Industrial Experience Report on the Formal Specification of a Packet Filtering Language Using the K Framework

Gurvan Le Guernic — DGA Maîtrise de l'Information, 35998 Rennes Cedex 9, France

Benoit Combemale, José A. Galindo — INRIA RENNES – BRETAGNE ATLANTIQUE, Campus universitaire de Beaulieu, 35042 Rennes Cedex, France

Many project-specific languages, including in particular filtering languages, are defined using non-formal specifications written in natural languages. This leads to ambiguities and errors in the specification of those languages. This paper reports on an industrial experiment on using a tool-supported language specification framework ($\mathbb{K}$) for the formal specification of the syntax and semantics of a filtering language having a complexity similar to those of real-life projects. This experimentation aims at estimating, in a specific industrial setting, the difficulty and benefits of formally specifying a packet filtering language using a tool-supported formal approach.

1 Introduction

Packet filtering (accepting, rejecting, modifying or generating packets, i.e. strings of bits, belonging to a sequence) is a recurring problematic in the domain of information systems security. Such filters can serve, among other uses, to reduce the attack surface by limiting the capacities of a communication link to the legitimate needs of the system it belongs to. This type of filtering can be applied to network links (which is the most common use), product interfaces, or even on the communication buses of a product. If the filtering policy needs to be adapted during the deployment or operational phases of the system or product, it is often required to design a specific language $\mathscr{L}$ (syntax and semantics) to express new filtering policies during the lifetime of the system or product. This language is the basis of the filters that are applied to the system or product. Hence, it plays an important role in the security of this system or product. It is therefore important to have strong guarantees regarding the expressivity, precision, and correctness of the language $\mathscr{L}$ (meaning that everything that need to be expressed can, and that everything that can be expressed has the most obvious semantics). Those guarantees can be partly provided by a formal design (and development) process.

Among diverse duties, the DGA (Direction Générale de l'Armement, a french procurement agency) is involved in the supervision of the design and development of filtering components or products. Those filters come in varying shapes and roles. Some of them are network apparatuses filtering standard Internet protocol packets (such as firewalls); while others are small parts of integrated circuits filtering specific proprietary packets transiting on computer buses. Their common definition is: "a tool sitting on a communication channel, analyzing the sequence of packets (strings of bits with a beginning and an end) transiting on that channel, and potentially dropping, modifying or adding packets in that sequence". Whenever the filtering algorithm applied is fixed for the lifetime of the component or product, this algorithm is often "hard coded" into the component or product with the potential addition of a configuration file allowing to slightly alter the behavior of the filter. However, sometimes the filtering algorithm to apply may depend on the deployment context, and may have to evolve during the lifetime of the component or product to adapt to new uses or attackers. In this case, it is often necessary to be able to easily

C. Dubois, P. Masci, D. Méry (Eds.): F-IDE 2016
EPTCS 240, 2017, pp. 38–52, doi:10.4204/EPTCS.240.3

© G. Le Guernic, B. Combemale & J.A. Galindo
This work is licensed under the
Creative Commons Attribution License.

write new filtering algorithms for the specific product and context. Those algorithms are then often described using a Domain Specific Language (DSL) that is designed for the expression of a specific type of filters for a specific product. The definition of the syntax and semantics of this DSL is an important task. This DSL is the link between the filtering objectives and the process that is really applied on the packet sequences. Often, language specifications (when there is one) are provided using natural language. In the majority of cases, this leads to ambiguities or errors in the specification which propagate to implementations and final user code. This is for example the case for common languages such as C/C++ or Java™ [12].

> "Unfortunately, the current specification has been found to be hard to understand and has subtle, often unintended, implications. Certain synchronization idioms sometimes recommended in books and articles are invalid according to the existing specification. Subtle, unintended implications of the existing specification prohibit common compiler optimizations done by many existing Java virtual machine implementations. [...] Several important issues, [...] simply aren't discussed in the existing specification."
>
> JSR-133 expert group [12]

Some of those ambiguities, as the memory model of multi-threaded Java™ programs [12], required a formal specification in order to be solved.

This paper is an industrial experience report on the use of a tool-supported language specification framework (the $\mathbb{K}$ framework) for the formal specification of the syntax and semantics of a filtering language having a complexity similar to those of real-life projects. The tool used to formally specify the DSL is introduced in Sect. 2. For confidentiality reasons, in order to be allowed by the DGA to communicate on this experimentation, the language specified for this experiment is not linked to any particular product or component. It is a generic packet filtering language that tries to cover the majority of features required by packet filtering languages. This language is introduced in Sect. 3 while its formal specification is described in Sect. 4. This language is tested in Sect. 5 by implementing and simulating a filtering policy enforcing a sequential interaction for a made-up protocol similar to DHCP. Before concluding in Sect. 7, this paper discusses the results of the experimentation in Sect. 6.

2 Introduction to the $\mathbb{K}$ Framework

Surprisingly, even if it is a niche for tools, there exists quite a number of tools specifically dedicated to the formal *specification* of languages (our focus in this work is on specifying rather than implementing DSLs). Those tools include among others: PLT Redex [6, 13], Ott [23], Lem [19], Maude MSOS Tool [3], and the $\mathbb{K}$ framework [20, 26]. All those tools focus on the (clear formal) specification of languages rather than their (efficient) implementation, which is more the focus of tools and languages such as Rascal [16, 2, 15] or its ancestor The Meta-Environment [14, 25], Kermeta [9, 10], and others. PLT Redex is based on reduction relations. PLT Redex is an extension (internal DSL) of the Racket programming language [7]. Ott and Lem are more oriented towards theorem provers. Ott and Lem allow to generate formal definitions of the language specified for Coq, HOL, and Isabelle. In addition, Lem can generate executable OCaml code. Ott is more programming language syntax oriented, while Lem is a more general purpose semantics specification tool. Ott and Lem can be used together in some contexts. The Maude MSOS Tool, whose development has stopped in 2011, is based on an encoding of modular structural operational semantics (MSOS) rules into Maude. Similarly to the Maude MSOS Tool, the $\mathbb{K}$ framework is based on rewriting and was also originally implemented on top of Maude.

