Secure Implementation of Asynchronous Method Calls and Futures

Peeter Laud

Cybernetica AS
peeter@cyber.ee

Abstract. Programming languages suitable for distributed computation contain constructs that should map well to the structure of the underlying system executing the programs, while being easily usable by the programmers and amenable to computer-aided verification. For object-oriented languages, *asynchronous method calls* returning *futures* that will be filled only after the called method has finished its execution have been proposed as a reasonably simple and analyzable programming construct. In this paper, we show how to map from a language with asynchronous method calls and futures to a language with explicit communication primitives and cryptographic operations. Our target language is reasonably similar to common process calculi, and translating it further to e.g. the applied pi calculus requires only known techniques. The translation is valid even for programs executing in *open* environments, where method calls and futures can be transmitted between the program and the environment.
Keywords: OO languages, process calculi, full abstraction

1 Introduction

One of the main issues in the interplay of object-orientation and distribution in programming languages is the handling of method calls and returns between loosely coupled objects without losing the benefits of synchronization [27]. In this paper we are studying a language that proposes to strike a suitable balance between these issues. The *Abstract Behavioral Specification* (ABS) language [22, 20] is an extension of Creol [23], currently used as the underlying formalism in a large-scale collaborative effort for formal verification of adaptable and evolvable software systems (`http://hats-project.eu`, [19]). The communication and synchronization abstractions of ABS have been carefully chosen to make the language convenient to use in modeling and specifying various concurrent systems [12, 21], while at the same time supporting formal analysis and verification [11, 28]. Thus the usage of ABS allows the design of *highly trusted* flexible software systems.

The inter-object messaging in ABS is based on asynchronous calls and futures. A method call immediately returns with a *future* that will be filled with the called method's result only after it has finished. At the same time, futures are first-class values and may be stored or passed around similarly to atomic data

or object references. This makes possible very varied communication patterns between objects. At the source level, this communication is secured — the semantics of the language does not allow someone to intercept a message between two objects or to replace it with a different one.

The issues of actually securing the inter-object communication were likely not considered during the design of ABS. The advanced concepts in the construction of the language do not map well into the concepts of security systems and it is not clear at all how this communication should be protected in an actual distributed implementation. A *walled garden* approach could be taken if the whole program runs under a single authority; in this case, the communication between all objects is protected by a single key and appropriate cryptographic protocols can be used to prevent replays by the adversary. If the program is *open* — different objects are controlled by different, mutually distrustful authorities — then this approach no longer works as long as calls between different authorities as possible.

In this paper we show how the communication between objects in ABS can be protected using cryptographic techniques. We propose a *fully abstract* translation from the ABS language to a language with explicit message-passing and cryptographic operations — two programs in ABS are indistinguishable iff their translations are indistinguishable. Hence this translation preserves all observable properties of programs, including all different kinds of security properties (integrity, secrecy, non-interference, etc.). In this paper, we are considering the symbolic semantics of cryptographic operations (perfect cryptography, or Dolev-Yao model) [15]. For computational soundness, the results of [13] may be applicable if encryption cycles are avoided. We believe that our implementation language can be straightforwardly translated into well-known process calculi, e.g. the applied pi calculus, such that a program in our implementation language and its translation are observationally equivalent. Besides being an interesting result in characterizing which communication models can be fully abstracted using cryptography (we discuss this more in the next section), our result also allows to carry over the verification results obtained for the ABS language to its implementation and thus serves as a validation of the design of ABS.

2 Related work

There has been a fair amount of work in securely translating the abstractions of communication back into the exchanging of messages on point-to-point channels between entities. This line of work was started by Abadi et al. [3, 4] who show how channels protected by the knowledge of names can be securely implemented using public channels where the messages are protected through cryptographic means. The translation is applied to processes specified in the join-calculus [16]; its restrictive scope extrusion rules help to simplify the translation. The source language is abstracted further by introducing authentication primitives in [5]. Authentication primitives in a π-calculus setting have been considered by Backes et al. [8]. Securing the channels with cryptographic mechanisms has also been explored by Bugliesi and Focardi [9], and by Mödersheim and Viganó [26] who

consider languages containing confidential and/or authentic channels between principals and ways to simulate them on unprotected channels using cryptography. Adão and Fournet [7] consider the translation of authentic channels between principals directly into channels protected by cryptography in the *computational* model [18]. Bugliesi and Giunti [10] give a translation for a variation of π-calculus with normal scope extrusion rules, but with capability types on channel names and a proxy service in the translated process that has no counterpart in the original.

Abstract information protection primitives and their implementation has also been considered in the language-based information flow security community. Vaughan and Zdancewic [31] consider the *packing/unpacking* of information at certain level of the information flow lattice; code executing with lower privileges cannot access high-packed data. Fournet and Rezk [17] consider programs with holes for adversary's code; the program data must be protected across these holes.

Abadi [1] has discussed the role full abstraction plays in the implementations of secure systems, as well as the difficulties in achieving it. He notes that for the full π-calculus, the unconstrained distribution of read-capabilities of channels, together with the requirement of forward secrecy (messages exchanged on a channel before the adversary obtained the read-capability for that channel remain secret) makes fully abstract translations to cryptographically protected channels hard to construct. On the other hand, in the join-calculus [3, 4] only write-capabilities can be distributed.

The constructs of ABS give rise to two different kinds of channels. An object reference can be used to send messages (invoke methods) to that object. This represents a channel where the write-capability can be freely distributed, but the read-capability is only at the object. This channel is similar to the channels of join-calculus. A future represents a channel where the read-capability can be freely distributed, but the write-capability is owned by a single task. According to π-calculus semantics, such channel would pose difficulties for a fully abstract translation, but the semantics of ABS makes it possible by not requiring forward secrecy. The handshake that takes place during a method invocation is possibly the trickiest feature to translate. Here a message on the first kind of channel causes the creation of a second kind of channel, such that the recipient has the write- and the sender the read-capability. A method invocation is atomic in ABS, hence the translation must be atomic, too.

3 The source language

3.1 Syntax

Our source language is a simplified version of (the object-oriented fragment of) ABS, retaining all the interesting details of inter-object communication and parallelism. We leave out the semaphore-based cooperative synchronization of tasks belonging to the same object, as our translation is orthogonal to those and

it is well-known how to express semaphores in π-calculus [25]. The (abstract) syntax of the source language is given in Figure 1. The notation $\overline{X}$ denotes a sequence of X-s.

$x \mid n \mid o \mid f$	local variable | task | object | field name
$Pr ::= \overline{Cl}\ B$	program
$Cl ::= \mathsf{class}\ C\{\overline{T\,f}\ \overline{M}\}$	class definition
$M ::= T\ m(\overline{T\,x})\ B$	method definition
$B ::= \{\overline{T\,x}\ s\}$	method body
$v ::= x \mid \mathsf{this} \mid f$	variable
$i ::= \ldots \mid -1 \mid 0 \mid 1 \mid \ldots$	integer
$e ::= v \mid \mathsf{null} \mid i \mid e \oplus e \mid e!m(\overline{e}) \mid e.\mathsf{get} \mid \mathsf{new}\ C$	expression
$s ::= v := e \mid \mathsf{return}\ e \mid cs$	statement
$cs ::= \mathsf{skip} \mid \mathsf{if}\ (e)\ s\ \mathsf{else}\ s \mid \mathsf{while}\ (e)\ s \mid s; s$	control statement
$T ::= \mathsf{Int} \mid C \mid \mathsf{Fut}(T)$	type

Fig. 1. Syntax of the source language

Let us explain the language constructs related to parallel distributed execution. Each object executes independently of others (i.e. the objects are the grains of distribution), the objects communicate *only* by asynchronous method calls (e.g. it is impossible to directly read or write some field of some object from inside another object). Each object belongs to some class; different classes have different methods. Each object may concurrently execute a number of tasks. In ABS, the scheduling of tasks belonging to the same object, is cooperative [20]. For simplifying the presentation in this paper, we let the tasks also execute independently of each other (the next task to make a step is chosen nondeterministically from the set of all tasks). The names of objects and tasks are picked from a countable set **N**.

The expression $e!m(\overline{e})$ denotes the *asynchronous* call of the method m. The call immediately returns a future. At the same time, a new task executing the code of m is started at the receiver of the call. The get-construct is used to read the value of that future, if it is available. If not, then get blocks. The expression $e \oplus e$ denotes the application of any binary operation to two expressions. In the actual implementations, there may be several different operations. In particular, $\oplus$ may denote the comparison of two values.

As specified in Fig. 1, the language does not contain means to prevent type errors. We assume that reasonable default values are used whenever a type error is detected at runtime (0 for integers, null for object references, a never-available future for futures, including the results of method calls on null).

3.2 Operational Semantics

The semantics of a program Pr is a labeled transition system (LTS). In general, a LTS is a quadruple $\mathcal{L} = (S, A, \rightarrow, s_0)$, where S is the set of *states*, A is the set of *labels* (both can be infinite), $\rightarrow$ is a subset of $S \times A \times S$, and $s_0 \in S$ is the starting state. We write $s \xrightarrow{\alpha} s'$ for $(s, \alpha, s') \in \rightarrow$. We assume that A contains a special label τ that we call the *silent label*. A LTS communicates with its outside environment through its transitions that carry non-silent labels.

The semantics $[\![Pr]\!]$ of ABS programs (as closed systems) is given in [20]. The semantics of a program as a closed system can be seen as a LTS where all transitions are labeled τ. An open-system semantics of an ABS-like language is considered in [6] and our treatment is a simplified version of that.

