On the Development and Formalization of an Extensible Code Generator for Real Life Security Protocols

Michael Backes[1,2], Alex Busenius[1], Cătălin Hriţcu[1,3]

[1]Saarland University [2]MPI-SWS [3]University of Pennsylvania

Abstract
This paper introduces Expi2Java, a new code generator for cryptographic protocols that translates models written in an extensible variant of the Spi calculus into executable code in a substantial fragment of Java, featuring concurrency, synchronization between threads, exception handling and a sophisticated type system with generics and wildcards. Our code generator is highly extensible and customizable, which allows it to generate interoperable implementations of complex real life protocols from detailed verified specifications. As a case study, we have generated an interoperable implementation of TLS v1.0 client and server from a protocol model verified with ProVerif. Furthermore, we have formalized the translation algorithm of Expi2Java using the Coq proof assistant, and proved that the generated programs are well-typed if the original models are well-typed. This constitutes an important step towards the first machine-checked correctness proof of a code generator for cryptographic protocols.

1 Introduction

Implementing cryptographic protocols is a difficult and notoriously error-prone task, where even the smallest error can cause very serious security vulnerabilities[1]. One way to prevent many such vulnerabilities is to model the protocol in a process calculus [4], check the security of the model [3], and then automatically generate a secure implementation from the protocol model. Automatic tools exist that can generate protocol implementations starting from such verified models together with configuration information that allows them to produce code that is interoperable with other implementations of the same protocol.

This paper introduces Expi2Java[2], a new tool that brings code generators for security protocols even closer to reality. Expi2Java is highly extensible and customizable: The user can easily add new cryptographic primitives, configure all the relevant parameters, customize implementation classes, and even customize the code generation process by editing the provided templates. Commonly used cryptographic primitives and data types are supported out of the box, and the user has full control not only over the high-level design of the protocol, but also over all the low-level details, including the precise bit-level representation of messages. To illustrate the flexibility of our tool we have generated an interoperable implementation of TLS v1.0 client and server from a protocol specification verified with ProVerif [3,9]. TLS is orders of magnitude more complex and sophisticated than the protocols used by the previous code generation experiments.

[1] For example, see the security advisories of OpenSSL (`http://www.openssl.org/news/`), a well-known implementation of various SSL/TLS versions.

[2] `http://www.infsec.cs.uni-saarland.de/projects/expi2java`

The types in the generated Java code cannot be synthesized out of thin air, so we ask the user to provide type annotations while writing the protocol model. Our code generator uses the type information in the model to generate the typing annotations needed by the Java type-checker. Additionally, it is important to detect all the typing errors as soon as possible, before the code generation phase, so that we can guide the user with helpful error messages. Our tool uses a type-checker with variant parametric types and subtyping to prevent the incorrect usage of cryptographic primitives in the protocol model. This source-level type-checker rejects models that could lead to ill-typed Java code early on, and produces errors in terms of the model the user has actually written, not in terms of the automatically generated code that the user would have a hard time understanding. Moreover, our source-level type-checker performs type inference, which greatly decreases the annotation burden on the user.

All the features that make Expi2Java a usable and useful tool in practice come at a price though: the tool is rather complex; it currently has more than 16.000 lines of Java code, and this code needs to be trusted to generate correct implementations that preserve the security of the original protocol models. In this paper we take the first step towards formally bridging the gap between secure protocol models and automatically generated protocol implementations: We provide the first mechanized formalization of a code generator for cryptographic protocols. And we do this for an idealized code generator that is fairly close to the core of our Expi2Java tool, without "sweeping under the carpet" the interesting details. We formalize the semantics of both the source language of the code generator, an extensible and customizable variant of the Spi calculus, and the target language, which is a substantial fragment of the Java programming language featuring concurrency, synchronization between threads, exception handling and a sophisticated type system with generics and wildcards. We formally prove that our translation generates well-typed programs when starting from well-typed protocol models. This confirms the validity of our source-level type-checker, and constitutes an important step towards the longer-term goal of proving the correctness of the translation.

Outline §2 discusses related work. §3 introduces the features and workflow of our tool while §4 reports on the TLS case study. §5 presents our source language, the Extensible Spi calculus. §6 describes the formalization of our target language, Variant Parametric Jinja. In §7 we give a high-level overview of our translation. In §8 we discuss our formalization, proofs, and some of the lessons we have learned. Finally, §9 discusses directions for future work and concludes. The implementation and documentation of Expi2Java are available online[3], together with the Coq formalization and proofs. The details that had to be omitted in this paper due to the lack of space are available in Appendix A.

2 Related Work

The idea of a code generator for protocol implementations is not new, and several such tools were developed in the past. One of the early approaches is the AGVI toolkit [21] that uses the Athena protocol analyzer, and can generate Java implementations. The CIL2Java tool [20] can also generate Java code from the CAPSL intermediate language CIL. SPEAR II [18] is a tool aimed at rapid protocol engineering. It has a graphical user interface and generates Java code that uses the ASN.1 standard for data encoding and various cryptographic libraries. The Sprite tool [25] translates simple protocols written in

[3] `http://www.infsec.cs.uni-saarland.de/projects/expi2java`

"standard notation" to the Spi calculus from which it can generate Java code. The CPPL compiler [19] generates OCaml code that can be deployed on stock Web servers.

Spi2Java [22,23] is the first code generator that attempts to be flexible and configurable. It uses the original Spi calculus [4] as the input language together with a very simple type system, and provides a (quite primitive) way to specify the low-level information needed for generating interoperable protocol implementations. The generated implementation of a simple SSH client [22] demonstrates that interoperability with standard implementations is indeed achievable. Nevertheless, the practicality of Spi2Java is quite limited: its customization mechanism is very hard to use, and requires manually editing huge highly-redundant XML files for associating configuration information to individual terms. Also, Spi2Java cannot handle all the cryptographic primitives involved in really complex protocols like TLS, and has no extension mechanism to circumvent this problem — this lead us to develop Expi2Java, initially as an extension of Spi2Java, but rewritten by now. Moreover, none of the existing tools has a thorough formalization of their translation or a proof that their translation preserves any interesting property of the original model.

We have based our formalization of the target language on Jinja with Threads [16, 17], since, to the best of our knowledge, this is the most complete and thorough formalization of Java that supports concurrency. In order to be able to use Jinja with our Coq formalization of the Extensible Spi calculus, we have manually translated the formalization of Jinja with Threads from Isabelle/HOL to Coq. We have extended Jinja with Threads with a type system based on the variant parametric types needed to express Java 5.0 generics and wildcards [26], which are pervasively used by our code generator. This very expressive type system was first introduced by Igarashi et al. [14] as an extension of Featherweight GJ, itself an extension of Featherweight Java [13].

3 Expi2Java: An Extensible Code Generator for Security Protocols

Expi2Java is a new code generator for security protocols designed to be highly extensible and customizable. It takes a protocol specification written in the Extensible Spi calculus (Expi calculus) together with a configuration that provides the low-level information needed for code generation and produces interoperable Java code.

The diagram in Figure 1 shows the workflow of Expi2Java. The Expi calculus model can be verified using ProVerif and type-checked with respect to the Expi2Java configuration (we defer the discussion about the type system to §5.3). The code generator takes the model and the configuration, and generates the symbolic library and the protocol implementation using code templates (special snippets of Java code).

The user can easily customize and extend the input language by adding new types or new cryptographic primitives (constructors and destructors) to the default configuration, and by instantiating them

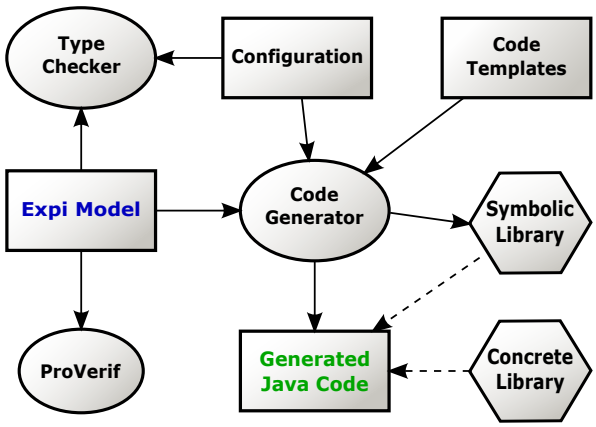

Figure 1. Expi2Java Workflow

with specific cryptographic algorithms (DES encryption, RSA signature, etc.). The Expi2Java configuration also specifies which Java class should be used to implement each of the cryptographic algorithms and allows to pass user-defined parameters to the implementation class. This can be used, for example, to specify which encryption mode and padding should be used to encrypt data or to specify the length and endianness of an integer nonce to match the protocol specification, basically giving the user full control over the low-level data format in the generated protocol messages.

The syntax used by Expi2Java for writing models is very similar to the one used by ProVerif [9] and includes support for ProVerif-specific constructs such as events and queries. The main difference is that our calculus is typed and we therefore need to annotate some terms with their types. Expi2Java can pretty-print the protocol specifications in ProVerif syntax, so that it can be formally verified before generating the implementation.

The translation used for code generation can also be customized, since Expi2Java uses templates to generate the protocol implementation. The user can change the templates and the class stubs to simplify integration of the generated code into existing applications. Our formalization targets the default translation used in the tool, with only minor simplifications.

The generated code relies on a runtime library containing classes that implement the cryptographic primitives used by the protocol specifications. We provide two interchangeable versions of the runtime library: a symbolic library that is used in our formalization and proofs, and a concrete library that implements real networking and cryptography. This separation allows us to abstract away from the complexity of network communication and cryptography when testing and debugging locally. The symbolic library is for the most part automatically generated by our tool and is described in more detail in §6. The concrete library contains implementations for most of the common cryptographic primitives and data types out of the box, and can easily be extended by the user. It uses the standard Java cryptographic providers to ensure interoperability with other existing protocol implementations.

In the six major releases over the last three years we made a lot of progress in the form of both practical features and usability improvements, turning Expi2Java from a prototype into a mature, useful and usable tool. We provide a detailed user manual and a tutorial that uses the Perrig-Song mutual authentication protocol as a running example [21]. More sample protocols such as Needham-Schroeder-Lowe, Andrew and Fiaba are provided in the Expi2Java distribution, together with the TLS implementation described in §4. Expi2Java is free software and is distributed under the terms of the GNU GPLv3.

4 TLS Case Study

In order to show the potential of Expi2Java we have generated a fully functional and interoperable implementation of the widely used Transport Layer Security (TLS) protocol [12] from a model verified with ProVerif. TLS provides a secure transport layer for many popular application protocols such as HTTP, FTP, SMTP, and XMPP.