The goal set for the experiment reported in this paper is to estimate the difficulty and benefits for an average engineer (i.e. an engineer with education and experience in computer science but no specific knowledge in formal language semantics) to use an "appropriate" tool for the formal specification of a packet filtering language. The "appropriate" tool needs to: be easy to use; be able to produce (or take as input) "human readable" language specifications; provide some level of correctness guarantees for the language specified; and be executable (simulatable) in order to test (evaluate) the language specified. The $\mathbb{K}$ framework seems to meet those requirements and has been chosen to be the "appropriate" tool after a short review of available tools. As there has been no in depth comparison of the different tools available, there is no claim in this paper that the $\mathbb{K}$ framework is better than the other tools, even in our specific setting.

This section introduces the $\mathbb{K}$ framework [21] by relying on the example of a language allowing to compute additions over numbers using Peano's encoding [8]. The $\mathbb{K}$ source code of this language specification is provided below.

```
module PEANO-SYNTAX
  syntax Nb ::= "Zero" | "Succ" Nb
  syntax Exp ::= Nb | Id | Exp "+" Exp        [strict,left]
  syntax Stmt ::= Id ":=" Exp ";"             [strict(2)]
  syntax Prg ::= Stmt | Stmt Prg
endmodule

module PEANO imports PEANO-SYNTAX
  syntax KResult ::= Nb

  configuration
    <env color="green"> .Map </env>
    <k color="cyan"> $PGM:K </k>

  rule N:Nb + Zero => N
  rule N1:Nb + Succ N2:Nb => ( Succ N1 ) + N2

  rule
    <env> ... Var:Id |-> Val:Nb ... </env>
    <k> ( Var:Id => Val:Nb ) ... </k>

  rule
    <env> Rho:Map (.Map => Var |-> Val ) </env>
    <k> Var:Id := Val:Nb ; => . ... </k>
    when notBool (Var in keys(Rho))

  rule
    <env> ... Var |-> ( _ => Val ) ... </env>
    <k> Var:Id := Val:Nb ; => . ... </k>

  rule S:Stmt P:Prg => S ~> P [structural]
endmodule
```

A 𝕂 definition is divided into three parts: the *syntax* definition, the *configuration* definition, and the *semantics* (rewriting rules) definition. The definition of the language *syntax* is given in a module whose name is suffixed with "-SYNTAX". It uses a BNF-like notation [1, 17]. Every non-terminal is introduced by a `syntax` rule. For example, the definition of the notation for numbers (`Nb`) in this language, provided on line 2, is equivalent to the definition given by the regular expression "`(Succ)`* `Zero`".

The *configuration* definition part is introduced by the keyword `configuration` and defines a set of (potentially nested) cells described in an XML-like syntax. This configuration describes the "abstract machine" used for defining the semantics of the language. The initial state (or configuration) of the abstract machine is the one described in this configuration part. The parsed program (using the syntax definition of the previous part) is put in the cell containing the `$PGM` variable (of type `K`). For the `Peano` language, the `env` cell is used to store variable values in a map initially empty (`.Map` is the empty map). From this definition, the 𝕂 framework can produce a graphical representation of the configuration, provided in Fig. 1

Figure 1: Peano's 𝕂 configuration

The *semantics* definition part is composed of a set of rewriting rules, each one of them introduced by the keyword `rule`. In the 𝕂 source file, rules are roughly denoted as "*CCF* => *NCF*" where *CCF* and *NCF* are configuration fragments. The meaning of "*CCF* => *NCF*" can be summarized as: if *CCF* is a fragment of the current abstract machine state (or configuration) then the rule may apply and the fragment matching *CCF* in the current configuration would then be replaced by the new configuration fragment *NCF*. In order to increase the expressivity of rules, *CCF* may contain free variables that are reused in expressions in *NCF*. If a specific valuation of the free variables *V* in *CCF* allows a fragment of the current configuration to match *CCF*, then this fragment may be replaced by *NCF* where the variables *V* are replaced by their matching valuation.

The rules for addition over numbers (`Nb` and not `Exp`), on lines 15 and 16, follows closely this representation. For those rules, *CCF* is a program fragment that can be matched in any cell of the configuration. For those two rules, the 𝕂 framework can then produce the following graphical representations:

RULE
$N{:}Nb$ + `Zero`
N

RULE
$N1{:}Nb$ + `Succ` $N2{:}Nb$
(`Succ` $N1$) + $N2$

For other rules, the configuration fragment matching is more complex and involves precise configuration cells that are explicitly identified. In order to compress the representation, *CCF* and *NCF* are not stated separately anymore. The common parts are stated only once, and the parts differing are again denoted "CCF_i => NCF_i", where CCF_i is a sub-fragment in *CCF* and NCF_i is the corresponding sub-fragment in *NCF*. Cells that have no impact on a rule *R* and are not impacted by *R* do not appear explicitly in the rule. Cells heads and tails (potentially empty) that are not modified by a rule can be denoted "...", instead of using a free variable that would not be reused.

For example, the rule which starts on line 18is the rule used to evaluate variables. The current configuration needs to contain a mapping from a variable `Var` to a value `Val` ("`X |-> V`" denotes a mapping from `X` to `V`) somewhere in the map contained in the `env` cell. It also needs to contain the variable `Var` at the beginning of cell `k`. This rule has the effect of replacing the instance of `Var` at the beginning of cell `k` by the value `Val`. For this rule, the 𝕂 framework generates the graphical representation given in Fig. 2.

The last rule on line 31 involves other internal aspects of the 𝕂 framework. It roughly states that, in order to evaluate a statement `S` followed by the rest `P` of the program, `S` must first be evaluated to a

KResult (defined on line 9) and then P is evaluated.

3 GPFL Context

The language specified in the experiment reported in this paper, named GPFL, is a generic packet filtering language. For confidentiality reasons, GPFL is not a language actually used in any specific real product. GPFL has been made-up in order to be able to communicate on the experimentation on tool supported formal specification of filtering languages reported in this paper. However, GPFL covers the majority of features needed in packet filtering languages dealt with by the DGA. GPFL can be seen as the "mother" of the majority of packet filtering languages.

