Metamodel-Aware Textual Concrete Syntax Specification

Frédéric Fondement [1]

École Polytechnique Fédérale de Lausanne (EPFL)
Lausanne, Switzerland

Rémi Schnekenburger, Sébastien Gérard [2,3]

Laboratoire d'Intégration des Systèmes et des Technologies (LIST)
Commissariat à l'Énergie Atomique (CEA)
Saclay, France

Pierre-Alain Muller [4]

Institut National de Recherche en Informatique et en Automatique (INRIA)
Rennes, France

Abstract

Metamodeling is raising more and more interest in the field of language engineering. While this approach is now well understood for the definition of abstract syntaxes, the formal definition of concrete syntaxes is still a challenge. Concrete syntaxes are traditionally expressed with rules, conforming to EBNF-like grammars, which can be processed by compiler compilers to generate parsers. Unfortunately, these generated parsers produce concrete syntax trees, leaving a gap with the abstract syntax defined by metamodels. This gap is usually filled by time consuming ad-hoc hand-coding. In this paper we propose a new kind of specification for concrete syntaxes that takes advantage of metamodels to generate tools (such as parsers or text generators) which directly manipulate abstract syntax trees. The principle is to map abstract syntaxes to concrete syntaxes via EBNF-like rules that explain how to render an abstract concept into a given concrete syntax, and how to trigger other rules to handle the properties of the concepts. The major difference with EBNF is that rules may have sub-rules, which can be automatically triggered based on the inheritance hierarchy of the abstract syntax concepts.

Key words: abstract concrete syntax mapping, metamodeling, textual concrete syntax, MDE, MDA, language engineering.

1 Introduction

Languages for the definition of metamodels such as MOF [15], model interchange facilities such as XMI [18] (to be seen as XML representation for abstract syntax trees) and tools such as Netbeans MDR [21] or Eclipse EMF [4] can be used for a wide range of purposes, including language engineering. While this approach is now well understood for the definition of abstract syntaxes, the formal definition of concrete syntaxes is still a challenge, even though concrete syntax definition is considered as an important part of metamodeling [1].

Being able to parse a text and transform it into a model, or being able to generate text from a model are concerns that are being paid more and more attention in industry. For instance, Microsoft with the DSL Tools [9] or Xactium with XMF Mosaic [2] in the domain-specific language engineering community, are two industrial solutions for language engineering that involve specifications used for the generation of tools such as parsers and editors. A new OMG standard, MOF2Text [16], is also being developed regarding concrete $\leftarrow$ abstract mapping. It is also likely that some other approaches concerning concrete syntax composition from a model, see for instance [14], may be improved to support analysis and synthesis of concrete syntax. However, at this time, we are not aware of a unified formal bidirectional specification of concrete syntax that would allow both concrete $\rightarrow$ abstract and concrete $\leftarrow$ abstract mappings.

In this paper, we explore such a bidirectional mapping by defining a language for the specification of textual concrete syntax in a context where abstract syntax is represented by metamodels. This language will be referred as TCSSL for Textual Concrete Syntax Specification Language. Concrete syntaxes written in TCSSL may be used by compiler compilers to generate text analyzers that produce models as instances of metamodels, instead of merely concrete syntax trees. It should also be possible to generate, again from that concrete syntax specification, pretty printers, IDEs, or even incremental synchronizers that update the textual views representing a model, and symmetrically update a model when the textual representation changes. Fig. 1 summarizes an example of usage of such specification.

Of course, a TCSSL specification dedicated to a given metamodel does not prevent defining another TCSSL specification dedicated to the same metamodel to provide another concrete syntax. One could thus imagine different teams working on the same model using different concrete syntaxes, i.e. textual views of the model, each one adapted to the different stakeholders' needs and preferences. Even though the model is viewed with different clothes, it is

[1] Email: frederic.fondement@epfl.ch
[2] Email: remi.schnekenburger@cea.fr
[3] Email: sebastien.gerard@cea.fr
[4] Email: pierre-alain.muller@irisa.fr

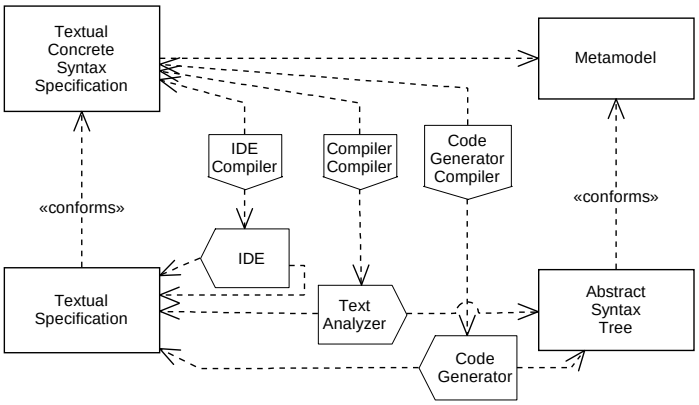


Fig. 1. Usage of a Formal Textual Concrete Syntax Specification Example

still the same model that everyone manipulates.