TLS is a very complex protocol, it supports many different cryptographic schemes, optional parameters and extensions. Our TLS model implements TLS v1.0 [12] with the AES extension [10] and the Server Name Indication (SNI) extension [8]. The model includes both client and server sides, the Handshake, the Application Data Protocol and the Alert Protocol. We support dynamic cipher suite selection between 6 different cipher suites (including AES, RC4 and 3DES encryption with different key lengths, SHA1 or MD5

HMACs and RSA key exchange). One of these cipher suites is dynamically chosen during the handshake. Supporting multiple cipher suites in TLS in older tools such as Spi2Java would require duplicating the whole protocol model for each of the cipher suites. In order to prevent this, Expi2Java allows parameterizing processes with respect to the employed cryptographic algorithm. Using the parameterized processes, we could add support for 6 cipher suites with only a few lines of code. The only noteworthy TLS features we have not implemented are: session resumption (a performance tweak where a vulnerability [1] was discovered in 2009), the client-authenticated handshake (a rarely used feature) and record fragmentation (unsupported by some popular servers and clients, and therefore a very rarely used feature). The handshake messages that are used for key exchange algorithms (other than RSA) are also not supported.

Our model of TLS consists of an Expi calculus process (about 850 lines) and a configuration file (625 lines). It includes all the steps of the Handshake and the consequent message exchange through the Application Data Protocol. We check the validity of each received message, including MACs and certificate chain (when provided with the list of trusted CA certificates) and respond with an alert message on errors. The structure of all messages is modeled completely in the Extensible Spi calculus, while the data formats of encryptions and certificates are defined in the configuration and implemented in corresponding Java classes. For comparison, the most sophisticated generated protocol implementation we are aware of is a much simpler SSH client generated by Spi2Java[4]. Their model is 250 lines in size and needs 1390 lines of XML configuration (i.e., more than 5-to-1 ratio between configuration and model).

We have verified some security properties of our TLS model with ProVerif. In particular, we have used 3 secrecy queries showing that the secret key of the server, the "pre-master secret" nonce used to derive the session keys and initialization vectors, and the request that the client sends to the server using the Application Data Protocol are not leaked. We have used 3 correspondence queries adapted from FS2PV [7] to show message authentication for the certificate and pre-master secret. Additionally, we have used 9 reachability queries providing a consistency check to ensure that different parts of the handshake are reached. The verification process for all 15 queries took about 9 minutes on a laptop with an Intel® Core2™ Duo P7450 CPU.

The generation of Java code for the TLS model (on the same hardware) takes only about 12 seconds. In addition to the sample web server and web client we generate a verified implementation of a TLS channel for the concrete library. We use this channel to generate a simple web server offering a web page over HTTPS and a web client downloading it. We have tested that the resulting implementation is interoperable with common browsers and web servers.

5 The Extensible Spi calculus

The source language of our translation is a variant of the Spi calculus [4], a process calculus for cryptographic protocols. We start with the variant of the Spi calculus by Abadi and Blanchet [3] – a subset of the language accepted by ProVerif [9]. We extend it with the Expi2Java configuration in §5.2 and define a type system for it in §5.3.

[4] The latest version (Apr. 19, 2011) is available at `http://spi2java.polito.it/`

5.1 Syntax

The syntax of the Extensible Spi calculus (Expi calculus) is defined in Table 1. In this calculus, *terms* are used to model cryptographic operations symbolically. Terms are obtained by applying *constructors* to other terms, *variables*, and *Expi names*. Constructor applications are parameterized by the name of the cryptographic algorithm A that should be used to implement the constructor in the generated code. Differently parameterized constructors are treated as different constructors, so for instance enc_{DES} is different than enc_{AES}. The global configuration (see §5.2) defines, amongst others, the set of constructor identifiers $\mathcal{F} = \{f_1, \ldots, f_n\}$ and their type, as well as the set of possible cryptographic algorithms that can be used to implement each constructor.

Table 1. The Syntax of the Extensible Spi calculus

$K, L, M, N ::=$	terms
a, b, m, n, k	Expi names
x, y, z, v, w	variables
$f_A\langle T_1, \ldots, T_m\rangle(M_1, \ldots, M_n)$	constructor application
$G ::= g_A\langle T_1, \ldots, T_m\rangle(M_1, \ldots, M_n)$	destructor applications
$P, Q, R ::=$	processes
$\mathsf{out}(M, N).P$	output
$\mathsf{in}(M, x).P$	input
$!\,\mathsf{in}(M, x).P$	replicated input
$\mathsf{new}\ a : T.P$	restriction
$P \mid Q$	parallel composition
$\mathbf{0}$	null process
$\mathsf{let}\ x = G\ \mathsf{in}\ P\ \mathsf{else}\ Q$	destructor evaluation

Notation: We use u to refer to both Expi names and variables.

In our calculus, *destructors* are partial functions that are applied to terms and can produce a term or fail. Similar to constructors, we parameterize destructors by a cryptographic algorithm name. The global configuration also defines the set of destructor identifiers $\mathcal{G}$, and the cryptographic algorithms supported for each destructor identifier. Constructors and destructors can have a parametric type, in which case we must provide a list of type annotations $\langle T_1, \ldots, T_m\rangle$ for instantiating the type variables. In our Expi2Java tool these annotations are automatically inferred in most cases.

Processes are used to model the behavior of protocol participants and the communication between them. A specific characteristic of our calculus is that replication (the bang symbol "!") can only appear before an input process [6]. This is the most common way to use replication in specifications, and at the same time it is reasonably easy to implement as a server that waits for requests on a channel and spawns a new thread to process each incoming message. The Expi name in the restriction process has a type annotation, otherwise the syntax of processes is standard [3, 4, 6].

In our Coq formalization, we use a *locally-nameless* representation [5] for Expi names (bound by the restriction process) and variables (bound by the let and input processes) to avoid the issues related to α-renaming. Nevertheless, for the sake of readability, throughout this paper we use the more familiar, named representation.

5.2 Configuration

The crucial features of the Expi calculus are its extensibility and customizability. The user can extend the sets of types, constructors and destructors, redefine the reduction relation

for destructors and provide different implementations for the constructors and destructors. Our whole calculus is parameterized over what we call a *configuration* – a collection of several sets and functions that define the behavior of types, constructors and destructors (see Table 2). We have defined a default configuration that is sufficient to model most cryptographic protocols, please refer to §A.2for more details.

Table 2. Configuration

$\Big(\mathcal{T}, \mathcal{F}, \mathcal{G}, alg(t), alg(f), alg(g), variance(t,i), Gen, f : \forall \widetilde{X}.\ (\widetilde{T}) \mapsto T, g : \forall \widetilde{X}.\ (\widetilde{T}) \mapsto T, \Downarrow\Big)$

$\mathcal{T} = \{t_1, \ldots, t_n\}$	a finite set of type identifiers
$\mathcal{F} = \{f_1, \ldots, f_m\}$	a finite set of constructor identifiers
$\mathcal{G} = \{g_1, \ldots, g_l\}$	a finite set of destructor identifiers
$alg(t_i) = \{A_i^1, \ldots, A_i^{n_i}\}$	cryptographic algorithms for each type
$alg(f_j) = \{A_j^1, \ldots, A_j^{n_j}\}$	cryptographic algorithms for each constructor
$alg(g_k) = \{A_k^1, \ldots, A_k^{n_k}\}$	cryptographic algorithms for each destructor
$variance(t,i) \in \{+, -, \circ\}$	the variance of each type identifier (see §5.3)
$Gen \subseteq \mathcal{T}$	a set of generative types
$f : \forall \widetilde{X}.\ (\widetilde{T}) \mapsto T$	the type of each constructor (see §5.3)
$g : \forall \widetilde{X}.\ (\widetilde{T}) \mapsto T$	the type of each destructor
$g(M_1, \ldots, M_n) \Downarrow N$	the destructor reduction relation

Notation: We use $\widetilde{T}$ to denote type sequences of form $T_1, \ldots, T_n$ for some n.

5.3 Type System

Since Java is an explicitly typed language, any code generator targeting Java needs to generate type annotations for variables, fields, method arguments and return values, etc. These type annotations cannot be generated out of thin air, and asking the user to manually annotate types in the automatically generated Java "spaghetti code" would be a usability disaster. We solve this problem by asking the user to provide type annotations while writing the protocol model. Our code generator uses the type information in the model to generate the typing annotations needed by the Java type-checker.

Additionally, we want to prevent that a user who makes mistakes in a protocol model finds out that the generated implementation does not even compile in Java because of typing errors in the automatically generated code, which the user does not understand. So we provide a type-checker for the Expi calculus that prevents generating ill-typed Java code. Our type-checker immediately reports inconsistent usage of terms in an understandable way – in terms of the specified protocol model the user has written. In §8.1 we show that if the original protocol model is well-typed with respect to our Expi calculus type system then our translation is guaranteed to generate a well-typed Java program.

Our type system for the Expi calculus features subtyping and parametric polymorphism [11]. This makes the type system very expressive and has allowed us to devise a very precise unification-based type inference algorithm. This decreases the type annotation burden on the user and improves the readability of the protocol models. Parametric polymorphism also allows us to have only a small number of "generically" typed constructors and destructors and still be able to specialize them. Parametric types can be nested, which naturally corresponds to the types of the nested terms and allows us to keep more information about the inner terms even after several destructor or constructor applications. The nested types allow us, for instance, to express the type of messages sent and received

over a channel, or to model the fact that an encryption and the corresponding key work on messages of the same type.

In order to ensure the correct use of cryptographic primitives we also parameterize types, constructors and destructors with respect to cryptographic algorithms. We use this feature to statically prevent mixing up different algorithms, e.g., decrypting an AES encrypted message with a DES key. As the result, our type system can express many complex types, such as specializations of channels, encryptions etc. using a reasonably small set of core types, which is better suited for formalization.

Table 3. Syntax of Expi Types

$$T, U ::= \quad X \quad | \quad \mathsf{Top} \quad | \quad \mathsf{Channel}_A\langle T \rangle \quad | \quad t_A\langle T_1, \ldots, T_m \rangle$$

Our type system has only two fixed types, which are required for the correct typing of processes: type Top and type $\mathsf{Channel}_A\langle T \rangle$. The type variables X are used in the parametric types of constructors and destructors. The types $t_A\langle T_1, \ldots, T_m \rangle$ represent user-defined parametric types defined using the set $\mathcal{T}$ from the configuration (see §5.2). Additionally, types $\mathsf{Channel}_A\langle X \rangle$ and $t_A\langle T_1, \ldots, T_m \rangle$ are parameterized by a cryptographic algorithm name A, which allows us to define subsets of related types such as TCP or TLS channels, AES keys and AES-encrypted messages. The configuration also defines a set of *generative* types, the only types that can be used in well-typed restriction processes.

The subtyping relation $<:$ is defined in Table 4. Type Top is a supertype of all other types. Subtyping for channel types $\mathsf{Channel}\langle X \rangle$ is invariant. The custom types $t_A\langle T_1, \ldots, T_m \rangle$ are subtyped according to the variance of their arguments, as defined in the configuration using function $variance(t, i)$. Subtyping is reversed for contravariant $(-)$ arguments, runs in the same direction for covariant $(+)$ arguments and requires the invariant $(\circ)$ arguments to be the same.