A *run-time* configuration —the state of the LTS— of a program is a set of objects. Each object is related to zero or more tasks. The configuration also records object and task names that are made known to the environment. Formally, the run-time configurations are as follows:

$$P ::= o[C, \sigma, \varphi] \mid n\,\langle o, \sigma, s\rangle \mid \mathsf{p}(o) \mid \mathsf{p}(n) \mid P \parallel P$$

Each object is represented by its identifier o, its class C, its state σ (the values of its fields), and the values of the futures it has received φ (a mapping from names to values). Each task is represented by its identifier n, its object o, the statement s that is yet to be executed in this task, and its state σ (the values of its local variables). Both o and n are *names*. The names of objects and tasks do not repeat inside a configuration. The operation $\parallel$ is considered to be commutative and associative.

The notation $\mathsf{p}(o)$ means that the name o is *public* — the environment knows it. It is possible for an environment to know the name o without the object $o[C, \sigma, \varphi]$ being part of the configuration. This means that the environment controls and executes that object. The notation $\mathsf{p}(n)$ means the same for task names n. For a configuration P, let $\mathcal{N}(P)$ denote all object and task names in the configuration. Let $\mathcal{N}_{\mathrm{p}}(P)$ and $\mathcal{N}_{\mathrm{l}}(P)$ denote all public names in P, and all local names in P (i.e. names of objects and tasks present in P), respectively.

The evaluation contexts for expressions and statements are defined as follows. We let c range over the *constants* — integers, null, and names. The hole $[]$ will be filled with the expression that will be evaluated during the next computation step.

$$e^{[]} ::= [] \mid e^{[]} \oplus e \mid c \oplus e^{[]} \mid e^{[]}!m(\overline{e}) \mid c!m(\overline{c}, e^{[]}, \overline{e}) \mid e^{[]}.\mathsf{get}$$
$$s^{[]} ::= v := e^{[]} \mid \mathsf{return}\ e^{[]} \mid \mathsf{if}\ (e^{[]})\ s\ \mathsf{else}\ s \mid \mathsf{while}\ (e^{[]})\ s \mid s^{[]}; s$$

The transition rules of the LTS $[\![Pr]\!]$ are given in Fig. 2. In these rules we make the following convention: if a task $n\langle o, \ldots\rangle$ is part of a configuration, then the object $o[\ldots]$ is also a part of it, even if it is not shown.

In the semantics, the rule (acall_1) describes a method call from an object o to o', both in the configuration. A new task is created which starts in a suspended state. The call immediately returns a future whose value is equal to the name of

$$\frac{}{n\langle o,\sigma,s[x]\rangle \xrightarrow{\tau} n\langle o,\sigma,s[\sigma(x)]\rangle}\ (\mathsf{rv}) \qquad \frac{}{n\langle o,\sigma,x:=c\rangle \xrightarrow{\tau} n\langle o,\sigma[x\mapsto c],\mathsf{skip}\rangle}\ (\mathsf{wv})$$

$$\frac{}{o[C,\sigma,\varphi] \parallel n\langle o,\sigma',s[f]\rangle \xrightarrow{\tau} o[C,\sigma,\varphi] \parallel n\langle o,\sigma',s[\sigma(f)]\rangle}\ (\mathsf{rf})$$

$$\frac{}{o[C,\sigma,\varphi] \parallel n\langle o,\sigma',f:=c\rangle \xrightarrow{\tau} o[C,\sigma[f\mapsto c],\varphi] \parallel n\langle o,\sigma',\mathsf{skip}\rangle}\ (\mathsf{wf})$$

$$\frac{}{n\langle o,\sigma,s[c_1\oplus c_2]\rangle \xrightarrow{\tau} n\langle o,\sigma,s[[\![\oplus]\!](c_1,c_2)]\rangle}\ (\mathsf{arith})$$

$$\frac{}{n\langle o,\sigma,\mathsf{skip};s\rangle \xrightarrow{\tau} n\langle o,\sigma,s\rangle}\ (\mathsf{skip}) \qquad \frac{n\langle o,\sigma,s_1\rangle \xrightarrow{\tau} n\langle o,\sigma,s_1'\rangle}{n\langle o,\sigma,s_1;s_2\rangle \xrightarrow{\tau} n\langle o,\sigma,s_1';s_2\rangle}\ (\mathsf{seq})$$

$$\frac{c\neq 0}{n\langle o,\sigma,\mathsf{if}\ (c)\ s_1\ \mathsf{else}\ s_2\rangle \xrightarrow{\tau} n\langle o,\sigma,s_1\rangle}\ (\mathsf{if}_1) \qquad \frac{}{n\langle o,\sigma,\mathsf{if}\ (0)\ s_1\ \mathsf{else}\ s_2\rangle \xrightarrow{\tau} n\langle o,\sigma,s_2\rangle}\ (\mathsf{if}_2)$$

$$\frac{c\neq 0}{n\langle o,\sigma,\mathsf{while}\ (c)\ s\rangle \xrightarrow{\tau} n\langle o,\sigma,s;\mathsf{while}\ (c)\ s\rangle}\ (\mathsf{while}_1)$$

$$\frac{}{n\langle o,\sigma,\mathsf{while}\ (0)\ s\rangle \xrightarrow{\tau} n\langle o,\sigma,\mathsf{skip}\rangle}\ (\mathsf{while}_2)$$

$$\frac{\mathrm{body}(m)=s \qquad s_{task}=s[\overline{c}/params(m)] \qquad n'\neq n}{o'[\ldots] \parallel n\,\langle o,\sigma,s[o'!m(\overline{c})]\rangle \xrightarrow{o\to o'} o'[\ldots] \parallel n\,\langle o,\sigma,s[n']\rangle \parallel n'\,\langle o',\sigma_{init},s_{task}\rangle \parallel \mathsf{p}(o) \parallel \mathsf{p}(o')}\ (\mathsf{acall}_1)$$

$$\frac{o'\neq o \qquad names(\overline{c})=\{c_1,\ldots,c_k\} \qquad n'\notin\{n,c_1,\ldots,c_k\}}{n\,\langle o,\sigma,s[o'!m(\overline{c})]\rangle \parallel \mathsf{p}(o') \xrightarrow{o\to n'[o'.m(\overline{c})]} n\,\langle o,\sigma,s[n']\rangle \parallel \mathsf{p}(o') \parallel \mathsf{p}(o) \parallel \mathsf{p}(c_1) \parallel \cdots \parallel \mathsf{p}(c_k)}\ (\mathsf{acall}_2)$$

$$\frac{n\notin names(\overline{c})=\{c_1,\ldots,c_k\} \qquad \mathrm{body}(m)=s \qquad s_{task}=s[\overline{c}/params(m)]}{o[\ldots] \parallel \mathsf{p}(o) \parallel \mathsf{p}(c_1) \parallel \cdots \parallel \mathsf{p}(c_k) \xrightarrow{n=o!m(\overline{c})} o[\ldots] \parallel \mathsf{p}(o) \parallel \mathsf{p}(c_1) \parallel \cdots \parallel \mathsf{p}(c_k) \parallel \mathsf{p}(n) \parallel n\langle o,\sigma_{init},s_{task}\rangle}\ (\mathsf{acall}_3)$$

$$\frac{\varphi(n') \text{ is undefined}}{o[C,\sigma,\varphi] \parallel n'\langle o',\sigma',\mathsf{return}\ c;s\rangle \xrightarrow{o\leftarrow o'} o[l,C,\sigma,\varphi[n'\mapsto c]] \parallel n'\langle o',\sigma',\mathsf{return}\ c;s\rangle \parallel \mathsf{p}(o) \parallel \mathsf{p}(o')}\ (\mathsf{return}_1)$$

$$\frac{}{\mathsf{p}(n) \parallel n\langle o,\sigma,\mathsf{return}\ c;s\rangle \xrightarrow{n[c]\leftarrow o} \mathsf{p}(n) \parallel n\langle o,\sigma,\mathsf{return}\ c;s\rangle \parallel \mathsf{p}(o) \parallel \mathsf{p}(names(c))}\ (\mathsf{return}_2)$$

$$\frac{n\neq n' \qquad \varphi(n') \text{ is undefined}}{o[C,\sigma,\varphi] \parallel \mathsf{p}(n') \parallel \mathsf{p}(names(c)) \xrightarrow{o\leftarrow n'[c]} o[C,\sigma,\varphi[n'\mapsto c]] \parallel \mathsf{p}(n') \parallel \mathsf{p}(o) \parallel \mathsf{p}(names(c))}\ (\mathsf{return}_3)$$

$$\frac{\varphi(n') \text{ is defined}}{o[C,\sigma,\varphi] \parallel n\langle o,\sigma',s[n'.\mathsf{get}]\rangle \xrightarrow{\tau} o[C,\sigma,\varphi] \parallel n\langle o,\sigma',s[\varphi(n')]\rangle}\ (\mathsf{get})$$

$$\frac{}{\square \xrightarrow{\nu n} \mathsf{p}(n)}\ (\mathsf{newn}) \qquad \frac{o'\neq o}{n\langle o,\sigma,s[\mathsf{new}\ C]\rangle \xrightarrow{\tau} n\langle o,\sigma,s[o']\rangle \parallel o'[C,\sigma_{init},\varphi_{empty}]}\ (\mathsf{newc})$$

$$\frac{}{\square \xrightarrow{\nu o} \mathsf{p}(o)}\ (\mathsf{newo}) \qquad \frac{P \xrightarrow{\alpha} P' \qquad \mathcal{N}_{\mathrm{f}}(\alpha)\cap\mathcal{N}_{\mathrm{l}}(P'')=\emptyset}{P \parallel P'' \xrightarrow{\alpha} P' \parallel P''}\ (\mathsf{frame})$$

Fig. 2. Operational semantics of the source language

the new task. This call is visible to the environment (o made a method call to o'), but called method and its arguments are supposed to be protected. The names o and o' both become known to the environment, too. This is our design choice, because protecting against eavesdropping is supposedly cheap, but protecting against traffic analysis is expensive. The rule (acall_2) describes a method call from an object o in the configuration to an object o' in the environment. In this case, the environment learns the caller, as well as all arguments of the method. The rule (acall_3) describes a call from the environment to an object in the configuration. A name can be among the arguments of the call only if it is already known to the environment. There is no constraint on the integer arguments.

