Prototyping Domain Specific Languages as Extensions of a General Purpose Language

Andreas Blunk and Joachim Fischer

Humboldt-Universität zu Berlin
Unter den Linden 6
D-10099 Berlin, Germany
{blunk,fischer}@informatik.hu-berlin.de

Abstract. Domain Specific Languages (DSLs) often consist of general constructs alongside domain-specific ones. A prominent example is a state machine consisting of states and transitions as well as expressions and statements. Adding general concepts to a DSL is a complex and time-consuming task. We propose an approach to develop such DSLs as extensions of a General Purpose Language (GPL). We believe that this approach significantly reduces development times. This is especially important in the first phases of DSL development when language constructs are evolving and not well conceived. Our development allows trying out different forms of constructs with an editor to be at hand at all times. The paper presents first results of the implementation of our approach on top of Eclipse. The feasibility is shown by applying it to the definition of state machines as an example DSL.

1 Introduction

General Purpose Languages (GPLs) are designed to be used in many different application domains. Their language constructs are universally applicable and not limited to a specific domain. In contrast, Domain Specific Languages (DSLs) include constructs created for a specific domain.

DSLs are divided into internal and external ones [1]. An internal DSL is represented within the syntax of a host GPL. Models expressed in the DSL are valid programs of the host language. They use the host language syntax in a stylised way for modelling within a given domain. Their advantage is that tools are already available. However, the representation of models in GPL syntax hinders their creation and their understanding.

In contrast, external DSLs are represented by a custom syntax. These DSLs are of special interest to us because models expressed in an external DSL are easier to create and easier to understand. In addition, DSLs often need to include general constructs known from GPLs, e.g., expressions and statements. Developing such DSLs is a difficult task today. We propose to tackle this problem by an approach based on extending a GPL with new domain-specific constructs. These constructs can be composed of and also used jointly with GPL constructs.

Ø. Haugen, R. Reed, and R. Gotzhein (Eds.): SAM 2012, LNCS 7744, pp. 72–87, 2013.

Our experience shows that the constructs of a DSL are usually not well conceived in the first place. DSLs are created in a number of iterations by talking to domain experts and letting them use the different stages of a DSL. This requires a rapid development process with tools available at all times.

Our approach supports the definition of new syntactic forms for different kinds of GPL constructs. It allows to refer to GPL constructs and to reuse them in DSL constructs. There is instant editor support with syntax checks and content assistance. When an extension definition is written, it is processed at runtime and added to the GPL. The editor instantly supports the syntactic forms of all extensions. We believe that our approach simplifies prototyping DSLs, which make use of general constructs.

In this paper we present the first results of the implementation of our approach. Until now, we have only implemented the possibility of syntactically extending a simple GPL. This GPL we refer to as the Base Language (BL). It includes a reasonably small set of well-known object-oriented language primitives. We plan to describe the semantics of extensions as a mapping to the BL in the future. In this paper, we only describe syntactic extensions of the BL. In this area, we demonstrate a successful application for the definition of a State Machine DSL. This is a neat example because state machines require general constructs like expressions and statements in their definition. A strong point in our approach is that the BL editor is able to provide instant content assistance for extensions. This is an exceptional feature amongst the existing DSL development frameworks. It can result in reduced development times because a DSL can instantly be applied to a problem at hand.

The remainder of the paper is structured as follows. Section 2 presents related approaches and describes their main deficiencies. The BL is introduced in Sect. 3. We summarise its main concepts and describe its definition. These explanations lay the foundation for the extension mechanism presented in Sect. 4. We detail our approach in a general way and give examples for the extension of the BL's syntax. Section 4 concludes with the implementation of an editor capable of instantly supporting extensions. In Sect. 5, we present an example of a definition of a State Machine DSL. The paper concludes in Sect. 6, followed by an outline of future development interests in Sect. 7.

2 Related Work

There are two prevailing groups of approaches to develop external DSLs. One group relies on pre-processor-based extensions to a GPL. These extensions are defined in a separate pre-processor language or in the GPL itself. Their instances are substituted by GPL code before models are executed. Usually, there is insufficient tool support with these approaches. Several representatives are compared with each other in [2].

The other group relies on metamodel-based techniques. Here, the constructs of a GPL are imported into the definition of the abstract or the concrete syntax of a DSL. In a separate step, DSL-aware tools, like an editor, are generated in an automatic way.