To achieve such requirements, we propose that each *representable* concept of the metamodel (i.e. each concept that should be possibly rendered using the concrete syntax to be described) is complemented with EBNF-like rules which state the mapping between a model element and a concrete syntax fragment. The major difference with EBNF is that properties of the concept, i.e. relations with other concepts, are interpreted as calls to other rules. In this case, at text analyzing time, the correct rule for the associated concept is triggered and returns a model element that has to be referenced by the property in the model. The text analyzer detects the appropriate rule for a given property by choosing between the rule of the awaited concept (if it exists), and all the rules corresponding to sub-concepts (i.e. rules of inheriting concepts). For instance, if a property references a real, the text analyzer has to trigger either the rule for reals, or the rule for integers, as the integer concept inherits from the real concept. Such behavior can be symmetrically applied for text generation. Of course, such adaptation of EBNF has side effects, and special features had to be added, such as a different system for rule choices or for loops. We will develop these points in the rest of the paper.

In section 2, we will discuss requirements of model-driven textual language engineering and provide a simple example that will be studied in the rest of the paper. Then, in section 3, we will present the concepts of the textual concrete syntax specification language (TCSSL) that we propose. Considerations about complex mapping requiring several passes will be given in section 4. An example of a parser and text generator which implements our approach will be developed in section 5. Finally section 6 draws some general conclusions and outlines future works.

2 Model-Driven Language Engineering

In this section, we introduce language engineering and its relations to model-driven engineering. We also present the basics of a single example, a textual

statechart language, that will be used to illustrate our proposal in the rest of the paper.

Model-driven methodologies [11,12] promote models as first class assets. They also promote iterative development by refinements and refactorings. This implies a number of different kinds of models, thus a number of different metamodels and variations of metamodels. A model is usually built with some purpose in mind. It can be source for (semi-)automatic computations (i.e. model transformations, refinement, refactoring, code generation...), or it can be merely documentary. Even if models are used in an automatic way, they clearly need to be represented in a human-readable way, so that they can be described and understood. This is where concrete syntax comes into play. Because of the number of metamodels and the necessity to edit or visualize models, it is desirable to find how to derive specific IDEs form concrete syntax specification. This paper concentrates on textual concrete syntaxes, although ongoing researches deal with graphical concrete syntaxes (see [3,8]).

To illustrate the requirements of model-driven engineering, one can study a simplified but yet illustrative version of statecharts [10], whose metamodel is shown in Fig. 2. A transition has exactly one source vertex and one target vertex. A vertex is either a pseudo state (initial state, choice, etc.) or a state, which is in turn either a composite state (i.e. containing other vertices and transitions), or a simple state. Transitions may be triggered by events. A state machine is given by its top state. Not shown in Fig. 2 are well-formedness rules that complement the metamodel and stipulate, for example, that an initial pseudo-state can never be the target vertex of any transition.

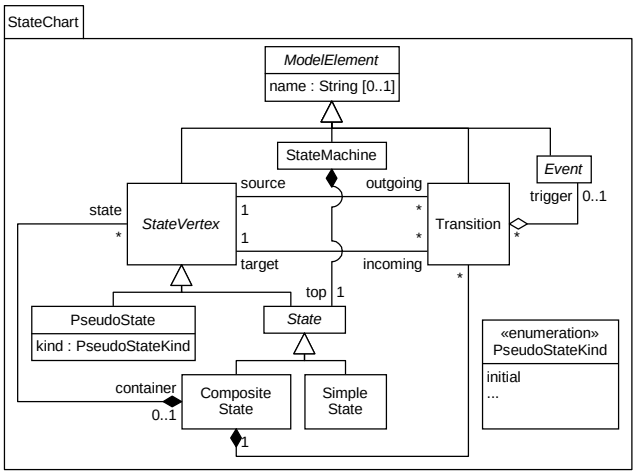
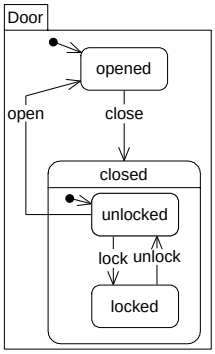


Fig. 2. Simplified State Chart Metamodel

Fig. 3 represents various possible representations for a simple state chart controlling a door. Part *(a)* uses a standard graphical state diagram notation. A door may be opened or closed. If the door is closed, it may be locked or not. A locked door cannot become opened. Part *(b)* represents the same model using a textual syntax that we propose to study below in the paper.

