phi(K)$ is defined as follows:

$$\begin{aligned}\Phi(K) := &\ \textstyle\bigwedge_{(M,E)\in K} [E : \mathbf{M}] \\ &\wedge \textstyle\bigwedge_{(M,E_i)\in K \wedge (M,E_j)\in \mathcal{S}(K) \wedge E_i \neq E_j} [E_i = E_j] \\ &\wedge \textstyle\bigwedge_{(M,E_i)\in K \wedge (inv(M),E_j)\in \mathcal{S}(K)} inv\,(E_i, E_j)\end{aligned}$$

The first conjunction clause checks that all expressions can be evaluated, the second checks that if there are several ways to build a message M, then all the corresponding expressions must evaluate to the same value. The third conjunction clause checks that if it was possible to generate a message M and its inverse $inv(M)$, then the corresponding expressions must also be mutually inverse.

The generation of the consistency formulas, implies comparing pairs taken from K with pairs taken from $\mathcal{S}(K)$. It can be shown that comparing pairs only in K is not sufficient. On the other hand, comparing any possible combination of pairs taken from $\mathcal{S}(K)$, would lead to an infinite formula. For this reason, knowledge sets can often be simplified without loss of information, i.e., without undermining the computation of the consistency formula (for the details refer to [14]).

Compilation The above notions are the elements required to compile the protocol into an executable protocol narration. The translation function keeps track of the global information on the used variables and hidden names, as well as the agent local information, about their knowledge on generated names.

In detail, statements like **private** k and A **generates** n check both that the local (or generated) name is fresh, but they are handled differently: whereas the construction A **generates** n increases the knowledge of A, the name k of **private** k is not added to any knowledge; this task is deferred to the explicit A **knows** k clauses for the intended A.

The compilation of $A \to B :\ M$ checks that M can be synthesized by A, instantiate a new variable x and adds the pair (M, x) to the knowledge of B.

Protocol	*Compile Time (sec)*			*Execution Time (sec)*			*Memory usage* (MB)*		
	(1)	(2)	(3)	(1)	(2)	(3)	(1)	(2)	(3)
2KP (orig)	97.75	2.25	2.43	1.080	0.813	0.715	3.46	8.24	8.69
2KP (AnBx)	1.03	0.13	0.14	1.013	0.945	0.799	0.91	0.88	0.98
3KP (orig)	6,945.43	13.20	13.10	100.848	0.826	0.779	411.52	33.64	36.13
3KP (AnBx)	8.88	2.25	2.43	1.255	0.908	0.738	1.05	1.04	1.04
SETv2 (orig)	513,827.20	3.04	3.51	4.052	1.107	0.888	226.07	7.91	10.63
SETv2 (AnBx)	0.84	0.05	0.05	1.080	0.889	0.816	0.74	0.83	0.81
H530	1.89	2.70	3.60	1.962	1.792	0.729	10.14	5.31	5.23
Google SSOv2	1.33	0.02	0.05	1.072	0.902	0.804	0.68	0.75	0.73
Kerberos PKinit	0.37	0.36	0.37	1.320	0.936	0.832	1.35	1.88	2.06

Table 7. Experimental results: (1) as in [14] w/o opt, (2) "prudent" and (3) "standard" optimization (* at compile time)

The consistency formula $\Phi(\mathcal{A}(K'_B))$ of the analysis of the updated knowledge K'_B defines the checks ϕ to be performed by B at run-time. The compilation process is formalized in [14] and the differences with the current implementation are reported in [28].

3.4 Performance issues and optimization of the generation of consistency checks

A preliminary version of our compiler, released in 2012 [26], implemented verbatim the method proposed in [14], only extending the analysis and synthesis rules in order to support cryptographic functions which were not available in [14], HMAC and Diffie-Hellman key agreements in particular. Unfortunately we found that, especially with some industrial-size protocols, it turned out to be very inefficient. In our experiments (Table 7, Figures 4 and 5), we found challenging working with the original specification of the e-commerce protocols SET [21] (we considered the unsigned variant denoted SETv2 in which the customer does not possesses asymmetric keys) and 3KP [13]. A clear symptom of inefficiency was the fact that these protocols required very long time to be compiled (around 1 hr and 55 min for 3KP and almost 6 days for SETv2 on a Windows 7 64-bit machine with CPU Intel Core i7-3770 3.40GHz, 8 GB RAM, JDK 7.0.45 64-bit, Haskell Platform 2013.0.0.2). We refer to this configuration throughout the paper. In same cases, applying the CSE optimization to the intermediate code was impossible. We faced a space explosion problem, leading to out of memory errors.

These protocols are characterized by several nested levels of encryption and by an extensive use of hash and MAC functions that are used to verify the consistency of data exchanged among the protocol participants. This leads to the generation of a high number of consistency checks involving comparison of message digests. In contrast, our revised versions [20,26] of SET and 3KP versions were performing rather well with compile time of less than 1 sec for SET and less than 9 sec for 3KP.

In addition we noticed that some of the computed checks were failing anyway. It turned out that the reason of this discrepancy was the different behavior, in the abstract and concrete model, of the cryptographic primitives. In fact, given two identical messages and encryption keys, a non-deterministic encryption scheme returns two different ciphertexts which are indistinguishable by an observer. This nice (in the real world) property was not properly captured by the model.

To address this issue, we could have refined our model adding to the abstract functions *enc* and *encS,* modelling symmetric and asymmetric encryption, a third argument, as proposed by Abadi and Fournet in [33], which represents a randomized quantity to distinguish between different ciphertexts of the same message and key. This would force the tool to synthesize checks that match the behavior of the non-deterministic encryption primitives, but at price of computing more expressions and with an extra parameter. Therefore, to solve the problem more efficiently, we found more straightforward to reason on how to avoid the generation of the expressions involved in these failing checks.

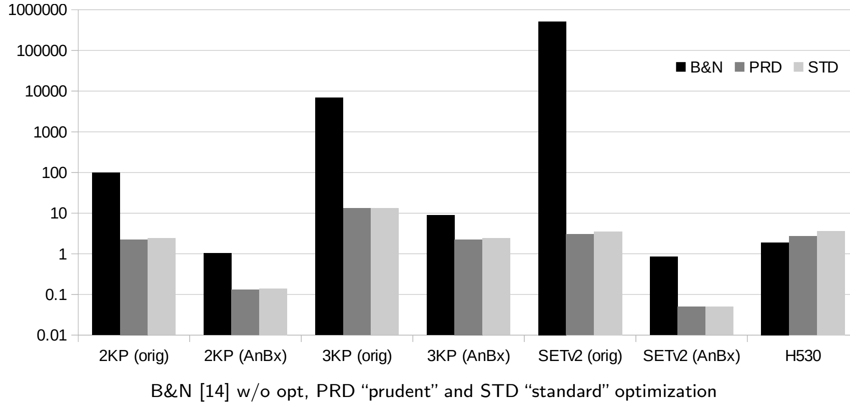

Fig. 4. Compile Time (sec), logarithmic scale

The key point is to observe that, if an agent knows the correct decryption key, he will deconstruct the ciphertext using the ANA-rules. In this case he will only store the decrypted message in his own knowledge, forgetting the ciphertext. The only exception is represented by forwarding channels, as in SET and iKP where, for example, the merchant must forward a message originating from the customer but secret for the acquirer. Given this, and assuming the non-determinism of the encryption scheme, if, to build an expression involved in an equality check, the agent needs to encrypt some data, the check is going to fail anyway because the new ciphertext will be different from the one to be compared with.

Therefore, if we prevent the agent from using the SYN-OP2 rule, for $op \epsilon \{enc, ensS\}$, when computing the checks, we substantially avoid computing any check which requires synthesizing new terms using symmetric and asymmetric encryption. With respect to the solution proposed in [33] we save a significant computational effort because there is not need to synthesize a lot of useless expressions that will be later discharged. It is important to underline that this does not undermine the robustness application because we just prune checks failing due to the over approximation of the abstract model.

The overall achievement here is two-fold: on the one hand we achieve a dramatic increase of speed in computing the checks; on the other hand we get rid of (most of) the checks would fail anyway because of the non-deterministic nature of the cipher schemes used in practice. In any case we leave the option to the developer to choose whether to model deterministic or non-determinist encryption schemes, distinguishing between symmetric and symmetric encryption.

The only (few) potential checks which are not still captured, are those involving terms which cannot be decrypted on reception and that are stored in the agent's memory. However, after the generation of the checks, they can easily be detected using pattern matching and type-checking, as we do. The practical advantage here is that, for complex protocols, we drastically reduce the check set and cut the computational effort. Therefore, the optimization (CSE) and check refinement can be performed, in the next steps, on a reduced number of checks. It should also be noted that the reordering performed by CSE allows for further optimization of the checks as we explain in section 4.

Experimental results From the experimental point of view, we were able to reduce the compile time for 3KP, from 1 hr 55 min to just 13 sec, decreasing the peak memory usage (measured by the Profiler of the Haskell Compiler 7.6.3) from 411 MB to just 16 MB. The execution time was also cut from 100 sec to just 1.34 sec. This large difference is explained by the fact that the standard algorithm compute more than 10,000 checks (but most of them are failing), while our version, generates about 50 checks. So far, in these experiments, we did not consider any optimization implemented in the following steps. However, memory usage reduction is a prerequisite to perform CSE on machines having a standard amount of RAM (4-8 GB). We also carried out an experiment performing CSE on the protocol 3KP and we found that, without our optimization, at least 48 GB of RAM were necessary complete the task.

For the protocol SETv2, the good news is that it can now be compiled in less than 3 sec while before it required almost 6 days. Peak memory usage was also decreased from 262 MB to just 6 MB. However, the execution time diminished just from 4.05 sec to 2.89 sec. This is interesting because in this case the

check set is not heavily pruned. Indeed our changes allows detecting the checks more efficiently than before.

In summary, the practical contribution of the work presented in this section is that now the tool can be efficiently used on standard desktop machines and its performance is more than adequate for agile prototyping.

4 Executable Narrations Optimization

The executable narration format describes in operational terms the actions and the checks to be performed by every agent. However, our goal was to generate efficient code. The idea is to use some compiler optimization techniques to speed up the execution time. In addition, we found useful, at this level, to do some analysis and refinement to make the generated code more compact and readable. We present here the strategies we implemented and report on some experimental results. It should be underlined that these optimizations have a significant impact mainly on large protocols, because in simpler protocols the number of checks is rather limited, so there is not much room for optimization. We run 10 consecutive times the protocols in Java and computed the average values. The reader must be aware that especially in the case of memory usage at compile time (Table 7), the data are just indicative, because the Haskell Profiler does the measurement of the "maximum residency" by sampling. Anyway, we found these values useful to understand the order of magnitude of the benefit of the different optimization techniques.

