Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science 304 (2014) 151–166
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Parsing $\mathbb{K}$ definitions

Radu Mereuţă[1] Gheorghe Grigoraş[2]

Faculty of Computer Science
Alexandru Ioan Cuza University
Iaşi, Romania

Abstract

This paper describes our approach in parsing a $\mathbb{K}$ definition. The difficulty of the problem is given by the nature of the $\mathbb{K}$-framework, where the user can define the operational semantics of a language by inserting pieces of concrete syntax in the $\mathbb{K}$ code in a natural way. Our main contribution shows how to make use of SDF and the disambiguation mechanisms in the context of $\mathbb{K}$-framework.

Keywords: parsing, $\mathbb{K}$-framework, syntax, ambiguities, disambiguation

1 Introduction

$\mathbb{K}$ [14] is a rewriting-based semantic definitional framework suitable for defining semantics for programming languages and calculi, as well as type systems or formal analysis tools in an executable environment. A main advantage of having the executable semantical definition of a language is the ability to use it with analysis tools to verify programs. An illustrative example is given by MatchC [21] (under development), a tool based on Matching Logic [13] that uses the K definition of a subset of C.

A definition in $\mathbb{K}$ includes *configurations*, *syntax declarations*, *computations* and *rules*. Configurations organize the rewrite system state in an XML style in units, labeled cells which can be nested. The syntax declarations specify the shape of the language. Computations are obtained by embedding the "computational meaning" of the original language in the rewrite system. $\mathbb{K}$ (rewrite) rules generalize conventional rewrite rules by making it explicit which part of the term they read, write or do not care about [14].

[1] Email: `radu.mereuta@info.uaic.ro`
[2] Email: `grigoras@info.uaic.ro`

http://dx.doi.org/10.1016/j.entcs.2014.05.008
1571-0661/© 2014 Elsevier B.V. All rights reserved.

K-Maude [15], the current tool supporting 𝕂, proved to be quite scalable and applicable to real world programming languages such as Scheme [17], Verilog [16], Java 1.4 [8] and C [7] (with others underway). However, since 𝕂-Maude relies on the Maude parser to parse 𝕂 definitions, there are cases when new definitions introduce ambiguities. Moreover, in order to be translated in Maude, these definitions need to be syntactically correct; therefore it is preferable to have a parser able to handle 𝕂 definitions. The design of such a parser is an intricate task because the 𝕂 definitions are quite complex, combining 𝕂 syntactical constructs with fragments of syntax from the defined language. After several experiments we quickly ruled out parsers that use a scanner before the parser because they do not offer the generality of parsing different styles of programs embedded into one another. The solution came in the form of SDF [9] and its scannerless generalized parser. We generate several parsers for different purposes: one to extract the syntax declarations from a definition, another one is used to parse programs, and the last is used to parse the semantic rules (this is the most complex as it must handle constructors from two languages at the same time).

This paper is divided into 6 sections where in Section 2 we introduce a representative example to demonstrate the main problem that we are trying to solve. Following in Section 3 we present an abstract view of the chosen solution as a way of dealing with ambiguities in languages that embed other languages. Section 4 provides a more in depth view of the technique used and explains some of the details that lead to the chosen method. Some tools that have similar solutions are presented in Section 5.

2 Challenges in parsing 𝕂 definitions

To understand the problem of parsing a 𝕂 definition, let us look at an example:

```
module EXAMPLE
    syntax Exp ::= Id | Int | Exp "=" Exp
    syntax Stm ::= Exp ";" | Stm Stm

    configuration <T> <k> .K </k> <env> .Map </env> </T>

    rule [store]: <k> I $\boxed{=}$ V => V:Int </k>
                  <env> I:Id |-> (_ => V) </env>
end module
```

Every 𝕂 definition is composed of modules, which start with the keyword ”module” and a unique name (by convention is all capital letters). Each module may contain sentences of three types: syntax declarations, configurations and rewrite rules. Each sentence will start with its corresponding keyword and restricts the use of keywords inside the sentence (as explained in Section 4.1). In this case we consider a small language that accepts addition and assignment over identifiers and integers.

To give structure to the rewrite system we will declare a configuration consisting of two cells: a cell `<k>` for computation structures and a cell `<env>` for binding the variables to values. We initialize them with empty (identity) elements. In general the cells can have one of the five main syntactic categories (or sorts) used in $\mathbb{K}$: `K`, `Bag`, `List`, `Set` and `Map`. The sort `K` is considered special as it can be extended with new syntactical constructs from the defined language and it is mainly used to match on computations.