The rules (return_i) describe the returning of the result of a completed task. If the object receiving the value belongs to the configuration, it saves the returned value in its φ-component. Later, the get-expression may read it from φ; this is no longer visible to the environment. We have made a design choice that the result of a completed task may be returned to an object at most once. Most importantly, this means that all expressions $n.\mathsf{get}$ in the tasks of some object o return the same value. In a different object o', the value of $n.\mathsf{get}$ may be different (if n is the name of a task managed by the environment). In an implementation, it is simple to ensure the uniqueness of returned values in a single object, while comparing them across all objects requires complex protocols, e.g. Byzantine agreement.

Rule (newc) describes the creation of a new object. The creation is invisible to the environment, which does not learn the name of the new object. This allows the new object to be initialized by its creator (the first method call to the newly created object must necessarily come from the program, not from the environment, unless the program chooses to leak the identity of the newly generated object before placing any calls). This also allows the identities of objects to serve as passwords (that the adversary does not know). So the programmer can control how much interaction with the environment is allowed/possible.

The environment can also generate new names for objects and tasks (rules (newn) and (newo)); $\Box$ denotes the empty environment (unit element for $\|$). Finally, the (frame) rule states that each of the steps can also occur in a larger context, if the context does not interfere with the step. The set $\mathcal{N}_{\mathrm{f}}(\alpha)$ denotes all names in α that cannot be local names in a configuration to which a transition labeled by α is made. It is defined by

$$\mathcal{N}_{\mathrm{f}}(o \to n'[o'.m(\bar{c})]) = \{o', n'\} \qquad \mathcal{N}_{\mathrm{f}}(\nu n) = \{n\}$$
$$\mathcal{N}_{\mathrm{f}}(o \leftarrow n'[c]) = \{n'\} \qquad \mathcal{N}_{\mathrm{f}}(\nu o) = \{o\} \ .$$

For other labels, $\mathcal{N}_{\mathrm{f}}(\alpha) = \emptyset$. Also, when writing $P' \parallel P''$, we assume the objects and tasks in P' and P'' have different names.

The initial configuration for the program $\overline{Cl}\ \{\overline{T\,x}\ s\}$ will consist of just the task $n_0\,\langle \mathsf{null}, \sigma, s\rangle$. This task is the only task that is not tied to an object (all tasks created later will be tied to some object).

We denote the set of all labels occurring in Fig. 2 by Act_{src}.

4 The implementation language

We now present our implementation language. Its main differences from the source language are

- explicit message passing replaces asynchronous method calls;
- the spawning of new tasks is decided locally;
- cryptographic techniques, not the knowledge of names is used to protect the communication between objects;
- objects and tasks do not have names, but are identified through their keys.

As such, the communication primitives of our implementation language are the same as in well-known cryptographic process calculi, e.g. the applied pi-calculus [2]. We retain the syntactic constructions for classes, methods, and program state from our source language. It is known how to translate these constructions into the π-calculus [24, 29]. Thus we believe that we are justified in considering our implementation language as a dialect of the applied pi-calculus.

4.1 Syntax

Our implementation language contains cryptographic operations for the construction of messages exchanged between objects. The exact set of available operations does (mostly) not affect the syntax or semantics of the language, hence we defer its specification to Sec. 5 where we explain how to translate from the source to the implementation language. We assume we are given a signature Σ containing possible operations together with their arities. Also, there is an equational theory over cryptographic messages that specifies the cancellation and other rules. We let F range over the operations in Σ. For each $F \in \Sigma$, there will be an an operation that can be invoked in expressions; this operation applies F to its arguments. By overloading the notation, we let F denote that operation as well (hence the rule (crypt) in Fig. 4).

Still, there are a couple of operations that we need Σ to contain. As we want to identify the objects by their public keys, we require Σ to contain unary operations $\cdot^+$ and $\cdot^-$. No cancellation rules are associated with just these operations.

The syntax of the implementation language is given in Fig. 3. Compared to the source language, the first difference that we spot is the extra method body — the "main method" — in the class declarations. We denote the main method of a class C by $C.run$. This method is executed immediately after an object has been created and it will remain active for the entire lifetime of the object. Another difference is, that we require each class to have fields k_{pub} and k_{priv} where the public and private key of a newly generated object are saved upon its creation.

The expressions $e!m(\overline{e})$ for method calls and $e.\mathsf{get}$ for reading the value of a future have disappeared from the language. Instead, there is a statement $\mathsf{send}(e_1, e_2)$ that sends the message e_2 to the object with the public key e_1. There is also an expression recv that returns a message that has been sent to this object. Note that messages are sent to objects, not tasks.

$x \mid a \mid f$	local variable | atom | field name
$Pr ::= \overline{Cl}\ B$	program
$Cl ::= \mathsf{class}\ C\{\overline{T\ f}\ \overline{M}\ B\}$	class definition
$M ::= T\ m(\overline{T\ x})\ B$	method definition
$B ::= \{\overline{T\ x}\ s\}$	method body
$v ::= x \mid \mathsf{this} \mid f$	variable
$i ::= \ldots \mid -1 \mid 0 \mid 1 \mid \ldots$	integer
$e ::= v \mid \mathsf{null} \mid i \mid e \oplus e \mid a \mid F(e, \ldots, e)$ $\mid \mathsf{new}\ C \mid \mathsf{recv} \mid \mathsf{newatom} \mid \mathsf{unbox}(e)$	expression
$s ::= v := e \mid \mathsf{send}(e, e) \mid \mathsf{spawn}\ m(\overline{e}) \mid cs$	statement
$cs ::= \mathsf{skip} \mid \mathsf{if}\ (e)\ s\ \mathsf{else}\ s \mid \mathsf{while}\ (e)\ s \mid s; s$	control statement
$T ::= \mathsf{Int} \mid \mathsf{Msg}$	type

Fig. 3. Syntax of the implementation language

The expression newatom picks a new atomic cryptographic message (usable as a nonce, key, etc.) from a countable set $\mathcal{A}$. The expression $F(e_1, \ldots, e_n)$ constructs a new cryptographic message by applying the constructor F to the messages $e_1, \ldots, e_n$. The argument e_i may also be an integer; in this case, it will automatically be boxed as a message. For the opposite conversion, there is an explicit operation unbox (returns 0 if the argument is not a boxed integer). Again we assume the existence of a reasonable type system that avoids the mixing up of values of type Int and Msg. Besides these two types, our implementation language may reasonably include values of other types, and our translation procedure given in the next section will indeed need more types. These types do not affect the translatability of our implementation language into the applied pi calculus.

Finally, the spawn-statement is used to spawn new tasks. It adds a new task *at the same object* where the spawn-statement was executed. This task will execute the body of the method m. Note that spawn is a statement, not an expression. Hence it does not return anything.

4.2 Operational semantics

In the implementation language, object and future names are replaced by cryptographic messages. In particular, each object is represented by its public key and each task is identified by an atom. The set of values c is now generated by the grammar

$$c ::= i \mid a \mid \mathsf{null} \mid F(c, \ldots, c)\ .$$

Here null has the type Msg. We may also denote a value by E if we want to stress that it is a cryptographic message. We can also use the notation $E(x_1, \ldots, x_l)$

which denotes a message containing the variables $x_1, \ldots, x_l$; such message becomes a value if these variables are substituted with values.

The semantics of a program Pr in the implementation language is again a LTS, albeit with a different set of labels. Its runtime configurations are given by the grammar

$$P ::= k^+[C, \sigma] \mid \langle k^+, \sigma, s \rangle \mid \mathsf{g}(a) \mid \mathsf{p}(x \backslash E) \mid P \parallel P \ .$$

Here $k^+[C, \sigma]$ is an object of class C with the public key k^+ and values of fields σ. A task $\langle k^+, \sigma, s \rangle$ of this object k^+ does not have a name, but it has the values of local variables σ and the yet-to-be-executed statement s. The runtime configuration $\mathsf{g}(a)$ records that an atomic message a has been generated. The configuration $\mathsf{p}(x \backslash E)$ records that the environment has learned the message E built from atomic messages (and integers) using the constructors in Σ, and this message is available to the environment through the variable x, picked from a countable set $\mathcal{X}$. The components $\mathsf{p}(x \backslash E)$ are similar to the *frame* of a process in the applied pi calculus. The transitions ensure that the environment of the program can only perform operations that are consistent with the model of symbolic cryptography.

Let $\mathsf{NV}(P)$ denote the set of all names and variables *defined* in the configuration. If $u \in \mathcal{A} \cup \mathcal{X}$ then $u \in \mathsf{NV}(P)$ iff $u^+[\ldots]$, $\mathsf{g}(u)$, or $\mathsf{p}(u \backslash E)$ belongs to P.

The transitions of the LTS $[\![Pr]\!]$ are given in Fig. 4. Similarly to the presentation of the semantics of the source language in Fig. 2, we use the convention that if a task $\langle K, \ldots \rangle$ is mentioned in the configuration, then the object $K[\ldots]$ is also a part of that configuration, even if it has been omitted from the rule.