Table 4. Subtyping: $T <: U$

$$\text{(Sub-Top)} \quad \frac{T \vdash \diamond}{T <: \mathsf{Top}} \qquad \text{(Sub-Channel)} \quad \frac{T <: U \qquad U <: T}{\mathsf{Channel}_A\langle T \rangle <: \mathsf{Channel}_A\langle U \rangle}$$

$$\text{(Sub-Nested)} \quad \frac{\begin{array}{ll} t \in \mathcal{T} \qquad \forall i \in [1, n]. & (variance(t, i) = + \;\Rightarrow\; T_i <: U_i) \\ & (variance(t, i) = - \;\Rightarrow\; U_i <: T_i) \\ & (variance(t, i) = \circ \;\Rightarrow\; U_i <: T_i \;\wedge\; T_i <: U_i) \end{array}}{t_A\langle T_1, \ldots, T_n \rangle <: t_A\langle U_1, \ldots, U_n \rangle}$$

The type of terms is defined by relation $\Gamma \vdash M : T$ from Table 5. This relation is very simple and general. The type of constructor applications is defined in the configuration and instantiated by the given type annotations. Finally, using the subsumption rule Term-Sub a term can be used in any context that expects a supertype of its type.

Table 5. Term Typing: $\Gamma \vdash M : T$

$$\text{(Term-Env)} \quad \frac{\Gamma \vdash \diamond \qquad T \vdash \diamond \qquad u : T \in \Gamma}{\Gamma \vdash u : T} \qquad \text{(Term-Sub)} \quad \frac{\Gamma \vdash M : T \qquad T <: U}{\Gamma \vdash M : U}$$

$$\text{(Term-Constr)} \quad \frac{f_A : \forall \widetilde{X}.\, (\widetilde{T}) \mapsto T \qquad \forall i \in [1, m].\, U_i \vdash \diamond \qquad \forall i \in [1, n].\, \Gamma \vdash M_i : T_i[\widetilde{X} := \widetilde{U}]}{\Gamma \vdash f_A\langle U_1, \ldots, U_m \rangle(M_1, \ldots, M_n) : T[\widetilde{X} := \widetilde{U}]}$$

6 Variant Parametric Jinja

Our target language, Variant Parametric Jinja (VPJ) is based on Jinja with Threads [16, 17] and the type system with variant parametric types by Igarashi et al. [14]. In this section, we briefly describe VPJ with an emphasis on the modifications we have made; please refer to the original papers for more details [14–17]. We chose Jinja with Threads over other formalized Java fragments because of its comprehensiveness. Expi2Java needs concurrency with shared memory and synchronization to model channels and message passing, a class hierarchy with inheritance and casting to model data types, a type system that supports Java generics and wildcards and, optionally, exceptions to simplify modeling destructor applications. Jinja with Threads supports all these features except for generics and wildcards, which we added in the form of variant parametric types [14]. We have also extended the values and types with support for a new base type, **string**, and removed support for arrays in order to simplify the semantics.

Syntax. The syntax of VPJ is very similar to Jinja with Threads, and therefore differs from Java in several ways. The most visible simplification is the absence of distinction between expressions and statements. Other differences inherited from Jinja are simplifications in the form of a few artificial expressions like $\mathbf{insync}\ (@n)\ \{e\}$ and **unit**, slightly different typing and reduction rules, and the lack of syntactic sugar [16]. A difference unique to VPJ is that the object instantiation expression **new** takes an additional list of expressions $e_1, \ldots, e_k$ that are used to initialize all fields declared in the corresponding class.

Table 6. Syntax of Variant Parametric Jinja (VPJ)

$v ::= \ \mathbf{unit} \mid \mathbf{null} \mid \mathbf{true} \mid \mathbf{false} \mid i \mid \text{"}\mathtt{xyz}\text{"} \mid @n$	VPJ value
$\otimes ::= \ = \mid \neq \mid < \mid \leq \mid > \mid \geq \mid + \mid - \mid \times \mid \wedge \mid \vee \mid \oplus$	binary operation
$e ::=$	VPJ expression
$\quad \mathbf{new}\ C\langle T_1^P, \ldots, T_n^P\rangle\ (e_1, \ldots, e_k)$	object instantiation
$\quad (T^J)e$	typecast
$\quad v$	literal value
$\quad e_1\ \otimes\ e_2$	binary operation
$\quad x, y, z$	variable access
$\quad x := e$	variable assignment
$\quad e.f\{C\}$	field access
$\quad e_1.f\{C\} := e_2$	field assignment
$\quad e.\langle T_1^P, \ldots, T_n^P\rangle m\,(e_1, \ldots, e_k)$	parametric method invocation
$\quad \{T^J\ x := \lfloor v \rfloor\ ;\ e\}$	variable declaration block
$\quad \mathbf{synchronized}\ (e_1)\ \{e_2\}$	**synchronized** statement
$\quad \mathbf{insync}\ (@n)\ \{e\}$	locked **synchronized** statement
$\quad e_1\ ;\ e_2$	sequential composition
$\quad \mathbf{if}\ (e_1)\ \{e_2\}\ \mathbf{else}\ \{e_3\}$	conditional
$\quad \mathbf{while}\ (e_1)\ \{e_2\}$	while loop
$\quad \mathbf{throw}\ e$	exception throwing
$\quad \mathbf{try}\ \{e_1\}\ \mathbf{catch}\ (C\langle\widetilde{T^P}\rangle\ x)\ \{e_2\}$	exception catching
$T^J ::= \mathbf{void} \mid \mathbf{boolean} \mid \mathbf{int} \mid \mathbf{string} \mid \mathbf{null_T} \mid T^P$	VPJ type
$T^P ::= X^J \mid T^C$	parametric type
$T^C ::= C\langle T_1^P\mathcal{V}, \ldots, T_n^P\mathcal{V}\rangle, D\langle T_1^P\mathcal{V}, \ldots, T_n^P\mathcal{V}\rangle$	variant parametric class type
$\mathcal{V} \ ::= + \mid - \mid \circ \mid \star$	co-, contra-, in- and bivariant subtyping
$L \ ::= \mathbf{class}\ C\langle\widetilde{X^J} \lhd \widetilde{U^C}\rangle \lhd D\langle\widetilde{T^P}\rangle\ \{\widetilde{T^J}\ \widetilde{f}\ ;\ \widetilde{M}\}$	class definition
$M ::= \langle\widetilde{X^J} \lhd \widetilde{U^C}\rangle\ T^J\ m(\widetilde{T^J}\ \widetilde{x})\ \{\mathbf{return}\ e\}$	method definition

Notation: We write $\widetilde{X^J} \lhd \widetilde{T^C}$ for the sequence $X_1^J \lhd T_1^C, \ldots, X_n^J \lhd T_n^C$

Just like Java, VPJ allows to define custom classes with methods and fields. The syntax is similar to the Java generics, with type variables and upper bounds (denoted by $\lhd$ in VPJ, and $\mathbf{extends}$ in Java). In VPJ variant parametric types require variance annotations on each type argument. In addition to the variance annotations from the Expi calculus the variance in VPJ can also be *bivariant*; any two instantiations of a bivariant type constructor are subtypes of each other. The type system of VPJ is a mixture of the Jinja type system [16] with variant parametric types [14]. We have extended the subtyping relation with two additional rules stating that the null type $\mathbf{null_T}$ is the subtype of $\mathbf{string}$ and all reference types T^P. Additionally, we require the subclass relation to be *well-founded* (i.e., that it does not have infinite descending chains) to ensure termination of subclass checks.

Memory Model. VPJ uses a realistic memory model with a shared heap and thread-local stacks. The stacks are maps from variable names to values. The heap maps memory locations $@n$ to a class type T^C and the field values of the corresponding instance. This heap model differs slightly from the one used in Java, since we store the exact parametric type of each object, while in Java the information about the type parameters is lost.[5] We have decided to store the parametric types to simplify the formalization of variant parametric types in Coq and avoid the problems arising from Java-like semantics such as the need for run-time type-checks to enforce type soundness.

Semantics. The semantics of VPJ can be split into two parts, the *single-threaded core* and the (very complex) *multi-threaded semantics* [16]. The single-threaded semantics is defined using the reduction relation $\rightarrow_J$, a *labeled* relation that takes a conversion function and reduces some expression together with some state (i.e., stack and heap) to another expression and a state, possibly producing a *thread action* (usually visualized as a label on the reduction arrow). The thread actions are used for inter-thread communication in the multi-threaded semantics. The multi-threaded semantics is exactly the same as in Jinja with Threads [16], we have only made changes that had to be done to adapt the formalization for Coq.

Symbolic Library. In addition to the standard classes like `Object` required by VPJ we have defined an additional set of classes that are used by the translated protocols. We call this the *symbolic library*, because the implementation of the cryptographic primitives and data types does not perform any "real" cryptography or networking. The symbolic library provides a symbolic abstraction of the cryptographic primitives and channel communication. It is designed to be simple enough to simplify proving the translation secure in the future. Finally, the symbolic library can also be used for debugging purposes. Our Expi2Java tool also has a concrete library that performs "real" cryptographic operations and uses the actual network, but this is not formalized in Coq. The two libraries (concrete and symbolic) can be used interchangeably by the generated code.

The fixed part of the symbolic library (i.e., the one that is not generated by our translation from the configuration) consists of the 7 classes shown in Figure 2. `AbstractBase` is the base class of the class hierarchy for translations of all Expi types. `AbstractGenerativeBase` is the superclass for generative types. The generated flat class hierarchy reflects the subtyping relation from the Expi calculus.

[5] In the Java community, this process is usually called "type erasure", although not all type information is erased, just the generics, and that just for backward compatibility.

`Semaphore` implements a counting semaphore using the synchronization primitives of VPJ. This class is used in the implementation of inter-thread message passing. Expi channels are modeled using the class `AbstractChannel`$\langle X^J \rangle$. It implements the synchronous semantics of Expi channels, where different processes are implicitly synchronized when a message is sent from one process to another over a channel.

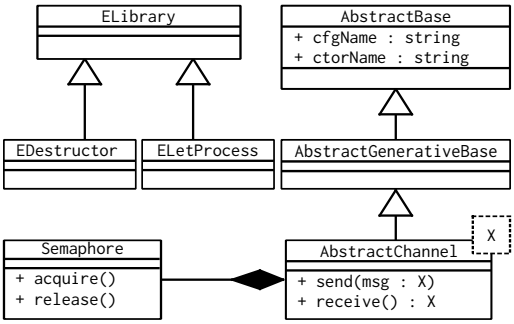

Figure 2. Symbolic Library Classes

Finally, `ELibrary` is the base class for `ELetProcess` and `EDestructor`, the two exception classes used to model failing destructor applications. The translation of the let $x = G$ in P else Q process uses the exception to decide when to take the else branch. We have proved that all symbolic library classes are well-typed with respect to our type system (see §8.1).