The operational semantics of the above syntactical constructs is given by $\mathbb{K}$ rules. They describe how the configuration is changed when these constructs are executed. For instance, the semantics of the assignment operator is given by the "store" rule presented above. The underscore denotes an anonymous meta-variable. We use different font styles to mark and exhibit that this simple example includes syntactical constructs belonging to different languages. The normal parts represent the grammar of $\mathbb{K}$, the bold face parts highlight the meta-variable declarations, and the only part that is actually of the defined language syntax is the boxed part (here given by "="). The left and right hand sides of the assignment are meta-variables ranging over two particular syntactical categories of the defined language. The variables I and V, once declared, keep their meaning, their sort and their bindings through the context of the entire rule. They can be declared anywhere in their visibility scope.

The back-end of the $\mathbb{K}$ tool is designed to accept $\mathbb{K}$ definitions represented as a collection of abstract syntactic trees (ASTs). An AST is the collapsed representation of the parse tree resulted after parsing a text. Working with the AST makes handling the input a lot easier as the unnecessary information is eliminated in the parsing process (e.g., white spaces and comments - which have no semantic meaning to the language). To obtain the AST we need a tool able to parse definitions as above, solve the ambiguities, infer the types for each construct and transform them into pure $\mathbb{K}$ definitions (using ASTs) like the following one:

```
rule(
    cell( "k",
          =>(=(IdVar("I"), IntVar("V")), IntVar("V")),
          "k"),
    cell(
          "env",
          |->( IdVar("I"), =>(IntVar("_"), IntVar("V"))),
          "env"))
```

All the variables should have a type now, the anonymous variables should have an inferred type, the priority of the operators should be resolved and all of the language constructs should be in a prefix form that can be handled by a rewrite engine.

The current implementation of the $\mathbb{K}$-tool [15] has a front-end that transforms an annotated BNF definition of a language into a Maude [4] module which is later used to parse the $\mathbb{K}$ rewrite rules. This solution often leads to ambiguities because of mixfix operators and the preregularity property that needs to be satisfied by the ordered sorts. The biggest downfall was the lack of support for solving the ambiguities and conflicts provided by Maude.

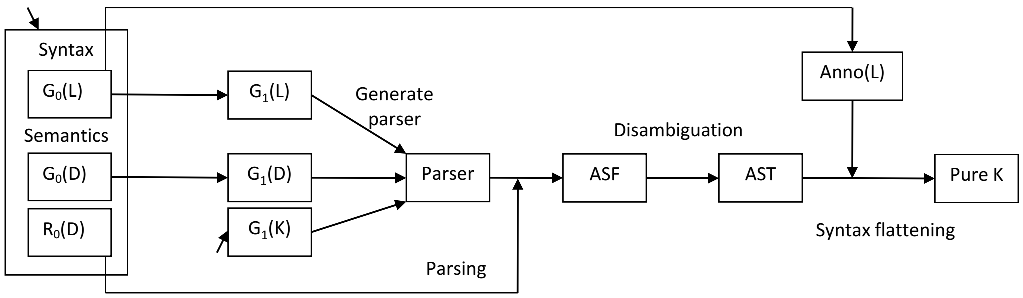

Fig. 1. The front end workflow (abstract)

3 Parsing technique

A $\mathbb{K}$ definition is composed out of three main parts: syntax declaration for the language in question - denoted in Figure 1 by G_0(L), extra syntactic constructs that extend the sort `K` - G_0(D) - and rewrite rules that give the semantics - R_0(D) .

The workflow of the solution we propose is represented in Figure 1. Level 0 represents $\mathbb{K}$ code and is one of the entry points in the compilation process. This is the input for the first processing step: basic parsing, which extracts the grammar for the defined language, namely G_0(L). To generate a parser for the complete definition, G_1(L) and G_1(D) will be composed with G_1(K) - the initial $\mathbb{K}$ definition without any syntactical constructs.

The embedding of language constructs in the $\mathbb{K}$ syntax is done with the help of the sort `K`. In fact this is the place where the name of the technique comes from. A term of sort `K` represents a computation which from a parsing perspective is a language construct. For the rewrite rules we also need meta-variables, which can take the place of non-terminals in syntactic constructs. In practice, putting all of these together is tricky because we want to offer the user easy access to the concrete syntax but also maintain the sanity of the definition.