Part *(c)* represents the model as an instance of the metamodel of Fig. 2 (i.e. the abstract syntax) using the MOF object diagram notation. For sake of readability, events have been omitted.

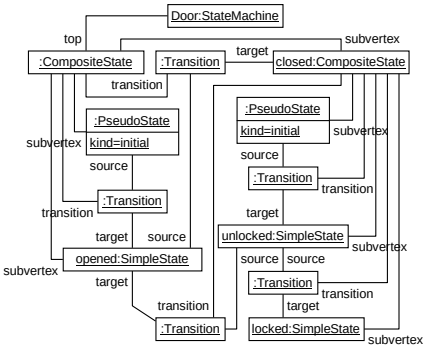


(a) Graphical Notation

```
StateMachine Door

CompositeState {
  initial State opened
  CompositeState closed {
    initial State unlocked
    State locked
    Transition from unlocked
        to locked on lock
    Transition from locked
        to unlocked on unlock
    Transition from unlocked
        to opened on open
  }
  Transition from opened
      to closed on close
}
```

(b) Textual Notation



(c) Abstract Syntax as Instantiation of Metamodel

Fig. 3. Three Representations of the Same State Chart

3 Textual Concrete Syntax Specification

This section is the core of the paper and presents the most important concepts of the Textual Concrete Syntax Specification Language (TCSSL). To illustrate these concepts, we use the example of section 2 and define the abstract ↔ concrete syntax mapping as presented in Fig. 4 using TCSSL.

As the TCSSL is a language, it needs its abstract syntax and concrete syntaxes to be defined. For sake of briefness, we will not introduce the concrete syntax although this would be possible using the TCSSL itself. Abstract syntax is defined using metamodeling, and presented using the MOF concrete syntax [15], which is close to UML class diagram notation. Concepts are thus represented using metaclasses, relations between them are modeled by associations (i.e. properties put face to face), and basic properties by attribute using basic data types (String, Boolean, Real, Integer).

The main concepts of the TCSSL are summarized in Fig. 5. A TCSSL syntax (`SyntaxDefinition`) is a set of TCSSL rules. It has a name (`name`), defines a default language interpreter for expressions (`defaultLanguage`), and a set of tokens that are ignored by the parser (like blanks or line feeds - `ignoredLexems`). A TCSSL rule (`Rule`) is assigned to each representable concept (`MetaElement`) of the metamodel. A rule may have a name (`name`). A rule may be a default rule for its concept (`isDefault`), which is the case in Fig. 4 line 9, which is the rule to be fired when no specific rule name is invoked. A rule may also be an entry rule (`isEntry`), as in Fig. 4 line 5, meaning that this may be the main rule of a model textual representation.

```
1:      syntax UMLSequenceCharts
2:      with    defaultLanguage KerMeta
3:      and     ignored "\ ", "\t", "\n", "\r", "\f";

5:      entry rule for StateMachine ::=
6:      "StateMachine" self.name
7:
8:      self.top;


9:      rule for SimpleState ::=
10:     <initial>"State" self.name
11:     ;

12:     rule for CompositeState ::=
13:     <initial>"CompositeState" self.name "{"
14:             mixed(self.state || self.transition)
15:     "}"
16:     ;

17:     macrorule initial ::=
18:     (<< {self}.isInitial := {true} >>"initial"
19:     |_
20:     );

21:     rule for Transition ::=
22:     "Transition" self.name "from" self.source<name>
23:           "to" self.target<name> (self.event opener "on")
24:     ;

25:     seekrule name for State with criterium {self.name} ::=
26:     [1](self.name);

27:     singletonrule for Event with criterium {self.name} ::=
28:     [1](self.name);
```

Fig. 4. The State Chart Language Specified Using the TCSSL

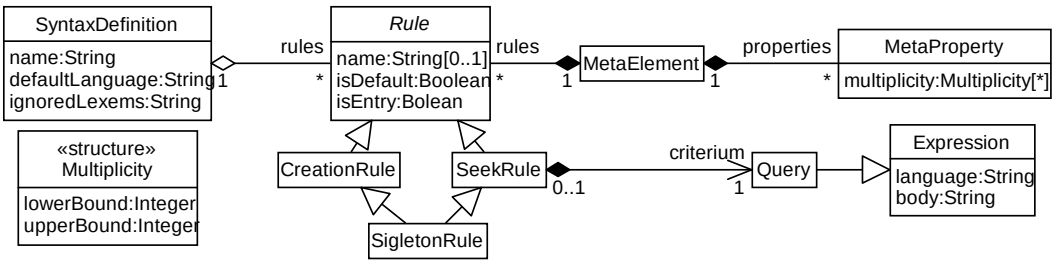


Fig. 5. Main TCSSL Concepts

There exist three kinds of rules, which behave differently only in case of concrete → abstract transformation. Macrorules are not real rules as they behave as syntactic sugar to avoid repetitive constructs. This is the case in Fig. 4 line 17, where the macro `initial` is used both in default rule for `SimpleState` and rule for `CompositeState`.