7 Translation Overview

The translation from the Expi calculus to VPJ is performed in two steps. The first step translates protocol models to the *Global Expi calculus*, a variant of the Expi calculus with a different semantics for name binders. The second and much more complex step translates Global Expi processes into VPJ programs.

7.1 First Step: Translating Expi Calculus Models to the Global Expi Calculus

The usual semantics of the Expi calculus heavily relies on α-renaming and scope extrusion to reduce processes. Scope extrusion allows to move name binders (i.e., restriction processes new $a : T$) out of parallel compositions ($P \mid Q$). The problem with this semantics is that it cannot be easily implemented in a mainstream programming language, because it would require changing the scope of variables and moving them across different threads. Instead of extruding restrictions around we define a variant of the Expi calculus in which restrictions generate globally fresh names. Such a global semantics was proposed for the pi calculus by Wischik [27], and we adapt this idea to the Expi calculus. The resulting Global Expi calculus uses the same terms and types as the Expi calculus and has only one different process, the generation process gen $x : T$ in P instead of the restriction process new $a : T.P$. The semantics of gen $x : T$ in P is to generate a globally fresh name a and substitute variable x with a in P, like for ML references. This first translation step brings our semantics closer to the one of the Java implementation, which should ease any future simulation proof.

7.2 Second Step: Translating Global Expi Models to VPJ Programs

In the second step of the translation we implement the behavior of Expi terms and processes in VPJ; this is much more complicated than the fist step.

Table 7. Translation Overview

Global Expi		**VPJ**
Configuration:		
- Type identifiers (t)	$\rightsquigarrow$	Class declarations
- Algorithm names (A)	$\rightsquigarrow$	String constants stored in fields
- Constructors (f)	$\rightsquigarrow$	Special methods in class `Fun`
- Destructors (g)	$\rightsquigarrow$	Special methods in class `Fun`
Expi Types	$\rightsquigarrow$	Variant parametric classes
Terms	$\rightsquigarrow$	Expressions (variables, method calls)
Processes	$\rightsquigarrow$	Expressions (variable declarations, control flow)
- Parallel composition ($P \mid Q$)	$\rightsquigarrow$	Threads that are spawned and joined
Free names	$\rightsquigarrow$	Variables in main (passed by reference to threads)

Table 7 gives an overview of how the different Expi calculus constructs are represented in VPJ. Expi calculus types are modeled as additional generated classes in our symbolic library, and variant parametric types are used to represent type parameters. The instances of these generated classes represent Expi terms of the corresponding type. The cryptographic algorithm names used in the destructor reduction relation are stored in fields of type $\mathbf{string}$ in the generated classes. Expi constructors and destructors are represented as special methods in a symbolic library class named `Fun`. We use a simple naming convention to distinguish constructor and destructor methods. Terms are translated to VPJ expressions that either access local variables or call constructor methods. Processes are translated into larger code blocks that create and modify the terms stored in local variables and use the symbolic library to interact with each other. We use threads to model parallel composition of processes and shared memory to pass data between them. Appendix A.7 gives a more detailed description of the translation process.

8 Proofs

We have used the Coq proof assistant [2] to develop the mechanized formalization of our translation and machine-check our proofs. We believe that this is the only way to stay honest with ourselves when proving something about a formalization that is so complex. A proof assistant helps organizing the formal development as a software project, ensures that the definitions are well-formed and in sync with each other, provides automation for proving routine tasks, and checks that the proofs are correct, without missing any corner-case and without forgetting any assumption that was made. This gives us high confidence in the final result, which one cannot easily obtain by "handwaving". The formalization encompasses about 17K lines of Coq code in total.

8.1 Symbolic Library and Generated Code are Well-typed

We have proved that the symbolic library and the VPJ code generated by the translation are well-typed. This is an important consistency check for our definitions (it helped us to find out, for instance, that the first version of our translation was using the field access in an inconsistent way) and justifies the usage of our Expi calculus type system to prevent generating ill-typed VPJ programs. In the longer-term perspective these proofs will be needed for proving the correctness of our translation.

We have shown that each class of the symbolic library is well-typed in a VPJ program containing a small number of Java standard library classes (`NullPointerException`, `Object`, etc.) and the symbolic library classes. We show that all types used in class declarations are well-formed and all declared methods are well-typed. The proof is by case analysis on the corresponding expression and using the right case of the expression typing relation. It is not complicated, but quite long and tedious, because we need to give the correct type of each subexpression and show all premises of the typing rules.

Showing that the code generated by the translation is well-typed was much harder. Most typing rules require providing the exact types of all subexpressions and only fail in the last moment if a wrong type was chosen. Another problem was finding the right invariants to type-check the code generated by recursive functions. An incorrect invariant (i.e., the type of the expression we are trying to type-check and the preconditions) usually becomes noticeable only when applying the induction hypothesis, after having constructed a big part of the failed proof attempt.

We prove that the expressions generated for the Global Expi processes, the constructor and destructor methods and the classes representing Expi types are well-typed assuming that our invariants hold. In the end, we use these results to show the following theorem:

Theorem 1 (Trans-WT). *If P is locally-closed, $\Gamma \vdash P$ and $P \rightsquigarrow (e, L)$, then L is well-typed and (e, L) is well-typed in a preallocated heap and an empty stack.*

This theorem shows that our translation generates a self-contained VPJ program (e, L) that is well-typed in the initial heap and stack. The assumptions about the heap and stack are inherited from Jinja. They are needed for the typing relation, because an arbitrary Jinja expression can contain local variables or throw system exceptions (e.g., `NullPointerException`). At the program start, we assume that the heap is "preallocated", i.e., it contains instances of system exception classes at some fixed addresses that are reserved for the system exceptions, and the stack contains no local variables.

The Coq proof of this theorem is done in great detail, and so are the proofs of most of the helper lemmas. Due to the lack of time we did not prove some rather obvious properties of the translation that are used in the proof of the theorem, e.g., that processes are translated only to expressions of type **void** or that all used types were generated and added to the program. Furthermore, we did not prove some list and substitution rewriting lemmas, and other similar helper lemmas in cases that looked trivially true but were tedious to prove in Coq. We have also assumed that the translation of terms and the declaration of free Expi names are well-typed.

8.2 Destructor Consistency Proof for Default Configuration

The flexible nature of the Expi calculus makes it impossible to prove the type system sound for an arbitrary configuration. The destructor reduction relation could be defined in a way that conflicts with the typing of the destructor or of some of the constructors used by the reduction rule. For instance, it would be possible to give an identity destructor (which returns its only argument) the type $(\mathsf{Int}) \mapsto \mathsf{Bool}$. Such a destructor would be inconsistent, because we cannot give any term two different types that are not even subtypes of each other. To prevent such inconsistency in our default configuration we have proved the following theorem in Coq:

Theorem 2 (Destructor Consistency). *If a destructor has type $g_A : \forall [X_i]_n.\ ([T_j]_m) \mapsto T$, $g_A\langle [U_i]_n \rangle ([M_j]_m) \Downarrow M$ and $\forall j \in [1, m].\ \Gamma \vdash M_j : T_j[\widetilde{X} := \widetilde{U}]$, then M can be typed to the instantiated return type of the destructor: $\Gamma \vdash M : T[\widetilde{X} := \widetilde{U}]$.*

Destructor consistency is the crucial step in the subject-reduction proof of the Expi calculus. It is the only part of the soundness proof that depends on the configuration and should therefore be re-proven if the user wants to change the default configuration. Once one proves destructor consistency, soundness follows directly.

8.3 Lessons Learned

Formalizing our code generator in a proof assistant turned out to be harder than we expected. Like the large majority of similar tools before it [18, 20, 21, 23–25], Expi2Java targets the Java programming language. This has many pragmatic advantages for building a usable and secure tool: from type and memory safety, an extensive standard library and the cryptographic service provider architecture, to the ease of integrating the generated code into existing applications. However, Java is a very complex programming language, and merely adapting an existing mechanized formalization of a subset of Java [16, 17] to suit our particular needs turned out to be a quite daunting task. About 7k LOC out of the 17k LOC of our formalization are solely concerned with defining our target language, VPJ.

Moreover, Java is an imperative language, and lacks certain functional features that would have greatly simplified the symbolic representation of terms: immutable data structures, structural equality, and pattern matching. A programming language like Scala[6] or F#[7], which integrate features of both functional and imperative object-oriented programming, would have been a better match for implementing our symbolic library, while preserving most of the pragmatic advantages of Java. On the other hand, we are not aware of any mechanized formalizations of comprehensive subsets of these languages, so the upfront effort needed just to formalize the target language would have been even larger.

9 Conclusion

In this paper we have introduced Expi2Java, an extensible code generator for security protocols. We have illustrated the flexibility of Expi2Java by generating interoperable implementations of a client and a server for TLS v1.0 from a protocol model verified with ProVerif. We have formalized our source and target languages as well as the translation between them using the Coq proof assistant, and proved that the generated code is well-typed if the original protocol model is well-typed. This increases our confidence in the translation, and justifies the usage of our Expi calculus type system to catch all type errors as early as possible and present understandable error messages. Additionally, we have proved the consistency of the destructors in our default configuration.

In the future it would be very interesting to show that the translation presented here preserves the trace properties, and, more ambitiously, the security properties (e.g., the robust safety) of the original protocol model. The former could be achieved by using weak labeled simulation to relate VPJ programs to Expi processes, while for the later one would have to show that this simulation is contextual and that each VPJ attacker can be mapped back to an attacker in the Expi calculus. We believe that the current work builds a solid ground on which the preservation of security properties can be formally investigated.

[6] `http://www.scala-lang.org/`
[7] `http://www.fsharp.net/`

Acknowledgments We thank the anonymous reviewers for their helpful feedback. Expi2Java originated as an extension of Spi2Java [22, 23]. Cătălin Hriţcu was supported by a fellowship from Microsoft Research and the International Max Planck Research School for Computer Science.