The chosen method is similar to the one used in ASF+SDF [19] in the sense that we add new transitions to connect the grammar of $\mathbb{K}$ and the grammar given by the user. For each non-terminal X defined by the user, we add two new productions: `X ->` K and `K ->` X. This technique will allow the flexibility of matching the syntactic constructs in contexts that normally would not be allowed by the type system. The idea is to allow the user to write rules as he/she would write the AST representation, but with the convenience of concrete syntax (mixfix form) which in some cases can be more intuitive than writing the constructor. Details about how this is being solved are presented in Section 4.2.

Because we combine two grammars at runtime (without the users intervention), some problems may occur at parsing time. Therefore a post-processor is required to solve some of these problems. In this case we will use disambiguation filters for context-free languages (procedures that choose from a range of possible parses for a sentence, the most appropriate one according to some criteria [6]). This type of disambiguation, based on late type-checking, has been applied in [3] where the authors use more information from the quoting-unquoting mechanism.

There are many ways for disambiguation of ambiguous grammars ranging from simple syntactic criteria to semantic criteria [6]. Some classes of disambiguation rules turn out to be adequate for declarative filters [1] (associativity, priority), but in the case of $\mathbb{K}$ definitions, a more complex approach was needed that uses both the declarative methods and custom procedures using semantic information.

Each of the main steps of the proposed disambiguation filter below represents a type of ambiguity that could be found in the AST. The following rules should be viewed as rewrite rules, in the sense that if the left pattern is being matched somewhere in the AST, then that term will be replaced with the right part if the `where` condition is met.

Let $\leq$ denote the subsorting relation, where `syntax A ::= B` (or `B -> A` in SDF) implies `B` $\leq$ `A`. In this case we will consider the relation to be transitively closed.

(i) update context sensitive information
  (a) collect configuration info and then apply disambiguation filter for every cell, in every rule:

$$\begin{aligned} &\mathit{rule}\left\langle\, \mathit{amb}(T : \{S_1, \ldots, S_n\}) \,\right\rangle_{\mathsf{cell}} \qquad\qquad \Rightarrow \\ &\mathit{rule}\left\langle\, \mathit{amb}(T : \{S_i | S_i \leq \mathit{type}(\mathit{cell})\}) \,\right\rangle_{\mathsf{cell}} \end{aligned}$$

  In other words, a constructor is eliminated if it is found in a cell rule after parsing and has a sort that is not smaller or equal to the type declared in the configuration.
  (b) collect variable declarations and then update the type of each appearance of the variables:

$$\mathit{Var} : S \Rightarrow \mathit{Var} : S' \quad \textbf{where} \text{ variable Var has type S'}$$

  If a variable *Var* has been typed by the user somewhere in the rule, then everywhere in the rule that variable will have the same type on every location it appears. Type inconsistencies will be solved in the next steps.

(ii) second type checking
  (a) choose well typed terms (top-down type checking):

$$\begin{aligned} &\mathit{term}(..., \mathit{amb}(T : \{S_1, \ldots, S_n\}), ...) \qquad\qquad \Rightarrow \\ &\mathit{term}(..., \mathit{amb}(T : \{S_i \mid S_i \leq \mathit{originalSort}\}), ...) \end{aligned}$$

  This filter is applied from top to bottom, and it behaves in a similar way to normal parsers in the sense that it deletes terms that would be eliminated by the type system in the original grammar. The difference here is that it applies the rule only if there is at least one term left.

(b) choose best fit (bottom-up type checking):

$$amb(T_1, \ldots, T_n) \Rightarrow T \quad \text{where } T = bestFit(T_1, \ldots, T_n)$$

applied bottom-up and chooses the trees that have the least type violations (described in more detail in the next paragraph).

(c) choose maximal sort for the other ambiguities:

$$amb(T : \{S_1, \ldots, S_n\}) \Rightarrow T : \max(\{S_1, \ldots, S_n\})$$

Because the type system is not active, some overloaded operators may generate an ambiguity for each declaration. This rule keeps only the most general sort for that operator.