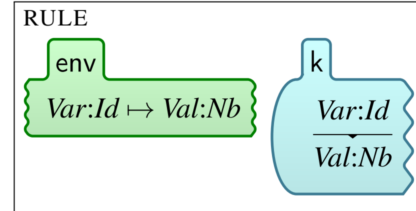
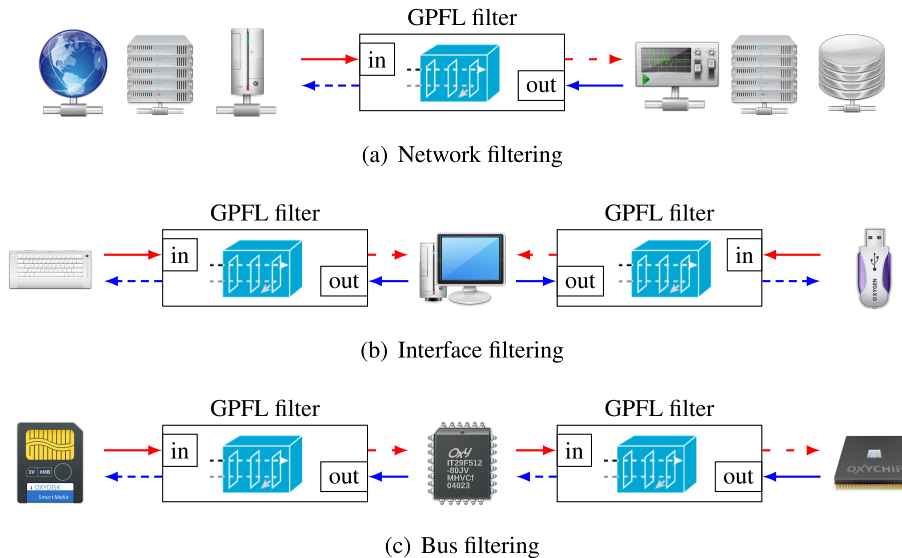


Figure 2: Peano's 𝕂 rule for variables

GPFL aims at expressing a wide variety of filters. Those filters can be placed at the level of network, interfaces, or even communication buses between electronic components. They can be applied on standard protocols such as IP, TCP, UDP, ... or on proprietary protocols, which are more common for component communication protocols. However, all those filters are assumed to be placed on a communication link. Messages (packets) that get through the filter can only get through in two ways, either "going in" or "going out"; there is no switching taking place in GPFL filters. Those different use cases are illustrated in Fig. 3.



(a) Network filtering

(b) Interface filtering

(c) Bus filtering

Figure 3: Use cases for GPFL-based filters

GPFL focuses on the internal logic of the filter. Decoding and encoding of packets is assumed to be handled outside of GPFL programs (filters), potentially using technologies such as ASN.1 [11, 5]. For GPFL programs, a packet is a record (a set of valued fields). A GPFL program (dynamically) inputs a sequence of records and outputs a sequence of records. Figure 4 describes the architecture of GPFL-based filters. An incoming packet (on either side) is first parsed (decoded) before being handed over to

the GPFL program. If the packet can not be parsed, depending on the type of filter (white list or black list), the packet is either dropped or passed to the other side without going through the GPFL program. Any packet (record) output by the GPFL program (on either side) is encoded before being sent out. In addition, the GPFL program can generate alarms due to packets not complying with the encoded filtering policy.

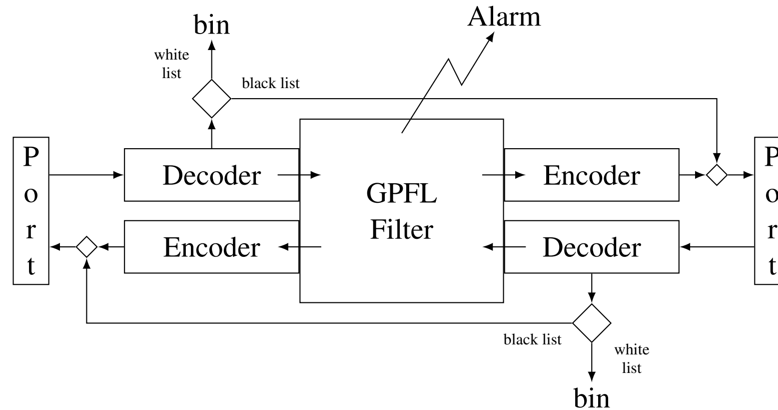


Figure 4: Architecture of GPFL-based filters

The GPFL language must allow to: drop, modify or accept the current packet being filtered; generate new packets; and generate alarms. GPFL must allow to base the decision to take any of those actions on information pieces concerning the current packet being filtered and previously filtered packets. Those information pieces must include: some timing information, current or previous packets directions through the filter ("in" or "out"), and characteristics of current or previous packets including field values and computed properties such as, for example, a packet "type" or total length. The computation of those properties and decoding of packet fields is outside of the scope of GPFL; it is left to the decoders.

In order to gradually build a decision, GPFL must allow to interact with variables (reading, writing, and computing expressions) and automata (triggering a transition in an automaton and querying its current state). The intent for automata is to be used to track the current step of sessions of complex protocols. GPFL must allow to combine filtering statements using: sequential control statements (executing two statements in sequence); conditional control statements (executing a statement only if a condition is true); iterating control statements (repeatedly executing a statement for a fixed number of repetitions). There is no requirement for a loop (or while) statement whose exit condition is controlled by an expression recomputed after every iteration. For the experiment reported in this paper (on formal specification of a filtering language), the iterating statement is considered sufficient for the intended use of GPFL and close enough to a loop statement from a semantics point of view, while exhibiting interesting properties for future analyses (for example, any GPFL program terminates).

4 GPFL's Specification

Due to lack of space, GPFL's specification and testing is only summarized in this paper. However, a full specification of GPFL and a testing section can be found in the companion technical report [18].

