Kant: A Domain-Specific Language for Modeling Security Protocols

C. Braghin*[a], M. Lilli[b], E. Riccobene[c], K. Notari and Marian Baba
Department of Computer Science, Università degli Studi di Milano, Italy

Keywords: Model-Driven Language Engineering, Security Protocol Specification and Analysis, Language Validation.

Abstract: Designing a security protocol is a complex process that requires a deep understanding of security principles and best practices. To ensure protocol effectiveness and resilience against attacks, it is important to strengthen *security by design* by supporting the designer with an easy-to-use, concise, and simple notation to design security protocols in a way that the protocol model could be easily mapped into the input model a verification tool to guarantee security properties. To achieve the goal of developing a DSL language for security protocol design, working as the front-end and easy-to-use language of a formal framework able to support different back-end tools for security protocol analysis, we present the abstract and concrete syntaxes of the *Kant* (Knowledge ANalysis of Trace) language. We also present a set of validation rules that we have defined to help the designer, already at design time, to avoid common security errors or to warn him/her regarding choices that might lead to protocol vulnerabilities. The effectiveness of Kant's expressiveness is discussed in terms of a number of case studies where Kant has been used for modeling protocols.

1 INTRODUCTION

Security protocols are a set of procedures defining how data should be transmitted, encrypted, authenticated, and protected to ensure the security and integrity of information during communication (Anderson and Needham, 1995). Despite their apparent simplicity, designing a security protocol is a complex process that requires a deep understanding of security principles and best practices. Although a large number of security protocols have been developed and implemented to provide security guarantees, the development of security protocols is particularly prone to errors: many communications protocols do not use up-to-date security features or implement them incorrectly (Basin et al., 2018). Security vulnerabilities and errors cannot be detected only by functional software testing because they appear in the presence of a malicious adversary. For this reason, many published protocols have proven to be faulty many years after their practical usage (Tobarra et al., 2008).

To ensure the protocol's effectiveness and resilience against attacks, it is important to strengthen security at design-time by achieving the so-called *security by design*. In fact, defects found after the design phase are expensive to fix, and the cost increases exponentially during the subsequent development phase (Haskins et al., 2004). Thus, protocol design should be complemented by rigorous analysis, and formal verification of security protocols has become a key issue.

Most of the tools developed in the past for protocol verification were not widely adopted by the industry, and the reason is strongly related to the fact that formal method usage requires a strong mathematical background, which many designers or engineers do not hold (Davis et al., 2013). Additionally, (i) validation tools are often based on modeling languages with poor usability, making the writing of the model error-prone as well; (ii) the verification process is not a *push-bottom* activity and requires user intervention along with the proof and knowledge of the analysis mechanism; (iii) the verification results are often difficult to interpret and to cross check with the original protocol; (iv) verification approaches have different goals and sometimes the integrated use of various tools could be useful for more complete protocol analysis. However, tools integration is difficult to achieve (Heinrich et al., 2021) since each tool has its own input language and modeling primitives, thus using different tools often requires starting modeling the same protocol from scratch each time.

[a] https://orcid.org/0000-0002-9756-4675
[b] https://orcid.org/0000-0001-7236-9171
[c] https://orcid.org/0000-0002-1400-1026
*This work was partially supported by project SERICS (PE00000014) under the MUR National Recovery and Resilience Plan funded by the European Union - NextGenerationEU.

Braghin, C., Lilli, M., Notari, E. and Baba, M.
Kant: A Domain-Specific Language for Modeling Security Protocols.
DOI: 10.5220/0012386400003645

In *Proceedings of the 12th International Conference on Model-Based Software and Systems Engineering (MODELSWARD 2024)*, pages 62-73
ISBN: 978-989-758-682-8; ISSN: 2184-4348

In this paper, we present the first results of an ongoing research project to provide practitioners with a formal framework able to support different kinds of analysis for security protocols, but at the same time easy to use and with a reduced gap between designer's background and formal notations.

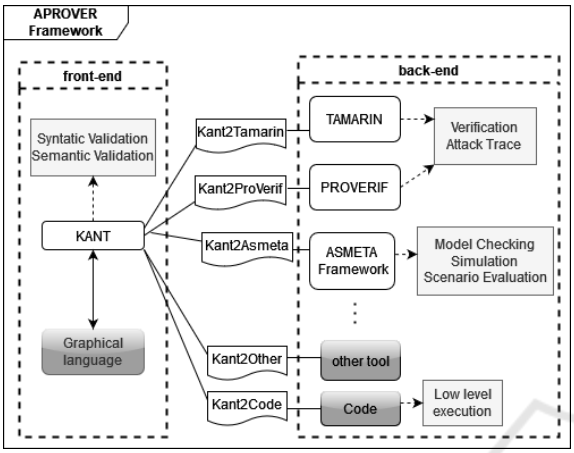

Figure 1: APROVER Framework.

Fig. 1 shows the architecture of the APROVER (Automatic PROtocol VERifier) framework under development. We envision it as composed of a modeling front-end on top of richer and more specific modeling, analysis, and implementation back-end frameworks. Following this rationale, the modeling front-end should provide an easy-to-use, concise, and simple notation to design security protocols. By exploiting the model-based software engineering approach, this could be achieved by defining a DSL (Domain Specific Language) for protocol specification, and possibly developing a textual and graphical notation (with interchangeable models) for it. More specific models or code in the target back-end frameworks/-platforms could be synthesized from these DSL models by defining/developing suitable model transformations towards input models of the target back-end tools. ASMETA (Arcaini et al., 2011) for model simulation and model checking, PROVERIF (Blanchet, 2001) and TAMARIN (Meier et al., 2013) for trace analysis, are the tools under consideration as primary back-end tools, but other back-ends could be integrated into the APROVER framework by exploiting model transformations.