The first part will be described in more detail in Section 4.4, but the second type checking requires a bit more explanations before continuing. Because we introduce new productions that links $\mathbb{K}$ productions with the grammar defined by the user, some anomalies may appear which would be normally handled by the type system. At this step, there is also more information regarding meta-variables and cell types. To make this case clearer, let us look at an example. First we declare two productions that define a polymorphic operator:

```
syntax Int    ::= Int    "+" Int
syntax String ::= String "+" String
```

To make the connection with the $\mathbb{K}$ language, we add the following constructs:

```
syntax K ::= Int | String
syntax Int ::= K
syntax String ::= K
```

This modification will allow for insertions of rewrites and meta-variables in any context. It also means that a rule like the following:

```
rule  A:Int +Int B:Int => A + B
```

will produce the following Abstract Syntax Forest (an AST with multiple valid results):

```
rule =>(+Int(A:Int, B:Int),
        amb(
            Int+Int(A, B),
            String+String(A, B)
           )
        )
```

The *bestFit* rule, described at step (ii)-(a), uses information from the original grammar concerning the types of the non-terminals of each production rule to eliminate some anomalies that could occur. The function *bestFit* returns the branch with least type violation. In the example above, after the first step where we update the context sensitive information (variable types), we can safely remove the `String+String` production because in the original grammar `Int` $\not\leq$ `String` (is not in a subsorting relation).

4 An SDF-based implementation

Here we describe an implementation of the above solution that uses SDF [9] and its scannerless generalized parser within Spoofax [5]. Currently a prototype is under development and it shows promising results. SDF's modularity proved to be really helpful at the level of integration between the K grammar and the defined language grammar.

A notable advantage of using this tool is the ability to generate Eclipse plug-ins that can recognize the language specified by the user. This goes as far as syntax coloring, error reporting, code folding, and seeing the parse tree resulted after the parsing step. This allowed us to experiment with several solutions that could be very close to the expected end result.

Because the chosen solution to parse the $\mathbb{K}$ definitions is very much dependent on the tool that does the actual parsing, this chapter will contain a detailed description of how the different stages of the compilation interact with the SDF and Stratego tools [5].

The input for every definition, is a text file with the extension ".k". This is passed to a primary parser which separates the different components of the definition into `syntax` declarations, `configurations` and `rules`. At this point only the information regarding the syntax declarations are completely available as they have a standard form for every definition. The configurations and rules are still stored as strings and will be processed in a later step. The precise details will be provided in Section 4.1.

Section 4.2 contains the motivation of choosing these technologies and shows how to generate a new parser starting from the result of the previous step. It consists of two parts because two grammars are being generated: one for the programming language described in the definition and one for the definition itself which require a few modifications from the form specified by the user.

In Section 4.4 is presented the infrastructure necessary to call the parser generated at the previous step and to disambiguate the Abstract Syntax Forest. To be able to eliminate the ambiguities, some extra information regarding the sort of the cells in a configuration is required, namely the list of subsorts and the list of syntactic constructs from the original grammar.

4.1 Basic parsing

For the first step in the compilation, the input file is sent to a parser that separates the text into modules. A 𝕂 module will contain a list of sentences of the form:

```
KModule ::= "module" ModName "is" KSentence* "endmodule"

KSentence ::= "rule" Hidden+
            | "configuration" Hidden+
            | "syntax" {SConstruct1 ">"}+
SConstruct1 ::= {SConstruct2 "|"}+
SConstruct2 ::= String | Id

Hidden ::= ~[\n\r\t\ ]+ - ("endmodule" | "rule" |
                           "syntax" | "configuration")
```

This grammar is enough to do the initial parsing and the result should be a non ambiguous one as the language constructs are parsed as *bubble terms* alongside 𝕂 syntax declarations. The idea comes from Maude and ASF+SDF [2] which also implement term rewriting on concrete syntax specified by the user. A bubble term starts with a keyword (here we have "rule" and "configuration") and continues matching words until the next keyword is found. In SDF this comes naturally because the parser is scannerless. In the grammar above, the `Hidden` construct matches any non-whitespace token, except for the keywords. This technique has been used by Moonen in [18] to extract code from texts that are not completely defined by the parser. In the case of 𝕂 we isolate the *water* (contents of rules that may include syntax from the language given by the user) from the *islands* (keywords that separate sentences).

The 𝕂 syntax declarations are a rearrangement of the SDF syntax to be closer to the BNF style. It starts with the keyword `syntax`, an identifier specifying the sort of the constructs and a list of production rules. The list of production rules can be separated by `">"` to specify that productions in the left hand side have a higher priority than the ones in the right. Inside a priority block there can be specified other productions separated by `"|"` and will be considered to have the same priority.