Common Subexpression Elimination (CSE) In general, during the protocol execution, operations like encryption, decryption, hashing can be performed on the same data more then once. This phenomenon affects in particular the verification of the checks because the recipient, intuitively, has to test all the possible ways to verify how his knowledge and expectations about the received messages match with the actual ones. This also implies that, on different equality checks, the same subexpression can often appear more than once. Here we assume that two syntactically equivalent expressions in the AnB model are intended to compute exactly the same value. Therefore, the elimination does not have side effects. Instead, if the repeating encryption operations rely on some randomization, this should be made explicit by the designer in the protocol specification, for example using a confounder.

Since it is well known [34,35,36] that cryptographic operations are computationally expensive, it is convenient to store the results of common subexpressions in variables that can be later employed to evaluate expressions in the rest of the protocol. We make here the reasonable assumption that storing and retrieving values from a volatile memory is less expensive in time and space than performing additional cryptographic operations on the same data. In terms of performance gain, CSE allows us to cut the execution time by 24% in 3KP, 60% for SETv2 but just 3% for H530. Our tool performs CSE in two passes:

1. During the first pass (*discovery phase*) the instructions in the narration are analyzed and the candidate expressions are found. The candidate expressions must include at least one of the following CPU intensive operations: encryption, decryption, hashing, hmac, functions and operations on keys like those performed during key agreements. Clearly, at this level of abstraction, is it impossible to assess the computational complexity of symbolic non-cryptographic functions, but we preferred to include them, considering their presence in the abstract model, as an signal of their relevance. Moreover, to improve code readability we also considered the projection operations. If, during the protocol execution, an agent computes the candidate expression more than once, a new variable is created and the corresponding expression is assigned to it. The expression is then substituted by the variable in all its occurrences. The mapping of variables and expression is stored.
2. In the second and final pass (*reordering phase*), the variable assignments are analyzed and reordered to further decrease the number of cryptographic operations avoiding recomputing more than once the same values. This pass is needed because during the first phase the statements are analyzed in sequential order, therefore here we discover dependencies among variables, and thus anticipate the assignment of variables which are subexpression of other variables.

For example, this sequence of statements performed by the customer in 3KP

$VarC_1 := dec_{inv(pk(C))}(\pi_4(dec_{inv(pk(C)}(VarC_0)))$

$VarC_2 := dec_{inv(pk(C))}(VarC_0)$

can be reordered as follows, saving one decryption operation

$VarC_2 := dec_{inv(pk(C))}(VarC_0)$
$VarC_1 := dec_{inv(pk(C))}(\pi_4(VarC_2))$

Although we do not have yet a formal proof of soundness, we believe that for CSE it could be based on the properties of the semantics of AnB, which is defined in terms of the AVISPA Intermediate Format IF [12]. In fact, the local state of the honest agents during the protocol execution is denoted by facts $\mathtt{state}_{\mathcal{A}}$(A,$m_1$,...,$m_n$), where $\mathcal{A}$ is the role of the agent, A is the name of the agent, and $m_1, ..., m_n$ is an unordered lists of terms, known by the agent, that grows monotonically. Being the list unordered and persistent, nothing prevents, in a safe protocol, the honest agents to build and store in advance the terms they need to be compute during the protocol execution. Therefore, changing the order in which terms are computed (and dropping the duplicated ones) should not introduce new attacks, as long as the agent's system is not compromised.

"Prudent" vs "Standard" optimization An additional option offered by the reordering in CSE is the possibility of pruning at compile time the check set. In fact, provided the success of a predecessor in a sequence of checks, it is safe to substitute the occurrence of the expression with a variable in all subsequent checks of the same step of the protocol, dropping the duplicated checks. This sound appealing but in any case the user must be aware that if, for any reason, the previous check is skipped, then the overall safety of the application can be compromised. To some extend this is comparable to what happened in the "*goto fail*" bug of the Apple TLS implementation [6]. In fact if there is no assurance that a previous check has been performed correctly, there is no guarantee that some meaningful checks has not been dropped by mistake. Therefore, we found useful to devise two approaches letting the user to choose between two kinds of "optimized" modes for building checks.

1. "*prudent*": does not assume the success of any previous check. This implementation is slower but more robust, since every check is independent from any other checks. In addition, the order of checks does not count, modulo the variable assignment and reordering done in CSE;
2. "*standard*": assumes the success of the previous checks in the form $var = expr$ that are used to prune the subsequent related checks. This an highly optimized implementation, and it is proper to speak about an ordered list of checks since here the order matters.

In the 3KP protocol we started from more 10,000 different checks in the unoptimized original algorithm [14] down to 22 for the "standard" and around 50 with the "prudent" implementation. In the case of protocol H530 we have 952 "prudent" checks vs 20 "standard" checks. Although not systematic, these numbers give an idea of the fact that for a professional programmer distilling manually and finding an optimal list of checks is not trivial. In terms of performance gain, we found for the protocol H530 a reduction of the execution time of 41%, 10% for 3KP, 11.5% for SETv2. For other protocols the average gain is between 5-10% but it is could be negligible if the set of generated checks in the two approaches are almost identical.

Pair ⇒ Tuples The solution proposed in [14] uses to represent tuple as nested pairs. While we found this useful to compute faster the checks, since both the source and the target language support concatenation and projection, we implemented a conversion function. This improves the code readability and it is widely supported, being the projection operator $\pi_i\,[\cdot]$ commonly available in many programming languages (e.g., the retrieval of the *i-th* element of an array).

Deferred Generation of Fresh Values In an executable narration, the actions associated with the generation of fresh values are placed in [14] at the beginning of the protocol, regardless the actual step in which they are used. It is hence useful to rearrange the order of these actions to defer the creation of fresh values until they are really necessary. In practice the effect is to perform each **new** action just before the action employing for the first time that fresh value. This could reduce to load of the processor, for example when a protocol session is aborted, due to a check failure or an ongoing Denial of Service attack.

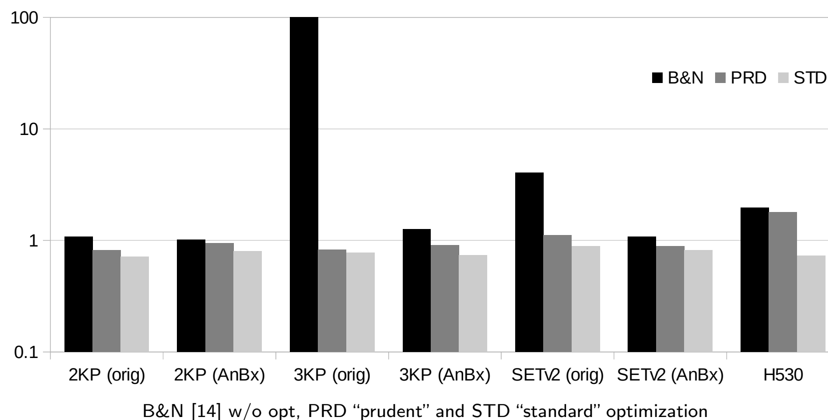

Fig. 5. Execution Time (sec), logarithmic scale

5 From Executable Narrations to Java Code

As we have seen in the previous sections we began from a specification of a security protocol in *AnBx* and through a series of translations we obtained a more detailed, but still abstract, representation of the protocol, called *optimized executable narration*. This intermediate protocol scheme that formalizes and refines the original specification, is an important step because it makes explicit the actions and the checks upon reception that each agent has to perform during the protocol execution. The way such actions and checks are written is suitable to be translated in an imperative programming language because they can be expressed by means of constructs that are generally available in such languages: variable assignments, functions and procedure calls, logical conditions such as equalities.

It should be clear that before to automatically generate the source code of an application, we need to fill several implementation details that are not caught by the formal model. If we consider the *optimized executable narration* as a complete, language independent, abstract representation of the protocol, we can devise a design approach that keeps apart the *protocol logic* from the *application logic*.

Informally we can say that, given a target programming language, the application logic is the part of code that is added to the protocol logic in order to generate the concrete and complete application. Separating the two logics (Figure 2) has the potential advantage of allowing the generation of source code in different programming languages. Although here we are focusing on Java we think that, applying the same strategy, the generation of code in other object-oriented or procedural typed languages might be done with a limited effort. The main idea is to plug the protocol logic in the application skeleton/template that is independent from the protocol itself.

To implement such approach, from the practical point of view, we need at least the following ingredients:

- a parametrized application template – the *application logic;*
- the information derived from the protocol – the *protocol logic* – employed in the preparation of the code generation. We denote such intermediate format as *JProtocol*;
- a software library accessible through an application programming interface.

Parametrized Application Template - Application Logic This component, written in the target language, is the skeleton of the application. The functions implemented by this template can include, for example, the parsing of command line arguments, the initialization of the agents, the configuration of the communication channels and the software components offering access to the cryptographic resources. These functionalities are typically used by every security application regardless the concrete protocol and hence they are suitable to be included in the application template. Clearly the concrete code must be parametrized according to the protocol specification, hence the protocol steps and relative actions will depend on the specific protocol. Additionally we can consider the fact that every agent must open at least one communication channel, but the exact number of channels depends on the protocol logic.

JProtocol - Protocol Logic From the optimized protocol narration we must derive the information that will be used to instantiate the application template, and hence a translation process must be defined. We found useful to introduce here another intermediate step because it makes simpler to extract from the protocol narration the information (i.e., the parameters) that will be used to instantiate the application template during the code generation. This format is still independent from the target language, and this comes out to be handy for the generation of the code in the target language.

The information extracted includes the parameters needed to instantiate each agent: the channels used, the local variables and the local procedures. It is worth noting that almost all the information can be directly derived from the optimized protocol narration, but, in a few cases, it might be necessary, or just more convenient, to retrieve the information from the *AnB* specification.

The intermediate protocol format that we call *JProtocol* is a 8-tuple including the following elements:

$$(name, roles, steps, channels, fields, methods, shares, actions)$$

- The *protocol name.* It is used thoroughly in the code generation. First of all it gives the name to the application. It is retrieved from the *AnB* specification, since the executable narration do not carry this data;
- The *roles* (agents) running the application. The role names are used by the agents to refer to their peers in the narration. They will also be bound to the concrete agent names/aliases by means of a mapping defined in the *application configuration file* (see 5.3). Since the set of roles is built analyzing the actions, agent names included in the agent knowledge, but not involved in the protocol, are ignored;
- The protocol *steps.* To improve the readability of the generated code, instead of writing the actions performed by each agent just as a sequence of statements, it is more convenient to group them according to the protocol steps to which they belong to. In this way each step will include only one communication action (send or receive), and zero or more actions of assignment or check type. The idea is to have a main procedure calling to a subroutine parametrized on the protocol step. This allows generating more meaningful logs and makes easier to debug the application. The steps are retrieved analyzing the protocol actions;
- The communication *channels* employed during the protocol execution. Each pair of communicating agents needs a channel, which, in the standard network environment, will be implemented as a TCP/IP socket where one agent will act as client and the other as server. As a design choice, the one who starts the communication will be the client, the other the server. If an agent exchanges messages with more than one agent, he will need to open a different channel with each peer. Again, the channel parameters are inferred from the protocol actions;
- For each role, the local variables, procedures and functions must be derived. In an object oriented terminology, they are the class *fields* and *methods* that are declared by each *role* class. These element will be used in statements that will implement the protocol actions in the target language;
- The *shares* are used to model the knowledge which is shared among agents prior the protocol execution (for instance a pre-shared symmetric key). These elements are derived from the *AnBx* specification;
- The *actions* performed by each agent during the protocol run. There are two issues to consider in the translation from the optimized executable narrations to the intermediate format: the translation of *actions*, and the translation of *expressions* used in such actions.