We here present *Kant* (Knowledge ANalyzing Tracer), a DSL explicitly designed as the front-end language of APROVER for the specification of security protocols. Kant is a high-level language endowed with primitives for protocol modeling and assumption expression. It has been conceived as a sort of *lingua franca* that allows simple textual descriptions to express protocols and easy translation into the input languages of various tools for protocol analysis. Since Kant has been engineered and developed by using the Langium platform[1], , its grammar definition comes with an editor and a number of other tools that allow checking Kant models not only for syntactic correctness, but also for consistency with respect to a given semantic model. The latter is expressed in terms of a set of validation rules, which describe both constraints and best practice principles in designing security protocols. Such a checking mechanism is extremely important at design-time to avoid common security errors (e.g., incorrect use of encryption keys), or to warn the designer regarding choices that might lead to protocol vulnerabilities (e.g., a principal's name not mentioned explicitly in the message). Note that in Fig. 1 blocks in grey are under development and the mapping from Kant to validation and verification tools is out of the scope of this work.

The paper is structured as follows. Section 2 recalls the concept of security protocol and presents a protocol that we use as a reference scenario throughout the paper. Section 3 introduces the Langium platform used to develop Kant as DSL, while Kant modeling primitives and constructs are given in Section 4. Section 5 presents the semantic model of Kant and shows the application of the validation rules. In Section 6 we show the use of Kant and the language validation mechanism on the reference scenario. Section 7 compares our results with respect to other research work in literature, and Section 8 concludes the paper and outlines future research directions.

2 REFERENCE SCENARIO

Although there is a wide range of protocols, differing by the number of principals (or actors), the number of messages, and the goals of the protocol (that may often be expressed with a list of desired security properties), they all share a common structure. Indeed, a communication protocol consists of a sequence of messages between two or more principals. Each message may be written by using the classical Alice&Bob notation in the form:

$$\mathsf{M1}.\ A \rightarrow B : message\_payload$$

which specifies:

- The *principals* (or *actors*) exchanging messages (in general, symbols A and B represent arbitrary principals, S a server). In particular, the direction of the arrow specifies the sender and the receiver of the message.

[1] https://langium.org/

- The order in which messages are sent, and their specific *payload*. In particular, M1 is a label identifying the message, whereas *message_payload* specifies the actual content of the message.

In secure protocols, payloads can be partially or totally ciphered, either by symmetric-key encryption (in this case, a key shared between actors is used both to encrypt and decrypt), or by asymmetric-key encryption (here, K_B and $K_B{}^{-1}$ is used to specify a public and private key of B, to encrypt and, respectively, decrypt). Message payloads can contain other information, such as nonces (N), timestamps (T), etc.

The security goals are often defined with respect to CIA (Confidentiality, Integrity, Authentication) triad. The most common are *confidentiality* or *integrity* of message payloads, or *entity authentication* (i.e., the process by which one entity is assured of the identity of a second entity that is participating in the same session of a protocol, thus, they share the same values of the protocol parameters, such as session keys, nonces, etc.).

Consider the classic Needham-Schroeder public-key protocol (NSPK, for short) that will be used throughout the paper as a running example to introduce the Kant notation.

M1. $A \rightarrow B:$	$\{A, N_A\}_{K_B}$
M2. $B \rightarrow A:$	$\{N_A, N_B\}_{K_A}$
M3. $A \rightarrow B:$	$\{N_B\}_{K_B}$

It was introduced in 1978 for mutual authentication (here, we omit the exchanges with the certification authority to get the public keys). It consists of three messages: in the first message, principal A sends to B a message containing her identity, A, and a nonce, N_A, to avoid replay attacks (i.e., reuse of old messages, often called a *challenge* message), that only B can decrypt with his private key. B's answer (message M2) is ciphered with A's public key and contains nonce N_A to authenticate B (he is the only one able to decrypt message M1 and obtain N_A in clear), and a nonce N_B to authenticate A with B. Since message M2 is encrypted with A's public key, she is the only one who can decrypt it, thus if B receives message M3 containing nonce N_B encrypted with his public key, A is authenticated, too.

3 Kant DEVELOPMENT

Kant (Knowledge ANalysis of Trace) is the domain-specific language we explicitly designed and implemented in Langium for the specification of security protocols. Using a tool to engineer a DSL offers several advantages that can streamline the development process and enhance the utility of the DSL, such as syntax highlighting, autocompletion, and debugging support.

In particular, Langium represents an innovative tool in the context of language engineering enabling DSL development in a web-based technology stack. When used in the context of a desktop app (e.g., VS Code, Eclipse IDE, etc.), Langium runs on the Node.js platform. Additionally, it can run in a web browser to add language support to web applications with embedded text editors (e.g., Monaco Editor). The interface between Langium and the text editor is the Language Server Protocol (LSP), allowing languages based on Langium to seamlessly interact with a range of popular IDEs and editors supporting LSP.

A *grammar language* is provided to specify the syntax and structure of the language. The grammar rules describe the concrete syntax by instructing the parser how to read input text, and also the abstract syntax in terms of meta-classes and their properties. For a given text document, Langium creates a data structure called Abstract Syntax Tree (AST): every grammar rule invocation leads to a corresponding node in the AST that is a JavaScript object, and Langium generates a TypeScript interface for every rule to provide static typing for these nodes. As programs written in the language are parsed, Langium automatically generates Abstract Syntax Trees based on these interfaces, making efficient manipulation and analysis of the parsed content possible. In particular, Langium allows to implement also custom validation without the need to involve complex external tools, thus ensuring that validation is defined alongside the grammar specification. It provides a cutomizable Validation Registry mapping validation functions with specific nodes in the AST, enabling targeted validation checks at relevant points in the AST (e.g., either internal or leaf nodes). Designer only need to define the validation functions encapsulating the desired checks on both the structural and semantic aspects of the language captured by the AST, and to update the Validation Registry with the new functions. This approach is great for rapid prototyping, when the focus is on designing the syntax of a new language.

Beyond parsing, Langium extends its capabilities to establish connections between different language elements, enabling cross-referencing and linking. These relationships play an important role, and Langium is capable of resolving them automatically thanks to built-in scoping and indexing.

Although Langium's grammar declaration language is similar to Xtext, they are built upon different open-source libraries and tools: Xtext is built upon

Eclipse and ANTLR, while Langium is built upon Visual Code and Chevrotain. Moreover, Xtext is heavily based on the Eclipse Modeling Framework, whereas Langium uses TypeScript interfaces, enabling language engineering in TypeScript, the same technology that is used for VS Code extensions. In contrast, building a tool that uses an Xtext-based language server with VS Code or Theia means creating a hybrid technology stack with some parts implemented in Java, others in TypeScript. Development and maintenance of such a mixed code base are more demanding, and long-term maintenance is likely more difficult compared to Langium's coherent technology stack.