References

1. CVE-2009-3555. Man-in-the-Middle Vulnerability in TLS via Session Renegotiation, 2009.
2. The Coq proof assistant, 2010. Version 8.3, Home page: `http://coq.inria.fr/`.
3. M. Abadi and B. Blanchet. Analyzing security protocols with secrecy types and logic programs. *Journal of the ACM*, 52(1):102–146, 2005.
4. M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The Spi calculus. *Information and Computation*, 148(1):1–70, 1999.
5. B. E. Aydemir, A. Charguéraud, B. C. Pierce, R. Pollack, and S. Weirich. Engineering formal metatheory. In *POPL 2008*, pages 3–15, 2008.
6. M. Backes, C. Hriţcu, and M. Maffei. Type-checking zero-knowledge. In *CCS 2008*, pages 357–370. ACM Press, Oct. 2008.
7. K. Bhargavan, C. Fournet, A. D. Gordon, and S. Tse. Verified interoperable implementations of security protocols. *ACM Transactions on Programming Languages and Systems*, 31(1), 2008.
8. S. Blake-Wilson, M. Nystrom, D. Hopwood, J. Mikkelsen, and T. Wright. *Transport Layer Security (TLS) Extensions*, April 2006. RFC 4366.
9. B. Blanchet. *ProVerif v1.14pl4 (Automatic Cryptographic Protocol Verifier) User Manual*, February 2008. `http://www.proverif.ens.fr/`.
10. P. Chown. *Advanced Encryption Standard (AES) Ciphersuites for Transport Layer Security (TLS)*, June 2002. RFC 3268 (Informational).
11. P.-L. Curien and G. Ghelli. Coherence of subsumption, minimum typing and type-checking in $F_{\leq}$. *Mathematical Structures in Computer Science*, 2(1):55–91, 1992.
12. T. Dierks and C. Allen. *The TLS Protocol Version 1.0*, January 1999. RFC 2246.
13. A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM TOPLAS*, 23(3):396–450, Mai 2001.
14. A. Igarashi and M. Viroli. Variant parametric types: A flexible subtyping scheme for generics. *ACM TOPLAS*, 28:795–847, September 2006.
15. G. Klein and T. Nipkow. A machine-checked model for a Java-like language, virtual machine and compiler. *ACM TOPLAS*, 28(4):619–695, 2006.
16. A. Lochbihler. Type safe nondeterminism - a formal semantics of Java threads. In *FOOL*, 2008.
17. A. Lochbihler. Verifying a compiler for Java threads. In *ESOP 2010*. Springer, 2010.
18. S. Lukell, C. Veldman, and A. Hutchison. Automated attack analysis and code generation in a multi-dimensional security protocol engineering framework. In *Southern African Telecommunications Networks and Applications Conference*, 2003.
19. J. A. McCarthy, S. Krishnamurthi, J. D. Guttman, and J. D. Ramsdell. Compiling cryptographic protocols for deployment on the web. In *WWW 2007*, pages 687–696, 2007.
20. J. Millen and F. Muller. Cryptographic protocol generation from CAPSL. Technical Report SRI-CSL-01-07, SRI International, December 2001.
21. A. Perrig, D. Song, and D. Phan. AGVI – Automatic Generation, Verification, and Implementation of security protocols. In *CAV 2001*, LNCS. Springer-Verlag, 2001.
22. A. Pironti and R. Sisto. An experiment in interoperable cryptographic protocol implementation using automatic code generation. In *ISCC*, 2007.
23. D. Pozza, R. Sisto, and L. Durante. Spi2Java: Automatic cryptographic protocol Java code generation from spi calculus. In *AINA*, pages 400–405. IEEE Computer Society Press, 2004.
24. D. X. Song, S. Berezin, and A. Perrig. Athena: A novel approach to efficient automatic security protocol analysis. *Journal of Computer Security*, 9(1/2):47–74, 2001.
25. B. Tobler and A. Hutchison. Generating network security protocol implementations from formal specifications. In *CSES*, 2004.

26. M. Torgersen, C. P. Hansen, E. Ernst, P. von der Ahé, G. Bracha, and N. M. Gafter. Adding wildcards to the Java programming language. In *SAC 2004*, pages 1289–1296, 2004.
27. L. Wischik. Old names for nu, 2004. Presented at Dagstuhl Seminar 04241.

A Appendix

A.1 Formalization Size

Table 8 gives an overview over the size of our formalization. The formalization encompasses about 17K lines of Coq code in total (not counting the infrastructure lemmas generated by LNgen[8]).

Table 8. Formalization Size

Module	Lines of code
Formalization of Expi and Global Expi calculi	953
Expi proofs	2864
Formalization of VPJ	6979
Symbolic library	510
Translation from Global Expi to VPJ	1066
VPJ proofs (symbolic library and the translation are well-typed)	3679
Auxiliary definitions and lemmas	1217
Total (without infrastructure lemmas generated by LNgen)	17268
Generated LNgen lemmas	12981

A.2 Default Configuration

In Table 9 we define the default configuration that contains commonly used cryptographic primitives (such as pairing, symmetric encryption, digital signatures, public-key encryption, hashing, etc.) and their corresponding types. These primitives are sufficient to model complex cryptographic protocols such as TLS.

Destructor Consistency Proof The proof of Theorem 2 for the default configuration is by case analysis on the destructor identifier. The interesting cases are the ones for the destructors pre, dec and deca. Here we need to prove inversion lemmas for typing the arguments of constructors succ, enc and enca respectively. We prove these lemmas by induction on the subtyping relation and distinguishing the cases where the constructor application has its type by either subsumption rule TERM-SUB or the constructor typing rule TERM-CONSTR. Proving pre consistent requires considering two reduction rules for the zero and succ cases and showing a helper lemma stating that any subtype of Int is also an Int. In the case of dec destructor we do an induction on the subtyping relation of the arguments and prove some additional helper lemmas stating the preservation of the configuration for arguments of enc.

Table 9. Default Configuration

[8] http://www.cis.upenn.edu/~sweirich/papers/lngen/

$$
\begin{array}{lll}
T_* & := & \mathsf{Top} \quad \mathsf{Channel}\langle X^\circ\rangle \\
 & & \mathsf{Bool} \quad \mathsf{Channel}\langle X^\circ\rangle \quad \mathsf{SymEnc}\langle X^\circ\rangle \quad \mathsf{PubEnc}\langle X^\circ\rangle \quad \mathsf{Signed}\langle X^\circ\rangle \\
 & & \mathsf{Int} \quad \mathsf{Pair}\langle X^+, Y^+\rangle \; \mathsf{SymKey}\langle X^-\rangle \; \mathsf{PubKey}\langle X^-\rangle \; \mathsf{SigKey}\langle X^-\rangle \\
 & & \mathsf{String} \; \mathsf{Hash}\langle X^\circ\rangle \quad \mathsf{KeyPair}\langle X^\circ\rangle \; \mathsf{PrivKey}\langle X^-\rangle \; \mathsf{VerKey}\langle X^-\rangle \\
f_* & := & \mathsf{enc}^2 \; \mathsf{enca}^2 \; \mathsf{pk}^1 \; \mathsf{sk}^1 \; \mathsf{sign}^2 \; \mathsf{sigk}^1 \; \mathsf{vk}^1 \\
 & & \mathsf{h}^1 \; \mathsf{hmac}^2 \; \mathsf{true}^0 \; \mathsf{false}^0 \; \mathsf{pair}^2 \; \mathsf{succ}^1 \; \mathsf{zero}^0 \\
g_* & := & \mathsf{dec}^2 \; \mathsf{deca}^2 \; \mathsf{msg}^1 \; \mathsf{ver}^2 \; \mathsf{id}^1 \; \mathsf{eq}^2 \; \mathsf{fst}^1 \; \mathsf{snd}^1 \; \mathsf{pre}^1 \\
Gen_* & := & \{\mathsf{Int}, \; \mathsf{Channel}\langle X\rangle, \; \mathsf{SymKey}\langle X\rangle, \; \mathsf{KeyPair}\langle X\rangle\}
\end{array}
$$

$f_* : \forall X_1, \ldots, X_m.\ (T_1, \ldots, T_n) \mapsto T \quad :=$

$$
\begin{array}{ll}
\mathsf{enc}_A : \forall X.\ (X, \mathsf{SymKey}_A\langle X\rangle) \mapsto \mathsf{SymEnc}_A\langle X\rangle & \mathsf{h}_A : \forall X.\ (X) \mapsto \mathsf{Hash}_A\langle X\rangle \\
\mathsf{enca}_A : \forall X.\ (X, \mathsf{PubKey}_A\langle X\rangle) \mapsto \mathsf{PubEnc}_A\langle X\rangle & \mathsf{true} : () \mapsto \mathsf{Bool} \\
\mathsf{pk}_A : \forall X.\ (\mathsf{KeyPair}_A\langle X\rangle) \mapsto \mathsf{PubKey}_A\langle X\rangle & \mathsf{false} : () \mapsto \mathsf{Bool} \\
\mathsf{sk}_A : \forall X.\ (\mathsf{KeyPair}_A\langle X\rangle) \mapsto \mathsf{PrivKey}_A\langle X\rangle & \mathsf{pair} : \forall X, Y.\ (X, Y) \mapsto \mathsf{Pair}\langle X, Y\rangle \\
\mathsf{sign}_A : \forall X.\ (X, \mathsf{SigKey}_A\langle X\rangle) \mapsto \mathsf{Signed}_A\langle X\rangle & \mathsf{succ}_A : (\mathsf{Int}_A) \mapsto \mathsf{Int}_A \\
\mathsf{sigk}_A : \forall X.\ (\mathsf{KeyPair}_A\langle X\rangle) \mapsto \mathsf{SigKey}_A\langle X\rangle & \mathsf{zero}_A : () \mapsto \mathsf{Int}_A \\
\mathsf{vk}_A : \forall X.\ (\mathsf{KeyPair}_A\langle X\rangle) \mapsto \mathsf{VerKey}_A\langle X\rangle & \\
\mathsf{hmac}_A : \forall X.\ (X, \mathsf{SymKey}_A\langle X\rangle) \mapsto \mathsf{Hash}_A\langle X\rangle &
\end{array}
$$

$g_* : \forall X_1, \ldots, X_m.\ (T_1, \ldots, T_n) \mapsto T \quad :=$

$$
\begin{array}{ll}
\mathsf{dec}_A : \forall X.\ (\mathsf{SymEnc}_A\langle X\rangle, \mathsf{SymKey}_A\langle X\rangle) \mapsto X & \mathsf{eq} : \forall X.\ (X, X) \mapsto X \\
\mathsf{deca}_A : \forall X.\ (\mathsf{PubEnc}_A\langle X\rangle, \mathsf{PrivKey}_A\langle X\rangle) \mapsto X & \mathsf{fst} : \forall X, Y.\ (\mathsf{Pair}\langle X, Y\rangle) \mapsto X \\
\mathsf{msg}_A : \forall X.\ (\mathsf{Signed}_A\langle X\rangle) \mapsto X & \mathsf{snd} : \forall X, Y.\ (\mathsf{Pair}\langle X, Y\rangle) \mapsto Y \\
\mathsf{ver}_A : \forall X.\ (\mathsf{Signed}_A\langle X\rangle, \mathsf{VerKey}_A\langle X\rangle) \mapsto X & \mathsf{pre}_A : (\mathsf{Int}_A) \mapsto \mathsf{Int}_A \\
\mathsf{id} : \forall X.\ (X) \mapsto X &
\end{array}
$$

