Proceedings of the 2013 Federated Conference on
Computer Science and Information Systems pp. 1515–1522

Declarative Specification of References in DSLs

Dominik Lakatoš*, Jaroslav Porubän†, Michaela Bačíková‡
Technical University of Košice
Letná 9, Košice, Slovakia
Email: {dominik.lakatos*, jaroslav.poruban,† michaela.bacikova‡}@tuke.sk

***Abstract*—The occurrence of identifiers and references in computer languages is a common issue. The same applies for domain specific languages, whose popularity is increasing and there is a need for aid in their design process. This paper analyses the problem of identifiers and references in computer languages. Current methods use an imperative approach for supporting references in languages; therefore a language designer is required to manually write reference resolving. The method proposed in this paper perceives references and identifiers as language patterns, which can be specified in a declarative manner with much less knowledge about the problem of resolving references in computer languages.**

I. Introduction

REFERENCES in languages are common. We use simple identifiers in our everyday life to identify objects, activities, abstract designs, etc. Even every one of us has an identifier, which we call "name". If we want to reference to somebody else, we use the name of that person. The problem arises when somebody uses short name *John* and there is more than one person with that short name known to him. How do we know, which *John* is he or she referring to? In that case we have to know more information and understand the mutual scope of names between the communicating parties to be able to properly identify the correct person.

The situation is not different in the area of computer languages. Majority of computer languages use references in a form of variable names, function names or any other named structures. How can we decide if the identifier used in our code refers to this particular structure, if we have more than one structure declared with the same name? In the theory of computer languages, the process of searching for these identifiers is called *reference resolving*. The basic solution of reference resolving is routine: A language designer creates a table of identifiers for storing every identifier declared in the code. Then he/she defines a lookup method for searching in the table for a suitable identifier based on the given textual reference in the code. Every language needs to solve the lookup process with its own function even when such a function is well-known and similar in most languages. Defining the lookup function is a common and routine task, but the implementation is poorly automatized.

In the last few years we can notice the increase in the language-oriented development [1]. Developers create special small or medium sized languages, called *domain specific languages (DSLs)* for many different areas. The development of DSLs is difficult [2] and identifiers and references are common in DSLs, which doesn't make it easier without proper support. The problem of a reference resolving is particularly acute in the area of external DSLs, as they need to function autonomically without any existing general purpose language (GPL). In this paper we aim to simplify the specification of DSLs with references by proposing declarative manner of specifying language constructs as identifiers and references.

II. Declaring References in Languages

Every GPL uses some sort of references. Even the oldest ones use some a form of named variables and they need a method to resolve the variable identifiers on their places of usage. It is possible to identify two types of statements concerning variables:

- *declaration* - a place where a name (and, optionally, a type) is assigned to a variable
- *usage* - a place where we use or change the value contained in a named variable by referring to its name

The declaration of one variable can occur only once in a language sentence scope but usage of the variable can occur on multiple places. A typical example of a declaration is:

```
float a;
```

It is a declaration of a variable `a` restricted to contain a decimal value. A typical example of usage is an assignment or reading of a value:

```
a = 10; //assignment
factor(a); //reading of a value
```

Supporting tools for processing and executing sentences from a language can be created in two different ways. A language designer can design a language and manually implement all the tools needed for language processing. The second option is to write a language specification using any of the existing compiler generators and then generate the processing tools. Use of a compiler generators is preferable because it provides better automation and is frequently used in the area of specification of DSLs.

Compiler generator tools help with specifying lexical units, syntax, semantic actions [3] and sometimes they even explicitly define language concepts by means of abstract syntax. There are many compiler generator tools, we can mention the most common known Yacc [4], or even more sophisticated ones such as ANTLR [5], Beaver [6], LISA [7], [8], JastAdd [9] or Xtext framework [10], [11]. If we look on the problem of resolving textual references to places of its declaration, usually it can be addressed within the semantic