In this section we review a selection of each approach's representatives. We describe the general idea of each approach first. After this, we describe each approach's major deficiencies regarding the definition of complex DSLs. These are DSLs whose constructs consist of or refer to other GPL or DSL constructs. An example is a State Machine DSL. It contains a transition construct for defining a transition from one state to another state. Amongst other things, the transition construct consists of action statements (GPL constructs) and a reference to a target state (a DSL construct).

2.1 Pre-processor-Based Approaches

The group of pre-processor based approaches shares a number of deficiencies: (1) there is none or insufficient editor support for extensions, and (2) the syntax extension capabilities are too restrictive for defining more complex DSL constructs. This includes missing description means for defining references between DSL constructs.

The Java Syntactic Extender (JSE) [3] is a pre-processor for the Java programming language. Extensions are limited to a few shapes with partially predefined syntax: function call macros and statement macros. For example, they have to begin with a name and have to end in a predefined way. Extensions can be composed of GPL constructs, but they cannot define references to GPL constructs defined elsewhere. JSE is more powerful compared to the very simplistic macro mechanism in the C language, because the type of GPL construct to be included can be defined, e.g., a for-each statement that is composed of expressions and statements.

Camlp4 [4] is a pre-processor for the multi-paradigm language Ocaml. In contrast to JSE, it performs extensions to GPL transformations by acting on the Ocaml abstract syntax tree (AST). It allows to augment the Ocaml grammar by new rules and to modify or delete existing ones. Because of its AST-based foundation, even extensions of expressions with regard to precedence and associativity are possible. A restriction is imposed by Ocamls type of grammar definition. Extensions have to be parseable by a recursive descent parser.

In the field of simulation modelling, there is a GPL with special simulation constructs called Simulation Language with Extensibility (SLX) [5]. Its extension capabilities are situated on the level of regular expressions. Compared to JSE and Camlp4, SLX is less powerful. The parts of an extension are plain strings, which do not reference GPL constructs. However, there is basic syntax assistance in SLX. The syntax of extensions gets highlighted after each successful compilation step.

2.2 Metamodel-Based Approaches

A prominent representative of the group of metamodel-based approaches is Xtext [6]. In Xtext, DSLs are defined by specifying their constructs in a concrete syntax, which ultimately is an EBNF-like grammar. From this grammar an abstract syntax description in the form of an object-oriented metamodel is

derived. The metamodel is used for further processing of DSL instances, e.g., for specifying additional constraints and for defining execution semantics. Another artefact which can be generated from the grammar is a text editor with support for syntax highlighting and content assistance.

Xtext also provides GPL-like constructs in a language called Xbase. Its constructs, e.g. expressions, can be included into a DSL grammar. However, GPL constructs cannot be extended by new ones, e.g. one cannot add a new type of expression useable jointly with all other expressions.

In addition, Xbase is limited to expressions in the form of operators and statements. There are no means for defining functional or structural abstractions. Each DSL which needs to include them has to define them again.

A more powerful approach than Xtext regarding the extension of a GPL is followed by the Meta Programming System (MPS) [7]. In MPS, the definition of a DSL begins with the specification of an abstract syntax in the form of a metamodel. Then for each construct a concrete syntax is defined as a projection to text. In the modelling process one does not write text but instantiates structures defined by the metamodel. These structures are represented in their text form.

As a result, there is no parsing of text necessary. However, the DSL editor is unusual to operate because one cannot enter the single characters forming a certain DSL construct directly. Instead, one has to choose from a set of possible constructs insertable at the current cursor position. After an option is chosen, the fixed textual parts of the construct are expanded and the cursor can be moved from one variable text fragment to the next. For example, when a class is to be added, the first fixed part is the `class` keyword followed by the name of the class as a variable part. Other variable parts are for example super classes and attributes.

In MPS, a GPL called BaseLanguage can be used to extend or to include general constructs into a DSL. Different kinds of GPL constructs can be extended. In addition, defining name-based references to DSL as well as GPL constructs is supported.

Xtext and MPS are able to generate a DSL editor from a DSL description. However, the process of developing a DSL is complex. One has to master an interactive framework in order to define the different aspects of a DSL at the appropriate places. In addition, the editor is not immediately available. Its software has to be generated and compiled first before the editor can be used. The generation step has to be initiated every time the language definition has changed. This manual step hinders rapid prototyping of DSLs.