The rule (send_1) describes the transmission of messages from one object in the configuration to another one. The first argument of send is the identity (the public key) of the receiver. If it equals null then anyone (including the adversary) may receive that message. Importantly, the sender will not proceed before the receiver has received the message. This is similar to the method call in the source language where, after making the call, the caller knows that the callee has received it. The adversary sees the identity of the sender, the identity of the receiver, and the message. All these are bound to new variables $k_1, k_2, x \in \mathcal{X}$ that the adversary may use afterwards. The rule (send_2) describes the transmission of a message where the sender is in the configuration, but the receiver is not. The adversary gets exactly the same information. If the receiver of the message is specified as null then both rules are applicable.

The rule (send_3) describes the reception of a message sent by the adversary. It is an example of how the adversary can use the messages it has received. Namely, the adversary specifies a message with variables $x_1, \ldots, x_l \in \mathcal{X}$ that are bound to values $c_1, \ldots, c_l$ in the configuration. These values will be substituted in place of the variables in the received message. This construction allows the adversary to specify precisely those messages that it can construct from the received messages according to the rules of symbolic cryptography.

The rule (crypt) handles the application of cryptographic constructors and the rule (unbox) the unboxing of integers. The function *unbox* returns i if its

$$\frac{}{\langle K,\sigma,s[x]\rangle \xrightarrow{\tau} \langle K,\sigma,s[\sigma(x)]\rangle}\ (\mathsf{rv}') \qquad \frac{}{\langle K,\sigma,x := c\rangle \xrightarrow{\tau} \langle K,\sigma[x\mapsto c],\mathsf{skip}\rangle}\ (\mathsf{wv}')$$

$$\frac{}{K[C,\sigma] \parallel \langle K,\sigma',s[f]\rangle \xrightarrow{\tau} K[C,\sigma] \parallel \langle K,\sigma',s[\sigma(f)]\rangle}\ (\mathsf{rf}')$$

$$\frac{}{K[C,\sigma] \parallel \langle K,\sigma',f := c\rangle \xrightarrow{\tau} K[C,\sigma[f\mapsto c]] \parallel \langle K,\sigma',\mathsf{skip}\rangle}\ (\mathsf{wf}')$$

$$\frac{}{\langle K,\sigma,s[c_1 \oplus c_2]\rangle \xrightarrow{\tau} \langle K,\sigma,s[[\![\oplus]\!](c_1,c_2)]\rangle}\ (\mathsf{arith}')$$

$$\frac{}{\langle K,\sigma,\mathsf{skip};s\rangle \xrightarrow{\tau} \langle K,\sigma,s\rangle}\ (\mathsf{skip}') \qquad \frac{\langle K,\sigma,s_1\rangle \xrightarrow{\tau} \langle K,\sigma,s_1'\rangle}{\langle K,\sigma,s_1;s_2\rangle \xrightarrow{\tau} \langle K,\sigma,s_1';s_2\rangle}\ (\mathsf{seq}')$$

$$\frac{c \neq 0}{\langle K,\sigma,\mathsf{if}\ (c)\ s_1\ \mathsf{else}\ s_2\rangle \xrightarrow{\tau} \langle K,\sigma,s_1\rangle}\ (\mathsf{if}_1') \qquad \frac{}{\langle K,\sigma,\mathsf{if}\ (0)\ s_1\ \mathsf{else}\ s_2\rangle \xrightarrow{\tau} \langle K,\sigma,s_2\rangle}\ (\mathsf{if}_2')$$

$$\frac{c \neq 0}{\langle K,\sigma,\mathsf{while}\ (c)\ s\rangle \xrightarrow{\tau} \langle K,\sigma,s;\mathsf{while}\ (c)\ s\rangle}\ (\mathsf{while}_1')$$

$$\frac{}{\langle K,\sigma,\mathsf{while}\ (0)\ s\rangle \xrightarrow{\tau} \langle K,\sigma,\mathsf{skip}\rangle}\ (\mathsf{while}_2')$$

$$\frac{K_2' = K_2 \vee K_2' = \mathsf{null}}{\begin{array}{c}\langle K_1,\sigma,\mathsf{send}(K_2',c);s\rangle \parallel \langle K_2,\sigma',s'[\mathsf{recv}]\rangle \xrightarrow{k_1\to k_2:x} \\ \langle K_1,\sigma,s\rangle \parallel \langle K_2,\sigma',s'[c]\rangle \parallel \mathsf{p}(x\backslash c) \parallel \mathsf{p}(k_1\backslash K_1) \parallel \mathsf{p}(k_2\backslash K_2)\end{array}}\ (\mathsf{send}_1)$$

$$\frac{K_2 \neq K_1}{\langle K_1,\sigma,\mathsf{send}(K_2,c);s\rangle \xrightarrow{k_1\to k_2:x} \langle K_1,\sigma,s\rangle \parallel \mathsf{p}(x\backslash c) \parallel \mathsf{p}(k_1\backslash K_1) \parallel \mathsf{p}(k_2\backslash K_2)}\ (\mathsf{send}_2)$$

$$\frac{K_2 \neq K_1 \quad \forall x_i : \theta(x_i) = c_i}{\begin{array}{c}\langle K_2,\sigma',s'[\mathsf{recv}]\rangle \parallel \mathsf{p}(k_2\backslash K_2) \parallel \mathsf{p}(x_1\backslash c_1) \parallel \cdots \parallel \mathsf{p}(x_l\backslash c_l) \xrightarrow{\to k_2:E(x_1,\ldots,x_l)} \\ \langle K_2,\sigma',s'[E\theta]\rangle \parallel \mathsf{p}(k_2\backslash K_2) \parallel \mathsf{p}(x_1\backslash c_1) \parallel \cdots \parallel \mathsf{p}(x_l\backslash c_l)\end{array}}\ (\mathsf{send}_3)$$

$$\frac{c = F(c_1,\ldots,c_k)}{\langle K,\sigma,s[F(c_1,\ldots,c_k)]\rangle \xrightarrow{\tau} \langle K,\sigma,s[c]\rangle}\ (\mathsf{crypt}) \qquad \frac{unbox(c) = i}{\langle K,\sigma,s[\mathsf{unbox}(c)]\rangle \xrightarrow{\tau} \langle K,\sigma,s[i]\rangle}\ (\mathsf{unbox})$$

$$\frac{a \neq K}{\langle K,\sigma,s[\mathsf{newatom}]\rangle \xrightarrow{\tau} \langle K,\sigma,s[a]\rangle \parallel \mathsf{g}(a)}\ (\mathsf{newa}) \qquad \frac{}{\Box \xrightarrow{\nu x} \mathsf{p}(x\backslash a) \parallel \mathsf{g}(a)}\ (\mathsf{newn})$$

$$\frac{\mathrm{body}(m) = s \qquad s_{task} = s[\overline{c}/params(m)]}{\langle K,\sigma,\mathsf{spawn}\ m(\overline{c});s\rangle \xrightarrow{\tau} \langle K,\sigma,s\rangle \parallel \langle K,\sigma_{init},s_{task}\rangle}\ (\mathsf{spawn})$$

$$\frac{body(C.run) = s_{\mathrm{init}} \qquad K \neq k \in \mathcal{A} \qquad \sigma'(k_{\mathrm{pub}}) = k^+ \qquad \sigma'(k_{\mathrm{priv}}) = k^-}{\langle K,\sigma,s[\mathsf{new}\ C]\rangle \xrightarrow{\tau} \langle K,\sigma,s[k^+]\rangle \parallel k^+[C,\sigma'] \parallel \langle k^+,\sigma'',s_{\mathrm{init}}\rangle \parallel \mathsf{g}(k)}\ (\mathsf{newc})$$

$$\frac{\triangle \in \{=,\neq\} \qquad E(E_1,\ldots,E_l)\ \triangle\ E'(E_1,\ldots,E_l)}{\mathsf{p}(x_1\backslash E_1) \parallel \cdots \parallel \mathsf{p}(x_l\backslash E_l) \xrightarrow{E(x_1,\ldots,x_l)\triangle E'(x_1,\ldots,x_l)} \mathsf{p}(x_1\backslash E_1) \parallel \cdots \parallel \mathsf{p}(x_l\backslash E_l)}\ (\mathsf{eq})$$

$$\frac{P \xrightarrow{\alpha} P' \qquad \mathsf{NV}(P'') \cap \mathsf{NV}(P') = \emptyset}{P \parallel P'' \xrightarrow{\alpha} P' \parallel P''}\ (\mathsf{frame})$$

Fig. 4. Operational semantics of the target language

argument was a boxed integer i, and 0 otherwise. The rules (newa) and (newn) describe the generation of new atoms (in $\mathcal{A}$) by the program or by the adversary. In both cases, the newly generated atom is recorded in the configuration. The rule (spawn) describes the spawning of new tasks. The new task belongs to the same object K as the task from where it was spawned. The rule (newc) describes the creation of a new object. As we see, the main method of the newly created object is immediately started.

The rule (eq) allows the adversary to compare messages it has received. These comparisons are the only means for the information to flow from the messages the adversary has received to the state of the adversary. In terms of applied pi calculus, if two states are such that all comparisons the adversary can perform give the same result, then these two states are *statically equivalent* [2, Sec. 4.2].

Finally, we again have the (frame)-rule that allows a transition to happen in a larger context. Similarly to the source language, the context may not interfere with the transition.

We denote the set of all labels occurring in Fig. 4 by Act_{imp}.