978-1-4673-4471-5/$25.00

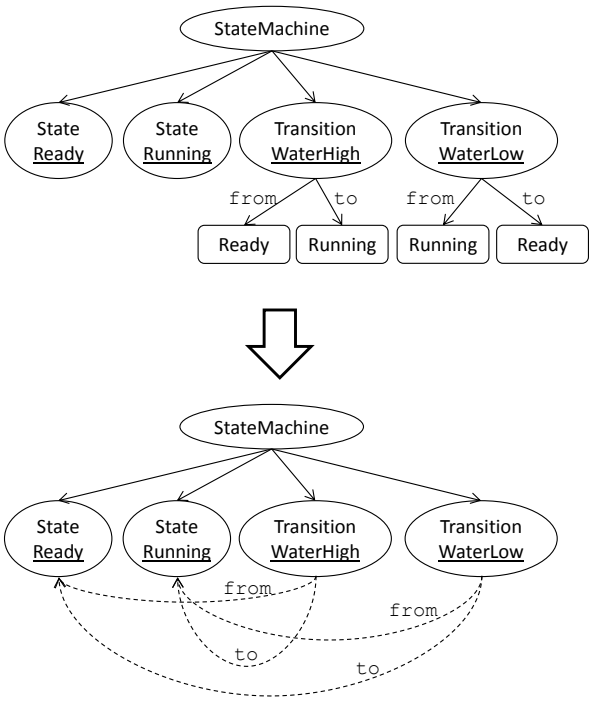

Fig. 1. Transformation from the abstract syntax tree to the abstract syntax graph

actions of the designed language. Some of the current compiler generators use attribute grammars in order to simplify the specification of semantic actions, for example in LISA. JastAdd uses an expanded version of attribute grammars [12] described by Hedin [13], [14] as reference attributed grammars (RAGs). RAGs aim to simplify attribute grammars for references by allowing attributes to be of reference type to other nodes in the syntax tree. RAGs only expand theoretical type model of attribute types and the actual process of discovering nodes in the tree needs to be manually implemented in the semantic actions of an attribute grammar. Xtext framework offers different approach with simple definition of referenced node types, scoping needs to be programmed with provided API.

We would like to show the usual process of declaring and resolving references on an example of a simple language of state machines. For the purposes of better comparison with the existing research we will use the example of the state machine language used in the JastAdd tutorial paper [15].

Listing 1. EBNF context-free grammar for a state machine language [15]

```
<statemachine> ::= <declaration>*
<declaration>  ::= <state> | <transition>
<state>        ::= "state" ID ";"
<transition>   ::= "trans" ID":" ID"->" ID ";"
ID = [a-zA-Z][a-zA-Z0-9]*
```

In the listing 1 is specified the grammar of the state machine language, which consists of *declarations*. A declaration can be a *state* or a *transition*. A non-terminal for a *state* contains only the state name represented by the named token *ID*. A *transition* contains the name (*ID*) of the transition, the name of the starting *state* and the name of the ending *state* of the transition. The abstract syntax of the state machine language is displayed in the listing 2. For every non-terminal there is one concept in the abstract syntax and instead of using just textual terminals for references (as it is common during the language desing phase) we used declarations of actual references.

Listing 2. Abstract syntax for the state machine language

```
concept StateMachine
  AS: declarations: list of Declaration

concept Declaration

concept State : Declaration
  AS: name: string

concept Transition : Declaration
  AS: name: string, from: State, to: State
```

In the abstract syntax and the grammar, it is possible to identify the point where there is a need for reference resolving during the language processing. Every *transition* has a textual name reference to the existing *state* concept. A sentence in the state machine language can be represented by a simple example of identifiers and references (see listing 3).

Listing 3. State machine language example sentence

```
state Ready;
state Running;
trans WaterHigh: Ready -> Running;
trans WaterLow: Running -> Ready;
```

Each *state* has a name, which acts as an identifier as well as each *transition* has a name serving as an identifier, but in this example we actually need only the state identifier because there will be no reference to the transition names. In the listing 3 we can see two states: *Ready* and *Running* and there are two transitions, each using the already defined state names as starting and ending states. During the language processing, the names represented as strings have to be connected to the actual state declarations. The process of interconnecting the referenced language concepts in any computer language can be perceived as a transformation from abstract syntax tree (AST) to abstract syntax graph (ASG). AST is a tree structure of language concepts created directly after parsing of the language textual form. Tree structure of textual forms of languages does not allow creation of direct references to existing declaration.Therefore we need textual placeholder for identifier to make references possible in later phase of language processing. ASG is a structure transformed from AST, which allows references from any node to any other node, so it means that a graph structure is created by extending the tree structure. The AST and ASG of our state machine code are displayed in fig. 1.