3 Base Language

The Base Language (BL) is a simplified GPL. It provides a set of object-oriented modelling constructs. The BL allows to solve a problem on a general level first. For similar problems, a library can be developed in the traditional way by using abstractions of the BL. In case other syntactic forms are needed, the BL can

be extended. New constructs can be added but existing constructs cannot be changed or deleted. Extensions can add expressiveness but they cannot mutate the BL into something completely different.

We use a subset of the Java programming language as the BL. It has a similar syntax and semantics. The following paragraphs give a short overview of the BL.

3.1 Basic Concepts

The basic modelling concepts are: classes, interfaces, procedures, variables, statements, and expressions. Classes and interfaces are used to define structural and operational parts of objects. Variables of these types have reference semantics. The primitive types are: int, double, boolean, string, and void. Variables of these types have value semantics. There are two collection types: (1) list for ordered and unique collections, and (2) sequence for ordered and non-unique ones. For each of them, a number of list operations is predefined: first-item, last-item, contains-item, index-of-item, item-at-index, before-item, after-item, and size-of-list.

Procedures can be defined in a global way and as methods of classes and interfaces. Variables can be defined global, as attributes of classes, as parameters of procedures, and as local variables inside procedures.

The following kinds of statements exist: local-variable-declaration, assignment, procedure-call, if-then-else, while, for-each, add-to-list, remove-from-list, clear-list, print, and return. There are expressions for logical computation (and, or), number comparisons (greater, less, equals), mathematical computation (e.g. plus, and minus), literals (e.g., 2, true, 3.2, "abc", null), object creation (new), object type related operations (cast, instance-of), element access by punctuation (e.g. xlist.first.getName()), and some predefined element accessors (self and super).

More advanced modelling concepts of Java, e.g., exceptions, threads, packages, and visibility, are not available.

3.2 Syntax Definition

It is important to understand the way the syntax of the BL is defined. This is essential for the definition of extensions, since these are defined by directly referring to constructs defined in the syntax of the BL.

There are two ways to refer to BL constructs. Extensions can add new forms of BL constructs, e.g. they can add a new kind of statement by referring to the BL statement. In addition, extensions can be composed of BL constructs, e.g., a for-statement defined as an extension can be composed of expressions and statements. The next paragraph gives an overview of the techniques used for defining the syntax.

The syntax of the BL consists of an abstract and a concrete syntax. The abstract syntax is defined by an object-oriented metamodel. It consists of classes and attributes. These classes are referred to as metaclasses in order to distinguish them from the classes in a language instance, whose structure they define.

```
extension ExtensionName {
   BaseLanguage_Rule -> Extension_Rule;
   Extension_Rule -> ... ;
   ...
}
```

Listing 1.1. General structure of an extension definition.

As an example, the metaclass `Clazz` defines the possible structure of class definitions in BL models. The metaclass contains attributes like a name and a list of `Variables`. Definitions of BL classes are instances of the metaclass `Clazz`. Each definition assigns specific values to attributes of metaclass instances. For example, the name of a BL class is assigned as a value of the `name` attribute. Attributes defined in a BL class become references to instances of the metaclass `Variable`.

By using metamodels, the structure of language concepts can be defined in an abstract way first. Then, a concrete representation is added by referring to classes and attributes in the metamodel. The metamodel of the BL is defined by using the Eclipse Modeling Framework (EMF) [8]. Its concrete syntax is defined in the Textual Syntax Language (TSL).

TSL is part of the Textual Editing Framework (TEF) [9]. It allows to define a concrete syntax as an attributed EBNF-like grammar. TSL definitions consist of rules, terminals, and non-terminals known from EBNF. In addition, TSL constructs have to be annotated by references to meta-classes and meta-attributes. These annotations define a mapping of the concrete textual representation to an instance of the metamodel.

A LALR parser is generated from a TSL description by using the parser generator RunCC [10] which is able to generate a parser at runtime. Additional code generation (including compilation) is not necessary to invoke the parser on an input stream. The implementation of our approach makes use of this runtime generation feature.

4 Extension Definition

The general structure of an extension definition is shown in Listing 1.1. Extensions can add new constructs, but they cannot redefine or delete existing ones. Hence, the modeller is assured that the basic concepts do not change in their meaning.

4.1 Syntax Definition

The syntax is defined in an attributed BNF-like description language, which is similar to TSL, but has a different syntax. It also provides some semantic additions. This language is named Simple Textual Syntax Language (STSL). It is tightly integrated into the extension concept of the BL.