Syntactic productions and blocks can be annotated with the typical SDF disambiguation filters: `left`, `right` or `non-assoc`. Also the `prefer` and `avoid` non-patterns can be provided for extra flexibility.

The final step of the primary parsing, is to generate the 𝕂 Intermediate Language (KIL). This is a generic and simple format for representing 𝕂 definitions. We use in this case the XML format (there is also a Java version later in the compilation flow) for it's flexibility and widespread support. A simple visitor, written in Stratego, that matches on every syntactic construct can print the abstract representation in the more readable and general format. To be able to provide better error messages for the users, each term is also annotated with the location information which has the form "(start-line, start-column, end-line, end-column)". The next module example:

```
module MOD-NAME
```

```
        syntax Exp ::= Exp "*" Exp [left]
                     > left:
                       Exp "+" Exp [left]
                     | Exp "-" Exp [left]
                     > Exp "=" Exp [right]
        rule A + B => A +Int B
end module
```

will be transformed into:

```
<module value="MOD-NAME" loc="(1,0,8,4)">
    <syntax loc="(2,1,6,37)">
        <sort value="Exp" loc="(2,8,2,10)" />
        <priority loc="(2,16,2,33)">
            <production loc="(2,16,2,33)">
                <sort     value="Exp" loc="(2,16,2,18)" />
                <terminal value="*" loc="(2,20,2,22)" />
                <sort     value="Exp" loc="(2,24,2,26)" />
                <annotations>
                    <tag key="left" loc="(2,29,2,32)" />
                </annotations>
            </production>
        </priority>
        ...
    </syntax>
    <rule loc="(7,1,7,22)" value="rule A + B =&gt; A +Int B" />
</module>
```

where each major syntactical construct is transformed into an XML node. Each node will have some location information and it may have other specific attributes; for example the module must have a name represented here by the `value` tag and so on.

4.2 Generate a new parser

The 𝕂 tool should allow the users to specify the grammar of a language and the semantic rules and in the background to take a program and then interpret it to find the result.

The first step in the list is to parse the program. This requires a new parser that will be generated from the grammar that the user specified in the definition. The transformation is straightforward as it is just a rearrangement of the original SDF syntax. The priority blocks are kept with their corresponding production rules. The only modifications that will be done is to move the subsortings (chain productions) and `bracket` productions outside the priority mechanism to avoid behaviors where the parser rejects correct programs.

The second part is a bit more complex as it involves the composition of two grammars. The first grammar is the default K syntax containing layout, cells,

variables, operations on top sorts, etc. The second grammar is parametric and is derived from the one specified by the user.
The generated SDF file will contain the following parts:

(i) import K-Technique - for the top sorts, cells, built-in operations;

(ii) import Common - for layout and variables;

(iii) generate the production rules for the language;

(iv) generate transitions for every user defined sort to and from `K`[3];

(v) generate variables - for each sort declare a typed variable that contains the name of the sort.

The third step from the above list generates a grammar similar to the one for programs. The difference is made at step four, where we add transitions to and from sort `K`. These will allow the user to insert and replace parts of code from the users language. To stop the parser from entering an infinite cycle, we will limit their use with the priority system (see: [19]).

For the example above, the generated SDF will look something like this:

```
context-free priorities
{
    Exp "*" Exp -> Exp {left, cons("C1Syn")}
} > { left:
    Exp "+" Exp -> Exp {left, cons("C2Syn")}
    Exp "-" Exp -> Exp {left, cons("C3Syn")}
} > {
    Exp "=" Exp {right, cons("C4Syn")}
}
context-free priorities
    K -> Exp > Exp -> K
context-free priorities
    Exp -> K > K -> Exp
context-free syntax
    VARID  ":" "Exp" -> K {cons("ExpVar")}
```

Each new production added to the base $\mathbb{K}$ grammar is attached a uniquely generated constructor name so it can be traced back to the original declaration.

4.3 Collect information

Because the type system has been altered (transitions from all declared sorts to and from `K`), some anomalies may appear in the form of ambiguities (see Section 3). To solve them, we need to gather some information regarding the original syntax.

The first thing that we need is the partial ordered set of subsorts. This is going to be generated in the form of a set of relations, transitively closed, that can be

[3] We would like to thank the anonymous reviewers for their input on this solution.

easily read by Stratego. This list contains the default `K` top sorts and the sorts defined by the user.