How local methods and field are retrieved and how actions are translated is explained in the next section.

Software Library and its Application Programming Interface Instead of fully generating the source code of every function used by the application, it is convenient to isolate a set of software components which can be reused among different applications. This approach is feasible because the class of programs we are considering performs actions that can share a common set of standard functions. Think about sending and receiving messages, encrypting and decrypting data, and so on. Therefore, it is advantageous to group and standardize these common functions and data structures, in what is usually called a *software library.* The access to the resources of the library is, as customary, defined by means of an application programming interface (API) (Section 6). This design strategy has several clear advantages:

- it relieves the application from dealing with a lot of implementation details;

– it makes considerably simpler the specification of the application templates;
– it makes more modular and cleaner the shape to the application;
– the maintenance of the code is simpler and more efficient: every program can benefit from the improvements and bug fixes in the library;
– it gives the chance to use alternative implementations of the library, provided the same standard API.

5.1 From Optimized Executable Narration to *JProtocol*

As we have seen, the intermediate representation of the protocol is a 8-tuple:

$$(name, roles, steps, channels, fields, methods, shares, actions)$$

In the previous section we shortly explained how the first four components are retrieved from the optimized executable narration. The procedure is simple and does not deserve any further explanation. Instead the last components require some care:

Local Procedures/Methods The set of local (private to each role) *methods* is built analyzing the *declarations* of the optimized protocol narration. If a function is part of the agent knowledge, excluding functions belonging to the library, such function must be available to the agent. In practice, in an object oriented language, a private method must be implemented in the role class. However, since in the protocol narrations the functions are abstract, only the skeleton of the method can be generated and a dummy value is returned. In this way, the generated code can be compiled and the application will be runnable, but the duty of filling the skeleton with a concrete implementation of the function is left to the user. This is the only "to be completed" portion of code produced by the code generator.

Local Variables/Fields The definition of the local variables, or class *fields*, is obtained by two components:

1. The first one derives from the declarations, and includes, for each agent, the known and the generated names. For each name a variable is declared, excluding the role names and the function names. For example, given this portion of the knowledge of agent `M`:

   ```
   M knows hash
   M knows hmac
   M knows C
   M knows price
   M generates tid
   ```

   the translation will include the declaration of two variables `price` and `tid`. The remaining names are ignored because they are either role names (`C`) or functions included in the library (`hash` and `hmac`). Along with the variable identifier, the type, as inferred by the type system (see 5.2), is stored.
2. The second component is built gathering information from the action list. For each assignment or reception action a variable will be declared. For example, these statements

   ```
   M: ?VAR_M_0
   M: VAR_M_1 := dec(VAR_M_0,priv(pkM))
   ```

   will imply two variables declarations: `VAR_M_0` and `VAR_M_1`. They will, respectively, store the value received by agent `M` and the result on an expression, namely the computation of the decryption of the first variable with the private key of agent `M`. Again, the type of the variable is evaluated by the type system.

Mapping of Expressions The last component of *JProtocol*, the mapping of the actions, requires the translation of *expressions* (Table 1) to a format that we call *jexpressions* (Table 8). In many cases the translation is straightforward, but when the expression includes functions like encryption and decryption some care is needed. The operators *encS*, *enc* and *dec* are commonly used to model the symmetric and asymmetric encryption. Moreover, in the latter we distinguish, as it happens in practice, between the digital signature operation and the standard encryption applied to achieve secrecy. The discrimination is done by means of the following *public keys conventions*: key identifier = prefix *sk* + agent name, for

$E, F ::=$	$encrypt(E, F)$	*asymmetr encryption*
$\mid$	$encryptS(E, F)$	*symmetric encryption*
$\mid$	$decrypt(E, F)$	*decryption*
$\mid$	$sign(E, F)$	*signature*
$\mid$	$verify(E, F)$	*verify*
$\mid$	$hmac(E, F)$	*hmac*
$\mid$	$hash(E)$	*hash*
$\mid$	$DHPubKey(E, F)$	*DH public key*
$\mid$	$DHSecKey(E, F)$	*DH secret key*
$\mid$	$(E_1, ..E_n)$	*tuple*
$\mid$	$\pi_i(E)$	*projection*
$\mid$	$f(E)$	*function*
$\mid$	$var(x, E)$	*variable binding*
$\mid$	id	*identifier*

Table 8. Syntax of *jexpressions*

the signature and key identifier = prefix *pk* + agent name, for the encryption. Key patterns not falling in these two classes are treated as a symmetric encryption. The type checker is in charge of verifying if the key type is coherent with the structure of the expression.

Here some examples showing how expressions are translated to *jexpressions*:

```
dec(X,priv(pkM)) => decrypt(X,priv(pk(M))
dec(X,pub(skM)) => verify(X,pub(sk(M))
enc(X,pub(pkM)) => encrypt(X,pub(sk(M))
enc(X,priv(skM)) => sign(X,priv(sk(M))
dec(X,K) => decrypt(X,K)
encS(X,K) => encryptS(X,K)
```

The translation of identifiers (names) and agent names maps each identifier to a pair *(identifier, type).* Since the *executable narration* is untyped, we rely on the declarations statements included in the *AnBx* specification. No ambiguity arises for types `Agent`, `Certified`, and `SeqNumber`. Identifiers of type `PublicKey` are conventionally mapped to public keys for encryption. Instead types like `Number` and `Symmetric_key` are too generic to be employed directly. For example, nonces and key agreement parameters including half-keys are all declared in *AnBx* as `Number` but in the target language they have, in general, a different type.

To overcome any possible ambiguity, the type is inferred by means of a naming convention (Table 9), provided that these identifiers are declared as `Number` or `Symmetric_key`. The same name convention is used to denote identifier added in the compilation from *AnBx* to *AnB*, hence the original type is preserved.

Although in some cases the type could be inferred by the message structure, we believe that forcing the designer to use the appropriate type would avoid accidental mistakes which could otherwise be discovered only at run-time. However, it should be noted that a designer, using only the *AnBx* specification language, can simply ignore these details because, in this case, the compiler is fully in charge of generating the concrete cryptographic implementation.

prefix	*type*
`Nx`	nonces
`Kx`	symmetric keys
`Xx,Yx`	Diffie-Hellman parameters and half keys
`Hx`	hmac keys
`SQNx`	sequence numbers

Table 9. Naming convention for the *AnB* identifiers of type `Number` and `Symmetric_key`

$$
\begin{array}{lll}
I ::= & i, A : \textbf{new}\ x : \mathbb{T} & \textit{fresh name generation} \\
\mid & i, A : \textit{send}(B, ch, E) & \textit{message emission} \\
\mid & i, A : \textit{receive}(B, ch, x : \mathbb{T}) & \textit{message reception} \\
\mid & i, A : x : \mathbb{T} := E & \textit{assignment} \\
\mid & i, A : \phi & \textit{check}
\end{array}
$$

where A,B agents, x:$\mathbb{T}$ variable of type $\mathbb{T}$, E expression, ch channel, i step, ϕ check

Table 10. Syntax of *jaction*

Mapping of Actions Having translated the expressions is now possible to translate the actions to the format we call *jaction* (Table 10). First of all, every action is labeled with the protocol step. As we mentioned earlier each step includes one send or one receive action, and zero or more actions of type assignment or check. Therefore the protocol step number is not unique but it is used to group consecutive actions in the generated code. Moreover, emission and reception actions are enriched with the reference to the communication channel used by them. This is necessary because in the concrete code the communications actions must be bound to a specific communication channel. The *steps* and the *channels* lists are available from the previous steps. Finally, the type of the variables created in actions – reception, assignment and fresh name generation – is inferred by the type system and stored along with the variable identifier.

5.2 The Type System

AnB is a typed language, but its set of types is too coarse compared to what is necessary in practice to encode a security protocol in Java. Some types like `Symmetric_key` can be useful to map the concrete types, other types like `Number` are sufficient to describe an abstract model suitable for verification but extremely vague in terms of a concrete implementation. For instance, while `Number` is used in *AnB* to model both nonces and key agreement half-keys, in Java nonces may be instances of the class `ByteArray,` while public half keys may be instances of the class `PublicKey`. As mentioned earlier, few naming convention are used (Table 9) to mark different types of key material and nonces. Moreover, we assume that the functions `pk` and `sk` are used in a dual key asymmetric cryptosystem were `pk` is used for encryption and `sk` is used for digital signature. These conventions are aimed at avoiding ambiguity which could arise even using type inference. In this way the type checker is able to detect errors like using `pk` instead of `sk`, or using the wrong type of keys. Given these assumptions we delegate the type system to infer all the other types. This reduces the need of type annotations and if type-checking of the specification succeeds it guarantees that the Java code generated is well-typed.

However, name conventions in general are not sufficient to handle complex terms. Let's look at the encryption and decryption operations. Agents should be able to encrypt any term[1]; in turn, decryption should output a term of the original type, the type of the data before the encryption. For example, in Java we could have a cryptographic engine exposing the following methods for symmetric encryption and decryption:

```
public SealedObject encryptS(Object obj, Key symmetricKey) {...}
public Object decrypt(SealedObject so, Key symmetricKey) {...}
```

The first methods encrypts any `Object` with an appropriate `Key` and outputs an encrypted object of type `SealedObject`. A snippet of code using such methods is:

```
private SealedObject SO = null;
private String Msg = new String ("msg");
private Key myKey = ... // the key is retrieved from the key store
SO = encryptS(Msg, myKey);
ch.send(SO); // the data is send to the network
```

It should be noted that the call to `encrypt()` does not require an explicit cast of `Msg.` In fact `String`, as any other type in Java, is a subtype of `Object`. On the contrary, the `decrypt()` method outputs a

[1] In our case we consider Java classes that implement the serializable interface.

value of type `Object`. Therefore in order to assign that value to a variable of type `String` an explicit cast is required. Otherwise the program will not compile.

```
private SealedObject SO = null;
SO = ch.receive(); // a SealedObject is received from the network
private String Msg;
private Key myKey = ... // the key is retrieved from the key store
Msg = (String) decrypt(SO, myKey);
```

However this does not guarantee the absence of run-time errors. For example, in the above code, we could have a run-time error (and the raise of cast exception) if the decrypted object is not of the expected type (`String`) or a subtype of it. What we need, in general, is the ability to infer the type of terms. This is used to declare new variables or compute the appropriate cast in assignments or within expressions. Such task can be accomplished by the type system and its type inference algorithm.