4 Kant SYNTAX

In this section, we present the abstract and concrete syntax of the Kant language and we show how it encompasses all the essential aspects of security protocols. In particular, it integrates useful features that can be helpful for the verification process and provide valuable insights for protocol correctness at the design phase.

The syntax includes constructs of the notations commonly used to express security protocols (e.g., the Alice&Bob notation); however, it has been extended with the concept of *knowledge exchange* between the parties involved in the security protocol session to help reasoning and analyzing the flow of information during the communication.

Kant has been developed by exploiting the MDE approach for a DSL definition, so its abstract syntax is given in terms of a metamodel from which a concrete textual notation is derived. Kant's meta-model (see Fig. 2) defines six mandatory meta-classes and two optional ones, which are necessary to specify a valid *protocol model* in Kant. The protocol must contain specific elements, including the definition of *principals* involved in the protocol (such as Alice, Bob, and Server), the *types* used in the function definitions and the *definition* of cryptographic *functions* and their inverse functions, the *knowledge* that each *principal* has during the message exchange, and the definition of the *communication mechanism* for messages exchange.

In addition to these mandatory model sections, two optional meta-classes can be used to define *shared knowledge* between principals, and to help the user writing, in a human-readable style, either *properties* or constraints on the model elements, and security *checks*. These statements are arguments for the validation and verification of the model.

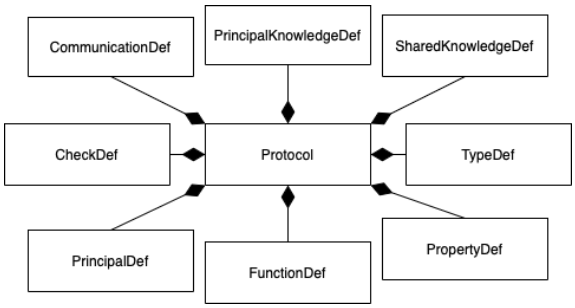

Figure 2: Protocol Meta-model.

Principal Definition. A protocol model (instance of the meta-model) written in Kant starts with the name declaration of the `principals` involved in the message exchange. Two examples of declarations follow (the user is free to select the most suitable name):

```
principal Alice
principal Alice, Bob, Server
```

Type and Function Definitions. We designed the language with static analysis in mind, so we included types it. The user can use built-in types that are declared in the language *prelude*, but he/she can also define other types. An example of `type definition` is the following:

```
type SymmetricKey, PublicKey, BitString, Group
```

where `SymmetricKey` and `PublicKey` are types of cryptographic keys used for symmetric and asymmetric encryption, respectively; `Bitstring` is a special type we call 'sink type' since a function that accepts a Bitstring as a parameter can accept any other type; `Group` represents elements of a Abelian group, i.e., a set of elements with commutative operations.

Types are used to categorize information in the principal's knowledge and allow the validation of the usage of piece of information through the protocol. Moreover, since some input languages of the back-end tools are typed, using types at the top-level of Kant model removes the need for a conversion step.

The `FunctionDef` meta-class enables users to define custom functions in agreement with the symbolic model (Blanchet, 2012) to describe both invertible and one-way functions. A function is defined by a

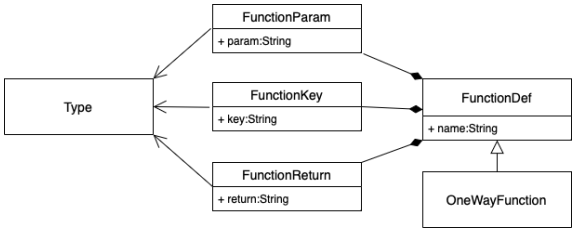

Figure 3: Function Definition Meta-model.

name, one or more parameters, and one or more return values. Each function component requires an identifier and a type, either pre-defined or custom-defined to suit the particular use case. Cryptographic functions, in addition, require the definition of a key used to encrypt data (which are the function parameters).

The separation of the FunctionDef in the composing meta-classes (FunctionParam, Functionkey and FunctionReturn, see Fig. 3) enables us to define validation rules (see Section 5) that are specific to each class and, in turn, helps to streamline the validation process.

Examples of function definitions are given below for the function ENC to encrypt data with a given key and the one-way cryptographic HASH function.

```
function ENC(content:BitString)
   with k:SymmetricKey -> [ enc:Ciphertext ]

function HASH(value:BitString)
                  -> [ hash:Digest ] one way
```

Property Definition. Properties can be added to a model to express constraints on model elements (e.g., on functions and their parameters) or equivalence properties (e.g., Diffie-Hellman exponentiation, as well as other equational theories). For example, the following property states the identity function as the result of applying description on encrypted data by using a symmetric key.

```
property forall x:BitString, k:SymmetricKey |
           DEC(ENC(x) with k) with k -> [ x ]
```

An example of a property stating function equivalence is the following that guarantees the Diffie-Hellman equivalence between two private keys a and b on a generator g in a finite cyclic group . [2]

```
property forall a:PrivateKey, b:PrivateKey,
            g:Group | DF(EXP(g,b),a)
            equals DF(EXP(g,a),b)
```

Knowledge Definition. Kant allows declaring the knowledge of the principals. There are two ways to accomplish this: (*i*) at the beginning of the protocol model, define the knowledge that is *shared* by selected principals in the initial state of the protocol run (by the construct share of the meta-class SharedKnowledgeDef); (*ii*) define *private* principal's knowledge at any point of the protocol model, but before sending a message (by the construct know in the meta-class PrincipalKnowledgeDef). In both cases of knowledge definition, two qualifiers can be used to distinguish between knowledge that is regenerated every time the protocol is executed (fresh knowledge at any protocol session), and knowledge that remains the same (constant knowledge during a protocol session) – both qualifiers are defined in the meta-class KnowledgeDefBuiltin on top of the meta-classes SharedKnowledgeDef and PrincipalKnowledgeDef.