A syntax definition in STSL consists of a set of rules by which the BL grammar is extended. Since the used parsing technique is LALR, conflicts can arise when adding new rules, e.g., shift/reduce conflicts. These conflicts are reported to the DSL developer, who has to correct them by changing the syntax description.

The first rule in a STSL description is special. It specifies which BL grammar rule (left side) is extended by a new extension rule (right side). The next step is the specification of the parts of this new extension rule and all subsequent rules. Each rule consists of terminals and non-terminals. Terminals can be a fixed sequence of characters or one of the predefined lexical tokens identifier (`ID`), integer number (`INT`), and string (`STRING`). Non-terminals are references to other rules.

In the description of other language aspects, e.g., semantics, the syntactical parts of an extension need to be accessed. Therefore, a mapping of the concrete syntax to an abstract syntax is defined. The mapping is realised in two stages. First, syntax pieces are prefixed by symbolic names, which allows to access their elementary or structured value. Second, a corresponding meta-class is created for each prefixed non-terminal, if one does not already exist. Already existing meta-classes are BL classes like `Statement` and `Expression`.

The structure, which is accessible by such a description, is an object tree. The nodes in this tree are the named non-terminals. Each type of non-terminal is an object described by a meta-class. The attribute structure of each such meta-class is defined by all the right-hand sides of rules, which have the same left-hand side. All the named parts on a right-hand side become attributes of the meta-class, which is named by the left-hand side. Attribute values are either references to other object nodes (for non-terminals) or elementary ones (for tokens).

As an example, an excerpt of an extension is shown in Listing 1.2. An `A` construct begins with the terminal `foo` (whose value is not accessible). An integer number must follow, which is assigned as value of an attribute named `n`. After the number, a `B` construct follows. The structure of `B` is not accessible. At the end, a `C` construct must be supplied. The structure of `C` can be accessed by the name `c`. From `c`, one can also navigate to structural parts of `C`, e.g., `c.d` refers to another number.

Two new meta-classes are created for the abstract syntax. The meta-class `A` is created for the rule `A`. It is defined as a sub-class of the BL rule meta-class `BaseRule`. `A` defines the following list of attributes: `n` of type `Integer`, `s` of type `String`, and `c` of type `C`. Note that there is no meta-class created for the non-terminal `B` because it is not prefixed by an attribute. The other meta-class is `C`. It defines an attribute `d` of type `Integer`.

The class structure derived from such a concrete syntax is an object-oriented description of the abstract syntax. This kind of structure is also known as a metamodel. Usually, a metamodel is the initial artifact in metamodel-based language development. Here, the metamodel is extracted from a concrete syntax description. It is intended to be used as a representation for specifying further processing of extensions, e.g. in the definition of an execution semantics.

```
BaseRule -> A;
A -> "foo" n:INT B c:C ;
B -> s:STRING;
C -> d:INT;
```

Listing 1.2. Excerpt of an example extension.

```
extension For {
   Statement -> ForStm ;
   ForStm -> "for" "(" variable:$Variable "=" value:Expression ";"
       condition:Expression ";" incStm:Assignment ")" "{"
          MultipleStatements
       "}";
   MultipleStatements -> ;
   MultipleStatements -> statements:Statement MultipleStatements;
}
```

Listing 1.3. A for-loop defined as an example extension.

A first realistic example is given in Listing 1.3. The extension defines a for-statement as an additional type of statement. A new same-named meta-class is created for the rule `ForStm`. The `ForStm` meta-class inherits from the meta-class `Statement`. It defines the attributes `variable`, `value`, `condition`, `incStm`, and `statements`. The structure of their values is further described by other meta-classes, e.g., `Variable`, and `Expression`.

The `Variable` non-terminal is notably different. It is prefixed by a dollar sign, which designates the non-terminal as a reference to an already existent object. So in the case of the non-terminal `$Variable`, an identifier referring to the name of an already existent `Variable` object has to be supplied. In case of BL constructs, e.g, variables, and procedures, the identifier can be a single name or a qualified one. Resolution of qualified identifiers is defined for the BL, but it cannot be specified for named extensions. If an extension element needs to be referenced, resolution is done on a global level, i.e., names of extensions have to be globally unique in order to refer to them. Global references are distinguished by using two dollar signs, e.g., `$$Variable` is used to refer to a `Variable` object by a single name only.

4.2 Kinds of Extensions

In this section, we present and discuss the most obvious kinds of extensions that seem to make sense. The discussion is only concerned with syntax here. Limitations of the approach are presented in the subsequent Section.