Types in Table 11 are those typically used in a wide range of security applications. It is worth noting that these types are still abstract, in the sense that their mapping to concrete types in the target language is postponed until the actual code is generated.

Some types (*Agent*,*Certified*,*SeqNumber*,*SymmetricKey*) simply map the *AnB* types. *PublicKey* and *PrivateKey* are parametric types where the parameter is used to distinguish keys with different purpose (signing and encryption). Three more parametric types are used for encrypted (sealed) and signed objects. In this case the parameter is needed to keep track of the original type of the object before the cryptographic operation. The parameter is employed when the inverse operation is performed.

In details: *SealedObject* is an encrypted object, *SealedPair* is used when a key is encrypted along with data as in the hybrid cryptography, *SignedObject* is a digitally signed object. *Hash* and *Hmac* are used to model digests. Keys, used to compute the HMACs, should have type *HmacKey*. A set of DH-types is used for values employed in key agreements. Tuples and functions are standard features of the language and they have their type counterparts here. Finally *Object* and *String* are the base types. *Object* can be thought as the default type, while *String* is a type commonly available in many programming languages, and it is useful in the generated code to produce human readable output.

The type inference algorithm is based on the typing rules shown in Table 12. The T-HASH and T-HMAC rules model the creation of digests. The original type is obfuscated and cannot be retrieved since hashing and macs are not invertible. T-FUN models functions, T-CAT the concatenation and T-PROJ the projection. T-ENC-* rules model encryption. It should be noted that we impose some constraints on the type of the key, depending on whether asymmetric or symmetric encryption is used. The resulting type is parametrized on T_1, the type of the original object before encryption. Symmetrically the T-DEC-* rules model the decryption. Here the parameter is used to determine the type of the object after the decryption. Similarly, some checks are done on the type of keys employed for decryption.

T-SIGN and T-VERIFY behave like T-ENC-* and T-DEC-*, but they are used to model the digital signature and its verification. The last three rules T-KAP, T-KAS-1 and T-KAS-2 are used to set the typing rules of the operations performed during the key agreements. Two T-KAS-* rules are provided because they model two ways the agents have to compute their shared secret keys, at the end of the key agreement protocol.

5.3 Application Template

The last component of our toolbox is the application template. This set of files represents the skeleton of the Java application, and we refer to it as the *application logic*. The template is filled with the *protocol logic*, the data synthesized from the specification of the executable narration, and stored in the *JProtocol* data structure. The template files must be written in the target language, Java in our case.

As a running example, we show portions of code taken from the revised *2KP* (protocol name: `Rev_2KP`). Additionally, to help the reader to understand the program structure, the UML class diagram of the resulting application is shown in Figure 6.

In general, assuming that the protocol name is `ProtName`, the full application is composed of the following Java file (classes):

`ProtName.java` The main file of the application defines the `ProtName` class, which implements only the method `main()`. This method is invoked when the application is started: the command line parameters

Fig. 6. UML class diagram of the revised *2KP* in Java

$T ::=$	$Agent$	*agent*
$\mid$	$Certified$	*certified agent*
$\mid$	$Nonce$	*nonce*
$\mid$	$SeqNumber$	*sequence number*
$\mid$	$SymmeticKey$	*symmetic encryption key*
$\mid$	$PublicKey\,\langle S \rangle$	*public key of type S*
$\mid$	$PrivateKey\,\langle S \rangle$	*private key of type S*
$\mid$	$SealedObject\,\langle T \rangle$	*sealed object of type T*
$\mid$	$SealedPair\,\langle T \rangle$	*sealed object of type T + key*
$\mid$	$SignedObject\,\langle T \rangle$	*signed object of type T*
$\mid$	$Hash$	*hash*
$\mid$	$HmacKey$	*hmac encryption key*
$\mid$	$Hmac$	*hmac*
$\mid$	$DHBase$	*DH parameter spec*
$\mid$	$DHKeyPair$	*DH key pair*
$\mid$	$DHPubKey$	*DH public key*
$\mid$	$DHSecKey$	*DH secret key*
$\mid$	$\left\{T_i{}^{i\in\{1..n\}}\right\}$	*tuple*
$\mid$	$T \rightarrow T$	*function*
$\mid$	$String$	*base type*
$\mid$	$Object$	*base type*
$S ::=$	PK	*key pair type for encryption*
$\mid$	SK	*key pair type for signing*

Table 11. Type system - Types

are passed, by means of the `Parse()` method, to the final class `ProtName_CommandLine_Parser`. The command line parameters `args` must include, along with other settings, the agent role that this instance of `ProtName` is playing during the protocol execution. For each agent an instance of `ProtName` must be created.

```
public class Rev_2KP {
 public static void main(String[] args) {
  Rev_2KP_CommandLine_Parser.Parse(args,Rev_2KP.class.toString());
  }
}
```

`ProtName_CommandLine_Parser.java` The purpose of the `ProtName_CommandLine_Parser` class is to initialize the application. The invocation of method `Parse()` causes the processing of the configuration file (details about its content are given below). The command line arguments are used to set up the parameters of the application. In detail, the main actions performed by this class are:

1. the mapping between the roles in the protocol and the identities of principals playing those roles. This is done by the `initRole` method

   ```
   Channel_SSLChannelType ct = Channel_SSLChannelType.SSL_PLAIN;
   Map<String,Channel_Settings> cs = new HashMap <String,Channel_Settings>();
   Map<String,String> aliases = new HashMap <String,String>();
   Rev_2KP_Principal Rev_2KP_pr = null;
   initRole(ct,role,cs,aliases);
   ```

   Here `ct` is SSL/TLS channel type we are running the application on top of it. The default value is the plain channel.
2. reading the configuration file and initializing the cryptographic engine

$$\text{T-HASH}\ \frac{t : T}{hash(t) : Hash}$$

$$\text{T-HMAC}\ \frac{t_1 : T_1 \quad t_2 : HmacKey}{hmac(t_1, t_2) : Hmac}$$

$$\text{T-FUN}\ \frac{f : T_1 \rightarrow T_2 \quad t : T_1}{f(t) : T_2}$$

$$\text{T-CAT}\ \frac{for\,each\,i \quad t_i : T_i \quad i \in \{1..n\}}{\{t_i{}^{i \in \{1..n\}}\} : \{T_i{}^{i \in \{1..n\}}\}}$$

$$\text{T-PROJ}\ \frac{t_1 : \left\{T_i{}^{i \in \{1..n\}}\right\}}{t_1.{}_j : T_j}$$

$$\text{T-ENC-ASYM}\ \frac{t_1 : T_1 \quad t2 : PublicKey\,\langle PK\rangle}{enc(t_1, t_2) : SealedPair\,\langle T_1\rangle}$$

$$\text{T-ENC-SYM}\ \frac{t_1 : T_1 \quad t2 : T_2}{encS(t_1, t_2) : SealedObject\,\langle T_1\rangle} \qquad T_2 \in \{SymmetricKey, DHSecKey\}$$

$$\text{T-DEC-ASYM}\ \frac{t_1 : SealedPair\,\langle T_1\rangle \quad t2 : PrivateKey\,\langle PK\rangle}{dec(t_1, t_2) : T_1}$$

$$\text{T-DEC-SYM}\ \frac{t_1 : SealedObject\,\langle T_1\rangle \quad t2 : T_2}{dec(t_1, t_2) : T_1} \qquad T_2 \in \{SymmetricKey, DHSecKey\}$$

$$\text{T-SIGN}\ \frac{t_1 : T_1 \quad t2 : PrivateKey\,\langle SK\rangle}{sign(t_1, t_2) : SignedObject\,\langle T_1\rangle}$$

$$\text{T-VERIFY}\ \frac{t_1 : SignedObject\,\langle T_1\rangle \quad t2 : PublicKey\,\langle SK\rangle}{verify(t_1, t_2) : T_1}$$

$$\text{T-KAP}\ \frac{t_1 : DHBase \quad t2 : DHKeyPair}{kap(t_1, t_2) : DHPubKey}$$

$$\text{T-KAS-1}\ \frac{t_1 : DHPubKey \quad t2 : DHKeyPair}{kas(t_1, t_2) : DHSecKey}$$

$$\text{T-KAS-2}\ \frac{t_1 : DHKeyPair \quad t2 : DHPubKey}{kas(t_1, t_2) : DHSecKey}$$

Table 12. Type system - Typing rules (the environment is omitted)

```java
private static void initRole(Channel_SSLChannelType ct,Rev_2KP_Roles role,
    Map<String,Channel_Settings> cs,Map<String, String> aliases) {
[...]
Properties configFile = new Properties();
AnBx_Debug.out(layer,"Reading config file: "+configFileName.toString());
InputStream propertiesStream = Rev_2KP_CommandLine_Parser.class.
    getResourceAsStream(configFileName);
if (propertiesStream != null) {
        try
        {
                configFile.load(propertiesStream);
                crypto_config = new Crypto_Config (configFile);
        } catch (IOException e)
        {
                AnBx_Debug.out(layer,"Error reading config file: "+
                    configFileName.toString());
                e.printStackTrace();
        }
[...]
}
```

3. reading, from the configuration file, the key path of the keystore, where keys and certificates can be retrieved, and the location where pre-shared information can be found. Additionally, the alias of the agent is set; this is necessary for self identification, for example to retrieve the private keys

```java
private static void initRole(Channel_SSLChannelType ct,Revised_2KP_Roles
    role,Map<String,Channel_Settings> cs,Map<String, String> aliases) {
[...]
        myAlias = configFile.getProperty(role.toString());
        keypath = configFile.getProperty("keypath");
        sharepath = configFile.getProperty("sharepath");
[...]
}
```

4. the creation and the initialization of the communication channels. Parameters like the hostname, the port, and the role played in the channel (client or server) are retrieved.

```java
private static void initRole(Channel_SSLChannelType ct, Revised_2KP_Roles
    role, Map<String, Channel_Settings> cs, Map<String, String> aliases) {
[...]
for (Rev_2KP_Roles peer : Rev_2KP_Roles.values()) {
 if (configFile.getProperty(peer.toString()) != null) {
    aliases.put(peer.toString(), configFile.getProperty(peer.toString()));
  }
  if (!peer.equals(role)) {
    ch = channelName(role,peer);
    String host = configFile.getProperty(ch + HOST_SUFFIX);
    if (host != null) {
      Integer port = Integer.parseInt(configFile.getProperty(ch +
          PORT_SUFFIX));
      if (configFile.getProperty(ch + ROLE_SUFFIX).equalsIgnoreCase("Client"
          )) {
        cs.put(peer.toString(),new Channel_Settings(ct, Channel_Roles.CLIENT
            , host, port));
      } else {
        cs.put(peer.toString(),new Channel_Settings(ct, Channel_Roles.SERVER
            , host, port));
      }
    }
  }
}
```