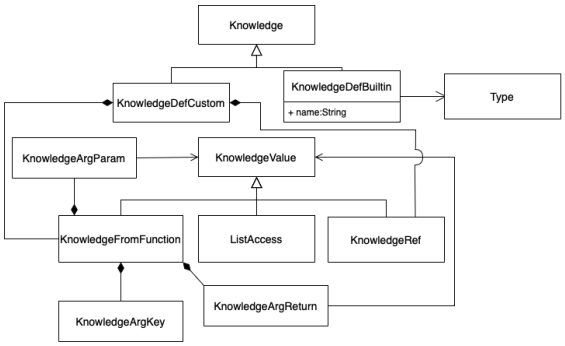

Figure 4: Knowledge Meta-model.

Private knowledge of a principal can include the specification of its own Finite State Machine (FSM): principal's initial state is specified as

```
state initial_state;
```

at the beginning of the protocol model; information on a reached state and its enabled transition can be specified at any point of the protocol model as:

```
state reached_state;
transition previous_state
=> reached_state;
```

Information relative to principal's FSM is primarily used in verification, but it is also useful in validation to analyze the knowledge flow of principals[3].

Two examples of shared and private knowledge specification follow:

```
Bob,Alice share{
    const key:SymmetricKey;}

Alice know{
    state second_state_Alice;
    transition first_state_Alice
    => second_state_Alice;
    fresh priv_a:PrivateKey;
    pub_a = PUB_GEN(priv_a);
    enc_mess = ENC(m) with key;}
```

[2]According to the Diffie-Hellman key exchange protocol, the shared secret between two parties is given by the formula $K = (g^a)^b \mod p = (g^b)^a \mod p$, being p a prime number and g a primitive root modulo p on which the two parties agree and that are public values, a and b secret values chosen by Alice and Bob, respectively.

[3]Since not a mature feature yet, Kant allows for drawing the FSM of each principal from his/her knowledge block.

The knowledge block is used to specify what a principal knows at the application of a given protocol rule. Fig. 4 shows the relation among the meta-classes for knowledge representation.

Each information is identified by a reference `priv_a` of a given type `PrivateKey` in `priv_a:PrivateKey` and can be constant or fresh (from the meta-class `KnowledgeDefBuiltIn`). Additionally, information of meta-class `KnowledgeDefCustom` is of the form `ref= ...` and can refer to the result of a function application (e.g., `puba=PUB_GEN(priva)`), or it can reference some saved information derived from the received messages (e.g., `dec_na = PKE_DEC(enc_na)`) or built to be sent as a message (e.g., `enc_na = PKE_ENC(na)`) (see the reference scenario model, in the following).

Particular importance is given to the two built-in functions that we have defined to facilitate list management (and that we use to model our reference scenario). The language prelude contains the basic types and functions: CONCAT and SPLIT of the meta-class `ListAccess`; the former yields the reference of a list concatenating a sequence of elements; the latter returns the list of elements from a given list reference. The property that derives from the two functions definition (see below) guarantees that concatenating a sequence of elements into a referenced list and then sliting such referenced list, holds the original list.

```
function CONCAT(...values: BitString)
                    -> [ value: BitString ]
function SPLIT(value: BitString)
                    -> [ values: BitString ]
property forall v: BitString |
                SPLIT(CONCAT(...v)) -> [ v ]
```

Principals Communication. A principal `Alice` builds within its knowledge a piece of information, `enc_mess`, that it wants to send as a protocol message to one of the other participants, `Bob`. To send the message, we exploit the concrete syntax of the meta-class `CommunicationDef`:

```
Alice->Bob: enc_mess
```

Security Checks. Kant grammar allows the user to specify security properties, which are verified once the model is transformed into a valid model for the back-end verification tools. The validation rules introduced in Sect. 5 can guarantee a lightweight form of static analysis of the information flow. Dynamic analysis can be performed only by back-end verification tools.

We provide three meta-classes, `ConfidentialityCheck`, `EquivalenceCheck`, and `AuthenticationCheck`, to express different kinds of security checks. The concrete syntax follows:

- *ConfidentialityCheck*: an information `m` must be known `only` by principals `Alice` and `Bob`

```
only Alice,Bob should know m
```

- *EquivalenceCheck*: this is used to check whether two pieces of information are equal or not; this is relevant for a security protocol when communication happens in an insecure channel and some information can be stolen or altered during the protocol run.

```
g_ab, g_ba should be equal
```

- *AuthenticationCheck*: the nounce `nb` allows `Alice` to `authenticate` principal `Bob`.

```
Alice should authenticate Bob with nb
```

Kant Model of the Reference Scenario. The following Kant model [4] is the specification of the NSPK security protocol described in Sect. 2: Alice and Bob have an associated FSM; they do not share any knowledge; each principal has private keys, uses a public key (generated from the corresponding private one) to encrypt messages, decrypts messages by using its own private key, generates fresh nonces and builds private knowledge according to the protocol rules.

```
principal Alice, Bob

Alice know {
    state sending_puba;
    const priva: PrivateKey;
    puba = PUB_GEN(priva);}

Alice -> Bob : puba

Bob know {
    state waiting_puba;
    const privb: PrivateKey;
    pubb = PUB_GEN(privb);}

Bob -> Alice : pubb

Alice know {
    state waiting_pubb;
    transition sending_puba=>waiting_pubb;
    fresh na: Nonce;
    enc_na = PKE_ENC(na) with pubb;}

Alice -> Bob : enc_na

Bob know {
    state waiting_enc_na;
    transition waiting_puba=>waiting_enc_na;
    fresh nb: Nonce;
    dec_na = PKE_DEC(enc_na) with privb;
```

[4] All the Kant language artifacts (grammar, models, validation rules, etc) are available at https://github.com/Aprover/kant on GitHub.

```
    nb_na = CONCAT(nb, dec_na);
    enc_nb_na = PKE_ENC(nb_na) with puba;}

Bob -> Alice : enc_nb_na

Alice know {
    state waiting_enc_na_nb;
    transition waiting_pubb =>
      waiting_enc_na_nb;
    dec_na_nb = PKE_DEC(enc_nb_na) with priva;
    rec_na_nb = SPLIT(dec_na_nb);
    enc_nb = PKE_ENC(rec_na_nb[1]) with pubb;}

Alice -> Bob : enc_nb

Bob know {
    state waiting_enc_nb;
    transition waiting_enc_na=>waiting_enc_nb;
    dec_nb = PKE_DEC(enc_nb) with privb;}

check nb, dec_nb should be equal
check Bob should authenticate Alice with nb
```

5 Kant VALIDATION RULES

A model written in a DSL is an instance of the language meta-model. Thanks to Langium, a model can be validated according to certain validation rules and constraints, covering both syntactic and semantic aspects. These rules have to be defined at the meta-model level. Error and warning messages can be reported upon model validation.