Statements. A kind of extension which immediately comes to ones mind is the introduction of new statements. For statement extensions, the BL grammar rule `Statement` is extended.

```
extension StateMachine {
    ClassContentExtension -> StateMachine;
    StateMachine -> "stateMachine" name:ID "{" StateListOptional "}";
    ...
}
```

Listing 1.4. Beginning of the definition of a state machine as an extension.

Statement extensions can only be used at certain places where a BL statement is allowed, e.g., in the body of a procedure or in structured statements. They either terminate with a semicolon, e.g., the print statement, or with curly braces, e.g., the while statement. Extensions should follow this style, but they are not forced to. It is possible to define another ending symbol or no ending symbol at all. Nevertheless, conflicts may occur for some combinations.

For example, in an extension with rules `Statement -> S1` and `S1 -> "s1" exp:Expression ">"` there is a shift/reduce conflict, because the final symbol > can be a part of an expression as well.

Another characteristic of statements is that they usually start with a keyword. But they can also begin with other kinds of tokens, e.g., with an identifier, an integer number, or an expression. It is also allowed to reuse keywords as long as there is some distinguishable part in the new grammar rule.

For example, the BL includes a for-each statement, which begins with the keyword for. In Listing 1.3, we added a traditional number-based for-loop. This extension is feasible and not in conflict with the for-each statement. Both statements reference a variable after the opening parenthesis of the for keyword. However, the for-each statement is followed by a colon, while the for-loop statement is followed by an equals sign.

Embedded. Extensions with a more declarative nature are those embedded into modules or classes. These are places where classes, variables, and procedures are defined. Extensions embeddable into modules have to extend the rule `Module ContentExtension`. For classes, the rule `ClassContentExtension` has to be extended.

An example for such an extension is the definition of a state machine inside a class, which specifies the behaviour of the objects of that class. The beginning of such an extension is shown in Listing 1.4.

Expressions. Extending expressions is more complicated, because of operator precedence rules. There are 8 priority classes defined in the BL. Table 1 gives an overview of these classes and the grammar rules that they are defined by.

An expression extension begins with a reference to a priority class rule on its left side. The right side is consists of other expression priority classes and terminals.

```
extension PreInc {
   L2Expr -> PreInc;
   PreInc -> "++" left:L1Expr;
}
```

Listing 1.5. Definition of a pre-increment expression.

```
extension Ternary {
   L9Expr -> Ternary;
   Ternary -> cond:L8Expr "?" trueCase:L8Expr ":" falseCase:L9Expr;
}
```

Listing 1.6. Definition of a ternary if-else operator.

As an example, Listing 1.5 shows the definition of the unary operation pre-increment. It has the same priority as an unary plus and an unary minus. Therefore, it extends the rule `L2Expr`. A same or lower priority expression must be provided on its right side.

Table 1. Operator precedence in BL expressions.

Priority	Operators	Operations	BL rule
1	. ()	member access, procedure call	L1Expr
2	+ - !	unary plus, minus, negation	L2Expr
3	* / %	multiplicative	L3Expr
4	+ -	additive	L4Expr
5	< > <= >= instanceof	relational	L5Expr
6	== !=	equality	L6Expr
7	and	logical and	L7Expr
8	or	logical or	L8Expr

Adding additional priority classes is currently not supported. It would be required to allow the redefinition of existing rules in order to insert a new priority class rule, which is not allowed in this setup. For example, a level 9 priority class is needed for the definition of a ternary if-else operator. It is necessary to redefine the rule `Expression -> L8Expr` to `Expression -> L9Expr` and to add the rule `L9Expr -> L8Expr`. Supporting priority class insertion in future versions should be possible. Then, the ternary if-else operator could be defined as shown in Listing 1.6.

4.3 Limitations

Extensions are limited to certain kinds of BL rules. In the BL grammar there are basically two kinds of rules: (1) *assigned rules*, and (2) *unassigned rules*.

An *assigned rule* is always used in connection with an attribute in some other rule. After an *assigned rule* is successfully parsed, an object of a same-named meta-class is created. This object is assigned to the attribute of another object corresponding to another rule from which the *assigned rule* was called. In contrast, an *unassigned rule* is used without an attribute. It solely defines one or more simple reductions to other rules. Extensions can only be defined for *assigned rules* because the objects created for an extension instance have to be held in an attribute of the abstract syntax graph.