The transformation from AST to ASG in current compiler generators is accomplished manually within semantic actions. An example of semantic actions defined via attributed grammar written in JastAdd for a simple name analysis of the state machine language is shown in listing 4. Attributes are specified

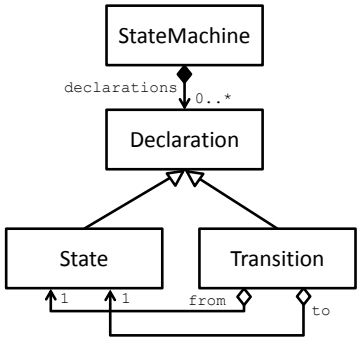

Fig. 2. Model of the state machine language

for *from Transition* named *source* and for *to Transition* named *target*. The *lookup* attribute is used for searching for a specific named *State* in the already specified list of *States*. Understanding of the provided example requires more knowledge about the specification notation used in JastAdd [15], but it shows the common amount of code needed to solve references in a language sentence and this problem is repeating in most of existing languages.

Listing 4. Code for name analysis for the state machine language specification in JastAdd [15]

```
aspect NameAnalysis {
 syn State Transition.source() =
       lookup(getSourceLabel());
 syn State Transition.target() =
       lookup(getTargetLabel());
 inh State Declaration
               .lookup(String label);

 eq StateMachine.getDeclaration(int i)
       .lookup(String label) {
   for (Declaration d :
               getDeclarationList()) {
     State match = d.localLookup(label);
     if (match != null) return match;
   }
   return null;
 }
 syn State Declaration
    .localLookup(String label) = null;
 eq State.localLookup(String label) =
   (label.equals(getLabel()))?this:null;
}
```

III. Language specification with annotated classes

There are different methods of specifying languages. The most common is in form of *grammars*. In our approach we use our own tool for language specification, which allows to model languages as classes in object-oriented programming paradigm [16]. This way the reuse of UML modeling tools for modeling languages [17] is possible.

In order to use our approach, the model of the state machine language has to be described by classes as described in Fig. 2. Each class in the model represents exactly one language concept. The connections between the language concepts are represented by the relationships in the model (inheritance, composition). Each UML composition relationship describes a required connection between the concepts, while the aggregation relationship means a connection created with identifiers. Generalization is used to represent alternation for the parent concept.

Modeling a language in a *graphical form* is great for general overview, but as we enrich a model with additional information such as concrete syntax and semantics, it becomes hard to understand. Execution of a plain model without any specified actions or supporting tool is usually not possible. In order to solve these problems and for practical purposes we define the language concepts in a textual form of classes of object oriented languages. The first class could be created for the root concept (*StateMachine*) of the state machine language.

Listing 5. StateMachine concept specification

```
class StateMachine {
  List<Declaration> declarations;

  StateMachine(List<Declaration> decls) {
    this.declarations = decls;
  }
}
```

Each class diagram is a representation of the abstract syntax with the recognized language concepts and their connections, as it is possible to notice in Fig. 2. Abstract syntax of *StateMachine* in listing 5 is represented by the definition of the class fields, in this case we have only one field named *declarations* for storing the list of *Declaration* concepts. Concrete syntax is represented by the class constructor. In this case there is only one concrete syntax representation (hence one constructor) which consists only of the list of *Declaration* concepts. It is a simple example for defining language constructs in a form recognizable to any programmer with a knowledge of object-oriented programming. In later examples we will discuss additional forms of concrete syntax specification. More details about this representation can be found in [16].

The next specification is a declaration of an abstract language concept *Declaration*, which in grammar form serves as a non-terminal for choosing between *State* and *Transition* non-terminals. It is possible to achieve the same effect in our language specification using inheritance and abstract classes. Therefore, the *Declaration* concept can be specified as a simple empty abstract class or an interface. We have chosen to use abstract class as this representation is closer to our model situation.

Listing 6. Declaration concept specification

```
abstract class Declaration {}
```

A. Declaring identifiers

The presence of identifiers and references is so common in computer languages, that we can consider it a language pattern. This pattern was already used in languages, but we have identified and distilled it as one of the building blocks of languages and therefore it can be extracted and later used

in any language in form of declaration. Our example of the state machine language uses a special form of unique named identifiers for the *State* concepts. And this is where our method comes forward and provides the capability to simply annotate a field which should be used as a special identifier for encapsulating the language concept. In our example we have the *State* language concept with the *label* property, which serves as an identifier of the *State* instance (see listing 7). In order to declare the *label* property to be used as identifier, we only need to annotate the property with the *@Identifier* annotation. The optional options available for the *@Identifier* annotation will be discussed later in this paper.