Algorithms	Types	Constructors	Destructors
$\star$	$\mathsf{Pair}\langle X, Y\rangle$	pair true false	fst snd id eq
TCP-IP HTTPS Memory	$\mathsf{Channel}\langle X\rangle$		
RSA	$\mathsf{PubEnc}\langle X\rangle$ $\mathsf{PrivKey}\langle X\rangle$ $\mathsf{PubKey}\langle X\rangle$ $\mathsf{Signed}\langle X\rangle$ $\mathsf{SigKey}\langle X\rangle$ $\mathsf{VerKey}\langle X\rangle$ $\mathsf{KeyPair}\langle X\rangle$	enca pk sk sigk vk	deca msg ver
3DES DES RC4 AES256 AES128	$\mathsf{SymEnc}\langle X\rangle$ $\mathsf{SymKey}\langle X\rangle$	enc	dec
SHA1 MD5 SHA1-HMAC MD5-HMAC	$\mathsf{Hash}\langle X\rangle$ $\mathsf{SymKey}\langle X\rangle$	h hmac	
UInt8 UInt16 UInt24 UInt32 Nonce	Int	succ zero	pre
ASCII UTF8	String		

$G \Downarrow_* M \quad :=$

$$
\begin{array}{lll}
\mathsf{dec}_A(\mathsf{enc}_A(M, K), K) & \Downarrow M & \mathsf{eq}(M, M) \quad \Downarrow M \\
\mathsf{deca}_A(\mathsf{enca}_A(M, \mathsf{pk}_A(K)), K) & \Downarrow M & \mathsf{fst}(\mathsf{pair}(M, N)) \quad \Downarrow M \\
\mathsf{msg}_A(\mathsf{sign}_A(M, K)) & \Downarrow M & \mathsf{snd}(\mathsf{pair}(M, N)) \Downarrow N \\
\mathsf{ver}_A(\mathsf{sign}_A(M, K), \mathsf{vk}_A(K)) & \Downarrow M & \mathsf{pre}_A(\mathsf{succ}_A(M)) \Downarrow M \\
\mathsf{id}(M) & \Downarrow M & \mathsf{pre}_A(\mathsf{zero}_A) \quad \Downarrow \mathsf{zero}_A
\end{array}
$$

A.3 Operational Semantics of the Expi Calculus

The semantics of the calculus is quite standard and is defined by a structural equivalence relation $\equiv$ and an internal reduction relation $\rightarrow$.

Structural equivalence relates the processes that are considered equivalent up to syntactic rearrangement. The only difference to the usual semantics [3] is the absence of the binder swapping rule that allows to rearrange restriction processes. This rule is not important for reduction and is difficult to define in a locally-nameless representation.

Table 10. Structural equivalence: $P \equiv Q$

(EQ-ZERO-ID) $\dfrac{}{P \mid \mathbf{0} \equiv P}$ (EQ-FORK-COMM) $\dfrac{}{P \mid Q \equiv Q \mid P}$ (EQ-FORK-ASSOC) $\dfrac{}{(P \mid Q) \mid R \equiv P \mid (Q \mid R)}$

(EQ-SCOPE) $\dfrac{a \notin fn(P)}{\mathsf{new}\ a : T.(P \mid Q) \equiv P \mid \mathsf{new}\ a : T.Q}$ (EQ-FORK) $\dfrac{P \equiv Q}{P \mid R \equiv Q \mid R}$ (EQ-FORK-NEW) $\dfrac{P \equiv Q}{\mathsf{new}\ a : T.P \equiv \mathsf{new}\ a : T.Q}$

(EQ-REFL) $\dfrac{}{P \equiv P}$ (EQ-SYMM) $\dfrac{Q \equiv P}{P \equiv Q}$ (EQ-TRANS) $\dfrac{P \equiv Q \qquad Q \equiv R}{P \equiv R}$

Internal reduction defines the semantics of communication and destructor application. The communication is defined in the usual way. On the other hand, the destructor application (see rules RED-DESTR and RED-ELSE) uses the destructor reduction relation $\Downarrow$ defined in the configuration. The reduction relation $\Downarrow$ defines the semantics of destructors: they can either succeed and provide some term as the result (written as $G \Downarrow N$) or fail (written as $G \not\Downarrow$).

Table 11. Internal reduction: $P \rightarrow Q$

(RED-I/O) $\dfrac{}{\mathsf{out}(a, M).P \mid \mathsf{in}(a, x).Q \rightarrow P \mid Q[x := M]}$ (RED-EQ) $\dfrac{P \equiv P' \qquad P' \rightarrow Q' \qquad Q' \equiv Q}{P \rightarrow Q}$

(RED-!I/O) $\dfrac{}{\mathsf{out}(a, M).P \mid !\,\mathsf{in}(a, x).Q \rightarrow P \mid Q[x := M] \mid !\,\mathsf{in}(a, x).Q}$ (RED-NEW) $\dfrac{P \rightarrow Q}{\mathsf{new}\ a : T.P \rightarrow \mathsf{new}\ a : T.Q}$

(RED-FORK) $\dfrac{P \rightarrow Q}{P \mid R \rightarrow Q \mid R}$ (RED-DESTR) $\dfrac{g(\widetilde{M}) \Downarrow N}{\mathsf{let}\ x = g(\widetilde{M})\ \mathsf{in}\ P\ \mathsf{else}\ Q \rightarrow P[x := N]}$ (RED-ELSE) $\dfrac{g(\widetilde{M}) \not\Downarrow}{\mathsf{let}\ x = g(\widetilde{M})\ \mathsf{in}\ P\ \mathsf{else}\ Q \rightarrow Q}$

Our default configuration (see Table 9) contains a set of destructors commonly used in cryptographic protocols and a default destructor reduction relation $\Downarrow_*$. The cryptographic algorithm name can be used by the destructor reduction relation $\Downarrow$. Our default destructor reduction relation $\Downarrow_*$ checks that the types of all arguments of a destructor have the same configuration name. This ensures for example that an AES-encrypted message cannot be decrypted with a DES key.

The process typing relation $\Gamma \vdash P$ checks that all used terms are well-typed, enforces the proper usage of destructors and channels and checks that only generative types are used in restriction processes.

A.4 Process Typing Relation for the Expi Calculus

The processes cannot directly have a type, instead we define a process typing relation $\Gamma \vdash P$ to check that all used terms are well-typed. This relation requires the channel terms to have a channel type and the type used in restriction process to be generative.

Table 12. Process Typing: $\Gamma \vdash P$

(PROC-OUT)
$$\frac{\Gamma \vdash M : \mathsf{Channel}_A\langle T\rangle \qquad \Gamma \vdash N : T \qquad \Gamma \vdash P}{\Gamma \vdash \mathsf{out}(M, N).P}$$

(PROC-PAR)
$$\frac{\Gamma \vdash Q \qquad \Gamma \vdash P}{\Gamma \vdash P \mid Q}$$

(PROC-[REPL]-IN)
$$\frac{\Gamma \vdash M : \mathsf{Channel}_A\langle T\rangle \qquad \Gamma, x : T \vdash P}{\Gamma \vdash [!]\mathsf{in}(M, x).P}$$

(PROC-NEW)
$$\frac{T \vdash \diamond \qquad T \in \mathit{Gen} \qquad \Gamma, a : T \vdash P}{\Gamma \vdash \mathsf{new}\ a : T.P}$$

(PROC-DES)
$$\frac{\begin{array}{c} g_A : \forall X_1, \ldots, X_m.\ (T_1, \ldots, T_n) \mapsto T \\ \forall i \in [1, m].\ U_i \vdash \diamond \qquad \forall i \in [1, n].\ T_i[\widetilde{X} := \widetilde{U}] \vdash \diamond \\ \forall i \in [1, n].\ \Gamma \vdash M_i : T_i[\widetilde{X} := \widetilde{U}] \qquad \Gamma, x : T[\widetilde{X} := \widetilde{U}] \vdash P \qquad \Gamma \vdash Q \end{array}}{\Gamma \vdash \mathsf{let}\ x = g_A\langle U_1, \ldots, U_m\rangle(M_1, \ldots, M_n)\ \mathsf{in}\ P\ \mathsf{else}\ Q}$$

(PROC-NULL)
$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash \mathbf{0}}$$

A.5 The Global Expi Calculus

The main difference to the Expi calculus is in the semantics of internal reduction and structural equivalence. The structural equivalence relation $\equiv_G$ defined in Table 13 is much simpler than the structural equivalence $\equiv$ used in the Expi calculus. The relation $\equiv_G$ is symmetric, reflexive and transitive and also keeps the two rules stating the commutativity and associativity of parallel composition, but omits the problematic scope extrusion rule.

Table 13. Structural equivalence: $P \equiv_G Q$

(GEQ-ZERO-ID)
$$\frac{}{P \mid \mathbf{0} \equiv_G P}$$

(GEQ-FORK-COMM)
$$\frac{}{P \mid Q \equiv_G Q \mid P}$$

(GEQ-FORK-ASSOC)
$$\frac{}{(P \mid Q) \mid R \equiv_G P \mid (Q \mid R)}$$

(GEQ-FORK)
$$\frac{P \equiv Q}{P \mid R \equiv_G Q \mid R}$$

(GEQ-REFL)
$$\frac{}{P \equiv_G P}$$

(GEQ-SYMM)
$$\frac{Q \equiv_G P}{P \equiv_G Q}$$

(GEQ-TRANS)
$$\frac{P \equiv_G Q \qquad Q \equiv_G R}{P \equiv_G R}$$

The reduction relation in the Global Expi calculus has the same rules for reducing the let process and communication using input and output processes, but has no rule to reduce underneath restriction processes like RED-NEW in Table 11. Instead, it defines a way to directly reduce the generation process $\mathsf{gen}\ x : T\ \mathsf{in}\ P$ to $P[x := a]$, where a is a globally fresh Expi name (see GRED-GEN). This semantics models the usual implementation of a restriction process in mainstream programming languages, it generates a globally fresh Expi name and substitutes it for occurrences of the variable that was bound by the generation process.

Table 14. Internal reduction: $P \rightarrow_G Q$

(GRED-GEN)
$$\frac{a \notin \mathit{fn}\ (\mathsf{gen}\ x : T\ \mathsf{in}\ P \mid R)}{\mathsf{gen}\ x : T\ \mathsf{in}\ P \mid R \rightarrow_G P[x := a] \mid R}$$

(GRED-DESTR)
$$\frac{g(\widetilde{M}) \Downarrow N}{\mathsf{let}\ x = g(\widetilde{M})\ \mathsf{in}\ P\ \mathsf{else}\ Q \mid R \rightarrow_G P[x := N] \mid R}$$

(GRED-EQ)
$$\frac{P \equiv P' \qquad P' \rightarrow_G Q' \qquad Q' \equiv Q}{P \rightarrow Q}$$

(GRED-ELSE)
$$\frac{g(\widetilde{M}) \not\Downarrow}{\mathsf{let}\ x = g(\widetilde{M})\ \mathsf{in}\ P\ \mathsf{else}\ Q \mid R \rightarrow_G Q \mid R}$$

(GRED-I/O)
$$\frac{}{\mathsf{out}(a, M).P \mid \mathsf{in}(a, x).Q \mid R \rightarrow_G P \mid Q[x := M] \mid R}$$

(GRED-!I/O)
$$\frac{}{\mathsf{out}(a, M).P \mid !\,\mathsf{in}(a, x).Q \mid R \rightarrow_G P \mid Q[x := M] \mid !\,\mathsf{in}(a, x).Q \mid R}$$

The rule RED-FORK is not explicitly present in $\rightarrow_G$, it is merged into the other rules (where applicable) in the form of an additional contextual process R. The reason for that is the "global freshness" condition in $\mathsf{gen}\ x : T\ \mathsf{in}\ P$ – we cannot allow to reduce subprocesses independently from each other, because the reduction could introduce free names that are already used in other subprocesses.