5. the creation of an object of class `ProtName_Principal`, according to the role the agent is playing, and the invocation of its method `run()`.

```
Rev_2KP_Principal Rev_2KP_pr = null;
Properties configFile = new Properties();
initRole(ct, role, cs, configFile, aliases);
if (myAlias != null && keypath != null) {
 Rev_2KP_pr = new Rev_2KP_Principal(myAlias, keypath, cs, aliases,
     crypto_config);
    switch (role) {
     case ROLE_C:
          Rev_2KP_pr.run(new Rev_2KP_ROLE_C(role, protname, sharepath));
          break;
     case ROLE_M:
          Rev_2KP_pr.run(new Rev_2KP_ROLE_M(role, protname, sharepath));
          break;
     case ROLE_a:
          Rev_2KP_pr.run(new Rev_2KP_ROLE_a(role, protname, sharepath));
          break;
        }
        } else {
                AnBx_Debug.out(layer, "Unable to initialize Rev_2KP
                    Principal");
                }
        }
```

`ProtName_Principal.java` This class extends the library class `AnB_Principal` and holds the agent knowledge about channels and cryptographic material (keys, certificates, aliases) and their parameters. Once the principal is initialized a call to the superclass method `run()` executes the sequence of steps the agent has to perform.

```
class Rev_2KP_Principal extends AnB_Principal {
  public Rev_2KP_Principal(String myAlias, String path, Map<String,
      Channel_Settings> cs, Map<String, String> aliases, Crypto_Config config) {
    super(myAlias, path, cs, aliases, config);
  }
}
```

`ProtName_Role_<X>.java` This class which extends the library class `AnB_Protocol` (Figure 12) is the core of the application. One file is generated for each role, where `<X>` is replaced with the role name. The method `run()` initialize the communication channels and executes the steps of the protocol. The two parameters of `run()` are the mapping of the communication channels and the mapping of role/aliases. We show the portion of code of the agent playing the Merchant role (`ROLE_M`) who directly communicate with two other agents, the customer (`ROLE_C`) and the acquirer (`ROLE_a`). Therefore the Merchant needs to open two channels. Here the exceptions are also caught. In particular the `ClassCastException` is raised if the incoming messages does not belong to the expected class, i.e., they do not have the expected structure.

```
public void run(Map<String, AnB_Session> lbs, Map<String, String> aliases) {
  this.aliases = aliases;
  AnB_Session ROLE_M_channel_ROLE_C = lbs.get("ROLE_C");
  AnB_Session ROLE_M_channel_ROLE_a = lbs.get("ROLE_a");
    try {
          init();
      ROLE_M_channel_ROLE_C.Open();
      ROLE_M_channel_ROLE_a.Open();
      do {
        executeStep(ROLE_M_channel_ROLE_C, Revised_2KP_Steps.STEP_0);
        [...]
```

```
            executeStep(ROLE_M_channel_ROLE_C, Revised_2KP_Steps.STEP_7);
             } while (loop);
          ROLE_M_channel_ROLE_C.Close();
          ROLE_M_channel_ROLE_a.Close();
              } catch (ClassCastException e) {
                    AnBx_Debug.out(layer, "Message format type error - Program will
                         be terminated!");
            [...]
                    }
            } catch (Exception e) {
                    AnBx_Debug.out(layer, "Generic error - Program will be
                        terminated!");
            [...]
                    }
}
```

The method `executeStep()` executes the actions specified by each steps (sending, receiving, variable assignments, checks on reception). The two parameters are the session `s` and the protocol `step` to be executed. The `AnB_Session` (Figure 11) is a library class giving access both to the communication channel and to the cryptographic functions. Several communication and cryptographic operations are performed in these two protocol steps, including the checks on reception.

```
private void executeStep(AnB_Session s, Rev_2KP_Steps step) {
  status(step);
  switch (step) {
  case STEP_0:
  // C -> M,(-|-|M): [can(C):a],[Desc:M]
  // C -> M: {hmac(can(C),H3a),{H3a}pk(a),hmac(Desc,H3M),{H3M}pk(M)}pk(M)
        VAR_M_R0 = (Crypto_SealedPair) s.Receive();
        VAR_M_DMR0VPM = (AnBx_Params) s.decrypt(VAR_M_R0);
        VAR_M_DJ4MDMR0VPMVPM = (SecretKey) s.decrypt((Crypto_SealedPair)
            VAR_M_DMR0VPM.getValue(3));
        VAR_M_MDESCDJ4DMR0VPMVPM = s.makeHmac(Desc,(SecretKey)
            VAR_M_DJ4MDMR0VPMVPM);
        eqCheckL("0.1",(Crypto_ByteArray) VAR_M_DMR0VPM.getValue(2),
            VAR_M_MDESCDJ4DMR0VPMVPM);
        invCheckL("0.2",(Crypto_SealedPair) VAR_M_DMR0VPM.getValue(1),
            Crypto_SealedPair.class);
        VAR_M_J1MDMR0VPM = (Crypto_ByteArray) VAR_M_DMR0VPM.getValue(0);
    break;
  case STEP_1:
    //  M -> C,(@M|C|-): TID,[Price,TID,[Desc:M],[can(C):a]]
    //  M -> C: {C,SQN4,TID,hash(Price,TID,hmac(Desc,H3M),hmac(can(C),H3a))}inv
        (sk(M))
        TID = new String("TID"+"_"+sessionID);
        Nx4 = s.getSeqNumber();
        VAR_M_HPRICETIDMJ1MDMR0VPMMMDESCDJ4DMR0VPMVPM = s.makeDigest(new
            AnBx_Params(Price,TID,VAR_M_J1MDMR0VPM,VAR_M_MDESCDJ4DMR0VPMVPM));
        s.Send(s.sign(new AnBx_Params(aliases.get("ROLE_C"),Nx4,TID,
            VAR_M_HPRICETIDMJ1MDMR0VPMMMDESCDJ4DMR0VPMVPM)));
    TID = s.getSeqNumber();
    SQN4 = s.getSeqNumber();
    s.Send(s.sign(new AnBx_Params(aliases.get("ROLE_C"),SQN4,TID,s.makeDigest(
        new AnBx_Params(Price,TID,VAR_M_MDESCDJI4DM0VPMVPM,(Crypto_HmacPair)
        VAR_M_DM0VPM.getValue(0))))));
    break;
  case STEP_2:
        [...]
   }
  status(step);
}
```

`ProtName_Setup.java` This class is a support class which generates the data shared among agents before the protocol execution. It uses serialization to write the objects in the location specified by the entry `sharepath` in the application properties file.

```
[...]
// create shared knowledge objects
AnB_Crypto_Wrapper.writeObject(new String("Price"),sharepath+"Price.ser");
AnB_Crypto_Wrapper.writeObject(new String("Desc"),sharepath+"Desc.ser");
[...]
```

`ProtName_Roles.java` This enumeration class contains the list of *roles* (agents) participating in the protocol.

```
public enum Revised_2KP_Roles {
  ROLE_C, ROLE_M, ROLE_a
}
```

`ProtName_Steps.java` This enumeration class contains the list of *steps* of the protocols.

```
public enum Rev_2KP_Steps {
  STEP_0, STEP_1, STEP_2, STEP_3, STEP_4, STEP_5, STEP_6, STEP_7
}
```

Configuration File The configuration file contains a set parameters that, along with the command line arguments, are used to initialize the application. This file can be easily modified by the end user without need to regenerate the code of the application. An example of configuration file is shown in Figure 7. The parameters include the path where the keystore is located in the file system. The keystore contains the keys and certificates of known agent. Moreover, the configuration file includes the mapping between protocol roles and agent aliases, the setting of the communication channels and the parameters used to initialize the cryptographic engine.

The path of the keystore, is used by the application to retrieve the key material which is used during the execution. It is assumed that the format of the key store is compatible with the cryptographic settings of the application. In general the application is designed to use the appropriate cryptographic algorithms based on the keys type available in the keystore. In this database, keys and certificates are associated to an alias which is used as an index to access to that cryptographic objects. It is hence necessary to provide an explicit mapping between protocol roles and the alias of the agent who is actually playing that specific role.

Channel parameters are used to initialize the TCP/IP sockets and they include the network role (client or server), the port and the hostname. Note that the hostname is used only for the client channels, because servers listen on a port in their own system, and they do not need that parameter to setup the socket. The default value for the hostname is the localhost IP address (127.0.0.1) but it can be freely changed if the user needs to run the processes on different machines. In this case every machine running the protocol must have its own copy of the configuration file. For security reasons it is not advisable to share the same file among different agents. Moreover, the client port and server port must match in order to establish a communication.

Changing the cryptographic engine settings allows using different cipher schemes and parameters without the need to regenerate the application. However, it is up to the user to check weather the settings and the schemes are actually supported by their own systems.

5.4 Code Generation

The last phase of the process is the code emission. As we have seen, we do not only write the security related code (the protocol logic) but also a complete application combining the information derived from the optimized executable narration with the application template (the application logic).

As a first step we must reconcile the two logics. In practice, this is done binding the abstract view of the types and the API calls with the concrete one. The binding depends on the target programming

```
# Protocol: Rev_2KP
# Java Config File: "C:/genAnBx/src/Rev_2KP/Rev_2KP.properties"
# Roles/Share
ROLESHARE = ROLE_C
# Roles/Aliases
ROLE_C = alice
ROLE_M = bob
ROLE_a = charlie
# Channels
ROLE_C_channel_ROLE_M_role = Client
ROLE_C_channel_ROLE_M_host = 127.0.0.1
ROLE_C_channel_ROLE_M_port = 6666
ROLE_M_channel_ROLE_C_role = Server
ROLE_M_channel_ROLE_C_host = 127.0.0.1
ROLE_M_channel_ROLE_C_port = 6666
ROLE_M_channel_ROLE_a_role = Client
ROLE_M_channel_ROLE_a_host = 127.0.0.1
ROLE_M_channel_ROLE_a_port = 6669
ROLE_a_channel_ROLE_M_role = Server
ROLE_a_channel_ROLE_M_host = 127.0.0.1
ROLE_a_channel_ROLE_M_port = 6669
# Paths
keypath = C:/genAnBx/keygen_dual/
sharepath = C:/genAnBx/bin/Rev_2KP/
# Cryptographic Engine default settings
cipherScheme = AES
asymcipherSchemeBlock = RSA
keySize = 128
keyGenerationScheme = AES
secureRandomAlgorithm = SHA1PRNG
hMacAlgorithm = HmacSHA1
messageDigestAlgorithm = SHA-1
keyAgreementAlgorithm = DH
```

Fig. 7. Protocol.Properties configuration file

language and on the support library. In our case the bindings are defined by the maps shown in Table 13. The abstract API calls are mapped to the concrete Java calls implemented by our *AnBxJ* library (Section 6); the abstract Types are mapped to the concrete Java and library types.