In Kant, the model validation phase helps the protocol designer to avoid common mistakes such as incorrect use of encryption keys, or definition of incorrect knowledge flows. Moreover, Kant grammar has some restrictions on the use of undeclared names of functions, types, and principals, and these restrictions are captured by the use of *cross-references* in Langium (see Sect. 3). Thus, it is possible to eliminate incorrect spelling of terms during model writing and to suggest, by means of auto-completion, only what is allowed.

Besides the advantages offered by the cross-reference mechanism in terms of revealing mistakes and making suggestions, in order to improve Kant model validation, we defined and implemented three sets of validation rules, whose violations result in errors or warnings. Rules for *syntactic checks* (see Sect. 5.1) are used to reveal syntactic errors that must be corrected. Rules for *semantic checks* (see Sect. 5.2) can reveal potential errors or violations of security properties. Rules for *prudent engineering practices* (see Sect. 5.3) work as guidelines for security protocol specification following some of the principles outlined in (Abadi and Needham, 1996); they help to identify and prevent common error patterns found in the literature, which might lead to attacks. Especially for semantic and prudent practice rules, a warning serves as a suggestion to improve the clarity of the model, or to adopt good practices in writing the protocol.

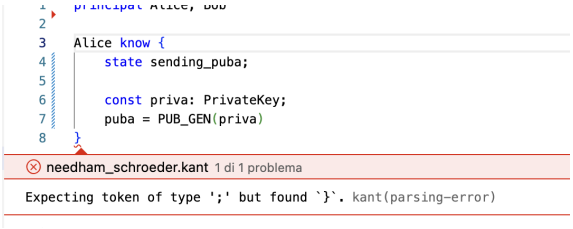

Figure 5: Semicolon placement error.

All validations are performed in real-time and are triggered by the user entering new knowledge. In section 5.1.1 we show an example taken from the GUI of Visual Studio Code, subsequent error reports are shown in plain text for better readability. We convey to mark incorrect elements by a *red* underline, and a warning by a *yellow* underline. We use fragments of the Kant model of the NSPK protocol to show the application of the validation rules.

5.1 Syntactic Checks

5.1.1 Syntax Well-Formedness

This set of validation rules checks for the correct formatting of parentheses, comments, and delimiters when writing a protocol in Kant. The example in Fig. 5 catches the error of a missing semicolon, which is used to declare separate knowledge.

5.1.2 Knowledge Declaration

New knowledge must be either fresh, constant or the result of a function. When declaring fresh or constant knowledge, the type must also be specified.

```
Alice know {
    state sending_puba;
    const priva;}
```

The error message specifies the expected token sequence:
Expecting: one of these possible token sequences:

1 `[ID : [Type:ID]]`

2 `[ID : [Type:ID], ID : [Type:ID]]`

but found: 'priva'.

5.1.3 Naming Convention

The naming convention in Kant requires that principals' names start with a capital letter, function names are capitalized, and variables are in lower case.

```
principal alice , Bob
```

In this case, it is raised a warning since this is a stylistic convention and not a real error:
Principal name alice should start with a capital letter.

5.1.4 Keyword Usage

Keywords can only be used in the way specified by the syntax (similarly to other programming languages).

```
Bob know {
     state state;}
```

This means that it is not possible to use "state" as a name for a FSM state. The error produced is:
Expecting token of type 'ID' but found 'state'.

5.1.5 Functions Definition

A function in Kant is declared as a name, with typed parameters and typed results. Invocations must follow the same structure as the definitions.

```
function PKE_ENC(content: BitString)
    with k: PublicKey -> [pke_enc: Ciphertext]
   Alice know {
    ...
    fresh na: Nonce;
    fresh nx: Nonce;
    enc_na = PKE_ENC(na,nb) with pubb;}
```

The validator produces the following error because the declaration has only one parameter:
"PKE_ENC" requires "1" argument, but "2" arguments are provided.

A similar error would be produced if the number of keys entered does not correspond to the number of keys declared.

5.1.6 Unused Knowledge or Principal

In order to increase the clarity of the model, we warn the user of all principals and knowledge declarations that remain unused in the writing of a protocol:

```
principal Alice, Bob, Server
```

Principal Server is declared but never used.

5.1.7 Check Fields

User-entered checks in the protocol may only target knowledge and principals, but not functions.

```
only Alice,Bob should know CONCAT(na,nb)
```

Knowledge check should target only knowledge references and list access.

5.2 Semantic Checks

5.2.1 Type Compatibility

The types of the parameters used in the function invocation are inferred and checked to be congruent with those of the declarations.

```
Bob know {
 ...
 dec_na = PKE_DEC(enc_na) with pubb;
 ...}
```

Incorrect key type: "PublicKey", the invoked function requires a key of type "PrivateKey".

5.2.2 Knowledge Scoping

The knowledge declared by a principal is only accessible by that principal or the ones it has shared it with. Furthermore, the names of new knowledge must be unique across the entire protocol.

```
Bob -> Alice : nx
```

Principal "Bob" doesn't know "nx".

5.2.3 Function Inversion

Only functions that are not one way can be inverted and a property must be specified for the inversion.

```
function PKE_ENC(content: BitString)
    with k: PublicKey
    -> [ pke_enc: Ciphertext ]one way
function PKE_DEC(pke_enc: Ciphertext)
    with k: PrivateKey
    -> [ content: BitString ]
property forall x: BitString, k: PrivateKey
| PKE_DEC(PKE_ENC(x) with PUB_GEN(k))
    with k -> [ x ]
```

PKE_ENC is a one way function, it cannot be inverted.