Listing 7. State concept specification

```
class State extends Declaration {
  @Identifier
  String label;

  @Before("state")
  @After(";")
  State(@Token("ID") String id) {
    this.label = id;
  }
}
```

We have used also other annotations, which are not new in our language specification. The *@Before* and *@After* annotations serve for defining terminal symbols in textual concrete syntax of *State*. The *@Token* annotation has a similar purpose, it can be used to concretize the lexical symbol used for checking and extracting data. If we compare this concrete syntax specification with the grammar defined for the state machine language in listing 1, it is simple to find connections between both specifications. The token *ID* is already defined in a special language configuration file in our tool and it serves for accepting names.

B. Declaring references

For the identifiers to have a logical meaning in the language, it is needed to reference to them from another place in the language sentence. The places used for referencing to existing identifiers are marked with the *@References* annotation. In our example we marked the constructor parameters *sourceLabel* and *targetLabel* as referencing names to the *State* concept. We need to define the concrete syntax according to the grammar from listing 1, therefore we are using the *@Before*, *@After* and *@Token* annotations as well.

Listing 8. Transition concept specification

```
class Transition extends Declaration {
  String label;
  State source;
  State target;

  @Before("trans")
  @After(";")
  Transition(
    @Token("ID")
    String label,

    @Before(":")
    @Token("ID")
    @References(State.class, field = "source")
    String sourceLabel,

    @Before("->")
    @Token("ID")
    @References(State.class, field = "target")
    String targetLabel
  ) {
    this.label = label;
  }
}
```

The usage of the *@References* annotation has an impact on some programming techniques. We are using *@References* to annotate constructor parameters and this way we are assigning special behavior to those parameters. Our language specification implementation allows us not to store the value of the annotated constructor parameter in any traditional programming way, actually this value is automatically stored for us during the language processing phase. Therefore, in the listing 8 it is possible to leave the constructor body with only one statement for storing the *label* of the transition. The requirement for the usage of the *@References* annotation is that it should be used on a *String* parameter and it is required to specify the referencing class (in the *value* parameter of the annotation), in our example it is the *State* class. The next parameter of *@References* is *field*, which allows us to define the name of the referenced class field (sometimes called property) used to store the specified language concept (object of the specified class). Both parameters are used later to filter all identifiers according to their corresponding language concepts, and they are used for injecting [18] appropriate object into the class field using reflection [19]. In our example of the *Transition* concept the objects of the *State* class are injected into the *source* and *target* fields, which are also checked for type consistency.

IV. Scope of References

References in computer languages are seldom created within one universal scope of availability. Even if there is a universal scope, called global scope, languages are not using only this one scope. Every variable in the language can be declared in a different scope and the rules for discovering a proper defined variable can be very difficult to define. An example of different scopes for the *count* variable is shown in listing 9. The *count* variable is declared in the global scope and at the same time it is also declared in the local scope of the *foo* function (D2), therefore the output of *foo* is `0` as we are using the variable declared closer to the place of its usage. The output of *goo* is `10` as the *count* variable name is referencing to the global variable declaration (D1) because there is no other closer variable declaration with this name.

Listing 9. Simple scope example with the *count* variable

```
int count = 10; // D1

void foo() {
  int count = 0; // D2
  print count;
}
```

```
void goo() {
  print count;
}
```

Support for using identifiers and references in scopes is very different in most compiler generator tools. Most of the tools are delegating this work to the designer by means of custom implementation (Yacc [4], Flex [20], JavaCC [21], Beaver [6]), they offer attribute grammars to help with the issue as it is done in JastAdd [9] and Lisa [7] or Xtext framework provides API for implementing scopes [11]. Still the language designer needs to know how to implement the scoping of references for his/her new language. ANTLR [22] provides a special support for defining scopes of variables for non-terminals and it allows easier access in order to find the declared variable in the tree of scoped variables, but still the user is required to specify proper variables and implement the use of scopes in the lookup functions. Therefore we can summarize, that all popular compiler generator tools require the user to write a custom solution for almost every issue concerning the reference resolving of identifiers within any scope.