The second step is the list of constructors and the original sort. For example the `Exp "*" Exp -> Exp {left, cons("C1Syn")}` construct will generate the following term: `("C1Syn", "Exp", ["Exp", "Exp"])`. This information is used in the second type checking filters presented in Section 3. The exact details will be presented in the next section.

4.4 Main parsing

This step is responsible for connecting all of the steps previously described, thus becoming the most complex one in the front end. Firstly, the grammar described in Section 4.2 is compiled into a new parser that can recognize the 𝕂 definition.

The first step is to parse the configurations and extract the sort of each cell. Typically a cell in configuration will contain other cells, or the empty element corresponding to each top sort. This is also a place to initialize some cells with terms specified in the syntax and this is the reason why a standard XML parser would not be of much help.

For the next example:

```
configuration <T>
                 <k> .K </k> <env> .Map </env>
                 <nextLoc> 0 </nextLoc>
                 <input> ListItem("Hello World!") </input>
              </T>
```

the following information about cell sorts will be extracted:

```
 - T       : Bag
 - k       : K
 - env     : Map
 - nextLoc : K
 - input   : List
```

The sort of a term is the most general sort that the term can be represented by. This information will be used later to disambiguate the cells in rules. From the AST obtained at this step, the 𝕂 Intermediate Language will be generated in the form of XML.

Secondly, this section describes the method of parsing the rules. This is the most difficult part to get right because here can be found the most complex combinations of 𝕂 syntax and user defined syntax. This is why the parser after this step will return an Abstract Syntax Forrest, that will be processed afterwards. The disambiguation filter is composed out of four main steps (introduced in Section 3):

(i) update context sensitive information
  (a) **disambiguate cell types**. This step uses information from the previous step and eliminates any parsing possibility that does not fit the sort type of the cell. For example the rule `rule <k> A => B </k> ...` have A and

B which will range through every top sort but the configuration says that only the `K` sort is allowed. Any type violation is reported to the user as an error, and the compilation process is stopped.

(b) **associate types to variables**. If a variable is declared as having a specific type in one part of the rule, then all of the other ambiguities in other places, due to that variable, are going to be eliminated. For instance, the statement `rule <k> A:Exp </k> <k> A </k> ...` - in the first part of the rule the user specified that the variable A has the type "Exp" and because the visibility space for variables is the entire rule every occurrence of `A` will be consider to have the declared sort. Again, any type violation is reported to the user as an error.

(ii) second type checking

(a) **choose well typed terms**. In this step, a tree traversal is performed (top-down) and at each node the following transformation is applied: check to see if the term is declared by the user, then apply the subsorting filter to all its children. The subsorting filter checks to see if the term is ambiguous and eliminates the terms that are not subsorted to the original sort. Because we do not want to completely reject the parse tree, a condition says that if the filter eliminates all of the children then the term is skipped and leaves it for the next filter. This filter comes close to the type system in conventional parsers with the difference that it doesn't reject parsing trees if it can still find a result.

(b) **choose best fit**. This final step is required to eliminate some anomalies that appear because the type system has been altered. This filter uses the list of constructors described in Section 4.4 and is applied like described at the end of Section 3.

(c) **choose maximal sort**. This step is applied on each ambiguous node and the most general sort is chosen from the list of parsing possibilities. The list of subsorts is required to eliminate every production that has a more general representative. This step is used to eliminate overloaded operators like `_,_` (list separator).

If after applying all of these steps ambiguities are still present in the parse tree, then an error is reported to the user to revise his grammar, or write some constructs in the prefixed, or labeled form. If everything executed correctly, then the AST is printed in the XML format ($\mathbb{K}$ Intermediate Language) and sent to the next stages of the compilation.

4.5 Performance and scalability

The tests we run, involve compiling real world $\mathbb{K}$ definitions on an Intel Core 2 Duo 2.66 GHz processor, with 4GB of RAM. The table below contains the various times and sizes of three of our definitions:

	imp	simple	modelink
files	24	24	34
size(kB)	34	40	57
basic parsing(s)	1.7	2.3	5.5
table generation(s)	2	2.9	12
sdf lines	1000	1100	2000
k rules	163	228	259
parsing rules(s)	2.8	3.8	9.1

These examples can be found in the Google Code repository of the 𝕂-framework: `http://code.google.com/p/k-framework/`. `Imp` is one of our smaller examples and thus it takes the least amount of time to compile. `Simple` is a bit more complex, with an expanded syntax and extra rules, this is why we can see an increase in the times for compilation and table generation. `Modelink` (still in progress) is one of our bigger definitions which stretches the limits of the prototype. These are only preliminary results, but show a promising start, already comparable to the 𝕂-Maude tool. We expect the execution times to drop slightly as we optimize different stages of the compilation.