In the setting of the pi calculus, the global semantics was shown to be equivalent to the usual scope-extrusion-based semantics [27]. This makes the Global Expi calculus a perfect intermediate language for our translation. This allows us to completely avoid dealing with scope extrusion when translating Global Expi to VPJ.

The translation from Extensible Spi calculus to Global Expi calculus is very simple. Both calculi use the same terms, types and configuration, we just use them as they are. The only interesting case is the translation of the restriction process $\mathsf{new}\ a : T.P$ to the generation process $\mathsf{gen}\ x : T\ \mathsf{in}\ P$. In this case we just substitute the name used by the restriction process with a fresh variable for the generation process.

A.6 More details about the VPJ type system

Expression typing in VPJ is the same as in Jinja with Threads, and therefore differs from Java in some details. For instance the conditional expression $\mathbf{if}\ (e_1)\ \{e_2\}\ \mathbf{else}\ \{e_3\}$ requires the type of e_2 to be the subtype of e_3 or vice versa and so resembles the Java's ternary conditional operator _ ? _ : _. On the other hand, method and class typing in VPJ are based on the definitions from Variant Parametric Featherweight Java [14]. All types used in the declaration of a well-typed method must be well-formed in a type variable environment that requires the method and class type parameters to be subtypes of their upper bounds (where method parameters override class parameters). The method body should be well-typed in the typing environment where the parameters have the corresponding types. The method declaration must override the same method declared in one of the superclasses (if any) with the same argument and return types. A class declaration is well-typed if all the types used in the declaration are well-formed and all method declarations are well-typed. Finally, a VPJ program is well-typed if all its class declarations and the main expression are well-typed.

A.7 Details of the Translation

Generating Classes for Expi Types. Expi type identifiers are translated into VPJ class declarations by rule TRANS-TCLASS in Table 15. The name of the resulting class is derived from the type identifier using the function $idtostr(x)$. We use either

`AbstractGenerativeBase` or `AbstractBase` as the superclass, depending on whether the type is generative or not. The type parameters of the Expi type are encoded as type parameters with fresh names in the generated VPJ class.

Table 15. Expi Types $\rightsquigarrow$ Variant Parametric Classes

(TRANS-TCLASS)

$$C := \mathit{idtostr}(t) \qquad \forall i \in [1,m].\ X_i^J \text{is fresh} \qquad D := \begin{cases} \texttt{AbstractGenerativeBase} & \text{if } t \in \mathit{Gen} \\ \texttt{AbstractBase} & \text{otherwise} \end{cases}$$

$$\left\{\left(U_1^J, x_1\right), \ldots, \left(U_k^J, x_k\right)\right\} := \left\{(\mathit{spitoj}_T(U_i), \mathit{idtostr}(f)\texttt{\_i}) \mid f \text{ and } U_i \text{ s.t. } f_A : \forall \widetilde{X}.\ (U_1, \ldots, U_i, \ldots, U_n) \mapsto t_A\langle\widetilde{X}\rangle\right\}$$

$$t_A\langle T_1, \ldots, T_m\rangle \rightsquigarrow$$

```
class C⟨[X_i^J ◁ AbstractBase]_m⟩ ◁ D {
  U_1^J x_1 ; … ; U_k^J x_k ;
  boolean ⟨⟩equals(C⟨X_1^J, …, X_m^J⟩ obj) {
    if (this = obj) { true }
    else if (obj = null) { false }
    else { if (this.x_i{C_i} = null) { this.x_i{C_i} = obj.x_i{C_i} }
           else { this.x_i{C_i}.⟨⟩equals (obj.x_i{C_i}) } ∧ … }
  } }
```

Notation: $[X_i]_n$ denotes the sequence $X_1, \ldots, X_n$

Instances of the generated classes are used to represent terms. Since terms are either generated by the process gen $x : T$ in P for a generative type T or constructed out of other terms using a constructor, we preserve this structure in our model to be able to access subterms in destructors. In particular, we store the name of the constructor that was used to create the term in a field named `ctorName` of type **string** (defined in `AbstractBase`). This field is set to **null** if the term is created using the gen process. We also store the arguments given to the constructor used to create the term in fields. We generate a field for each argument of every Expi constructor that returns the corresponding type.

Each class representing an Expi type has a method called `equals`, which takes one argument of the same type as the class where it is declared and returns a **boolean**. This method implements structural equality on Expi terms by comparing the constructor names of the terms and recursively comparing their arguments. We use pointer equality to compare terms created by the gen process.

Type annotations, for instance in restrictions, are translated into the corresponding VPJ class type as follows: Type Top is translated into `AbstractBase`. Type Channel is translated into the parametric type `AbstractChannel` instantiated with the translation of the type parameter. Custom types $t_A\langle\widetilde{X}\rangle$ are translated into corresponding classes. Translating the variances of parametric types is, however, quite tricky. The type systems of the Expi calculus and VPJ both support parametric types with variances, but use two completely different approaches to define *which* variance should be used. In the Expi calculus, the variance of each type parameter is fixed by the definition of the type. VPJ uses a more flexible approach coming from Java, here the class declaration only declares the names of type parameters and the variance is chosen when the class is *used*, i.e., in the **new** expression or in the declaration of method parameters or local variables. In both cases, the variances have a similar effect on the subtyping relation, and we translate them in the way that makes the VPJ type system behave like the one for Expi. In particular, there are only a few cases where subtyping plays a role in the Expi calculus: when a term is sent over a channel and when it is used in a constructor or destructor application. We ignore the case with the channel, because the channel type is invariant. This leads to the following

strategy: we translate all types used in processes as invariant and only use variances in the parameter declarations of the Expi constructor and destructor methods. This is achieved by two slightly different functions: $spitoj_T$, which ignores variances, and $spitoj_{T\mathcal{V}}$, which uses the variance for each type parameter from the configuration.

Generating Expi Constructor Methods. Expi constructors are translated into special methods in class `Fun`, as defined by the rule TRANS-CTOR. This class has no other purpose but being a container for constructor and destructor methods. If the constructor has a parametric type, the resulting method declaration is parameterized by type variables with `AbstractBase` as their upper bounds.

Table 16. Expi Constructors $\rightsquigarrow$ Constructor Methods

(TRANS-CTOR)

$$\frac{\begin{array}{l} T^J := spitoj_T(T) \qquad \forall i \in [1,k].\ X_i^J := spitoj_T(X_i) \qquad mthd := idtostr(f)\texttt{\_Ctor} \\ m := \begin{array}{l} T^J\ \langle \left[X_i^J \lhd \texttt{AbstractBase}\right]_k \rangle mthd(\textbf{string}\ \texttt{algName}, \left[spitoj_{T\mathcal{V}}(T_j)\ \texttt{arg}_j\right]_n) \\ \quad \{\ \textbf{new}\ T^J\ \left(\texttt{algName}, \text{"}\texttt{f}\text{"}, \left[\texttt{arg}_j\right]_n\right)\ \} \end{array} \end{array}}{f : \forall\, [X_i]_k.\ ([T_j]_n) \mapsto T \quad \rightsquigarrow \quad m}$$

We pass the algorithm name used in the constructor application as the first argument to the method. The types of the constructor arguments may contain variances, and we use the function $spitoj_{T\mathcal{V}}$ to translate them. The body of an Expi constructor method simply creates an instance of its return type and initializes all fields using the corresponding values, i.e., the algorithm name, constructor name and the remaining arguments.

Generating Expi Destructor Methods. Destructors are also translated into parameterized methods in class `Fun`. The destructor reduction relation in the Expi calculus works by pattern matching the arguments given to the destructor and then creating the resulting term from the matched terms using constructor applications. Unfortunately neither VPJ nor real Java support this form of pattern matching, so we need to implement it by hand. We do this by generating the code for destructor methods that performs the pattern matching for the corresponding reduction rule. Our pattern matching algorithm is repeated for each reduction rule until one of them applies. For each rule we perform the following three steps: First, we compare the structure of each method argument and the corresponding reduction rule argument. We compare the constructor name and cryptographic algorithm names stored in the objects and abort if one of the terms does not match. We then build a list of "access expressions" that can be used to access the subterms matched by the variables contained in the reduction rule. Finally, we compare the corresponding variables with each other using their `equals` methods and, provided all equality checks succeeded, construct the return term using constructor method calls and the matched variables.

Terms. We translate terms into VPJ expressions using function $spitoj_M$ defined in Table 17. Expi names and variables are represented by local variables of the corresponding type. These local variables are declared by the translations of the corresponding processes (see Table 18). Constructor applications are translated into calls to the corresponding constructor method using a temporary instance of the `Fun` class.

Table 17. Terms $\rightsquigarrow$ VPJ Expressions

$$\begin{array}{ll} spitoj_M(u) & = \mathtt{x}_u \\ spitoj_M(f_A\langle [T_i]_m \rangle ([M_j]_n)) & = (\textbf{new}\ \texttt{Fun}\,()).\langle [spitoj_T(T_i)]_m \rangle idtostr(f)\texttt{\_Ctor}\ \left(A, [M_j]_n\right) \end{array}$$

Processes. Each processes in the Expi calculus is translated into a VPJ expression and a set of class declarations (see Table 18 and please refer to §A.8 for all the rules). In most cases, the translation is quite obvious, all subprocesses, subterms and types are translated using the corresponding functions and no new class declarations are generated.