5 Translation

For our proposed translation, we need some standard cryptographic operations in our implementation language — symmetric encryption, public-key encryption and signatures. Hence we require the signature Σ to contain the binary operations senc, sdec, penc, pdec, sig and vfy. These are related to each other and to the operations $\cdot^+$ and $\cdot^-$ by the following cancellation rules:

$$\begin{aligned} \mathsf{pdec}(x^-, \mathsf{penc}(x^+, m)) &= m \\ \mathsf{sdec}(x, \mathsf{senc}(x, m)) &= m \\ \mathsf{vfy}(x^+, \mathsf{sig}(x^-, m)) &= m, \end{aligned}$$

these equalities hold for all messages x and m. Additionally, we need the pairing operation $(\cdot, \cdot)$ and the projections π_1, π_2 with the cancellation rules $\pi_i((x_1, x_2)) = x_i$. Longer tuples can be modeled as the results of repeated applications of pairing. We again stress that we are working in the symbolic (Dolev-Yao, perfect) model of cryptography.

It is possible to check whether a message m is a pair: in this case, m is equal to $(\pi_1(m), \pi_2(m))$. We need similar checks also for encryption and signatures. Hence we require Σ to contain unary operations is_penc?, is_senc? and is_sig?. These are related to the previous operations by cancellation rules $\mathsf{is}_X?(X(\ldots)) = \mathsf{true}$, where $\mathsf{true} \in \Sigma$ is a nullary operation.

We need more complex data types with associated operations for our translation. These types could be simulated by just using the Msg type, and building up the structure using the pairing operations. It may be hard to ensure the atomicity of operations using just the constructs of the implementation language, though. As mutual exclusion is easy to model in π-calculus, we believe that the addition

of these data types does not change the straightforwardness of translation from our implementation language to applied pi-calculus.

We let Set be a type that has sets of messages as possible values. We let Map be a type that has finite maps from messages to messages as possible values. We have the operations $\emptyset : \mathsf{Set}$, $\{\cdot\} : \mathsf{Msg} \to \mathsf{Set}$, $\in\colon \mathsf{Msg} \times \mathsf{Set} \to \mathsf{Bool}$, $\cup, \cap, \setminus : \mathsf{Set} \times \mathsf{Set} \to \mathsf{Set}$ for sets. We also have the operations for finite maps: $\mathsf{empty} : \mathsf{Map}$, $\cdot[\cdot \mapsto \cdot], \cdot[\cdot \mapsto \cdot]_{\mathrm{w}} : \mathsf{Map} \times \mathsf{Msg}^2 \to \mathsf{Map}$, $\cdot(\cdot) : \mathsf{Map} \times \mathsf{Msg} \to \mathsf{Msg}$, $\mathit{dom} : \mathsf{Map} \to \mathsf{Set}$. Here $\varphi[E \mapsto E']_{\mathrm{w}}$ is the *weak update* operation — the value of $\varphi(E)$ is changed only if it was undefined before. If $\varphi(E)$ is undefined then the application $\varphi(E)$ returns null.

We show how to map each program Pr in the source language to the corresponding program $\{\{Pr\}\}$ in the target language. The translation is largely syntax-directed: to translate a program, one has to translate its classes, classes are translated by translating its methods, a method is translated by translating its body, and control statements in the source language have equivalents in the target language. In the following we present the translation of these parts that are not straightforward.

Object fields In addition to the fields declared in the source program, each object in the implementation language has an additional field φ of type Map. It is initialized to empty and used to store the return values from methods (analogous to futures).

The representation of object and task names is elaborated more in the next paragraphs. Shortly, however: an object is referred to by its public key k^+. Each task is associated with a separate symmetric key K. A reference to a task is stored as a pair (k^+, K) of the keys of the object owning this task, and of this task itself.

The value of the field φ is a finite map that maps pairs of the form (k^+, K) to messages representing the value returned by the task identified by (k^+, K). If $\varphi((k^+, K))$ is undefined then the current object has not yet learned the identity of the task (k^+, K). If $\varphi((k^+, K)) = \mathsf{null}$ then the identity has been learned but the message containing the return value has not been parsed.

Types The types in source and target language are different. We put $\{\{\mathsf{Int}\}\} = \mathsf{Int}$ and $\{\{C\}\} = \{\{\mathsf{Fut}(T)\}\} = \mathsf{Msg}$.

Method invocation An asynchronous call to an object must be replaced with an explicit message. For simplicity, we assume that a call $e!m(\overline{e})$ does not occur as a subexpression in some larger expression. The statement $v := e_0!m(e_1, \ldots, e_n)$ is translated as follows:

$$x_0 := e_0; \cdots ; x_n := e_n; K := \mathsf{newatom}; v := (x_0, K); \varphi := \varphi[v \mapsto \mathsf{null}]; \\ \mathsf{send}(x_0, \mathsf{penc}(x_0, (K, m, x_1, \ldots, x_n))) \quad (1)$$

Here $K, x, x_0, \ldots, x_n$ are temporary variables that are not used elsewhere. After computing the values of the expressions $e_0, \ldots, e_n$, we generate a new atom that

we store in K, and intend to use as a symmetric key. This is the key that is going to identify the task that is created as the result of the method invocation. Note the following interesting aspect — the new task is spawned at the object identified by x_0, but the key identifying it is generated at the calling object. This ensures that the key identifying the new task does not have to be communicated back and the "protocol" for invoking a new method consists of a single message only. In this way the translation from the source to the target language is greatly simplified.

After creating the key K, we define the variable v, set the point in the mapping φ storing the values received from other tasks, and send a message to the object identified by x_0. The message lists the invoked method m, its arguments $x_1, \ldots, x_n$, and the identity of the task K.

Method declaration Compared to the source language, each method of the implementation language receives one additional argument — the symmetric key associated with the task executing that method. The declaration $T\ m(\overline{T\,x})$ is translated into $\{\{T\}\}\ m(\mathsf{Msg}\,K, \overline{\{\{T\}\}\,x})$.

Returning a value A value returned by a task may be read by anyone knowing the key associated with that task. The reader must be sure of the identity of the object returning the value. The return of the value of the variable v at the end of a method is translated as

$$\mathsf{while}(\mathsf{true})\ \{x := \mathsf{newatom}; \mathsf{send}(\mathsf{null}, (k^+, \mathsf{sig}(k^-, \mathsf{senc}(K, (k^+, v, x)))))\} \quad (2)$$

Here k^+ and k^- are the fields of the object owning this task, containing this object's public and private keys. The variable K contains the symmetric key associated with this task. We see that the finished task sends out an unbounded number of messages, all different, containing the return value. These messages are receivable by any object.

Getting a returned value $\{\{v := e.\mathsf{get}\}\}$ is defined as

$$x := e; \varphi := \varphi[x \mapsto \mathsf{null}]_{\mathrm{w}}; \mathsf{while}(\varphi(x) = \mathsf{null})\{\mathsf{skip}\}; v := \varphi(x)$$

(for simplicity we assume that $e.\mathsf{get}$ does not occur as a subexpression).

The main method of an object Whenever a new object is created, its main method is executed. In our translation, this method is responsible for receiving the messages and acting on them. This method is the same for all classes, only the list of methods it has to consider is different. The method has a local variable ψ of type Set (initially $\emptyset$) that contains the messages that may contain returned values that this object is as of yet incapable of decrypting. It also has a local variable $\mathcal{K}$ of type Set (initially $\emptyset$) that contains the symmetric keys associated with the tasks of this object.

The main method executes an infinite loop that performs the following operations.

– Receive a message from the network, store it in the variable r.
– Let $mc := \mathsf{pdec}(k^-, r)$. If the second component m of mc is equal to the name of some method of class C, and the first component K is not contained in $\mathcal{K}$, then let $v_1, \ldots, v_n$ be the 3rd, 4th, etc. component of mc. Add K to $\mathcal{K}$ and execute $\mathsf{spawn}\ m(K, v_1, \ldots, v_n)$.
– Otherwise we handle r as a returned value. Let $s := \pi_1(r)$ and $vm := \mathsf{vfy}(s, \pi_2(r))$. If the pair (s, vm) is not yet an element of ψ, then add it there.
– For each element (s, vm) of ψ and each (k^+, K) in the domain of φ, such that $\varphi((k^+, K)) = \mathsf{null}$ and $s = k^+$: let $dm = \mathsf{sdec}(K, vm)$. If $\pi_1(dm) = k^+$, then set φ to $\varphi[(k^+, K) \mapsto \pi_2(dm)]$ and remove (s, vm) from ψ. (this step also requires *iterators* over Set and Map)

6 Equivalence

The semantics of programs in both source and implementation language are given as labeled transition systems, albeit with different set of labels. An *adversary* is any other LTS running in parallel and *synchronizing* on (a subset of) these labels. As the labels are different, the possible adversaries are also different and we cannot compare the LTS-s $[\![Pr]\!]$ and $[\![\{\{Pr\}\}]\!]$ directly. Still, we show that for each program Pr in the source language, $[\![Pr]\!]$ can be translated to one equivalent to $[\![\{\{Pr\}\}]\!]$ by running a suitable LTS $\overrightarrow{\mathcal{L}}$ in parallel with it. *Vice versa*, $[\![\{\{Pr\}\}]\!]$ can be translated to one equivalent to $[\![Pr]\!]$ with the help of some LTS $\overleftarrow{\mathcal{L}}$.