Syntax. To the exception of expressions and expression fragments, GPFL's syntax is formally defined by the $\mathbb{K}$ source fragment provided below.

```
18  syntax Cmd ::= "nop" | "accept" | "drop" | "send(" Port "," Fields ")"
               | "alarm(" Exp ")"                                    [strict(1)]
20             | "set(" Id "," Exp ")"                               [strict(2)]
               | "newAutomaton(" String "," AutomatonId ")"
22             | "step(" AutomatonId "," Exp "," Stmt ")"            [strict(2)]
    syntax Stmt ::= Cmd
24              | "cond(" Exp "," Stmt ")"                           [strict(1)]
                | "iter(" Exp "," Stmt ")"                           [strict(1)]
26              | "newInterrupt(" Int "," Bool "," Stmt ")"
                | Stmt Stmt                                          [right]
28              | "{" Stmt "}"                                       [bracket]

30  syntax AutomataDef ::= "AUTOMATA" String AutomataDefTail
    syntax AutomataDefTail ::= "init" "=" AStateId ATransitions | ATransitions
32  syntax ATransitions ::= List{ATransition,""}
    syntax ATransition ::= AStateId "-" AEvtId "->" AStateId
34  syntax AStateId ::= String
    syntax AEvtId ::= String
36  syntax InitSeq ::= "INIT" Stmt
    syntax PrologElt ::= AutomataDef | InitSeq
38  syntax Prologues ::= PrologElt | PrologElt Prologues

40  syntax Program ::= "PROLOGUE" Prologues "FILTER" Stmt
```

A GPFL program is composed of a prologue, executed only once in order to initialize the execution environment, and a filter statement, executed once for every incoming packet. A prologue is composed of automaton kind definitions and initialization sequences. An automaton kind definition specifies an identifier K, an initial state for automata of kind K and a set of transitions for automata of kind K. A transition definition is composed of: two automaton states F and T, and an automaton event that triggers the transition from F to T.

A GPFL statement is composed of GPFL commands or statements combined sequentially. Some statements can be guarded by an expression and executed only if that expression evaluates to true (`cond`). Some statements (`iter`), associated with an expression e, are exectued v times, where v is the value of e before the first iteration. Finally, `newInterrupt` statements register a statement to be executed in the future, potentially periodically.

GPFL commands are the basic units having an effect on the execution environment. The `nop` command has no effect and serves mainly as a place holder. The `accept`, resp. `drop`, command states to accept, resp. drop, the current packet and stop the filtering process for this packet. The `send` command sends a packet on one of the ports. The `alarm` command generates a message on the alarm channel. The `set` command sets the value of a variable. The `newAutomaton` command initializes an automaton of the provided kind, and assigns this newly created automaton to the provided identifier. The `step` command tries to trigger an automaton transition by sending an event e to an automaton a. If there is no transition from the current state of a triggered by the event e, then the associated statement is executed.

Semantics The full formal specification of GPFL's semantics can be found in the companion technical report [18]. GPFL's semantics rules are defined on the configuration presented graphically in Fig. 5. The `prg` cell contains the GPFL program. After initialization of the program, automaton kind definitions are stored in the `automatonKindDefs` cell and the `filter` cell contains the filter (GPFL statement)