5.2.4 List Access

It is possible to access a list by using only the names resulting from the application of a SPLIT. The validation of the SPLIT requires it to be used only on the results of a CONCAT:

```
Alice know {
 ...
 dec_na_nb = PKE_DEC(enc_nb_na) with priva;
 rec_na_nb = SPLIT(enc_nb_na);
 ...}
```

The "SPLIT" function is called on a parameter that is not the result of "CONCAT".

5.3 Prudent Engineering Practices

5.3.1 Same Key for Encryption and Authentication

The usage of the same key for symmetric encryption and signing can allow an attacker to use the signing algorithm to decrypt messages, so it is crucial to use a different key for each method. In this example, Alice uses the same key `priva` both for decrypting the message `enc_bit` and to sign the plaintext `dec_mess`. This causes a potential vulnerability when the signature is sent out.

```
Alice know {
    fresh bit:BitString;
    fresh priva:PrivateKey;
    puba = PUB_GEN(priva);}
...
Alice know {
    dec_mess = PKE_DEC(enc_bit) with priva;
    sign_mess = SIGN(dec_mess) with priva;}
```

5.3.2 Hash Used as Encryption

A message should not contain a signed hash of a plaintext as it could have been generated by a third party and sent to another principal. An example of the problem can be shown by the example below, which specifies the following message:

$$Alice \rightarrow Bob : \{X\}_{K_b}, \{HASH(X)\}_{K_a{}^{-1}}$$

where $\{X\}_{K_b}$ represents the asymmetric encryption of secret X (in plaintext) using Bob's public key K_b and $\{HASH(X)\}_{K_a{}^{-1}}$ represents the signature of the hash function applied to secret X using Alice's private key.

```
Alice know {
    enc_x = PKE_ENC(x) with pubb;
    hash_x = HASH(x);
    sign_hash_x = SIGN(hash_x) with priva;
    mess = CONCAT(enc_x,sign_hash_x);}

Alice -> Bob: mess
```

5.3.3 Encrypt then Sign

The authentication pattern *encrypt then sign* is vulnerable to attacks: the attacker can remove the signature and replace it with its own, claiming ownership of the message that the recipient receives. Thus, this pattern is generally insecure and should be avoided. To ensure greater security, it would be better to use the sign-then-encrypt method and add the identity of the recipient to the signature. The example below shows a message `mess` built by using asymmetric encryption with a public key `pubb`. The resulting cyphertext `enc_x` is signed and concatenated to the encryption.

```
Alice know {
    enc_x = PKE_ENC(x) with pubb;
    sign_enc_x = SIGN(enc_x) with priva;
    mess = CONCAT(enc_x,sign_enc_x)}
Alice -> Bob: mess
```

5.3.4 Add Recipient Identity to Signature

According to (Abadi and Needham, 1996), it is crucial never to infer the principal's identity from the content or the sender of a message, as this can lead to *impersonation attacks*. To prevent such attacks, it is essential to mention the identity of the receiver in the message's signature. Additionally, although it can be deduced from the key used to sign the message, the identity of the sender should also be included. In the example below, Alice sends a message `enc_sign` encrypted with asymmetric encryption and containing a signature `sign_kab_ta` that does not include `Bob`'s identity.

```
Alice know {
    fresh kab:SymmetricKey;
    fresh ta:Nonce;
    kab_ta=CONCAT(kab,ta);
    sign_kab_ta = SIGN(kab_ta) with priva;
    enc_sign = PKE_ENC(sign_kab_ta) with pubb;}

Alice -> Bob: enc_sign
```

5.3.5 Avoid Double Encryption

Double encryption, also known as *cascading encryption*, is a practice that has fallen out of favor in modern cryptographic security. While it may seem like a logical way to enhance data security, it is often counterproductive. Double encryption introduces complexity, consumes additional computational resources, and poses key management challenges. Rather than enhancing security, it can lead to marginal improvements while increasing the potential for implementation errors and vulnerabilities. The following example uses double encryption. The first symmetric encryption with key `kab` is applied on `bit`, and the result is encrypted using public encryption with the key `pubb`.

```
Alice know {
 fresh bit:BitString;
 enc1 = ENC(bit) with kab;
 enc2 = PKE_ENC(enc1) with pubb;}
```

6 Kant EFFECTIVENESS

In order to evaluate the ability of Kant language to capture the common concepts and primitives of security protocols, besides the NSPK protocol, we modeled classical security protocols such

as SSL, Needham-Schroeder, Needham-Schroeder-Lowe, Woo-lam, Yaholm, BAN-Yaholm, Otway-Rees, Denning-Sacco, and other protocols are under development. Kant's models of all the case studies have been validated by using the set of validation rules and, as expected, some warnings were raised on those protocol aspects that might cause vulnerabilities – not surprisingly since the rules on prudent engineering practice (see Sect. 5.3) were defined following the guidelines suggested in (Abadi and Needham, 1996) as measures to prevent known attacks against classical protocols.

Besides modeling constructs in common with other notations for security protocol description, Kant has also primitives to model the principal's knowledge and the FSM of its execution of a given protocol session. Such concepts are relevant for protocol verification by back-end tools, and the advantage of being already specified at the level of Kant model facilitates the translation of these models into models suitable for verification and allows better integration of such crucial activity into the development process of robust protocols. Typically, verification is undertaken after the protocol has already been released (Zhang et al., 2020; Cremers et al., 2016; Lilli et al., 2021), and the effort of making changes in the protocol is expensive. Our idea in developing Kant was to facilitate the integration of verification early in the development process. In this way, not only one develops consistent documentation across iterations, but also avoids the introduction of vulnerabilities when adding new features or making changes to cryptographic primitives.

To assess the potentialities of Kant in modeling knowledge information flow among parties, we selected the WPA3 Simultaneous Authentication of Equals (SAE) protocol[5], a password-based authentication and password-authenticated key agreement method. The choice was due to the fact that this protocol includes in its documentation a FSM of the protocol instances, and it is one of the first protocols to expose this feature. In addition, it provides advanced security features such as Forward Secrecy, Eavesdropping, and Dictionary attack resilience.

The overall picture of the WPA3 SAE protocol execution is given, in the official documentation, in terms of the FSM reported in Fig.6. The path in green describes the transitions performed by each principal (and are captured by our model) involved in the protocol for getting authentication; the other transitions are performed by other entities described in the WPA3 documentation, and we here abstract from them.