In our method we support automatic reference resolution with scoping rules. Result of parsing language sentence is provided in a form of AST. References with scopes are allowed by selecting valid nodes in the required scope. In order to select nodes in AST it is useful to use a tree quering method. Our current solution uses the XPath querying language [23] as it is specially designed for efficient filtering of XML nodes [24] and use simple syntax. AST is usually not represented in XML, but XML is also a tree structure, therefore, the transformation from AST to XML is straighforward.

Each language is different and each reference in the language can be defined in a different scope. In order to characterize scoping, we have divided references to three levels of scoping:

- Global scope
- Simple local scope
- Complex scope

A. Global scope

The state machine language used in the previous sections uses references without scope or we can say that it uses global scope. Every *State* language concept is uniquely named within the full scope of a language sentence. Therefore it is a language within the first level of scoping and there is no need to declare it with any additional parameters. Previous listings (5, 6, 7 and 8) of class specifications of the state machine language are sufficient for our compiler generator with automatic resolution of references.

B. Simple local scope

Declaration of a referenceable language concept in the scope of parent concept is an example of a simple local scope level. Any scoping with a simple XPath query is considered to be a simple local, scope level. We can illustrate the problem on an example of a language for description of departments with cars and employees. Every car has a name, a fuel type and a year of acquisition. Every employee has a name and he/she can have a name of a car, which can be used by him/her. Employees can have a permission to drive the car owned by the same department, but it is not possible to use cars from different departments. Name of the car in the employee specification line is actually a reference to a declared car name in the department. Textual representation of this language is displayed in listing 10.

Listing 10. Department language with cars and employees

```
department: Accounting
  car: Audi A4; diesel; 2008
  car: Ford Focus; gas; 2012
  empl: John Smith {Audi A4}
  empl: Emma Fisher {Ford Focus}
  empl: Jim Parker

department: Technical services
  car: VW Golf; diesel; 2010
  car: Ford Focus; gas; 2005
  empl: Steve Cosby {Ford Focus}
```

Specification of the *Employee* language concept in our class form is displayed in listing 11. Scoping in references is allowed with the *path* parameter in the *@References* annotation. The value of the *path* parameter is an XPath query for accessing all relevant *Car* concepts. In our example we have used the XPath `parent::Department/Car`, which means that we are expecting to traverse the AST from the actual *Employee* node to the parent node named *Department* and then to find all child nodes named *Car*. This way, we have specified scoping for referencing to the *Car* concept within the same *Department* concept. The method of finding the *Car* node with a proper identifier is included in our method and it is not necessary to specify it in the XPath query, the user only needs to define the path for all relevant language concepts.

Listing 11. Specification of the *Employee* concept in the department language

```
class Employee {
  String name;
  Car car;

  @Before("empl:")
  Employee(
    String name,
    @Before("{")
    @After("}")
    @References(value=Car.class,
        path="parent::Department/Car")
    String car
  ) {
      this.name = name;
  }

  @Before("empl:")
  Employee(String name) {
    this.name = name;
  }
}
```

In the context of our approach, the *simple local scope* can be perceived as any local scope, which can easily be defined with an XPath query inside the *@References* parameter *path*. The complexity of a possible query is depending on the user's knowledge of XPath. Although, we are not restricted only to XPath, it is possible to use any other language for traversing the tree structures, however it needs to be included in our language tool.

C. Complex scope

In our research we aim for tool support of DSLs, which are usually not as complex as general purpose languages and therefore do not need to use complex scopes. Although, sometimes it is useful to have complex scope rules. In this section we will discuss the power of XPath expressions in our language design method and we provide an overview for advanced scoping.

A usual behavior of reference resolving using an XPath expression in the *path* parameter of the *@References* annotation consists of adding an XPath predicate with a test of the name of an identifier to the end of the original expression. For example an expression for finding a *Car* with the name `Audi A4` from listing 11 is converted to the following XPath expression.

```
parent::Department/Car[_id="Audi A4"]
```

In some XPath queries it is useful to specify a place for filtering nodes before getting all nodes. In order to write an XPath expression for searching for appropriate variable `count` declaration from listing 9 we can use an XPath expression

```
(ancestor::*[Variable/_id="count"])[last()]/
    Variable[_id="count"]
```

In order to parameterize this XPath expression we can use `##cmp##` as a placeholder for `_id = "NAME"`, therefore our extended XPath expression can be written as follows:

```
(ancestor::*[Variable/##cmp##])[last()]/
    Variable[##cmp##]
```

This XPath expression first chooses the appropriate ancestor with expected named variable declaration.