Next, we use the *JProtocol* data and the bindings to generate syntactically correct Java statements (or portions of them) to be injected in the application template. This task is performed with the support of the HStringTemplate [37] library, the Haskell port of the StringTemplate Java library [38,39].

Figure 9 shows the template file (`ROLE_x.st`) for the ROLE_x class. Terms between the $ delimiters are the templates which are instantiated during the protocol generation. The task performed by StringTemplate is to replace these templates with the actual code. For example

```
$fields:{n|private $n.typeof$ $n.name$ = null;
}$
```

is a template used to declare the private fields of the ROLE_x class. It is filled with the type and the name of the each element taken from the fields component of *JProtocol*.

The resulting code for role *Merchant* in the revised 2KP protocol is the following:

```
private String Price = null;
private String Desc = null;
private String TID = null;
private Crypto_ByteArray Nx4 = null;
private Crypto_ByteArray Nx8 = null;
private Crypto_SealedPair VAR_M_R0 = null;
```

abstract API call	*AnBxJ/Java API calls*	*abstract Type*	*AnBxJ/Java Types*
APISend	Send	SealedPair	Crypto_SealedPair
APIReceive	Receive	SealedObject	SealedObject
APIEncrypt(S)	encrypt	SignedObject	SignedObject
APIDecrypt	decrypt	HmacPair	Crypto_HmacPair
APISign	sign	JHash	Crypto_ByteArray
APIVerify	verify	AnBx_Params	AnBx_Params
APIHash	makeDigest	JString	String
APIHmac	makeHmac	JObject	Object
APISQN	getSeqNumber	JHmacKey	SecretKey
APINonce	getNonce	JSymmetricKey	SecretKey
APISymKey	getSymmetricKey	JNonce	Crypto_ByteArray
APIHmacKey	getHmacKey	JSeqNumber	Crypto_ByteArray
APIDHPubKey	getKeyEx_PublicKey	JDHBase	DHParameterSpec
APIDHKeyPair	getKeyEx_KeyPair	JDHPubKey	PublicKey
APIDHSecKey	getKeyEx_SecretKey	JDHKeyPair	KeyPair
APIEqCheck	eqCheck	JDHSecKey	SecretKey
APInvCheck	invCheck	JVarArgs	Object ...
APIwffCheck	wffCheck		

Table 13. API and type bindings

```
private AnBx_Params VAR_M_DMR0VPM = null;
private SecretKey VAR_M_DJ4MDMR0VPMVPM = null;
private Crypto_ByteArray VAR_M_MDESCDJ4DMR0VPMVPM = null;
private Crypto_ByteArray VAR_M_J1MDMR0VPM = null;
private Crypto_ByteArray VAR_M_HPRICETIDMJ1MDMR0VPMMMDESCDJ4DMR0VPMVPM = null;
private Crypto_SealedPair VAR_M_R2 = null;
private String VAR_M_R4 = null;
private Crypto_SealedPair VAR_M_R6 = null;
private SignedObject VAR_M_DMR6VPM = null;
private AnBx_Params VAR_M_DDMR6VPMUSA = null;
private String VAR_M_J5MDDMR6VPMUSA = null;
private Crypto_ByteArray VAR_M_J6MDDMR6VPMUSA = null;
```

This example shows that StringTemplate does not provide just a simple string substitution but also makes possible to use more complex patterns like attributes with properties ($n.name$). Moreover, it allows applying an anonymous template ({n|private $n.typeof$ $n.name$ = null;}) to each element of an attribute ($fields:<anonymoustemplate>$).

Following the same approach it possible to generate even more structured code. This is how the method executeStep() is specified in the same template file ROLE_X.st

```
protected void executeStep(AnB_Session $sessname$, $prot$_Steps step) {
  status(step);
  switch (step) {
    $stepactions:{n|
      case $n.astep$:
        $n.action$
```

```
            break;
                    }$
            }
            status(step);
}
```

In this case, along with simple substitutions of the session (`$sessnane$`) and protocol name (`$prot$`), we can use an anonymous template - `{n|...}` - to generate the cases of the switch statement, namely the actions (`$n.action$`) to be performed in each step (`$n.astep$`). Here, we found more productive to directly generate a sequence of actions as a string in Haskell and pass it to the template property `$n.action$` rather than managing templates for all the kind of possible actions.

Finally, the generated configuration and application files are written to the disk. As an example of this, we show the names of files (left side the template name, right side the application name) for the revised *2KP* protocol (roles are: A acquirer, M merchant, C customer):

- `CommandLine_Parser.st -> Rev_2KP_CommandLine_Parser.java`
- `Principal.st -> Rev_2KP_Principal.java`
- `ROLE_x.st -> Rev_2KP_ROLE_A.java, Rev_2KP_ROLE_M.java, Rev_2KP_ROLE_C.java`
- `Roles.st -> Rev_2KP_Roles.java`
- `Steps.st -> Rev_2KP_Steps.java`
- `main.st -> Rev_2KP.java`
- `setup.st -> Rev_2KP_Setup.java`
- `[] -> Rev_2KP.properties`
- `[] -> build.xml`

The `build.xml` file is an Ant build file which allows compiling, running and testing the application. In particular the target `run` can be used to run in parallel all the agent's programs in order to verify that the application is executable. All the Java files must belong to the same package (`rev_2kp` in this case).

Ultimately, one of the advantages of this approach of code generation is that it is possible to change the application templates without need to adapt the tool, as long as the template interface (parameters) is maintained. On the one hand, we think this is an advantage for the user because he has the possibility to work on the application template in the target language. One the other hand, the generation of code in another procedural typed language would require a reasonable extra effort and would consist of the following tasks:

- writing new templates files
- defining the syntax of statements like variable declaration, variables initialization (constructors), imperative actions
- setting the binding between the abstract types and the API calls and the concrete one.

A prerequisite would be the availability of a security and communication library similar to the one we designed. Since all the modern programming languages offer these features, the work would consist of building a wrapper library, around the existing language features, having the same interface of our library (Figures 10 and 11).

6 API - Java Security Library (*AnBxJ*)

We developed a Java library to experiment with our approach and validate its practical effectiveness. The package provides an application programming interface (API) that implements the primitives discussed in the previous sections. To support a high degree of flexibility, the API does not commit to any specific cryptographic solution (algorithms, libraries, providers). Instead, it is structured as a modular, easily configurable framework that leaves the developer free (at compile, deployment or even at runtime) to decide which cryptographic scheme to use, according to the security, robustness and performance requirements the application must satisfy. By default for asymmetric encryption, the system uses the algorithms and the key lengths specified in the digital certificates of the public keys used for encryption and signature. For the symmetric encryption the pre-set schemes can be changed by the user editing the configuration file. This simplifies the practical use of the library.

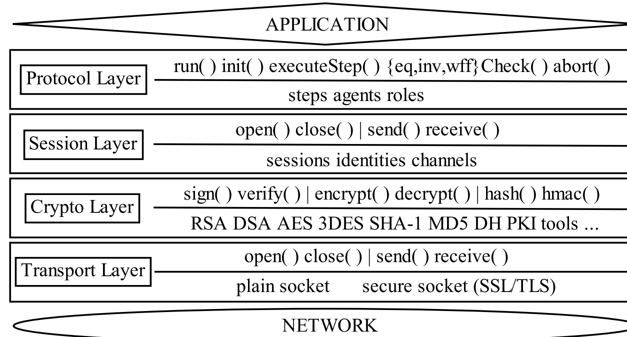

Fig. 8. *AnBxJ* Java Library Architecture

Several encryption and digital signature algorithms, hash and digest functions, and different key sizes are made available to the application by means of a standard interface. This is done by interacting with the cryptographic libraries, almost transparently, using the standard interface specified by the Java Cryptographic Architecture [22,23]. Changing the cryptographic protocols and settings is easy, because it does not modify the source code of the application, but only the configuration of the encryption engine in use. The framework is extensible making possible to add new cryptographic libraries, or replacing faulty implementation, or compromised algorithms. This approach leads to a clear design of the application, focusing on the application logic, abstracting the programmer from the complexity of the underlying network protocols and infrastructure. Communication and cryptographic run-time errors are handled at this level, and exceptions are raised.

The API is structured in the layered architecture described in Figure 8, whose main components are described as follows:

- The *transport layer* provides all the networking functionality necessary to transport messages over the network, using both plain and secure sockets (SSL/TLS). Although the enforcement of the security properties is often delegated to the cryptographic layer, is it also possible to run applications over a secured channel rather than over a plain one.
- The *cryptographic layer* essentially provides procedures to encrypt and decrypt, sign and verify, digest messages using the facilities included in libraries like `java.security` and `javax.crypto`. The public key infrastructure (PKI) binds public keys with their respective user identities by means of a certificate authority (CA). Trusted certificates are stored in keystores, and *identities* are defined associating *aliases* with a pair of public keys (one for encryption and one for digital signature).
- The *session layer* offers the functions `send()` and `receive()`, which map, respectively, the output and the input primitives. Any *serializable* object can be a *message* exchanged by means of these primitives, thus it is possible to transmit a wide range of object classes across a network connection link. Primitives to `open()` and `close()` sessions are also provided. Moreover, shielding the details of the cryptographic layer, the `AnB_session` class (Figure 11) offers methods for calling, in a simplified manner, the primitives of the cryptographic layer extending the class `AnB_Crypto_Wrapper` (Figure 10).
- The *protocol layer* gives an abstract description of the protocol: data flow and control flow, steps and principal roles, checks on reception. The main class is the abstract generic class `AnB_protocol` (Figure 12) which must be extended by each role class.

7 Related Work and Conclusions

Other Java code generators for security protocols have been proposed in the past. An early project [40] allows for automatic generation of Java code from a specification written in CASPL [41] or in its intermediate language CIL. Although standard Java cryptographic providers are used, this tool has some noticeable limitations. Since at that time a RSA implementation was not publicly available in Java, the tool does not handle public-key encryption. *AGVI* [15] is another of the earliest tools which generates the Java implementation of a security protocol. It also generate security protocols from user requirements but

when dealing with complex security protocols, may suffer from state space explosion problem. *Spi2Java* [17] is a framework to semi-automatically generate Java security protocol implementations from verified Spi Calculus formal specifications. The aim of the framework is to provide high correctness confidence on the generated code, thus making a step towards bridging the gap between the verified abstract formal models, and their concrete implementations. *JavaSPI* [18] is an evolution of Spi2Java. The main novelty of this approach stands in the use of Java as both a modeling language and an implementation language. The tool is able to generate interoperable code by implementing serialization and marshaling methods that must be implemented manually. *Expi2Java [19]*, which started as an extension of Spi2Java, takes models of security protocols written in Expi culculus (an extensible variant of Spi culculus) and translate them into interoperable Java code.