The crucial steps of the protocol message ex-

[5]802.11-2020 - IEEE Standard for Information Technology

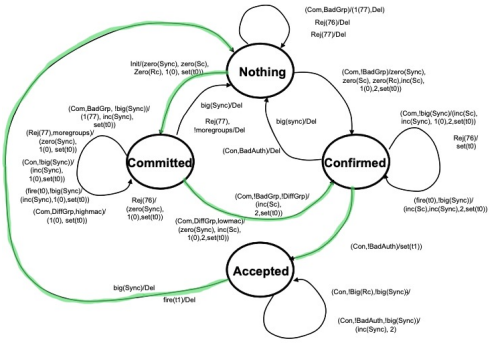

Figure 6: WPA3 SAE FSM.

changes depicted in Fig. 7 are mainly two: (1) the *commitment exchange*, to move from state *Committed* to state *Confirmed*, where each party commits to a single password guess; (2) the *confirmation exchange*, to move from state *Confirmed* to state *Accepted*, which validates the correctness of the guess. The state *Nothing* is the initial state where a principal is when it is created and immediately moves to state *Committed* when starts the protocol exchange.

The main rules of the protocol are the following:

- A party can commit at any point during the exchange.
- Confirmation can only be done after a party and its peer have committed.
- Authentication is only accepted when a peer has successfully confirmed.
- The protocol ends successfully when each participating party has acknowledged and accepted the authentication process.

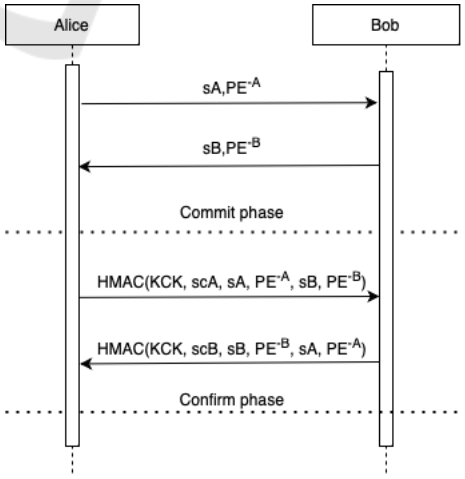

Figure 7: WPA3 SAE key exchange.

The protocol uses modular arithmetic primitives

to calculate numbers sA and sB by summing:

$$sA = (a + A) mod q$$

$$sB = (b + B) mod q$$

where a, A, b, B are random numbers, and q is the (prime) order of the group. Password Equivalent (PE) is an hashed value of the password that Bob and Alice know and is raised to the power of $-A$ for Alice and $-B$ for Bob. Once the commit messages have been exchanged, the two agents calculate the value k as:

$$k = (PE^{sB} * PE^{-B})^a = (PE^{sA} * PE^{-A})^b = PE^{ab} \quad (1)$$

The two principals compute KCK as the hash of the concatenation between k and the sum modulo q of sA and sB. Finally, they hashed all received information together with KCK and a counter called *send-confirm* (scA, scB) to build the commit messages.

To model in Kant the above computations, we defined the following functions:

```
function SCALAR_OP(r:Number,y:Group)->[z:Group]
function ELEMENT_OP(x:Group,y:Group)->[z:Group]
function INV_OP(h:Group)->[o:Group]
function SUM_MOD(r:Number,n:Number)->[t:Number]
property forall r:Number, t:Number, p:Group |
    ELEMENT_OP(SCALAR_OP(SUM_MOD(r,t),p),
    INV_OP(SCALAR_OP(t,p)))-> SCALAR_OP(r,p)
property forall x:Group, y:Group |
    ELEMENT_OP(x,y) equals ELEMENT_OP(y,x)
property forall r:Number, t:Number |
    SUM_MOD(r,t) equals SUM_MOD(t,r)
```

In both Finite Field Cryptography (FFC) and Elliptic curve cryptography (ECC) groups, WAP3 SAE employs three arithmetic operators: the element function `ELEMENT_OP` that produces a group from two groups, the scalar function `SCALAR_OP` that generates a group from an integer and a group, and the inverse function `INV_OP` that produces a group from a group. The protocol uses modular arithmetic, we added the function `SUM_MOD` has been added for this purpose (we omit the module as argument since it is of public domain and not relevant for the analysis). We then added various properties to address the commutativity of functions `ELEMENT_OP` and `SUM_MOD`, and a property that allows us to perform the mathematical simplification as described in the Formula 1.

The model of the WPA3 protocol in Kant, upon checking syntactic and semantic validation rules, is available at WPA3_SAE. This case study shows the expressive potential of the language in modeling a protocol having very high mathematical complexity. Moreover, thanks to the capability to express state end transition in the knowledge of a principal, the model is able to reflect the structure of the FSM in Fig. 6.

7 RELATED WORK

Formal verification of security protocols has been a critical area of research and development in computer security in the last twenty years. Many techniques and tools have been proposed for verifying protocols. Depending on the specific tool, either the protocol has been translated into the tool input language, or a new language has been defined to model the protocol. In most of the cases, the language was not user-friendly, requiring expertise in formal methods and protocol analysis. There are indeed some examples of domain-specific languages that inspired us when deciding which features needed to be included in the language, such as the knowledge notion, or the state. For example, in the seminal paper (Burrows et al., 1990) introducing one of the first formalisms designed to reason about protocols, the authors recognize the importance of make explicit the assumptions (called principal's beliefs and assumptions) taken before the execution of the protocol, which we expressed with the principal's knowledge.

AVISPA (Automated Validation of Internet Security Protocols and Applications) (Armando et al., 2005) supports the editing of protocol specifications and allows the user to select and configure the different back-ends of the tool, similarly to our long-term goal. The protocol is given in the High-Level Protocol Specification Language HLPSL (Chevalier et al., 2004), an expressive, modular, role-based, formal language, thus not really user-friendly; HLPSL specifications are then translated to the so-called Intermediate Format (IF), an even more mathematical-based language at an accordingly lower abstraction level and is thus more suitable for automated deduction.

Also Sapic+ (Cheval et al., 2022) aims at exploiting the strengths of some of the tools that reached a high degree of maturity in the last decades, e.g., TAMARIN and PROVERIF, offering a protocol verification platform that permits choosing the tool. However, the input language is an applied π calculus similar to PROVERIF, thus it requires high expertise in equational theories and rewrite systems.