Thus, the LTS-s $[\![Pr]\!]$ and $[\![\{\{Pr\}\}]\!]$ satisfy exactly the same security properties. Indeed, if there were an adversary $\mathcal{A}$ demonstrating the violation of the property $\mathcal{P}$ of the translated program $\{\{Pr\}\}$ (i.e. the parallel composition of $[\![\{\{Pr\}\}]\!]$ and $\mathcal{A}$ does not have the property $\mathcal{P}$) then the property $\mathcal{P}$ is also violated for the source program Pr and the parallel composition of $\overleftarrow{\mathcal{L}}$ and $\mathcal{A}$ is the adversary demonstrating this. Similarly, any adversary violating a security property for Pr can be transformed to an adversary violating the same property for $\{\{Pr\}\}$. The satisfaction of exactly the same security properties means that if we have succeeded to prove that Pr satisfies some security property, then the equivalence result of this section allows us to deduce that $\{\{Pr\}\}$ also satisfies this property.

The formalization of the claims above requires more definitions. Let $\mathcal{L} = (S, A, \rightarrow, s_0)$ be a LTS. A symmetric relation $\mathcal{R}$ on S is a *branching bisimulation* if for all $s, s', t \in S$ and $\alpha \in A$, $s \xrightarrow{\alpha} s'$ and $s\ \mathcal{R}\ t$ implies either $\alpha = \tau$ and $s'\ \mathcal{R}\ t$, or the existence of $t_1, t_2, t' \in S$, such that $s \xrightarrow{\tau}{}^* t_1 \xrightarrow{\alpha} t_2 \xrightarrow{\tau}{}^* t'$, $s\ \mathcal{R}\ t_1$, $s'\ \mathcal{R}\ t_2$, and $s'\ \mathcal{R}\ t'$ [30]. Two states of an LTS are *(branching) bisimilar* if they are related by a branching bisimulation. Two LTS-s $\mathcal{L}_1$ and $\mathcal{L}_2$ are bisimilar (denoted $\mathcal{L}_1 \approx \mathcal{L}_2$) if in their disjoint union, their starting states are bisimilar.

Let $\mathcal{L} = (S, A, \rightarrow, s_0)$ and $\mathcal{L}' = (S', A', \rightarrow', s'_0)$ be two LTS-s. Let $B \subseteq (A \cup A') \backslash \{\tau\}$. The *parallel composition* of $\mathcal{L}$ and $\mathcal{L}'$ *synchronized on* B is a LTS $\mathcal{L} \times_B \mathcal{L}' = (S \times S', (A \cup A') \backslash B, \Rightarrow, (s_0, s'_0))$ where a transition $(s, s') \overset{\alpha}{\Rightarrow} (t, t')$ exists iff one of the following holds:

– $\alpha \in A$, $s \xrightarrow{\alpha} t$ and $s' = t'$;
– $\alpha \in A'$, $s' \xrightarrow{\alpha}{}' t'$ and $s = t$;
– $\alpha = \tau$ and exists $\beta \in B$, such that $s \xrightarrow{\beta} t$ and $s' \xrightarrow{\beta}{}' t'$.

We can now state the correctness result for our translation. Informally, it states that no behavior of a program Pr is lost, and no new behavior introduced during the translation, as long as these behaviors can be described through branching bisimulation. Hence, if we consider Pr to have some desirable properties, then $\{\{Pr\}\}$ has the same properties.

Theorem 1. *For every program Pr in the source language there exist LTS-s $\overrightarrow{\mathcal{L}}$ and $\overleftarrow{\mathcal{L}}$, such that*

$$[\![Pr]\!] \times_{Act_{\mathsf{src}}} \overrightarrow{\mathcal{L}} \approx [\![\{\{Pr\}\}]\!]$$
$$[\![\{\{Pr\}\}]\!] \times_{Act_{\mathsf{imp}}} \overleftarrow{\mathcal{L}} \approx [\![Pr]\!] \ .$$

As an immediate corollary we get an equivalence result similar to [3].

Corollary 1. *Let Pr_1 and Pr_2 be two programs in the source language. Then $[\![Pr_1]\!] \approx [\![Pr_2]\!]$ iff $[\![\{\{Pr_1\}\}]\!] \approx [\![\{\{Pr_2\}\}]\!]$.*

Proof. Let $[\![Pr_1]\!] \approx [\![Pr_2]\!]$. Then $[\![\{\{Pr_1\}\}]\!] \approx [\![Pr_1]\!] \times_{Act_{\mathsf{src}}} \overrightarrow{\mathcal{L}} \approx [\![Pr_2]\!] \times_{Act_{\mathsf{src}}} \overrightarrow{\mathcal{L}} \approx [\![\{\{Pr_2\}\}]\!]$. The other direction is analogous. □

The rest of this section is devoted to proving Theorem 1.

6.1 The LTS $\overrightarrow{\mathcal{L}}$

The LTS $\overrightarrow{\mathcal{L}}$ must "translate" the actions of a program Pr in the source language to the actions in Act_{imp}. Also, the environment-initiated actions (belonging to Act_{imp}) must be translated back to the actions in Act_{src} that the program Pr understands.

One may want to specify $\overrightarrow{\mathcal{L}}$ in some programming language. The semantics of this language would then give us an LTS. But only the actual LTS matters for the purposes of the proof, hence we specify it without the detour thorough a programming language. At the same time, we may still think of $\overrightarrow{\mathcal{L}}$ as a machine, executing a program, and having memory. All possible contents of the memory would then be mapped to different states of the LTS. In particular, the LTS $\overrightarrow{\mathcal{L}}$ keeps the following records in the memory:

– A table T that matches the object and task names in source language semantics with the keys in target language semantics. Each entry in this table has the fields *msg* (a cryptographic message denoting the identity of an object or a task, similarly to our translation into the implementation language), *name* (a name in **N** for that object/task in the source language), *type* (a boolean showing whether this entry is for an object or a task name), and *local* (a

boolean showing whether this object/task is in the runtime configuration of Pr). If the entry is an object then there is also the field sk for the secret key of that object (in the implementation language). Recall that an object is identified by its public key and a task by a pair of its object's public key and its own symmetric key.
- A (finite) map $\mathbf{P}$ from variables to cryptographic messages, having the same role as the $\mathsf{p}(x \backslash E)$ components in the runtime configurations of the implementation language.
- A set **Ret** of pairs (o, n) of object and task names. A pair belongs to this set if the object o in the configuration of the source language has received the return value of the task n that does not belong in that configuration.

With these records, the translation between the source program and the environment for programs in the implementation language is straightforward. If Pr performs a transition labeled with $o \to o'$, then find the keys k and k' corresponding to o and o' from the table T (if they're not there then generate new name[1] k / k' and add new rows to T), let $E = \mathsf{penc}(k', dummy)$, generate new variables x_1, x_2, x_3, add new bindings $x_1 \mapsto k$, $x_2 \mapsto k'$ and $x_3 \mapsto E$ to $\mathbf{P}$, and perform a transition with the label $x_1 \to x_2 : x_3$. If Pr performs a transition labeled $o \to n'[o'.m(\overline{c})]$ then $\overrightarrow{\mathcal{L}}$ similarly finds the keys k and k' of o and o'. It will also translate the names of any objects and futures mentioned in $\overline{c}$ to the cryptographic messages denoting them, generating new names in the process as necessary. As next, $\overrightarrow{\mathcal{L}}$ generates a new name K that will be used as the symmetric key of the new task. It adds K and n' into a new row of the table T. Finally, it prepares a message E as in (1), generates new variables x_1, x_2, x_3, binds k, k', E to them in $\mathbf{P}$, and invokes the transition labeled $x_1 \to x_2 : x_3$. If keys corresponding to new objects have to be created, then the public key is stored in the field msg and the secret key in the field sk of a new row in the table T.

If $[\![Pr]\!]$ performs a transition labeled $o' \leftarrow o$ then $\overrightarrow{\mathcal{L}}$ finds k and k' denoting these objects, finds the secret key $\tilde{k}$ of the object o from the table T, constructs a message $E = (k, \mathsf{sig}(\tilde{k}, \mathsf{senc}(dummy_1, dummy_2)))$, and indicates the message send as before (performing the transition $x_1 \to x_2 : x_3$ with x_1, x_2, x_3 bound to k, k', E). The translation of the $n[c] \leftarrow o$ action by $[\![Pr]\!]$ consists of finding the messages k and K denoting the object and task from the table T, translating c (if it is an object/task name), constructing a message E similarly to (2), *nondeterministically* selecting the recipient k' of the message among the non-local objects in T, and transmitting E from k to k'.

If the environment performs an action $\to k : E(x_1, \ldots, x_l)$ then $\overrightarrow{\mathcal{L}}$ parses the message $\mathbf{P}(E)$ (note that $\overrightarrow{\mathcal{L}}$ has necessary secret keys for that). If it is an invocation of method m (as in (1)) and no task with the same symmetric key K for the object identified by k_2 has been invoked before, then $\overrightarrow{\mathcal{L}}$ performs the action $n = o!m(\overline{c})$ where $\overline{c}$ is obtained by parsing $\mathbf{P}(E)$ and translating with the help of T. If some object/task names are missing in T, then these are generated

[1] The generation of new names is an internal action of $\overrightarrow{\mathcal{L}}$

with actions νn and νo. Also, $\overrightarrow{\mathcal{L}}$ adds a new row with n and K to T. Similar checks and translations are performed if $\mathbf{P}(E)$ is the return value of some task according to (2). In this case, the set **Ret** is additionally consulted and updated.

The LTS $\overrightarrow{\mathcal{L}}$ always has the transitions $E \triangle E'$ enabled. Here $\triangle$ is either $=$ or $\neq$, depending on whether $\mathbf{P}(E) = \mathbf{P}(E')$ or not. The transition νx is also always enabled. When it is invoked, $\overrightarrow{\mathcal{L}}$ creates a new name and updates $\mathbf{P}$.