For example in the grammar `A -> b:B C; B -> "b"; C -> "c";`, there is an *assigned rule* named `B` and *unassigned rule* named `C`. An instance of `B` is assigned to the attribute `b` of an `A` object. In contrast, an instance of `C` cannot be assigned to an attribute of an `A` object. So the rule `C` cannot be extended.

Some of the *unassigned rules* are specially prepared for extension. For example, there is the *unassigned rule* `ClassContent`, which is used inside the rule `Clazz`. In order to allow extending the content area of a class, another rule `ClassContentExtension` and a corresponding meta-class are defined. Furthermore, an attribute `extensions` of type `ClassContentExtension` is added to the meta-class `Clazz`.

4.4 Difficulties

A major difficulty results from the use of the Eclipse Modeling Framework (EMF). EMF expects a metamodel to be complete and not changing when instances of its meta-classes are created. The problematic part is the generation of Java code for an EMF metamodel. For each meta-class a corresponding Java class is generated. At runtime, the instances of a metamodel are internally represented as objects of the generated Java classes.

However, in the case of extensions it is complicated to generate and compile these Java classes, and make them useable inside a running Eclipse. As a workaround, the Java class corresponding to an extended BL rule is instantiated instead. For example, in the case of statement extensions, there exists a meta-class as well as a Java class with the name `Statement`. In the workaround, an instance of a statement extension, e.g. the for-loop defined in Listing 1.3, is internally represented as a Java object of type `Statement`. In the next step, the respective meta-class of the Java object is set to the special meta-class of the extension rule. For example, in the case of the for-loop the meta-class is set to `ForStm`. Attributes defined by the extension meta-class are still accessible by using EMF's reflection mechanism. It allows to access an attribute by using generic get and set methods instead of generated ones.

4.5 Editor Implementation

The editor is instantly aware of all defined extensions. An additional software generation step is not needed in order to use the editor. It supports syntax highlighting and content assistance. Its implementation is based on a parser, which is extensible at runtime. Extension definitions are instantly recognised by

the BL editor. For each extension, the grammar rules defined by an extension are added to the grammar of the BL. The extended BL editor and its parser continue to work with the extended version of the grammar. When an extension definition is modified, the corresponding rules in the BL grammar are updated as well.

The extensible BL editor is implemented by using the Textual Editing Framework (TEF) and the Eclipse Modeling Framework (EMF) [8]. TEF is used for the definition of the BL concrete syntax and for the BL editor. EMF is used for the definition of a metamodel for the BL, which is required by TEF for describing notations. Implementing an extensible version of TEF is feasible since TEF's implementation is based on a runtime parser generator, called RunCC. It can generate parsers of type LALR at runtime.

5 State Machines as an Example

State machines [11] provide a fair level of abstraction when modelling the behaviour of stateful objects. A DSL for creating state machines provides the necessary modelling constructs in the vocabulary of the domain. In this case, the domain is specifying behaviour in a special way. The major constructs of this domain are states, transitions, and events. In addition, general constructs are necessary for optionally specifying the condition under which an event may take place and for defining actions to be taken when an event occurs. Here, expressions and statements known from GPLs are good abstractions.

State machines are domain-specific in terms of the way they allow to model behaviour. However, they are general-purpose in the sense that they can be used to specify behaviour as a part of modelling in many different domains. For example, they can be used to specify telecommunication protocols as in the Specification and Description Language (SDL) [12]. Another application domain, which we are particularly interested in, is there usage for defining workflows in manufacturing systems.

An example of a simple state machine is depicted in Fig. 1. There are four states: an initial state, the states `A` and `B`, and a final state. State transitions take place when events occur. For example, when a `Start` event occurs in state `A`, the state machine transitions to state `B`. For each state transition, actions are specified after the slash symbol `/`. In addition, a guard condition is defined by placing an expression of type boolean in square brackets `[ ]` after an event. For example, when a `Tick` event occurs in state `B` and the condition `i >= 3` is satisified, then the next state will be the final state.

On the one hand, state machines only contain a small set of constructs. On the other hand, they contain general constructs which makes their definition difficult. In SDL, state machines are defined as an external DSL. This way, models can be expressed in a custom syntax which improves understandability. However, handcrafting the necessary tools is very time consuming.