In (Jacquemard et al., 2000; Mödersheim, 2009), the authors define and give the semantics of AnB, a formal protocol description language based on the classical Alice&Bob notation we introduced in Sect. 2. However, in the translation, part of the readability is lost since a mathematical notation is used.

The work most related to ours is Verifpal (Chevalier et al., 2004), which uses a user-friendly high-level language that allows users to model cryptographic protocols and security properties in a rather intuitive way. The tool uses symbolic analysis techniques, and

can automatically generate formal verification code in the TAMARIN prover's input language, making it compatible with TAMARIN for further in-depth analysis and verification. However, they provide only a text editor with a syntax highlighter, whereas in our case we do both syntax and semantic checking.

8 CONCLUSIONS

We presented Kant (Knowledge ANalysis of Trace), a DSL we explicitly designed and developed in Langium for the specification of security protocols. Kant has been conceived as a front-end and easy-to-use language of a formal framework under development to support different back-end tools for security protocol analysis. The Kant grammar encompasses constructs of the notations commonly used to express security protocols, but it also has primitives to model information that is fundamental for formal analysis (done by back-end tools), i.e., (1) the knowledge flow exchanged between the parties during a protocol session, and (2) the FSM model that is behind the execution of each participant. A further innovative feature of Kant w.r.t. other notations for security protocol modeling is its embedded mechanism of model validation against a set of validation rules, which helps the designer avoid common security errors or make design choices leading to protocol vulnerabilities.

In future work, our first goal is to automate the transformation of a Kant model in input models of the back-end tools (ASMETA, TAMARIN, and PROVERIF are the first choices, see Fig. 1) by exploiting the advantages of the Model-driven Language Development. We also plan to develop a graphical front-end for APROVER for a visual rendering of Kant models.

REFERENCES

Abadi, M. and Needham, R. (1996). Prudent engineering practice for cryptographic protocols. *IEEE Transactions on Software Engineering*, 22(1):6–15.

Anderson, R. and Needham, R. (1995). *Programming Satan's computer*, pages 426–440. Springer Berlin Heidelberg, Berlin, Heidelberg.

Arcaini, P., Gargantini, A., Riccobene, E., and Scandurra, P. (2011). A model-driven process for engineering a toolset for a formal method. *Software: Practice and Experience*, 41(2):155–166.

Armando, A., Basin, D., Boichut, Y., Chevalier, Y., Compagna, L., Cuellar, J., Drielsma, P. H., Heám, P. C., Kouchnarenko, O., Mantovani, J., Mödersheim, S., von Oheimb, D., Rusinowitch, M., Santiago, J., Turuani, M., Viganò, L., and Vigneron, L. (2005). The AVISPA Tool for the Automated Validation of Internet Security Protocols and Applications. In *Computer Aided Verification (CAV'05)*, pages 281–285.

Basin, D., Dreier, J., Hirschi, L., Radomirovic, S., Sasse, R., and Stettler, V. (2018). A Formal Analysis of 5G Authentication. In *Proc. of the ACM SIGSAC Conf. on Computer and Communications Security (CCS'18)*, pages 1383–1396.

Blanchet, B. (2001). An efficient cryptographic protocol verifier based on prolog rules. In *Proc. of IEEE Computer Security Foundations Workshop*, pages 82–96.

Blanchet, B. (2012). Security Protocol Verification: Symbolic and Computational Models. In *Principles of Security and Trust*, pages 3–29.

Burrows, M., Abadi, M., and Needham, R. (1990). A Logic of Authentication. *ACM Trans. Comput. Syst.*, 8(1):18–36.

Cheval, V., Jacomme, C., Kremer, S., and Künnemann, R. (2022). Sapic+: protocol verifiers of the world, unite! Cryptology ePrint Archive, Paper 2022/741. https://eprint.iacr.org/2022/741.

Chevalier, Y., Compagna, L., Cuellar, J., Drielsma, P. H., Mantovani, J., Mödersheim, S., and Vigneron, L. (2004). A High Level Protocol Specification Language for Industrial Security-Sensitive Protocols. In *Workshop on Specification and Automated Processing of Security Requirements (SAPS'04)*.

Cremers, C., Horvat, M., Scott, S., and van der Merwe, T. (2016). Automated analysis and verification of tls 1.3: 0-rtt, resumption and delayed authentication. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 470–485.

Davis, J., Clark, M., Cofer, D., and et al. (2013). Study on the barriers to the industrial adoption of formal methods. *LNCS*, 8187:63–77.

Haskins, B., Stecklein, J., Dick, B., Moroney, G., Lovell, R., and Dabney, J. (2004). 8.4.2 Error Cost Escalation Through the Project Life Cycle. *INCOSE International Symposium*, 14:1723–1737.

Heinrich, R., Bousse, E., Koch, S., Rensink, A., Riccobene, E., Ratiu, D., and Sirjani, M. (2021). Integration and orchestration of analysis tools. In *Composing Model-Based Analysis Tools*, pages 71–95.

Jacquemard, F., Rusinowitch, M., and Vigneron, L. (2000). Compiling and Verifying Security Protocols. In *Logic for Programming and Automated Reasoning*, pages 131–160.

Lilli, M., Braghin, C., and Riccobene, E. (2021). Formal Proof of a Vulnerability in Z-Wave IoT Protocol. In *Int. Conf. on Security and Cryptography*.

Meier, S., Schmidt, B., Cremers, C., and Basin, D. (2013). The TAMARIN Prover for the Symbolic Analysis of Security Protocols. In *Computer Aided Verification (CAV'13)*, pages 696–701.

Mödersheim, S. (2009). Algebraic Properties in Alice and Bob Notation. In *Int. Conf. on Availability, Reliability and Security*, pages 433–440.

Tobarra, L., Cazorla, D., Pardo, J. J., and Cuartero, F. (2008). Formal Verification of the Secure Sockets Layer Protocol. In - *Proc. of the Int. Conf. on Enterprise Information Systems (ICEIS'08)*, pages 246–252.

Zhang, J., Yang, L., Cao, W., and Wang, Q. (2020). *IEEE Access*, 8:23674–23688.