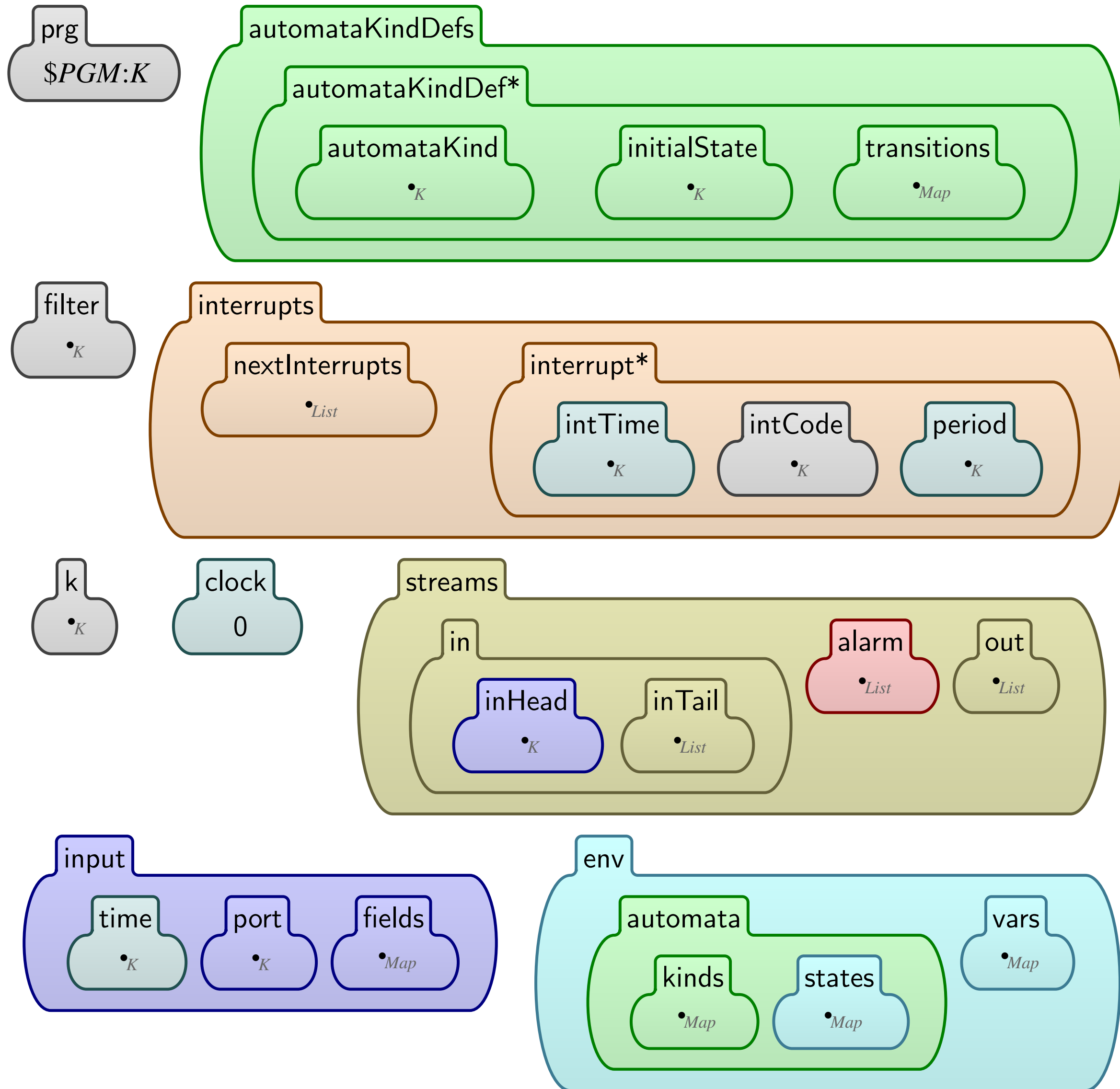


Figure 5: $\mathbb{K}$ configuration of GPFL

that is to be executed for every packet. The `interrupts` cell contains a set of interrupt definitions (`interrupt*`). An interrupt is a triplet composed of: the time when the interrupt is to be triggered, the code (statement) to be executed, and a "Time" value equal to the interruption period for a periodic interruption (or nothing for a non-periodic interruption). In addition, the `interrupts` cell contains an ordered list of the next "times" when an interrupt is to be executed. The `clock` cell registers the current "time". The configuration also contains a `k` cell that holds the GPFL statement under execution. Each time a new packet is input, the content of the `k` cell is replaced by the content of the `filter` cell, and the newly arrived packet is stored in the `input` cell with its arrival time and port.

Packets are input from the `streams` cell which contains: the packet input stream divided into the next packet to arrive (`inHead`) and the rest of the stream (`inTail`); the packet output stream; and the alarm output stream. In the input stream, resp. output stream, packets arriving, resp. leaving, on both

ports are mixed together, but contains information on the port of entry, resp. exit. Some choices made to represent those streams are not an intrinsic part of GPFL's formal specification. The division of the input stream into a head and a tail is such a choice. Those choices are made in order to be able to execute the specification. It is then required to implement, in the $\mathbb{K}$ framework, a mechanism to retrieve and parse strings describing packet sequences sent to the filter. In order to help distinguish between the formal specification of GPFL and the mechanisms put in place to execute it, whenever possible, implementation choices, such as the format of strings describing packets, are defined in another file which is loaded in the main specification file with the `require` instruction.

Finally, the `env` cell is the main dynamic part of the execution environment. It corresponds to a "record" of maps that associate: automaton kind and current state to automaton identifiers (`automata` cell); and values to variables.

5 Testing GPFL's Specification

GPFL's specification, introduced above and contained in the companion technical report [18], is not necessarily perfect. By a matter of fact, imperfections of GPFL's specification are of interest to the experimentation reported in this paper. Indeed, the goal of the experimentation is to see how a tool such as the $\mathbb{K}$ framework can help to spot and correct imperfections in filtering language specifications. One way to do so is by "testing" the new language specified, which is possible if the framework used to specify the language supports the execution or simulation of language specifications, which is the case for the $\mathbb{K}$ framework.

The test scenario used assumes a network of clients and servers. The clients request resources to servers using a made-up protocol, called "DHCP cherry", summarized in Fig. 6. The test scenario as-

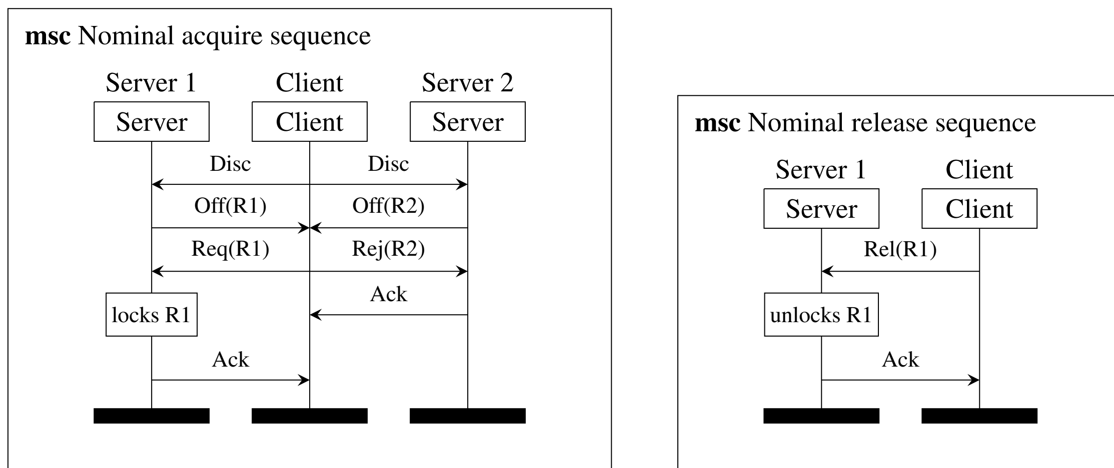


Figure 6: Nominal packet sequences of DHCP cherry protocol

sumes that servers behave poorly when interacting concurrently with different clients. The objective of the test scenario is then to filter communications in front of servers in order to prevent any concurrent client-server interactions with any given server. This test scenario is obviously made-up for this experimentation, which is a requirement due to confidentiality issues. However, it is still covering the most frequently used features of filtering languages similar to GPFL, while remaining simple enough for a first experimentation.

From the point of view of servers, non-concurrent interactions are sequential instances of only three generic atomic packet sequences. Those atomic packet sequences are the ones accepted by the automaton in Fig. 7. In this automaton, "in:*MP*", resp. "out:*MP*", is a transition trigger matching any incoming