5 Related work

The 𝕂-framework is not the first tool to use rewriting logic as it is only an optimization for specifying the executable semantics of programming languages. In fact, the current K-Maude tool is based on the Maude system. The legacy is most visible in the syntax module, where priorities between syntactic constructs are being specified globally with an index from 0 to 133. The logical kernel of the current version of Maude's parser is based on the SCP parsing algorithm [20]. SCP is a bidirectional, bottom-up and event-driven parser for unrestricted context-free grammars. From a computational perspective, it allows for an elegant and very efficient manipulation of a wide set of CFGs. Another very good advantage of Maude's parser, is reflection. This can be called *linguistic reflection*, that is, the possibility of parsing a term from which then can be extracted a grammar to parse some unanalyzed portions of that term–for example, parsing the top-level syntax of a module in a language allowing user-definable syntax, to obtain the grammar in which to parse expressions in that module.

Although Maude has a powerful system, it still has a few disadvantages. First of all, and the most important one, is that the lexer of the parser is standard and often terms need to be correctly delimited by spaces to obtain the correct results. Also, in the lexer category, it can be specified that terminals have to be declared as operators before an equation can be parsed. A second disadvantage would be that the parser is pretty much inaccessible, limiting the possibility of creating IDE extensions.

An older tool, but applied successfully for defining domain-specific languages and code analyses is the specification formalism ASF+SDF [2][10] which is a combination of the algebraic specification formalism ASF and the syntax definition formalism SDF. ASF+SDF specifications consist of modules, each module has an SDF-part (defining lexical and context-free syntax) and an ASF-part (defining equations). The SDF part corresponds to signatures in ordinary algebraic specification formalisms. However, syntax is not restricted to plain prefix notation since arbitrary context-free grammars can be defined. The syntax defined in the SDF-part of a module can be used immediately when defining equations, the syntax in equations is thus user-defined.

ASF+SDF will use variable declarations to later replace sorts in the rewrite equations. So far, it is the most expressive syntax for rewrite rules, that match on concrete syntax, and thus is the aim of $\mathbb{K}$-framework.

One of the newest tools that uses syntax definition and rewrite rules that match on concrete syntax, is Rascal [11]. This tool is the continuation of the ASF+SDF tool set and is based on similar theory when it comes to syntax definition. The program transformation is a bit different, as it is based on customizing visitors. This solution allows for more control, but loses conciseness.

```
switch (t) {
    case ‘<Exp e1> + <Exp e2>‘ : {
        print("<e1> plus <e2>");
    }
   ...
}
```

In the above example, one can notice that the syntactic construct is delimited by backquotes. The terminals ("+") are kept unchanged, but the variables are enclosed by < and >. This may be a safe way of managing any kind of syntax, but it makes it longer and generally speaking less elegant.

6 Conclusion

This work started because of the need to replace the K-Maude tool, currently used to parse $\mathbb{K}$ definitions. After comparing different parser generators, the most promising solution turned out to be SDF and its generalized parser. Having the possibility to write modular grammars, allowed us to integrate easily the two grammars in discussion (the K grammar and the defined language grammar). Because the downside of the context-free declarative grammars are ambiguities, special procedures needed to be developed to cope with the nondeterminism of the parsing step (most of these problems are generated because of the integration method).

This paper describes how to use SDF and the disambiguation mechanisms in the context of $\mathbb{K}$-framework. The first step is to read the definition and extract the syntax declarations. From these, a new parser is generated that can cope with the complexity of a $\mathbb{K}$ rewrite rule. The second step is to parse the entire definition and get a parse forest. The last step is necessary to filter the unwanted parsing possibilities. The final result should be a clean AST that represents the intended definition and which can now be used in the next steps of the compilation, towards a rewrite engine.

Because SDF has a very good connection with Eclipse with the help of Spoofax, future work in this direction includes a user friendly interface that will speed up the editing and testing phase. We are also working towards giving more intuitive error messages. This involves working with more permissive grammars that accept a bigger language, but with more complex disambiguation filters can reach the same result.

Acknowledgment

The research presented in this paper was supported in part by the DAK project Contract 161/15.06.2010, SMIS-CSNR 602-12516.