There is always a possibility to completely forget our declarative way of solving references and to implement it on our own. Our language method allows it without any problems, but the user would lose a great advantage. Still there can be some very specific scoping rules, which cannot be properly described with XPath and in this case we recommend using the traditional manual implementation of referencing to identifiers. The user has to be aware that he needs to manage his own storage of identifiers as well as the method for connecting places of identifier usage to the language concepts with the declared identifiers. Manual solution is possible and sometimes needed, although we are sure it is not necessary in most DSLs and our identifiers and references patterns are sufficient in such scenarios.

V. Process of Reference Resolving

Our method for reference resolving consists of three main steps:

- Unique identifier
- Reference searching and injecting
- Creation of undefined identifiers

A. Unique identifier

The first step in our reference resolving method is to check for uniqueness of identifiers. Every identifier for one language concept needs to be unique in its own scale. If identifiers are not unique it would cause a problem in reference resolving process, as one reference needs to reference only one language concept. This part of our method is providing functionality, which is not commonly used in other tools and where the user needs to implement it on his own using a table of identifiers or any other similar method. The default behavior of the *@Identifier* annotation without parameters is that the concept name is unique in the global scale of a language sentence and for language concept. Standard setting for uniqueness of identifiers used for specifying the *state* concept in the state machine language (see listing 7) is using the *@Identifier* annotation without parameters, therefore during language processing it is tested if every *state* has a unique name.

In order to specify other scale of identifier uniqueness, we are using XPath expressions as it was described in the section IV-B. It is possible to define a query for nodes of AST in which there is only one occurrence of an identifier of the specified type. This way we can define automatic checking for uniqueness of identifiers of *car* in *department* in our department language example (listing 10).

Listing 12. Specification of the *Car* concept in the department language

```
class Car {
 @Identifier(unique="parent::Department")
  String name;
  String fuelType;
  int year;

 @Before("car:")
  Car(
   @Token("NAME")
    String name,

   @Before(";")
   @Token("NAME")
    String fuelType,

   @Before(";")
   @Token("YEAR")
    int year) {
      this.name = name;
      this.fuelType = fuelType;
      this.year = year;
  }
}
```

Listing 12 specifies the language concept *car*, but our main focus is on the *@Identifier* annotation with the *unique* parameter. With this parameter we have specified that every *car*

has a unique name only in the scope of its parent *department* language concept. Therefore it is possible to have cars with the same name in different departments and the parsing process will finish without any error. Otherwise, in the case of not specifying the *unique* parameter, the uniqueness of *car* name would be tested in the global scale, producing an error even if different departments would have a car with the same name.

Scoping of references defined in the section IV-B and scoping of identifier uniqueness is based on similar XPath expressions. It is recommended to scope uniqueness of an identifier when we are using scopes for referencing. Identifier XPath scopes are usually simpler then XPath expressions for finding proper identifiers in *@References*. As an example, consider uniqueness scope of identifier in a function for listing 9, which can be defined as a simple XPath for variable identifier: `parent::Function` and if we compare it with XPath expression for finding proper variable reference shown in the section IV-C, it is obvious that the XPath expression for uniqueness scope is much simpler to write and understand.

B. Reference searching and injecting

The second and the most important part of our method is actual resolving of textual references. Details about textual reference resolving were already described in this paper in section III and scoping details in section IV. In our method we are using declarative specification of references with the *@References* annotation, which marks the textual parameters containing names of referencing language concepts defined in the fields marked with the *@Identifier* annotation. Referencing can be constrained with the scope rules defined by XPath. Our method discoveres referenced language concepts and inject them into fields specified in the *field* (for specifying the field name of class) and *value* (used to define the referencing type) parameters of the *@References* annotation.

Reference resolving can be carried out in two ways:

- after parsing an entire input sentence (explicit)
- during the parsing process (implicit)

The explicit approach is about an explicit execution of reference resolvining at the end of the parsing process. After execution we get an exact information about any inconsistencies in identifiers and references. Any error can be detected right after explicit execution call and propagated to the user. On the other hand it needs this one explicit execution of reference resolving and it makes this solution less modular.