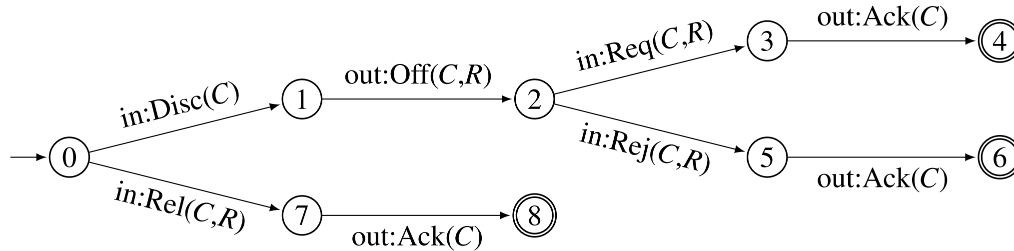


Figure 7: Automaton of server-side atomic packet sequences

packet (from the rest of the network to the server), resp. outgoing packet, matching packet pattern *MP*. *C*, resp. *R*, is a client, resp. ressource, identifier variable. *C*, resp. *R*, has to be instantiated in the same way (have the same value) for any packet of the same atomic packet sequence accepted by the automaton. The automaton of Fig. 7 is refined into a filtering policy automaton described in Fig. 8. Variables *C* and *R* have the same constraints as for the automaton of Fig. 7. The variable "$*$" matches any value, packet pattern "out:$*$" matches any outgoing packet, and packet pattern "out:$*$ - Ack(*C*)" matches any outgoing packet except Ack(*C*). This filtering policy accepts every outgoing packet; thus having no

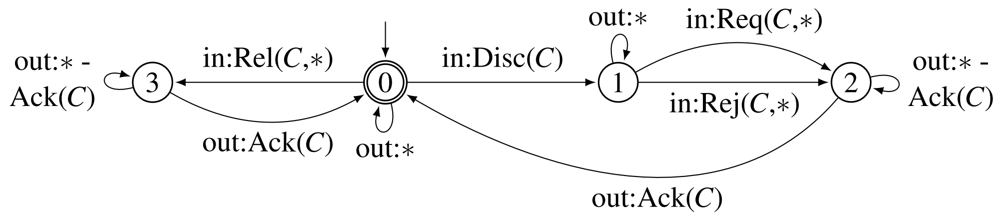


Figure 8: Filtering policy automaton

effect on the packets generated by the server. For incoming packets, if the current state of the automaton has no transition whose trigger matches the packet then the packet is discarded; otherwise, the packet is accepted and the associated transition is triggered. This filtering policy assumes that clients comply with the DHCP cherry protocol and ensures only that the filtered server only interacts sequentially with clients. If there is no idle server ready to receive a packet from a client, this client gets no answer and is expected to retry later.

This policy has been encoded in GPFL and executed using the following command (in Linux Bash): "`krun dhcp.gpfpl < dhcp_input-dataset.txt > dhcp_output.txt`" where the file `dhcp_input-dataset.txt` contains a sequence of packets already "parsed" (decoded packets, Fig. 4) input to the filter. The output of the simulation of the code (`dhcp.gpfpl`) written in the specified language (GPFL) is written in `dhcp_output.txt`.

6 Discussion on the Experimentation

The primary goal of this paper is not to set out the filtering policy described in Sect. 5 or, even, GPFL's specification described in Sect. 4. This paper is an industrial experience report on a primary evaluation of the cost and benefits of using formal specification tools in general, and the $\mathbb{K}$ framework in particular, to formally specify the syntax and semantics of filtering languages. Overall, it seems to the authors that using the $\mathbb{K}$ framework helped greatly to improve GPFL's specification quality. It forced the specification authors to be precise, and helped spot various errors and missing specification fragments.

With regard to the "cost", this experimentation argues in favor of tool supported formal specifications for high quality specifications of filtering languages. Of course, using natural language, it is possible to produce a cheaper, but ambiguous and approximate, specification. However, from the authors' natural language based experiences with packet filtering language specifications, using natural language to produce a specification with a similar level of precision and correctness would be more costly for engineers with operational semantics knowledge. With a decent knowledge of operational semantics concepts, the cost for newcomers to the $\mathbb{K}$ framework is relatively low, thanks to the numerous tutorials (in text and video), manuals and examples. In fact, having been exposed to operational semantics concepts (apart from general computer science concepts) seems to be the only prerequisite to efficiently using the $\mathbb{K}$ framework.

From the authors' previous experiences at formal specification of packet filtering language specifications without tool support, the cost of the constraints imposed by the $\mathbb{K}$ framework seems to the authors to be lower than the benefits provided by the tool support. Typically, the ability to simulate[1] the formal specification of the filtering language requires a particular handling of input/output related rules. However, this same ability to simulate the formal specification of the filtering language is highly beneficial when validating the correctness of the specification and expressivity of the language by "executing" test and documentation programs.

Other benefits of tool supported formal specifications of languages are numerous. In natural language documents specifying new languages, it is too common for program examples to be inconsistent with the language grammar. It is easily explained by the modifications brought to the language grammar during the specification document development. Examples directly related to the modified statements are usually modified accordingly. However, examples related to other aspects of the language are often forgotten. Using a tool supported formal specification, it is easy to adopt a "continuous/frequent integration" approach where examples are: written in separate files, regularly parsed to verify that they comply with the current grammar, and automatically imported in the specification document (the creation of this paper used this approach).

Additionally, use of a tool-supported formal specification approach modifies the workflow often applied when using natural language specification documents. With natural language specifications, the specification document writing process usually starts early after a short engineering phase (it may not be true for a language *development* process, however it is often the case in pure language *specification* processes), and the main part of the language specification is done during the specification document writing process. With a tool-supported formal specification approach, the specification of the language tend to be first developed inside the tool, and then the language specification is clarified during the specification document writing process. With a tool-supported formal specification approach, the language specification becomes a two phases process with two different views on the language specification. The

[1]The authors prefer to talk of "simulation" rather than "execution", as the loading time of the execution environment and limited ability to interact with other components would most likely prevent to use such an execution in a real world setting.

"two different views" aspect is particularly true with the $\mathbb{K}$ framework were semantics rules are entered textually in the source file and can be rendered graphically for the specification document. This two phases workflow (development then clarification and documentation) helps spot: differences of treatments (in particular for configuration cells), generalization and reuse opportunities (for example, in this experimentation, the use of only two internal commands, `iSend` and `iHalt`, to encode the three packet commands `accept`, `drop` and `send`), different concepts that are candidates to modularization (for example, in this experimentation, the externalization of packet data type definitions and string conversions), errors that manifest themselves in rare occasions (for example, in an earlier version of GPFL, automaton states and variable values where stored in the same map, which could trigger a key clash caused by variable and automaton identifiers having the same "name" part), or general simplifications (for example, during this report writing process, GPFL's configuration has been heavily reformatted to simplify the language specification and be closer to the concepts manipulated). From the authors experience, in general and compared to a natural language approach, a tool-supported formal specification process helps simplify and clarify a language specification.