State machines can also be defined as an internal DSL, e.g., as a library or framework in a GPL like Java. The library defines structural and functional

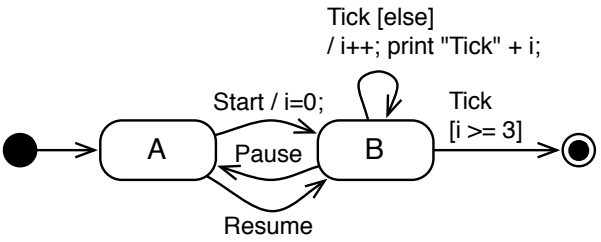

Fig. 1. Example state machine

abstractions which have to be used in a certain way in order to create domain-specific models. The expressiveness of internal DSLs is limited by the abstraction means offered by the underlying GPL. The basic modelling constructs of the GPL cannot be changed. In addition, domain-specific models have to be represented in the syntax of the GPL. This makes understanding the model more difficult as opposed to representations specifically created for a certain domain.

In order to use a state machine framework, one has to know how to apply its structural and functional abstractions in the right way. The state machine itself gets encoded by an unsuitable representation. It is an advantage that an editor is available and that the Java compiler can be used to execute state machines. In addition, Java itself offers necessary general constructs like expressions and statements. However, creating and understanding state machines becomes more complicated.

In our approach, the State Machine DSL is defined as an extension of the BL. In Listing 1.7, the syntax definition of a state machine extension is shown. State machines define the behaviour of stateful objects. Therefore, a good place for their definition is within a class. The definition begins with the keyword `stateMachine` followed by a name, a set of events, and the definition of states. An example for its use is depicted in Fig. 2. The definition of a custom syntax helps creating and understanding state machines. In addition, there is instant editor support.

6 Conclusion

We presented an approach that supports the syntactic extension of a simple GPL by domain-specific constructs. These constructs can be composed of or refer to GPL constructs themselves. Extensions are recognised by the GPL editor, which instantly provides content assistance for them. We believe that such a feature simplifies the development of DSLs and reduces development times in the first phase. In addition, the editor is a usual text editor which can be operated in a familiar way.

```
extension StateMachine {
    ClassContentExtension -> StateMachine;

    StateMachine -> "stateMachine" name:ID "{"
            EventDeclarations initialState:InitialState
                StateListOptional "}";

    EventDeclarations -> "events" ":" EventDeclList ";";
    EventDeclList -> events:EventDecl EventDeclListOptional;
    EventDeclListOptional -> ;
    EventDeclListOptional -> "," EventDeclList;
    EventDecl -> name:ID;

    StateListOptional -> ;
    StateListOptional -> states:Vertex StateListOptional;

    Vertex -> State;
    Vertex -> EndState;

    InitialState -> "initial" "->" target:$$Vertex ";";
    State -> "state" name:ID TransitionsOptional ";";
    EndState -> "end" name:ID ";";

    TransitionsOptional -> ;
    TransitionsOptional -> "(" OutgoingList ")";

    OutgoingList -> outgoing:Transition OutgoingListOptional;
    OutgoingListOptional -> ;
    OutgoingListOptional -> "," OutgoingList;

    Transition -> event:$$EventDecl GuardOptional EffectsOptional
        TargetStateOptional;

    GuardOptional -> ;
    GuardOptional -> "[" condition:Expression "]";

    EffectsOptional -> ;
    EffectsOptional -> "/" effect:Effect;
    Effect -> oneLine:Statement;
    Effect -> multiLine:CodeBlock;
    TargetStateOptional -> ;
    TargetStateOptional -> "->" target:$$Vertex;
}
extension ElseGuardExpr {
    L1Expr -> ElseGuardExpr;
    ElseGuardExpr -> "else";
}
```

Listing 1.7. State Machine DSL defined as BL extension.