The implicit approach is about resolving references and identifiers after each language concept creation (usually during the parsing process). For each parsed language concept it is trying to resolve the unresolved references and to match them with identifiers. The advantage of such solution is elimination of explicit execution of reference resolving, on the other hand it is hard to check for inconsistencies as it cannot tell if the registration of the new language concepts has finished or not. It is possible only to check for the actual state of reference resolution and to get information about the non-resolved references. At the end of the process of language parsing we should not have any non-resolved references. It is optional to check for non-resolved references, when we are using the implicit approach to resolution of references.

Both presented approaches to reference resolution have advantages and disadvantages and it is possible to use either of them. A language designer should decide which approach would be preferable to his new language. The implicit approach does not need any action for resolution of the references, but cannot guarantee that all references have been resolved properly without any optional check. The explicit approach is better in performance as the resolution of references is done only once and it can propagate the error in case of unresolved references, but it is required to execute this method explicitly and therefore there is a direct dependency between the parsing process and the reference resolution process.

C. Creation of undefined identifiers

The third part of our method is focused on the solution of one specific problem concerning references and identifiers within languages. In languages it is possible to have a reference to an identifier without the previous declaration of the identifier. It is a common occurrence in untyped or dynamically typed languages to use a variable without previous declaration (see listing 13). It is safe to claim, that a declaration of an identifier is required only if the declaration contains additional information about the language concept except the identifier name.

Listing 13. Example of a language without variable declaration

```
x = 10; // it is possible to use this

var x;  // instead of this
x = 10;
```

The previous example represents a GPL sentence fragment, but the same situation can be found when designing a DSL language. If we would look on the sentence written in the state machine language in listing 3 it is clear that the declaration of *state* does not contain any additional information except the name of a *state*. Taking that information into account, we can write the same state machine without any state declaration as it is shown in the following listing 14.

Listing 14. State machine language example without state declaration

```
trans WaterHigh: Ready -> Running;
trans WaterLow: Running -> Ready;
```

The *Ready* and *Running* states are declared in the place of their usage and the later usage is used as a reference to the existing state. This feature allows simplification of language sentences for common language users, as DSLs usually require syntax, which is more practical than technical.

Our method supports the automatic creation of language concepts in the place of their reference in case of non-existent variable declaration in the language sentence. It can be specified by the *create* parameter in the *@References* annotation (example in listing 15). This parameter is set to *false* by default, therefore a non-existent referenced concept is not created. Setting the *create* parameter to *true* allows the automatic creation of language concepts. The only requirement

is the existence of a constructor with a string parameter and the usage of global scope for referencing as well as identifier uniqueness.

Listing 15. References annotation with create parameter

```
@References(State.class,
            field = "source", create = true)
```

Automatic creation of undefined identifiers is the last phase of our method for resolving identifiers and references. It runs after the phase of resolving references and only if there is at least one *create* parameter with value *true*. After each creation of a new language concept it is required to resolve references as AST has been modified.

VI. Conclusion

Current language tools support creation of languages with one or more methods, but they fail in the area of an exact declaration of references and identifiers in the created languages. A language designer has to implement checking of identifier uniqueness and lookup methods for finding proper referenced identifiers manually, as it is common with other tools such as Beaver, JavaCC, JastAdd, ANTLR or at least programatically select language concepts from the scope using API in Xtext framework.

In this paper we have presented the method for declarative specification of computer languages with identifiers and references. We have discovered and described language patterns for identifiers and references. These language patterns have different possible parameters to adjust their impact. It is possible to define the scope of uniqueness of an identifier. We have described different levels of scopes for referencing other language concepts and we have explained different levels of scoping on illustrative examples. The scoping levels we discovered are global scope, simple local scope and complex scope. Complex scope is used mostly for advanced scoping of variable names in programming languages. During our research, we have analyzed various languages and discovered the pattern for identifiers without explicit declaration of identifier before the usage. In order to cope with such form of languages we provide an option to automatically create language concept on the first occurrence of identifier and to use it as a reference on every other occurrence.

Every mentioned pattern and option has been integrated in our method for working with references and identifiers in computer languages, with a special orientation to DSLs. Our method allows declarative definition of identifiers and references instead of more common imperative solutions of other methods. A summary of our method is a declarative transformation of AST to ASG. The method has been implemented in our YAJCo[1] tool, which can be found on the central Maven repository. All languages mentioned in this paper has been successfully tested in YAJCo and they serve as a proof of concept for the proposed method.