With respect to the these tools, our compiler generates Java code which includes the checks on reception. We think this is very important to build defensive implementations of security protocols and has a practical impact. However, this makes difficult to compare the compile time performance with other tools because in [15,17,42,19] the checks must be written manually. In [40] there is a notion of "receivability" which only models the ability to decrypt the received messages but does not compute other checks. In contrast to the tools requiring process calculi as input language [17,42,19], we use a more intuitive language *AnB*, making our tool suitable for a larger audience of developers. In addition, our abstract specification is the most compact. Using SPI requires long specification files [19] and type annotations, [40] requires type annotations as well. Instead, we use a simple naming convention to make the protocol specification extremely succinct and the tool delegates the duty to generate well-typed code to the type system.

Future work could take several directions. It would be important to make a formal proof of the soundness of the translation process. Our abstract models are verified with OFMC and a starting point could be to use the existing semantics [14,27] to complete the formalization of the compiler front-end. Another important extension could be the generation of interoperable implementations. We think that the most effective way is to use custom serialization methods as in [17,18], but they require manual coding. A further opportunity would be to plug the tool into an existing Integrated Development Environment (IDE), such as Eclipse [43], making the *AnBx compiler* suitable to be used and tested in a more professional environment.

Acknowledgments Part of this work was carried out while the author was a Ph.D. candidate at Università Ca' Foscari Venezia (Italy), under the valuable supervision of Prof. Michele Bugliesi. This work was partially supported by the EU FP7 Project n. 318424, "FutureID: shaping the Future of Electronic Identity" (futureid.eu). The author wishes to thank Thomas Groß for his helpful discussions and comments.

References

1. Avalle, M., Pironti, A., Sisto, R.: Formal verification of security protocol implementations: a survey. Formal Aspects of Computing **26**(1) (2014) 99–123
2. Georgiev, M., Iyengar, S., Jana, S., Anubhai, R., Boneh, D., Shmatikov, V.: The most dangerous code in the world: validating SSL certificates in non-browser software. In: Proceedings of the 2012 ACM conference on Computer and communications security, ACM (2012) 38–49
3. Langley, A.: Unfortunate current practices for HTTP over TLS (2011) http://www.ietf.org/mail-archive/web/tls/current/msg07281.html.
4. Poll, E., Schubert, A.: Verifying an implementation of SSH. In: WITS. Volume 7. (2007) 164–177
5. Cassidy, S.: Existential type crisis : Diagnosis of the OpenSSL Heartbleed Bug (2014) http://blog.existentialize.com/diagnosis-of-the-openssl-heartbleed-bug.html.
6. Ducklin, P.: Anatomy of a "goto fail" - Apple's SSL bug explained, plus an unofficial patch for OS X! (2014) http://nakedsecurity.sophos.com/2014/02/24/anatomy-of-a-goto-fail-apples-ssl-bug-explained-plus-an-unofficial-patch/.
7. Blanchet, B.: An efficient cryptographic protocol verifier based on Prolog rules. In: Computer Security Foundations Workshop, IEEE, IEEE Computer Society (2001) 0082–0082
8. Basin, D., Mödersheim, S., Viganò, L.: OFMC: A symbolic model checker for security protocols. International Journal of Information Security **4**(3) (2005) 181–208

9. Armando, A., Arsac, W., Avanesov, T., Barletta, M., Calvi, A., Cappai, A., Carbone, R., Chevalier, Y., Compagna, L., Cuéllar, J., et al.: The AVANTSSAR platform for the automated validation of trust and security of service-oriented architectures. In: Tools and Algorithms for the Construction and Analysis of Systems. Springer (2012) 267–282
10. Cremers, C.J.: The scyther tool: Verification, falsification, and analysis of security protocols. In: Computer Aided Verification, Springer (2008) 414–418
11. Abadi, M., Gordon, A.: A calculus for cryptographic protocols: The spi calculus. In: Proceedings of the 4th ACM Conference on Computer and Communications Security, ACM (1997) 36–47
12. Mödersheim, S.: Algebraic properties in Alice and Bob notation. In: International Conference on Availability, Reliability and Security (ARES 2009). (2009) 433–440
13. Bellare, M., Garay, J., Hauser, R., Herzberg, A., Krawczyk, H., Steiner, M., Tsudik, G., Van Herreweghen, E., Waidner, M.: Design, implementation, and deployment of the iKP secure electronic payment system. IEEE Journal on Selected Areas in Communications **18**(4) (2000) 611–627
14. Briais, S., Nestmann, U.: A formal semantics for protocol narrations. Theor. Comput. Sci. **389** (December 2007) 484–511
15. Song, D., Perrig, A., Phan, D.: Agvi-automatic generation, verification, and implementation of security protocols. In: Computer Aided Verification, Springer (2001) 241–245
16. Jeon, C.W., Kim, I.G., Choi, J.Y.: Automatic generation of the C# code for security protocols verified with Casper/FDR. In: Advanced Information Networking and Applications, 2005. AINA 2005. 19th International Conference on. Volume 2., IEEE (2005) 507–510
17. Pozza, D., Sisto, R., Durante, L.: Spi2Java: Automatic cryptographic protocol Java code generation from spi calculus. In: Proceedings of the 18th International Conference on Advanced Information Networking and Applications-Volume 2, IEEE Computer Society (2004) 400
18. Avalle, M., Pironti, A., Sisto, R., Pozza, D.: The JavaSPI framework for security protocol implementation. In: Availability, Reliability and Security (ARES 11), IEEE Computer Society, IEEE Computer Society (2011) 746–751
19. Backes, M., Busenius, A., Hriţcu, C.: On the development and formalization of an extensible code generator for real life security protocols. In: NASA Formal Methods. Springer (2012) 371–387
20. Bugliesi, M. and Modesti, P.: AnBx-Security protocols design and verification. In: Automated Reasoning for Security Protocol Analysis and Issues in the Theory of Security: Joint Workshop, ARSPA-WITS 2010, Paphos, Cyprus, 2010, Revised Selected Papers, Springer-Verlag (2010) 164–184
21. Bella, G., Massacci, F., Paulson, L.: Verifying the SET purchase protocols. Journal of Automated Reasoning **36**(1) (2006) 5–37
22. Gong, L., Ellison, G., Dageforde, M.: Inside Java 2 Platform Security: Architecture, Api Design, and Implementation. Addison-Wesley (2003)
23. Pistoia, M., Nagaratnam, N., Koved, L., Nadalin, A.: Enterprise Java 2 Security: Building Secure and Robust J2EE Applications. Addison Wesley (2004)
24. Hailpern, B., Tarr, P.: Model-driven development: The good, the bad, and the ugly. IBM systems journal **45**(3) (2006) 451–461
25. Armando, A., Basin, D., Boichut, Y., Chevalier, Y., Compagna, L., Cuéllar, J., Drielsma, P.H., Héam, P.C., Kouchnarenko, O., Mantovani, J., et al.: The AVISPA tool for the automated validation of internet security protocols and applications. In: Computer Aided Verification, Springer (2005) 281–285
26. Paolo Modesti: Verified Security Protocol Modeling and Implementation with AnBx. PhD thesis, Università Ca' Foscari Venezia (Italy) (2012)
27. Bugliesi, M. and Calzavara, S. and Mödersheim, S. and Modesti, P.: Security protocol specification and verification with AnBx. Under review (2013)
28. Paolo Modesti: Efficient Java code generation of security protocols specified in AnB/AnBx. Technical Report CS-TR-1422, School of Computing Science, Newcastle University (2014)
29. Apache Foundation: The Apache Ant Project http://ant.apache.org.
30. Bellare, M., Garay, J., Hauser, R., Herzberg, A., Krawczyk, H., Steiner, M., Tsudik, G., Waidner, M.: iKP a family of secure electronic payment protocols. In: Proceedings of the 1st USENIX Workshop on Electronic Commerce. (1995)
31. Bella, G., Massacci, F., Paulson, L.: An overview of the verification of SET. International Journal of Information Security **4**(1) (2005) 17–28
32. Denker, G., Millen, J.: CAPSL and CIL language design. Technical Report SRI-CSL-99-02, SRI International Computer Science Laboratory (1999)
33. Abadi, M., Fournet, C.: Mobile values, new names, and secure communication. ACM SIGPLAN Notices **36**(3) (2001) 104–115
34. Ferguson, N., Schneier, B.: Practical cryptography. Wiley New York (2003)
35. Menezes, A., Van Oorschot, P., Vanstone, S.: Handbook of applied cryptography. CRC (1997)
36. Schneier, B., Sutherland, P.: Applied cryptography: protocols, algorithms, and source code in C. John Wiley & Sons, Inc. (1995)

37. Clover, S.: HStringTemplate www.haskell.org/haskellwiki/HStringTemplate.
38. Parr, T.: A functional language for generating structured text. (2006) www.cs.usfca.edu/ parrt/papers/ST.pdf.
39. Parr, T.: Enforcing strict model-view separation in template engines. In: Proceedings of the 13th international conference on World Wide Web, ACM (2004) 224–233
40. Millen, J., Muller, F.: Cryptographic protocol generation from CAPSL. Technical Report SRI-CSL-01-07, SRI International (December 2001)
41. Denker, G., Millen, J., Rueß, H.: The CAPSL integrated protocol environment. Technical Report SRI-CSL-2000-02, SRI International, Menlo Park, CA (2000)
42. Tobler, B., Hutchison, A.: Generating network security protocol implementations from formal specifications. Certification and Security in Inter-Organizational E-Service (2005) 33–54
43. Eclipse Foundation: Eclipse IDE http://www.eclipse.org.