Whenever $\overrightarrow{\mathcal{L}}$ "translates" an action in Act_{src} to an action in Act_{imp} or *vice versa*, it first performs the transition with the label in Act_{imp} (non-deterministically guessing some parameters, if necessary), and afterwards the transition with the label in Act_{src}. This keeps the branching behaviors of $[\![Pr]\!] \times_{Act_{\mathsf{src}}} \overrightarrow{\mathcal{L}}$ and $[\![\{\{Pr\}\}]\!]$ the same.

6.2 The LTS $\overleftarrow{\mathcal{L}}$

The LTS $\overleftarrow{\mathcal{L}}$ translates the actions of a program in the implementation language to the actions in Act_{src}. We do not have to consider all programs in the implementation language, but only those of the form $\{\{Pr\}\}$, where Pr is a program in the source language.

Internally, $\overleftarrow{\mathcal{L}}$ keeps the following records.

- The table T^{o} relates the cryptographic messages with object names in $\mathbf{N}$ (note that creating new names in $\mathbf{N}$ is under the control of $\overleftarrow{\mathcal{L}}$). Each entry in this table has the field *name* for storing the name $o \in \mathbf{N}$. It also has the field *msg* that contains the cryptographic expression (with variables) that evaluates to the public key of that object if these variables are substituted with the messages associated with them by the runtime configuration of $\{\{Pr\}\}$ (through the $\mathsf{p}(x\backslash E)$-parts).
- The table T^{myo} relates the object identities and names for those objects that $\overleftarrow{\mathcal{L}}$ itself has caused to be created. It has the fields *var* and *name* with the latter having the same meaning as in T^{o}. The field *var* contains the variable whose value is an atomic message, such that var^+ is the public key and var^- the secret key of the object.
- The table T^{t} relates the cryptographic messages with task names in $\mathbf{N}$. Each entry in this table has the fields *name*, msg^{t} and msg^{o}. The two *msg*-fields contain cryptographic expressions that resolve to the symmetric key identifying the task, and public key of the owning object, respectively.
- The table **Ret** having the same role as in $\overrightarrow{\mathcal{L}}$.

The transitions of $[\![\{\{Pr\}\}]\!]$ and the environment (which makes transitions labeled with elements of Act_{src}) are translated as follows. If $\{\{Pr\}\}$ performs a transition labeled with $k_1 \to k_2 : x$ then $\overleftarrow{\mathcal{L}}$ finds the objects o_1 and o_2 corresponding to the keys k_1 and k_2 by comparing k_i with $R.msg$ for all entries R in T^{o}. The comparison is performed with the help of $[\![\{\{Pr\}\}]\!]$, by attempting the transitions labeled with $k_i \triangle R.msg$ for $\triangle \in \{=, \neq\}$. If T^{o} does not contain k_1 or k_2 then a new entry is added to T^{o} with a new name from $\mathbf{N}$. If o_2 is not

in T^{myo} then this message is between two objects in the runtime configuration of $\{\{Pr\}\}$. It will be translated either as $o_1 \to o_2$ or $o_2 \leftarrow o_1$, depending on the structure of the message pointed to by x.

If o_2 is in T^{myo}, then the message x is from an object in the runtime configuration of $\{\{Pr\}\}$ to an object outside it. If it is a method call then $\overleftarrow{\mathcal{L}}$ can decrypt the message x because it has the secret key corresponding to k_2. It finds the method m to be called, the symmetric key K identifying the task that the object o_2 is supposed to spawn, as well as the arguments $\overline{c}$. The arguments can be decoded with the help of the tables T^{o} and T^{t}. The LTS $\overleftarrow{\mathcal{L}}$ will generate a new task name n, add a new entry (n, K, k_2) to the table T^{t} and perform the transition labeled with $o_1 \to n[o_2.m(\overline{c_{decoded}})]$. Similarly, if x is the return of a value then it can be decrypted because the symmetric key K of the task that returned it can be found in T^{t} (see below for handling method calls coming from the environment). The task name n is found from the entry containing K and k_1, the returned value c is decoded and the transition $n[c_{decoded}] \leftarrow o_1$ is performed.

If the environment performs a transition labeled with $n = o!m(\overline{c})$ then $\overleftarrow{\mathcal{L}}$ will add a new entry to the table T^{t} with n, the public key k of the object o (found from T^{o}) and a new symmetric key K, obtained by performing a transition labeled with νx. It will then encode the arguments $\overline{c}$ with the help of the tables T^{o} and T^{t}. If some of the arguments are missing from these tables then new entries are added, obtaining the necessary keys through transitions labeled with νx. The LTS $\overleftarrow{\mathcal{L}}$ will then construct a new message E in the shape of (1) and perform a transition labeled with $\to k : E$.

If the environment performs a transition labeled with $o \leftarrow n[c]$ with o being found in T^{o} and n in T^{t}, and the pair (n, o) not belonging to **Ret** (if some of these conditions do not hold then $\overleftarrow{\mathcal{L}}$ will have no such transition for the environment to synchronize upon) then $\overleftarrow{\mathcal{L}}$ finds from T^{t} the symmetric key k^+ of the object having executed the task n. The object was running in the environment, thus its secret key can be found from T^{myo}. The LTS $\overleftarrow{\mathcal{L}}$ will now construct a message E according to (2) and perform a transition labeled with $\to k_o : E$, where k_o is the public key of o found from T^{o}.

If the environment performs a transition labeled with νo then $\overleftarrow{\mathcal{L}}$ will obtain a new key k by performing a transition labeled with νx, add (k, o) to the table T^{myo} and (o, k^+) to the table T^{o}. If the environment performs a transition labeled with νn then $\overleftarrow{\mathcal{L}}$ will obtain new keys K and k by performing two transitions labeled with νx. It will generate a new name $o \in \mathbf{N}$, add the entry (k, o) to T^{myo}, the entry (o, k^+) to T^{o} and (n, K, k^+) to T^{t}. This is a valid translation for names of tasks controlled *and initiated* by the environment because a program Pr has no means to make sure whether two of such tasks are executed by the same object or by different objects.

Similarly to $\overrightarrow{\mathcal{L}}$, whenever $\overleftarrow{\mathcal{L}}$ "translates" an action in Act_{imp} to an action in Act_{src} or *vice versa*, it first performs the transition with the label in Act_{src} and afterwards the transition with the label in Act_{imp}.

6.3 Bisimulations

The reachable states of the LTS $[\![Pr]\!] \times_{Act_{\mathsf{src}}} \overrightarrow{\mathcal{L}}$ satisfy invariants stating that the runtime configuration C of $[\![Pr]\!]$ and the data (T, $\mathbf{P}$, $\mathbf{Ret}$) kept by $\overrightarrow{\mathcal{L}}$ are consistent. Namely, T contains an entry about the object o or task n iff $\mathsf{p}(o) \in C$ or $\mathsf{p}(n) \in C$. The values of the fields *local* and *type* match with the type and locality of the elements in C. A pair (o, n) is in $\mathbf{Ret}$ iff $o.\varphi(n)$ is defined. The data kept by $\overrightarrow{\mathcal{L}}$ provides an easy translation between the states S of $[\![Pr]\!] \times_{Act_{\mathsf{src}}} \overrightarrow{\mathcal{L}}$ and the states C' of $[\![\{\!\{Pr\}\!\}]\!]$, giving us a bisimulation between these LTSs. Roughly, the states $S = (C, T, \mathbf{P}, \mathbf{Ret})$ and C' are related iff there exists a permutation θ of $\mathcal{A}$, such that

- C and C' have the same objects with the same names (public keys), where the translation between object names and public keys is given by T and θ. For an object name o, if T contains an entry R with $R.name = o$, then the object $o[\ldots] \in C$ corresponds to the object $(T.msg)\theta[\ldots] \in C'$. Objects $o[\ldots] \in C$ with names o not occurring in T correspond to objects $k^+[\ldots] \in C'$ with $k^+(\theta^{-1})$ not occurring in T in arbitrary one-to-one manner. Two corresponding objects must have the same class and the same values for their fields. For fields of type "object" or "task", the translation is again given through T and θ. The component φ of an object in C must match with the union of the field φ of the corresponding object in C', and the local variable ψ in this object's task executing the main method.
- C and C' have the same tasks (except for the tasks executing the main methods of objects in C'). Again, the translation between task names and symmetric keys is given by T and θ with tasks not present in T mapped to each other. A task's symmetric key is stored in its first input parameter. The local variables of the corresponding tasks have the same values, and the execution is at the same program point.
- The mapping $\mathbf{P}$ matches the elements $\mathsf{p}(x\backslash E)$ in C': for some $x \in \mathcal{X}$, $\mathbf{P}(x)$ is defined iff $p(x\backslash E) \in C'$ for some E. Moreover, $\mathbf{P}(x)\theta = E$.

The use of the permutation θ is justified by the fact that actual atomic messages in $\mathcal{A}$ are inaccessible to environments synchronizing over Act_{imp}. This avoids the problem that keys $k \in \mathcal{A}$ identifying objects are created at different times in $[\![\{\!\{Pr\}\!\}]\!]$ and $[\![Pr]\!] \times_{Act_{\mathsf{src}}} \overrightarrow{\mathcal{L}}$ (time of creation vs. time of becoming public). These identifiers are accessed through variables which are created at the same time in both LTSs (time of becoming public).