We would also like to thank the anonymous reviewers for their input in improving the solution presented in this paper.

References

[1] M.G.J. van den Brand, J. Scheerder, J. J. Vinju and E. Visser. Disambiguation Filters for Scannerless Generalized LR Parsers. Compiler Construction CC02, pages 143–158, Springer-Verlag. 2002

[2] M.G.J. van den Brand, J. Heering, P. Klint, and P.A. Olivier. Compiling Rewrite Systems: The ASF+SDF Compiler. *ACM Transactions on Programming Languages and Systems*, 24(4):334, 2002.

[3] Martin Bravenboer and Rob Vermaas and Jurgen J. Vinju and Eelco Visser, Generalized Type-Based Disambiguation of Meta Programs with Concrete Object Syntax, In Generative Programming and Component Engineering, 4th International Conference, 3676:157-172, 2005

[4] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer and Carolyn Talcott. The maude 2.0 system. In Robert Nieuwenhuis, editor, *Rewriting Techniques and Applications (RTA 2003)*, number 2706 in Lecture Notes in Computer Science, pages 76-87. Springer-Verlag, June 2003.

[5] Lennart C. L. Kats and Eelco Visser. The Spoofax Language Workbench. Rules for Declarative Specification of Languages and IDEs. Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, October 17-21, 2010, Reno, NV, USA. Pages 444-463

[6] Paul Klint and Eelco Visser. Using Filters for the Disambiguation of Context-free Grammars. Proc. ASMICS Workshop on Parsing Theory, pages 1–20. 1994

[7] Chucky Ellison and Grigore Roşu, A Formal Semantics of C with Applications, University of Illinois, URL: http://hdl.handle.net/2142/17414, November, 2010

[8] Azadeh Farzan, Feng Chen, José Meseguer and Grigore Roşu, Formal Analysis of Java Programs in JavaFAN, Proceedings of Computer-aided Verification (CAV'04), LNCS, volume 3114, pages 501 - 505, 2004

[9] J. Heering, P.R.H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism sdf - reference manual, 2001.

[10] J. Heering J. A. Bergstra and P. Klint. Algebraic specification. *ACM Press/Addison-Wesley*, 1989.

[11] Paul Klint, Tijs Storm and Jurgen Vinju. RASCAL: a Domain Specific Language for Source Code Analysis and Manipulation. SCAM 2009

[12] G. Rosu, CS322, Fall 2003 - Programming Language Design: Lecture Notes, Tech. Rep. UIUCDCS-R-2003-2897, Department of Computer Science, University of Illinois at Urbana-Champaign, lecture notes of a course taught at UIUC (December 2003).

[13] Grigore Roşu and Andrei Ştefănescu. Matching Logic: A New Program Verification Approach. Proceedings of the 2010 Workshop on Usable Verification (UV'10), Microsoft Research, 2010

[14] Grigore Roşu and Traian Florin Şerbănuţă. An overview of the K semantic framework. *Journal of Logic and Algebraic Programming*, 79(6):397-434, 2010.

[15] Grigore Roşu and Traian Florin Şerbănuţă. K-Maude: A Rewriting Based Tool for Semantics of Programming Languages. *Rewriting Logic and Its Applications - 8th International Workshop, WRLA* , 6381:104-122, 2010.

[16] Patrick O'Neil Meredith, Michael Katelman, José Meseguer and Grigore Roşu, A Formal Executable Semantics of Verilog, Eighth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE'10), IEEE, pages 179-188, doi:10.1109/MEMCOD.2010.555863, 2010

[17] Patrick Meredith, Mark Hills and Grigore Roşu, A K Definition of Scheme, University of Illinois at Urbana-Champaign, Department of Computer Science UIUCDCS-R-2007-2907, 2007

[18] Leon Moonen, Generating Robust Parsers using Island Grammars, Proceedings of the 8th Working Conference on Reverse Engineering", IEEE 2001

[19] Jurgen J. Vinju, Type-Driven Automatic Quotation of Concrete Object Code in Meta Programs, Rapid Integration of Software Engineering Techniques, Second International Workshop, 3943:97-112, RISE 2005

[20] J.F. Quesada. *The SCP parsing algorithm based on syntactic constraint propagation.* PhD thesis, 1997.

[21] Andrei Ştefănescu. MatchC: A Matching Logic Verifier using the $\mathbb{K}$-framework. In this volume.