Alpha ::=	$['A'..'Z'] \cup ['a'..'z']$	*alpha chars*
		$Alpha \subset String$
Ident ::=	$Alpha + String$	*identifier*
Agent ::=	*Ident*	*agent's name*
IdentList ::=	*Ident*	*list of identifiers*
	| *Ident*, *IdentList*	
Opererator ::=	**inv**	*inverse function*
	| **exp**	*exponential function*
	| **crypt**	*asymmetric encryption*
	| **scrypt**	*symmetric encryption*
	| **cat**	*concatenation*
	| **xor**	*xor*
	| **apply**	*function application*
Msg ::=	*Ident*	*identifier*
	| *Operator MsgList*	*operator on msg list*
MsgList ::=	*Msg*	*list of messages*
	| *Msg*, *MsgList*	
Type ::=	**Agent**	*base types*
	| **Number**	
	| **Function**	
	| **PublicKey**	
	| **Symmetic_key**	
TypeList ::=	*Type IdentList*	*list of types*
	| *Type IdentList*; *TypeList*	
Types ::=	**Types** : *TypeList*	*types*
KnowItem ::=	*Agent* : *Msg*	
KnowList ::=	*KnowItem*	*list of agents' know.*
	KnowItem; *KnowList*	
Knowledge ::=	**Knowledge** : *KnowList*	*knowledge*
Action ::=	*Agent ChType Agent* : *Msg*	*action*
ActionList ::=	*Action*	*list of actions*
	| *Action*; *ActionList*	
Actions ::=	**Actions** : *ActionsList*	*actions*
ChType ::=	$\rightarrow \mid \bullet\rightarrow$	*plain* | *authentic*
	| $\rightarrow\bullet \mid \bullet\rightarrow\bullet$	*secret* | *secure*
	| $\bullet\twoheadrightarrow$	*fresh authentic*
	| $\bullet\twoheadrightarrow\bullet$	*fresh secure*
Goal ::=	*Agent ChMode Agent* : *Msg*	*channel goal*
	| *Agent* **weakly authenticates** *Agent* **on** *Msg*	*weak authentication goal*
	| *Agent* **authenticates** *Agent* **on** *Msg*	*authentication goal*
	| *Msg* **secret between** *AgentList*	*secrecy goal*
Goals ::=	*Goal*	*goals*
	Goal; *Goals*	
Protocol ::=	**Protocol** *Ident*	*protocol definition*
	Types Knowledge Actions Goals	

Table 14. Syntax of *AnB*

Alpha	::=	[. . .]	*alpha chars*
Ident	::=	[. . .]	*identifier*
IdentList	::=	[. . .]	*list of identifiers*
Opererator	::=	[. . .]	*operators*
Agent	::=	*Ident*	*agent's name*
	|	'−'	*null agent*
AgentList	::=	*Agent*	*list of agents*
	|	*Agent*, *AgentList*	
Type	::=	[. . .]	*base types*
	|	**Certified**	*certified agent*
Def	::=	*Ident* (*IdentList*) : *Msg*	*definition w pars*
		Ident : *Msg*	*definition w/o pars*
DefList	::=	*Def*	*list of definitions*
		Def; *DefList*	
Defs	::=	ϵ	*empty definitions*
	|	**Definitions** : *DefList*	*non empty definitions*
Digest	::=	[*MsgList*]	*standard digest*
	|	[*MsgList* : *Agent*]	*verifiable digest*
Msg	::=	*Ident*	*identifier*
	|	*Operator MsgList*	*operator on msg list*
	|	*Digest*	*digest*
	|	*PPar Ident MsgList*	*definition w pars in msg*
	|	*PParId Ident*	*definition w/o pars in msg*
KnowItem	::=	*Agent* : *Msg*	*agent's knowledge*
	|	*IdentList* **share** *MsgList*	
KnowList	::=	*KnowItem*	*list of agent's knowledge*
	|	*KnowItem*; *KnowList*	
Knowledge	::=	**Knowledge** : *KnowList*	*knowledge*
Action	::=	*Agent ChType Agent* : *Msg*	*AnB action*
	|	*Agent* $\rightarrow$ *Agent*, *ChMode* : *Msg*	*AnBx action*
ActionList	::=	*Action*	*list of actions*
	|	*Action*; *ActionList*	
Actions	::=	**Actions** : *ActionsList*	*actions*
fresh	::=	ϵ | @	*emtpy* | *fresh*
forward	::=	ϵ | $\uparrow$	*empty* | *forward* (*opt.*)
Vers	::=	*AgentList*	*verifiers*
ChMode	::=	*forward fresh*(*Agent*, *Vers*, *Agent*)	*AnBx channel modes*
ChType	::=	[. . .]	*AnB channel types*
Goal	::=	[. . .]	*goal*
	|	*Agent* **confidentially sends** *Msg* **to** *Agent*	*confidential exchange*
Goals	::=	[. . .]	*goals*
Protocol	::=	**Protocol** *Ident*	*protocol definition*
		Defs Types Knowledge Actions Goals	

Table 15. Syntax of *AnBx* defined as an extension to the standard syntax of *AnB* ([. . .])

Reserved identifiers	::=	**pk**	*public key function for encryption*
	|	**sk**	*public key function for encryption*
	|	**hash**	*hash function*
	|	**hmac**	*hmac function*
	|	**g**	*DH base*
	|	**empty**	*sync message*

Table 16. Syntax of *AnBx* - Reserved identifiers

```
public final class $prot$_$role$ extends AnB_Protocol<$prot$_Steps,$prot$_Roles
    > {
  private static boolean loop = false;
  private static long $sessionID$ = 0;
  // local knowledge - constants
  $fieldsstatic:{n|private static $n.typeof$ $n.name$ = $n.pars$;
  }$
  // local vars
  $fields:{n|private $n.typeof$ $n.name$ = null;
  }$
public $prot$_$role$($prot$_Roles role, String name) {
    super();
    this.role = role;
    this.name = name;
    this.sharepath = sharepath;
    $sessionID$++;
  }
  protected void init() {
    // init shared vars
    $fieldsinit:{n| $n.name$ = ($n.typeof$) AnB_Session.readObject(sharepath+\"
        $n.name$$serExt$\");
  }$};
  public void run(Map<String, AnB_Session> lbs, Map<String, String> aliases) {
    this.aliases = aliases;
    $channelroles:{n|AnB_Session $n.chname$ = lbs.get(\"$n.chrole$\");
    }$
    try {
      init();
      $channels:{n|$n$.Open();
      }$
      do {
        $channelsteps:{n|executeStep($n.channel$, $prot$_Steps.$n.step$);
        }$
      } while (loop);
          $channels:{n|$n$.Close();
          }$

     } catch (ClassCastException e) {
       [...]
     } catch (Exception e) {
       [...]
     }
  };
  protected void executeStep(AnB_Session $sessname$, $prot$_Steps step) {
    status(step);
    switch (step) {
      $stepactions:{n|
        case $n.astep$:
              $n.action$
                  break;
      }$
    }
    status(step);
  }
  $rolemethods:{n|private $n.rettype$ $n.mname$($n.mpars$) {
    // TODO Auto-generated method stub
          return ($n.rettype$) $n.retvalue$;
  \}}$
}
```

Fig. 9. `Role_X.st` file template

```
public class AnB_Crypto_Wrapper {
// implements a class supporting cryptographic operations
// a wrapper for the encryption engine

protected Crypto_EncryptionEngine ee;
protected AnBx_Agent me;
private final static AnBx_Layers layer = AnBx_Layers.LANGUAGE;
public AnB_Crypto_Wrapper(Crypto_EncryptionEngine ee)
public AnB_Crypto_Wrapper(Crypto_KeyStoreSettings_Dual kssd)
public AnB_Crypto_Wrapper(Crypto_KeyStoreSettings_Dual kssd,Crypto_Config
    config)
public Crypto_KeyStoreSettings_Dual getKeyStoreSettings_Dual()
public void Setup(Crypto_KeyStoreSettings_Dual kssd)
public void Setup(Crypto_KeyStoreSettings_Dual kssd,Crypto_Config config)
public static void getInfo()

// ------------------------ cert/identities -----------------------
protected Certificate getRemoteCertificate_enc(String alias)
protected Certificate getRemoteCertificate_sig(String alias)
protected AnBx_Agent getMyIdentity()
protected void setMyIdentity()
// ------------------------ send/receive -----------------------
protected void Send_Id(AnBx_Agent id, Channel_Abstraction c)
protected void Send(Object obj, AnBx_Agent id, Channel_Abstraction c)
protected Object Receive(AnBx_Agent id, Channel_Abstraction c)
protected AnBx_Agent Receive_RemoteId(Channel_Abstraction c)
// ------------------------ encrypt/decrypt -----------------------
public Object decrypt(Crypto_SealedPair sc)
public Crypto_SealedPair encrypt(Object object, String alias)
public Object decrypt(SealedObject so, Key symmetricKey)
public SealedObject encrypt(Object object, Key symmetricKey)
// ------------------------ sign/verify -----------------------
public SignedObject sign(Object object)
public Object verify(SignedObject so, String alias)
// ------------------------ nonces, keys, seqnumbers -------------------
public Crypto_ByteArray getNonce()
public Crypto_ByteArray getSeqNumber()
public SecretKey getSymmetricKey()
public SecretKey getHmacKey()
// ------------------------ key exchange ------------------------------
public KeyPair getKeyEx_KeyPair()
public PublicKey getKeyEx_PublicKey(KeyPair keyPair)
public SecretKey getKeyEx_SecretKey(KeyPair keyPair, PublicKey publicKey)
public SecretKey getKeyEx_SecretKey(PublicKey publicKey, KeyPair keyPair)
// ------------------------ digest hash/hmac ----------------------------
public Crypto_ByteArray makeDigest(Object obj)
public Crypto_ByteArray makeDigest(Object obj, String str)
public Crypto_ByteArray makeHmac(Object obj, SecretKey sk)
// ------------------------ serialization ----------------------------
public static void writeObject(Object obj, String filename)
public static Object readObject(String filename)
}
```

Fig. 10. *AnBxJ*: Crypto API (`AnB_Crypto_Wrapper` class)

```java
public class AnB_Session extends AnB_Crypto_Wrapper {
// implements a session supporting cryptographic operations

private final static AnBx_Layers layer = AnBx_Layers.SESSION;
private Channel_Abstraction c;
private AnBx_Agent id_Remote = null;
private Boolean exchange_id = false;
// allow agents to exchange their aliases

public AnB_Session(Crypto_KeyStoreSettings_Dual kssd, Channel_Settings cs,
    boolean exchange_id, Crypto_Config config)
public AnB_Session(Crypto_KeyStoreSettings_Dual kssd, Channel_Settings cs,
    AnBx_Agent id_Remote, Crypto_Config config)
public AnB_Session(Crypto_KeyStoreSettings_Dual kssd, Channel_Settings cs,
    String id_Remote_alias, Crypto_Config config)
void initChannels(Channel_Settings cs)

// ----------------------- open/close ----------------------
public void Open()
public void Close()
// ------------------------ send/receive ---------------------
public Object Receive()
public void Send(Object obj)
public AnBx_Agent Receive_RemoteId()
public void Send_Id()
// ----------------------- setters/getters ----------------------
public Channel_Abstraction getC()
public AnBx_Agent getId_Remote()
public void setC(Channel_Abstraction c)
public void setId_Remote(AnBx_Agent id_Remote)
}
```

Fig. 11. *AnBxJ*: Communication API (`AnB_Session` class)

```java
public abstract class AnB_Protocol<S, R> {

private final static AnBx_Layers layer = AnBx_Layers.PROTOCOL;
protected String name = null;
protected String sharepath = null;
protected R role;
protected Map<String, String> aliases;
private boolean abortOnFail = false;
private enum CheckType {EQ,INV,WFF;};

abstract public void run(Map<String, AnB_Session> lbs, Map<String, String>
    aliases);
abstract protected void executeStep(AnB_Session lbs, S step);
abstract protected void init();

protected void abort(String msg)
protected void eqCheck(Object obj1, Object obj2)
protected void invCheck(Object obj1, Object obj2)
protected void invCheck(Object obj, Class<?> cls)
protected void wffCheck (Object obj)
protected void status(S step)
}
```

Fig. 12. *AnBxJ*: `AnB_Protocol` class