Table 18. Processes $\rightsquigarrow$ VPJ Expressions and Class Declarations (excerpt)

(TRANS-BANGIN)

$$\frac{P \rightsquigarrow (e, L) \qquad \texttt{x} \text{ is fresh} \qquad C \text{ is fresh} \qquad V := \left\{(\texttt{y}, T^J) : \texttt{y} \text{ is in scope} \wedge y : T^J\right\} \qquad \mathit{spitoj}_T\,(\mathit{typeof}(c)) = \texttt{AbstractChannel}\langle T^C\rangle}{!\,\mathsf{in}(c,x).P \rightsquigarrow \left(\begin{array}{l}\textbf{while}\,(\textbf{true})\,\{\\ \quad T^C\ \texttt{x};\ x := c.\langle\rangle\texttt{receive}\,();\quad ,\, L \cup \{\mathit{thread}(C,V,e)\}\\ \quad (\textbf{new}\ C\langle\rangle\,(V)).\langle\rangle\texttt{start}\,()\ \ \}\end{array}\right)}$$

(TRANS-GEN)

$$\frac{P \rightsquigarrow (e, L) \qquad T^J = \mathit{spitoj}_T\left(t_A\langle\widetilde{T}\rangle\right)}{\mathsf{gen}\ x : t_A\langle\widetilde{T}\rangle\ \mathsf{in}\ P \rightsquigarrow \left(\{T^J\ x\ ;\ x := \textbf{new}\ T^J\,(A, \textbf{null}, \ldots, \textbf{null});\ e\}, L\right)}$$

(TRANS-FORK)

$$\frac{P \rightsquigarrow (e_P, L_P) \qquad Q \rightsquigarrow (e_Q, L_Q) \qquad C_P, C_Q \text{ are fresh} \qquad x_P, x_Q \text{ are fresh} \qquad V := \left\{(\texttt{y}, T^J) \mid \texttt{y} \text{ is in scope and } \texttt{y} : T^J\right\}}{P \mid Q \rightsquigarrow \left(\begin{array}{l}\{\ C_P\langle\rangle\ x_P;\ C_Q\langle\rangle\ x_Q;\\ \quad x_P := \textbf{new}\ C_P\langle\rangle\,(V);\\ \quad x_Q := \textbf{new}\ C_Q\langle\rangle\,(V);\\ \quad x_P.\langle\rangle\texttt{start}\,();\quad x_Q.\langle\rangle\texttt{start}\,();\\ \quad x_P.\langle\rangle\texttt{join}\,();\quad x_Q.\langle\rangle\texttt{join}\,()\ \}\end{array}\ ,\ \left(\begin{array}{l}L_P \cup L_Q\\ \cup\{\mathit{thread}(C_P,V,e_P)\}\\ \cup\{\mathit{thread}(C_Q,V,e_Q)\}\end{array}\right)\right)}$$

The semantics of the replication process $!\,\mathsf{in}(c,x).P$ is to spawn a new copy of the continuation process P whenever a message x is received on channel c. We implement this using an endless loop where we block on the channel trying to receive some data and spawn a new thread implementing the process P as soon as we receive something (see TRANS-BANGIN). The translation of the $\mathsf{gen}\ x : T\ \mathsf{in}\ Q$ process declares a new local variable and stores a newly generated instance of the corresponding type in it (see TRANS-GEN).

Parallel composition is modeled using threads and has the most complicated translation (see TRANS-FORK). We declare two new thread classes as using the $\mathit{thread}(C, V, e)$ function, which defines two direct subclasses of `Thread` and overrides the method `run` (see §A.8). The generated expression starts these threads and calls `join` on them. We copy all currently visible local variables V into each thread and store them in fields with the same name to mimic the semantics of the Expi calculus, where all previously defined Expi names and variables are visible in all subprocesses.

The Final Result. We translate the configuration, a list of fresh class names, a Global Expi process and a list of free Expi names that occur in that process into a VPJ program (i.e., class declarations) and a target expression. The resulting program contains declarations of the standard VPJ classes (i.e., `Object`, `Thread` etc.), the symbolic library classes (see Figure 6) and the custom classes generated by the rule TRANS-TCLASS. We declare class `Fun` and add the translations of all constructors and destructors to it. Finally, we translate the free Expi names occurring in the Global Expi process to local variable names and translate the Global Expi process in the scope of these local variables.

A.8 Process Translation Rules

The $\mathbf{0}$ process is translated into a "do-nothing" operation. The process let $x = G$ is translated into a try-catch block where we declare a new local variable x, assign it the result of the destructor application (translated into the destructor method call) and proceed with the else-branch if an `ELetProcess` exception is caught (see TRANS-LET). The output process $\mathsf{out}(c, M)$ calls the `send` method on the channel variable c and gives it the translation of the term $spitoj_M(M)$ as an argument. The $\mathsf{in}(c, x)$ process declares a new local variable x and assigns the result of a call to `receive` method on the channel variable c to x.We have two almost identical translation rules (TRANS-GEN and TRANS-GENCH) because we need to distinguish between the custom Expi type and Channel when choosing the name of cryptographic algorithm used to initialize the fields of the new object.

Table 19. Processes $\rightsquigarrow$ VPJ Expressions and Class Declarations

(TRANS-NULL)
$$\frac{}{\mathbf{0} \rightsquigarrow (\mathbf{unit}, \emptyset)}$$

(TRANS-BANGIN)
$$\frac{P \rightsquigarrow (e, L) \quad \texttt{x} \text{ is fresh} \quad C \text{ is fresh} \quad V := \{(\texttt{y}, T^J) : \texttt{y} \text{ is in scope} \wedge y : T^J\} \quad spitoj_T\,(typeof\,(c)) = \texttt{AbstractChannel}\langle T^C\rangle}{! \,\mathsf{in}(c, x).P \rightsquigarrow \left(\begin{array}{l} \mathbf{while}\ (\mathbf{true})\ \{ \\ \quad T^C\ \texttt{x};\ x := c.\langle\rangle\texttt{receive}\,();\ \ , L \cup \{thread(C, V, e)\} \\ \quad (\mathbf{new}\ C\langle\rangle\,(V)).\langle\rangle\texttt{start}\,()\ \} \end{array}\right)}$$

(TRANS-OUT)
$$\frac{P \rightsquigarrow (e, L)}{\begin{array}{c}\mathsf{out}(c, M).P \quad \rightsquigarrow \\ (c.\langle\rangle\texttt{send}\,(spitoj_M(M))\,;\ e, L)\end{array}}$$

(TRANS-IN)
$$\frac{P \rightsquigarrow (e, L) \qquad x \text{ is fresh} \qquad spitoj_T\,(typeof\,(c)) = \texttt{AbstractChannel}\langle T^C\rangle}{\mathsf{in}(c, x).P \rightsquigarrow \left(\{T^C\ x\ ;\ x := c.\langle\rangle\texttt{receive}\,();\ e\}, L\right)}$$

(TRANS-LET)
$$\frac{\begin{array}{c}P \rightsquigarrow (e_P, L_P) \quad Q \rightsquigarrow (e_Q, L_Q) \quad G = g_A\langle[X_i]_m\rangle\,\left([M_j]_n\right) \quad mthd := idtostr(\texttt{g})\texttt{\_Dtor} \\ \forall i \in [1, m].\ X_i^J = spitoj_T(X_i) \qquad \forall j \in [1, n].\ e_j = spitoj_M(M_i)\end{array}}{\mathsf{let}\ x = G\ \mathsf{in}\ P\ \mathsf{else}\ Q \rightsquigarrow \left(\begin{array}{l} \mathbf{try}\ \{\ T^C\ \texttt{x}; \\ \quad \texttt{x} := (\mathbf{new}\ \texttt{Fun}\,()).\langle\left[X_i^J\right]_m\rangle mthd\,\left([e_j]_n\right);\ , L_P \cup L_Q \\ \quad e_P\ \}\ \mathbf{catch}\ (\texttt{ELetProcess ex})\ \{\ e_Q\ \} \end{array}\right)}$$

(TRANS-GEN)
$$\frac{P \rightsquigarrow (e, L) \qquad T^J = spitoj_T\left(t_A\langle\widetilde{T}\rangle\right)}{\mathsf{gen}\ x : t_A\langle\widetilde{T}\rangle\ \mathsf{in}\ P \rightsquigarrow \left(\{T^J\ x\ ;\ x := \mathbf{new}\ T^J\,(A, \mathbf{null}, \ldots, \mathbf{null});\ e\}, L\right)}$$

(TRANS-GENCH)
$$\frac{P \rightsquigarrow (e, L) \qquad T^J = spitoj_T\,(\mathsf{Channel}_A\langle T\rangle)}{\begin{array}{c}\mathsf{gen}\ x : \mathsf{Channel}_A\langle T\rangle\ \mathsf{in}\ P \quad \rightsquigarrow \\ \left(\{T^J\ x\ ;\ x := \mathbf{new}\ T^J\,(A, \mathbf{null}, \mathbf{new}\ \texttt{Semaphore}\,(0), \mathbf{new}\ \texttt{Semaphore}\,(0));\ e\}, L\right)\end{array}}$$

(TRANS-FORK)
$$\frac{\begin{array}{c}P \rightsquigarrow (e_P, L_P) \qquad Q \rightsquigarrow (e_Q, L_Q) \qquad C_P, C_Q \text{ are fresh} \qquad x_P, x_Q \text{ are fresh} \\ V := \{(\texttt{y}, T^J) : \texttt{y} \text{ is in scope and } y : T^J\}\end{array}}{P \mid Q \rightsquigarrow \left(\begin{array}{ll} \{\ C_P\langle\rangle\ x_P;\ C_Q\langle\rangle\ x_Q; & \\ \quad x_P := \mathbf{new}\ C_P\langle\rangle\,(V); & L_P \cup L_Q \\ \quad x_Q := \mathbf{new}\ C_Q\langle\rangle\,(V); & , \cup \{thread(C_P, V, e_P)\} \\ \quad x_P.\langle\rangle\texttt{start}\,();\ \ x_Q.\langle\rangle\texttt{start}\,();\ & \cup \{thread(C_Q, V, e_Q)\} \\ \quad x_P.\langle\rangle\texttt{join}\,();\ \ x_Q.\langle\rangle\texttt{join}\,()\ \} & \end{array}\right)}$$

The $thread(C, V, e)$ function generates a class declaration of a thread that can execute given VPJ expression. It is used in process translation rules TRANS-BANGIN and TRANS-FORK defined in Table 19 to update the set of class declarations. The semantics of thread classes in VPJ follows the semantics of Java. VPJ defines the class `Thread` with methods `start`, `run` and `join`. The semantics of `start` is to spawn a new thread and call the method `run` in that thread. We simply override the method `run` to start the expression we need.

Table 20. Definition of $thread(C, V, e)$

$$
\begin{aligned}
thread(C, \left\{\left(\widetilde{\mathtt{x}}, \widetilde{T^J}\right)\right\}, e) := \quad & \mathbf{class}\ C\langle\rangle \lhd \mathtt{Thread}\ \{ \\
& \quad \widetilde{T^J}\ \widetilde{\mathtt{x}}\ ; \\
& \quad \mathbf{void}\ \langle\rangle \mathtt{run}()\ \{ \\
& \quad\quad \widetilde{T^J}\ \widetilde{\mathtt{x}}; \\
& \quad\quad \mathtt{x} := \mathbf{this}.\mathtt{x}\{C\}; \\
& \quad\quad \mathbf{try}\ \{e\}\ \mathbf{catch}\ (\mathtt{ELibrary}\ \mathtt{ex})\ \{\mathbf{unit}\} \\
& \quad \} \\
& \}
\end{aligned}
$$