The bisimulation between $[\![\{\!\{Pr\}\!\}]\!] \times_{Act_{\mathsf{imp}}} \overleftarrow{\mathcal{L}}$ and $[\![Pr]\!]$ is similar. Again, the reachable states of $[\![\{\!\{Pr\}\!\}]\!] \times_{Act_{\mathsf{imp}}} \overleftarrow{\mathcal{L}}$ satisfy consistency invariants: if C' is the runtime configuration of $[\![\{\!\{Pr\}\!\}]\!]$ and $(T^{\mathrm{o}}, T^{\mathrm{myo}}, T^{\mathrm{t}}, \mathbf{Ret})$ is the internal state of $\overleftarrow{\mathcal{L}}$, then an entry R in T^{o} means that an object with the public key $R.msg$ is a component of C', unless $R.msg$ also occurs in T^{myo}. The states $S' = (C', T^{\mathrm{o}}, T^{\mathrm{myo}}, T^{\mathrm{t}}, \mathbf{Ret})$ and C are related, if

- C' and C have the same objects, where the translation between object names and public keys is given by T^{o}. These objects must be equivalent — have

the same class and values of fields, including the component/field φ and the local variable ψ of the main method of the object. The objects in C' whose public keys are not present in T^{o} are mapped to the objects o in C where $\mathsf{p}(o)$ is not part of the configuration C.
- C and S' know the same objects in the environment. For each row R in T^{myo} there is a component $\mathsf{p}(R.name)$ in C without the object $R.name[\ldots]$ itself being a component of C. Similarly, if there exists an object name o, such that $\mathsf{p}(o)$ is part of C, but $o[\ldots]$ is not, then there is a row R of T^{myo}, such that $R.name = o$.
- C' and C have the same tasks (except for the tasks executing the main methods of objects in C'), where the translation between task names and symmetric keys is given through T^{t}. Tasks in C with names not occurring in T^{t} are matched with tasks in C' with keys not occurring in T^{t}. If there is an entry (n, K, k^+) in T^{t}, but there is no task K of the object k^+ running in C', then there is also no task $n\langle\ldots\rangle$ in C, but $\mathsf{p}(n)$ is a component of C.

7 Conclusions

We have shown how to translate a distributed object-oriented language arising in the programming language and formal verification community to (a dialect of) applied pi calculus, such that the security properties of programs are preserved. Compared to other similar results, our source language does not so explicitly include communication between processes in different locations. Rather, the communication happens when needed to execute the language constructs. In this sense, the gap our translation has to cross is larger. On the other hand, this may also simplify the translation because it somewhat restricts the possible communication patterns. Still, the "channels" (object references and futures) can be freely communicated and this requires some non-obvious tricks to securely translate.

Besides the security, other aspects of implementing ABS are also worthy of studying. In parallel to our work, the *routing* of messages (which is made highly non-trivial by the fact that futures are first-class values and a finishing task does not know where its result is needed), as well as the mobility of objects is being studied by Dam and Palmskog [14].

References

1. Martín Abadi. Protection in programming-language translations. In Kim Guldstrand Larsen, Sven Skyum, and Glynn Winskel, editors, *ICALP*, volume 1443 of *Lecture Notes in Computer Science*, pages 868–883. Springer, 1998.
2. Martín Abadi and Cédric Fournet. Mobile values, new names, and secure communication. In *POPL*, pages 104–115, 2001.
3. Martín Abadi, Cédric Fournet, and Georges Gonthier. Secure Implementation of Channel Abstractions. In *LICS*, pages 105–116. IEEE Computer Society, 1998.

4. Martín Abadi, Cédric Fournet, and Georges Gonthier. Secure Communications Processing for Distributed Languages. In *IEEE Symposium on Security and Privacy*, pages 74–88, 1999.
5. Martín Abadi, Cédric Fournet, and Georges Gonthier. Authentication Primitives and Their Compilation. In *POPL*, pages 302–315, 2000.
6. Erika Ábrahám, Immo Grabe, Andreas Grüner, and Martin Steffen. Behavioral interface description of an object-oriented language with futures and promises. *J. Log. Algebr. Program.*, 78(7):491–518, 2009.
7. Pedro Adão and Cédric Fournet. Cryptographically sound implementations for communicating processes. In Michele Bugliesi, Bart Preneel, Vladimiro Sassone, and Ingo Wegener, editors, *ICALP (2)*, volume 4052 of *Lecture Notes in Computer Science*, pages 83–94. Springer, 2006.
8. Michael Backes, Agostino Cortesi, Riccardo Focardi, and Matteo Maffei. A calculus of challenges and responses. In Peng Ning, Vijay Atluri, Virgil D. Gligor, and Heiko Mantel, editors, *FMSE*, pages 51–60. ACM, 2007.
9. Michele Bugliesi and Riccardo Focardi. Language based secure communication. In *CSF*, pages 3–16. IEEE Computer Society, 2008.
10. Michele Bugliesi and Marco Giunti. Secure implementations of typed channel abstractions. In Martin Hofmann and Matthias Felleisen, editors, *POPL*, pages 251–262. ACM, 2007.
11. Dave Clarke, Nikolay Diakov, Reiner Hähnle, Einar Broch Johnsen, Germán Puebla, Balthasar Weitzel, and Peter Y. H. Wong. Hats - a formal software product line engineering methodology. In Goetz Botterweck, Stan Jarzabek, Tomoji Kishi, Jaejoon Lee, and Steve Livengood, editors, *SPLC Workshops*, pages 121–128. Lancaster University, 2010.
12. Dave Clarke, Nikolay Diakov, Reiner Hähnle, Einar Broch Johnsen, Ina Schaefer, Jan Schäfer, Rudolf Schlatte, and Peter Y. H. Wong. Modeling spatial and temporal variability with the hats abstract behavioral modeling language. In Marco Bernardo and Valérie Issarny, editors, *SFM*, volume 6659 of *Lecture Notes in Computer Science*, pages 417–457. Springer, 2011.
13. Hubert Comon-Lundh and Véronique Cortier. Computational soundness of observational equivalence. In Peng Ning, Paul F. Syverson, and Somesh Jha, editors, *ACM Conference on Computer and Communications Security*, pages 109–118. ACM, 2008.
14. Mads Dam and Karl Palmskog. A Foundation for Network-Adaptive Execution of Distributed Objects, 2012. Work in progress.
15. Danny Dolev and Andrew Chi-Chih Yao. On the Security of Public Key Protocols. *IEEE Transactions on Information Theory*, 29(2):198–207, 1983.
16. Cédric Fournet and Georges Gonthier. The reflexive chemical abstract machine and the join-calculus. In *POPL*, pages 372–385, 1996.
17. Cédric Fournet and Tamara Rezk. Cryptographically Sound Implementations for Typed Information-Flow Security. In *POPL 2008, Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, San Francisco, California, January 2008. ACM Press.
18. Shafi Goldwasser and Silvio Micali. Probabilistic Encryption. *Journal of Computer and System Sciences*, 28(2):270–299, April 1984.
19. Reiner Hähnle. Hats: Highly adaptable and trustworthy software using formal methods. In Tiziana Margaria and Bernhard Steffen, editors, *ISoLA (2)*, volume 6416 of *Lecture Notes in Computer Science*, pages 3–8. Springer, 2010.

20. Reiner Hähnle, Einar Broch Johnsen, Bjarte M. Østvold, Jan Schäfer, Martin Steffen, and Arild B. Torjusen. Report on the Core ABS Language and Methodology: Part A. Highly Adaptable and Trustworthy Software using Formal Models (HATS), Deliverable D1.1A, April 2010.
21. Michiel Helvensteijn, Radu Muschevici, and Peter Y. H. Wong. Delta modeling in practice: a Fredhopper case study. In Ulrich W. Eisenecker, Sven Apel, and Stefania Gnesi, editors, *VaMoS*, pages 139–148. ACM, 2012.
22. Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin Steffen. Abs: A core language for abstract behavioral specification. In Bernhard K. Aichernig, Frank S. de Boer, and Marcello M. Bonsangue, editors, *FMCO*, volume 6957 of *Lecture Notes in Computer Science*, pages 142–164. Springer, 2010.
23. Einar Broch Johnsen, Olaf Owe, and Ingrid Chieh Yu. Creol: A type-safe object-oriented model for distributed concurrent systems. *Theoretical Computer Science*, 365(1-2):23–66, 2006.
24. Cliff B. Jones. A pi-calculus semantics for an object-based design notation. In Eike Best, editor, *CONCUR*, volume 715 of *Lecture Notes in Computer Science*, pages 158–172. Springer, 1993.
25. Robin Milner. *Communicating and mobile systems: the π-calculus.* Cambridge University Press, 1999.
26. Sebastian Mödersheim and Luca Viganò. Secure pseudonymous channels. In Michael Backes and Peng Ning, editors, *ESORICS*, volume 5789 of *Lecture Notes in Computer Science*, pages 337–354. Springer, 2009.
27. Michael Philippsen. A survey of concurrent object-oriented languages. *Concurrency: Practice and Experience*, 12(10):917–980, 2000.
28. Ina Schaefer and Reiner Hähnle. Formal methods in software product line engineering. *IEEE Computer*, 44(2):82–85, 2011.
29. Jean-Guy Schneider and Markus Lumpe. Synchronizing Concurrent Objects in the π-Calculus. In Roland Ducournau and Serge Garlatti, editors, *Proceedings of Languages et Modèles à Objets*, pages 61–76. Hermes, Roscoff, 1997.
30. Rob J. van Glabbeek and W. Peter Weijland. Branching Time and Abstraction in Bisimulation Semantics. *Journal of the ACM*, 43(3):555–600, 1996.
31. Jeffrey A. Vaughan and Steve Zdancewic. A cryptographic decentralized label model. In *IEEE Symposium on Security and Privacy*, pages 192–206. IEEE Computer Society, 2007.