[1]https://code.google.com/p/yajco/

Acknowledgment

This work was supported by VEGA Grant No. 1/0305/11 Co-evolution of the artifacts written in domain-specific languages driven by language evolution.

References

[1] A. Kleppe, *Software Language Engineering: Creating Domain-Specific Languages Using Metamodels*, 1st ed. Addison-Wesley Professional, 2008.

[2] M. Mernik, J. Heering, and A. M. Sloane, "When and how to develop domain-specific languages," *ACM Comput. Surv.*, vol. 37, no. 4, pp. 316–344, Dec. 2005. [Online]. Available: http://doi.acm.org/10.1145/1118890.1118892

[3] M. Tofte, *Compiler generators: what they can do, what they might do, and what they will probably never do*. New York, NY, USA: Springer-Verlag New York, Inc., 1990.

[4] S. C. Johnson, *Yacc: Yet another compiler-compiler*. Bell Laboratories Murray Hill, NJ, 1975, vol. 32.

[5] T. J. Parr and R. W. Quong, "Antlr: a predicated-ll(k) parser generator," *Softw. Pract. Exper.*, vol. 25, no. 7, pp. 789–810, Jul. 1995. [Online]. Available: http://dx.doi.org/10.1002/spe.4380250705

[6] A. Demenchuk, "Beaver-a lalr parser generator," 2006.

[7] M. Mernik, N. Korbar, and V. Žumer, "Lisa: a tool for automatic language implementation," *SIGPLAN Not.*, vol. 30, no. 4, pp. 71–79, Apr. 1995. [Online]. Available: http://doi.acm.org/10.1145/202176.202185

[8] M. Mernik, M. Lenič, E. Avdičaušević, and V. Žumer, "Lisa: An interactive environment for programming language development," in *Compiler Construction*. Springer, 2002, pp. 1–4.

[9] T. Ekman, G. Hedin, and E. Magnusson, "Jastadd," 2008.

[10] M. Eysholdt and H. Behrens, "Xtext: implement your language faster than the quick and dirty way," in *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, ser. SPLASH '10. New York, NY, USA: ACM, 2010, pp. 307–309. [Online]. Available: http://doi.acm.org/10.1145/1869542.1869625

[11] "Xtext @ONLINE," http://www.eclipse.org/Xtext/, Jun. 2013.

[12] J. Paakki, "Attribute grammar paradigmsa high-level methodology in language implementation," *ACM Comput. Surv.*, vol. 27, no. 2, pp. 196–255, Jun. 1995. [Online]. Available: http://doi.acm.org/10.1145/210376.197409

[13] G. Hedin, "Reference attributed grammars," 1999.

[14] T. Ekman and G. Hedin, "Rewritable reference attributed grammars," in *ECOOP 2004–Object-Oriented Programming*. Springer, 2004, pp. 147–171.

[15] G. Hedin, "An introductory tutorial on jastadd attribute grammars," in *Generative and Transformational Techniques in Software Engineering III*. Springer, 2011, pp. 166–200.

[16] J. Porubän, M. Forgáč, and M. Sabo, "Annotation based parser generator," in *Computer Science and Information Technology, 2009. IMCSIT '09. International Multiconference on*, Oct., pp. 707–714.

[17] J. Rumbaugh, I. Jacobson, and G. Booch, *Unified Modeling Language Reference Manual*, 2nd ed. Addison-Wesley Professional, 2010.

[18] M. Fowler, "Inversion of control containers and the dependency injection pattern, jan. 2004," *URL: http://martinfowler. com/articles/injection. html*.

[19] I. R. Forman, N. Forman, D. J. V. Ibm, I. R. Forman, and N. Forman, "Java reflection in action," 2004.

[20] G. Nicol, *Flex: the lexical scanner generator*. Free Software Foundation, 1993.

[21] V. Kodaganallur, "Incorporating language processing into java applications: A javacc tutorial," *Software, IEEE*, vol. 21, no. 4, pp. 70–77, 2004.

[22] T. Parr, *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Bookshelf, 2007.

[23] J. Clark, S. DeRose *et al.*, "Xml path language (xpath) version 1.0," 1999.

[24] C.-Y. Chan, P. Felber, M. Garofalakis, and R. Rastogi, "Efficient filtering of xml documents with xpath expressions," *The VLDB Journal*, vol. 11, no. 4, pp. 354–379, 2002.