Moreover, the ability to execute the formal specification allows to adopt an incremental approach for the specification of the different statements semantics. In such an approach, the syntax of the language is first specified. Then a program example making use of all the statements of the language in as much context as reasonable is written. The semantics of the statements is then defined statements by statements. The program is executed using $\mathbb{K}$'s run time; and the execution stops when reaching a statement whose semantics is not defined yet. All the semantics rules associated to this statement are then defined. When stopping an execution, $\mathbb{K}$'s run time displays the current state of the configuration which can help specify the missing semantics rules. As the test program execution goes further and further during the language semantics specification process, this incremental approach is more rewarding for people in charge of the specification. The impact of using this incremental approach (which is not required by the $\mathbb{K}$ framework) on the quality of the specifications produced remains to be investigated.

Finally, the ability to execute the formal specification allows to test and validate the language specification. Two important points to validate are: the expressivity of the language and its expected semantics. GPFL's test code (Sect. 5) provided in the companion technical report [18] emphasizes the limitations of the simple automata that can be defined using GPFL. It could be useful to have automaton state variables, and triggering conditions that test and check automaton state variable values. However, adding automaton state variables would complexify automata definitions. Similarly, GPFL's test code contains a recurring code sequence to handle alarms which is triggered only when a threshold of a specific event occurrences is reached. It could be useful to add a specific command to GPFL which would have the same semantics as this recurring sequence. The ability to test programs does not solve expressivity questions (which have to be answered on a per language basis), however it helps explicit those questions. With regard to expected semantics, writing test programs helps validate that programs have the semantics that users would expect. The initial version of GPFL's test code did not behave as expected. It ended up being a misplaced statement in the filter code, but could also have been a problem with the semantics specification. Discovering the cause of a misbehavior of a test program (error in the semantics or the program) could be greatly simplified by $\mathbb{K}$'s debugger which can "execute" formal specifications step by step; especially as Domain Specific Languages (specifications and implementations) usually have limited debugging facilities (which is in accordance with their philosophy of limited expressivity for the sake of simplification). However, sadly, $\mathbb{K}$'s debugger crashed on our program with the version of the $\mathbb{K}$ framework used for this experimentation (version 3.6). This can be explained by the fact that $\mathbb{K}$ development effort was focused on the next version to come (version 4.0 which exited the beta stage at the end of July 2016). Finally, the ability to execute the formal specification helps to validate a set of test programs that

can be used as smoke test for language implementations.

7 Conclusion

This paper reports on an industrial experiment to formally specify the syntax and semantics of a filtering language (GPFL) using the tool-supported framework $\mathbb{K}$. For confidentiality reasons, the filtering language specified in this report has been made up for this experimentation; however, it covers the majority of concepts usually encountered in filtering languages. No comparison between different tools is made in this experiment. The goal of the experiment is to study the feasibility of using a tool-supported formal approach for the specification of domain-specific filtering languages having a complexity similar to filtering languages encountered in real-life projects.

The $\mathbb{K}$ framework proved to be sufficiently expressive to naturally express the syntax and semantics of GPFL in a formal way. The effort required by this formal specification is judged reasonable by the authors, and within reach of average engineers which have been exposed previously to operational semantics theories. Newcomers life is made easier by the numerous manuals, examples and tutorials available for the $\mathbb{K}$ framework. The tool support is a welcome help during the specification process. In particular, the ability to execute (or simulate) $\mathbb{K}$ formal specifications helps greatly when developing and fine tuning the language specification, and when producing smoke tests for the implementation.

Following such a specification process may seem to be in complete contradiction to any agile development principles [4]. However, using a tool-supported *executable* specification methodology allows to comply with one of the pillars of agile development: *early feedback*. As the language specification is executable, it is possible to ask final users (if some are available) to test the language and provide feedbacks on different aspects of the language, including its expressivity. In fact, IBM's Continuous Engineering development methodology [24] advocates for the use of executable models at every steps of the development.

To summarize, with regard to the benefits of putting the effort to produce a *formal* specification, the authors opinion, on improved quality and usefulness of formal specifications compared to non formal specifications written in natural language, is relatively well summarized in the following statement by David Schmidt [22], which is supported by the numerous ambiguities (and their consequences) in natural language specifications of common programming languages like C/C++ or Java [12].

> "Since data structures like symbol tables and storage vectors are explicit, a language's subtleties are stated clearly and its flaws are exposed as awkward codings in the semantics. This helps a designer tune the language's definition and write a better language manual. With a semantics definition in hand, a compiler writer can produce a correct implementation of the language; similarly, a user can study the semantics definition instead of writing random test programs."
>
> David Schmidt in ACM Computing Surveys [22]

In the experimentation reported in this paper, no formal analysis of the formal specification produced has been attempted. In future work, the authors plan to try some of the experimental tools available with the $\mathbb{K}$ framework on GPFL's specification. If time allows, a similar experimentation could be repeated with other tools oriented toward the formal specification of languages.

References

[1] John W. Backus (1959): *The Syntax and Semantics of the Proposed International Algebraic Language of the Zurich ACM-GAMM Conference*. In: *Proc. Int. Conf. Information Processing*, UNESCO, pp. 125–132.

[2] H. J. S. Basten, J. van den Bos, M. A. Hills, P. Klint, A. W. Lankamp, B. Lisser, A. J. van der Ploeg, T. van der Storm & J. J. Vinju (2015): *Modular Language Implementation in Rascal – Experience Report*. *Science of Computer Programming* 114, pp. 7–19, doi:10.1016/j.scico.2015.11.003.

[3] Fabricio Chalub & Christiano Braga (2007): *Maude MSOS Tool*. In: *Proc. Int. Work. Rewriting Logic and its Applications*, *Electronic Notes in Theoretical Computer Science* 176, Elsevier Science Publishers B. V., pp. 133–146, doi:10.1016/j.entcs.2007.06.012.

[4] Alistair Cockburn (2007): *Agile Software Development: The Cooperative Game*, 2nd edition. Pearson Education.