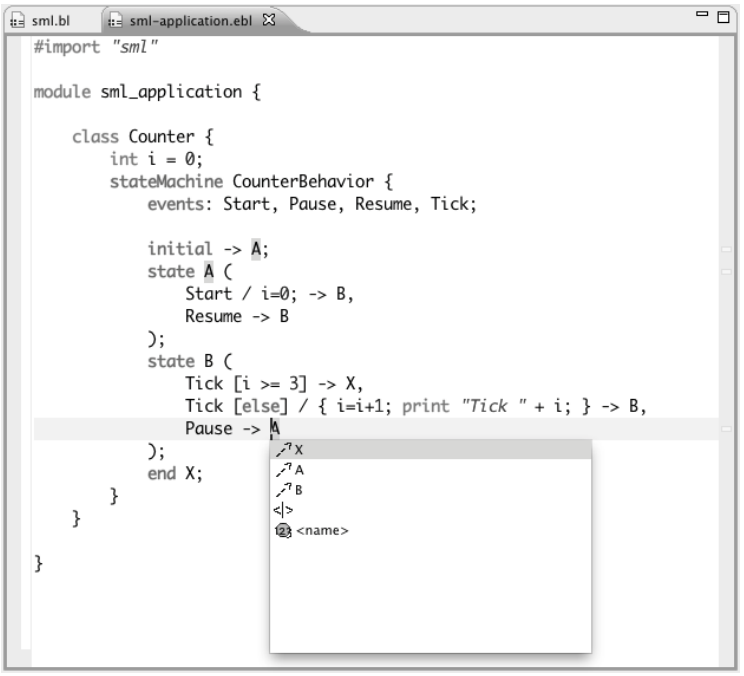

Fig. 2. Example state machine.

7 Future Work

Our next step will be to provide a description for the semantics of extensions. We plan to support this by a mapping to the BL. When an extended model is to be executed, extensions are translated to BL constructs first. Then the BL model is translated to an executable target language. Finally, the program in the target language is executed.

Limitations imposed by the semantics have to be investigated as well. We only tested the approach with respect to syntax extensions. This was done for a number of small example extensions including the presented simple State Machine DSL. In future, we plan to conduct a larger case study by applying the approach to a more powerful State Machine DSL. We intend to use this DSL for modelling the behaviour of manufacturing systems. Description means for specifying time passage and state changes based on conditions are necessary modelling constructs to be included.

Another aspect is language composability. It could be possible to combine several DSLs into one. In principle, the extension mechanism supports such modular DSL development. However, further investigation of this aspect is needed.

Beyond that, there is also room for improving the usability of extensions. One such aspect is debugger support. We intend to examine how a DSL-aware

debugger can be provided. We already gained experience on automatically deriving DSL debuggers in [13].

An aspect which was not paid much attention to is identifier resolution. When a DSL gets more complex, it may include the concept of a namespace. In this case, DSL constructs cannot be referred to by a globally unique identifier anymore. Instead, identifiers are structured and a context-dependent resolution algorithm has to be described. To our best knowledge, this is always done in a GPL. However, we already identified patterns in these descriptions and we believe that identifier resolution can be described in a more concise way by using an appropriate DSL. We intend to create such a DSL using the extension approach presented in this paper.

References

1. Fowler, M.: Domain-Specific Languages. Addison Wesley (2011)
2. Zingaro, D.: Modern Extensible Languages. Technical report, McMaster University, Hamilton, Ontario, Canada (2007)
3. Bachrach, J., Playford, K.: The Java Syntactic Extender (JSE). In: Proceedings of the 16th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2001, pp. 31–42. ACM Press (2001)
4. de Rauglaudre, D.: Camlp4, http://caml.inria.fr/pub/old_caml_site/camlp4
5. Henriksen, J.O.: SLX - The X is for Extensibility. In: Proceedings of the 32nd Conference on Winter Simulation, WSC 2000, vol. 1, pp. 183–190. Society for Computer Simulation International (2000)
6. Xtext: Xtext Documentation, http://www.eclipse.org/Xtext/documentation/
7. JetBrains: Meta Programming System (MPS), http://www.jetbrains.com/mps/
8. EMF: Eclipse Modeling Framework (EMF), http://www.eclipse.org/modeling/emf
9. Scheidgen, M.: Integrating Content Assist into Textual Modelling Editors. In: Modellierung. Lecture Notes in Informatics, vol. 127, pp. 121–131. Gesellschaft fr Informatik E.V. (2008)
10. Ritzberger, F.: RunCC - Java Runtime Compiler Compiler, http://runcc.sourceforge.net/
11. Harel, D.: Statechars - A Visual Formalism for Complex Systems. Science of Computer Programming 8(3), 231–274 (1987)
12. International Telecommunication Union (ITU): Recommendation Z.100, Specification and Description Language - Overview of SDL 2010 (2010), http://www.itu.int/rec/T-REC-Z.100/en
13. Blunk, A., Fischer, J., Sadilek, D.A.: Modelling a Debugger for an Imperative Voice Control Language. In: Reed, R., Bilgic, A., Gotzhein, R. (eds.) SDL 2009. LNCS, vol. 5719, pp. 149–164. Springer, Heidelberg (2009)