[5] Olivier Dubuisson (2000): *ASN.1 – Communication between Heterogeneous Systems*. OSS Nokalva. Available at `http://www.oss.com/asn1/dubuisson.html`. Translated from French by Philippe Fouquart.

[6] Matthias Felleisen, Robert Bruce Findler & Matthew Flatt (2009): *Semantics Engineering with PLT Redex*. The MIT Press.

[7] Matthew Flatt & PLT (2010): *Reference: Racket*. PLT-TR 2010-1, PLT Design Inc. `https://racket-lang.org/tr1/`.

[8] Jean van Heijenoort (2002): *From Frege to Gödel: A Source Book in Mathematical Logic, 1879-1931*, chapter Peano (1889). The principles of arithmetic, presented by a new method. Source Books in the History of the Sciences, Harvard University Press. A translation and excerpt of Peano's 1889 paper "Arithmetices principia, nova methodo exposita".

[9] Jean-Marc Jézéquel, Olivier Barais & Franck Fleurey (2011): *Summer School on Generative and Transformational Techniques in Software Engineering*, chapter Model Driven Language Engineering with Kermeta, pp. 201–221. *Lecture Notes in Computer Science* 6491, Springer Berlin Heidelberg, doi:10.1007/978-3-642-18023-1_5.

[10] Jean-Marc Jézéquel, Benoit Combemale, Olivier Barais, Martin Monperrus & François Fouquet (2013): *Mashup of metalanguages and its implementation in the Kermeta language workbench*. *Software & Systems Modeling* 14(2), pp. 905–920, doi:10.1007/s10270-013-0354-4.

[11] Joint Technical Committee ISO/IEC JTC 1, Information technology, Subcommittee SC 6, Telecommunications and information exchange between systems (2015): *Information technology – Abstract Syntax Notation One (ASN.1): Specification of basic notation*. International Standard 8824-1, ISO/IEC. ISO/IEC version of ITU-T X.680 (08/2015).

[12] JSR-133 expert group (2004): *JSR-133 Java™ Memory Model and Thread Specification Revision*. Java Specification Request (JSR) 133, Sun Microsystems, Inc. `https://jcp.org/en/jsr/detail?id=133`.

[13] Casey Klein, John Clements, Christos Dimoulas, Carl Eastlund, Matthias Felleisen, Matthew Flatt, Jay A. McCarthy, Jon Rafkind, Sam Tobin-Hochstadt & Robert Bruce Findler (2012): *Run Your Research: On the Effectiveness of Lightweight Mechanization*. In: *Proc. Symp. Principles of Programming Languages*, *SIGPLAN Not.* 47, ACM, New York, NY, USA, pp. 285–296, doi:10.1145/2103656.2103691.

[14] P. Klint (2009): *Tribute to a great Meta-Technologist: from Centaur to The Meta-Environment*. In Y. Bertot, G. Huet, J.-J. Levy & G. Plotkin, editors: *From Semantics to Computer Science, Essays in Honour of Gilles Kahn*, Cambridge University Press, pp. 235–264, doi:10.1017/CBO9780511770524.012.

[15] P. Klint, J. J. Vinju & M. A. Hills (2011): *RLSRunner: Linking Rascal with K for Program Analysis*. In: *Proc. Int. Conf. Software Language Engineering*, Springer, doi:10.1007/978-3-642-28830-2_19.

[16] Paul Klint, Tijs van der Storm & Jurgen Vinju (2009): *RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation*. In: *Proc. Int. Working Conf. Source Code Analysis and Manipulation*, IEEE Computer Society, pp. 168–177, doi:10.1109/SCAM.2009.28.

[17] Donald E. Knuth (1964): *Backus Normal Form vs. Backus Naur Form*. *Commun. ACM* 7(12), pp. 735–736, doi:10.1145/355588.365140.

[18] Gurvan Le Guernic & José A. Galindo (2016): *Experience Report on the Formal Specification of a Packet Filtering Language Using the K Framework*. Research report 8967, Inria. `https://hal.inria.fr/hal-01385541v1`.

[19] Dominic P. Mulligan, Scott Owens, Kathryn E. Gray, Tom Ridge & Peter Sewell (2014): *Lem: Reusable Engineering of Real-world Semantics*. In: *Proc. Int. Conf. Functional Programming*, *SIGPLAN Not.* 49, ACM, pp. 175–188, doi:10.1145/2692915.2628143.

[20] Grigore Roşu & Traian Florin Şerbănuţă (2010): *An Overview of the $\mathbb{K}$ Semantic Framework*. *The Journal of Logic and Algebraic Programming* 79(6), pp. 397–434, doi:10.1016/j.jlap.2010.03.012.

[21] Grigore Roşu & Traian Florin Şerbănuţă (2014): $\mathbb{K}$ *Overview and SIMPLE Case Study*. In: *Proc. Int. Work. K Framework and its Applications (K 2011)*, *Electronic Notes in Theoretical Computer Science* 304, pp. 3–56, doi:10.1016/j.entcs.2014.05.002.

[22] David A. Schmidt (1996): *Programming Language Semantics*. *ACM Computing Surveys* 28(1), doi:10.1145/234313.234419.

[23] Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar & Rok Strniša (2010): *Ott: Effective Tool Support for the Working Semanticist*. *J. Functional Programming* 20(1), pp. 71–122, doi:10.1017/S0956796809990293.

[24] Cathleen Shamieh (2014): *Continuous Engineering For Dummies®*. IBM Limited Edition, John Wiley & Sons, Inc.

[25] T. van der Storm & J. J. Vinju (2008): *Using the Meta-Environment for Domain Specific Language Engineering*. Technical Report SEN-R0805, CWI Software Engineering.

[26] Traian Florin Şerbănuţă, Andrei Arusoaie, David Lazar, Chucky Ellison, Dorel Lucanu & Grigore Roşu (2014): *The $\mathbb{K}$ Primer (version 3.3)*. *Electronic Notes in Theoretical Computer Science* 304, pp. 57–80, doi:10.1016/j.entcs.2014.05.003. Proc. Int. Work. K Framework and its Applications (K 2011).