Simple rules (`CreationRule`), like in Fig. 4 line 9, are rules that will instanciate the corresponding model element when triggered at text analyzing time. This is the case, as in Fig. 3 *(b)*, when it is stated `State locked`: as specified by the rule, a new instance of the `SimpleState` concept in metamodel of Fig. 2 is instanciated in the model, with its `name` property filled with the "locked" string value.

Seek rules (`SeekRule`) are rules that do not create anything in the model. They merely look for existing concept instances in the model that satisfy a criterion given in the `with criteria` clause, as for Fig. 4 line 25. In the example, the rule looks for the name of the state, as it can be found in the textual representation. That kind of rule may be interesting for referencing an already existing model element. Here, it is a rule for concept `State` ; as the rule has a name, it is not considered as the default rule for that concept. So, to call that rule one needs to explicitly name it. This is the case in Fig. 4 line 22 and 23, when the rule calls the `self.source<name>` and `self.target<name>` properties that are in relation with the `State` concept.

Singleton rules (`SingletonRule`) have almost the same behavior as creation rules, but will avoid instanciation in case a given criterion is satisfied. If there is no model element that satisfies the criterion, then that element will be created, otherwise, the existing element is returned by the rule for referencing. This is the case for the `Event` rule of Fig. 4 line 27. If the event already exists, there is no need to create it, otherwise, a new `Event` instance will appear in the model.

An expression (`Expression`), or its side-effect free variant (`Query`), is expressed in any kind of language able to navigate and, if necessary, to alter the model. Examples of such language are Kermeta [13] and MTL [22] for languages able to alter the model, or OCL [17] for queries. If necessary, as for most compiler compiler specifications, expressions may be included within rule atoms for some purpose (by the mean of the `SideEffectRuleAtom` concept). Of course, altering the model is only necessary for concrete → abstract syntax mapping, and the latter concept is inactive during a concrete ← abstract syntax mapping.

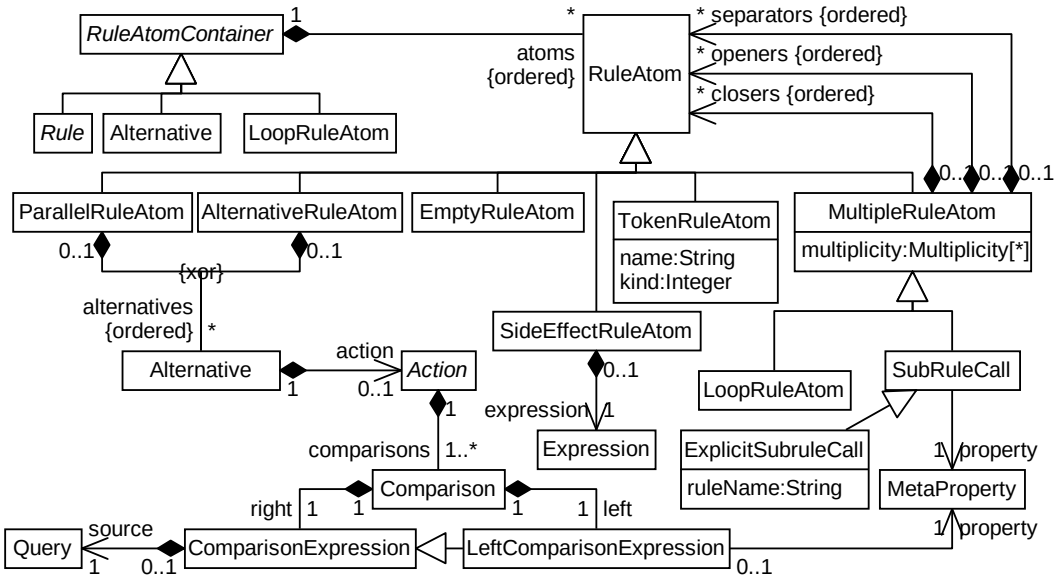


Fig. 6. TCSSL Rule Atoms

Rule elements are presented by Fig. 6. A rule contains an ordered suite

of rule atoms (`RuleAtom`). A choice (`AlternativeRuleAtom`) is composed of alternatives (`Alternative`) of rule atoms. An alternative can be guarded (`Action`). A guard explains a comparison, or a conjunction of comparisons (`Comparison`), between a property of a model element found by an expression (`LeftComparisonExpression`) and a value (`RightComparisonExpression`). If triggered, that guard behaves differently depending on the direction of the mapping. In case of concrete → abstract mapping, the property in the model is affected with the value. In case of concrete ← abstract mapping, the guard selects the correct rule alternative according to information stored in the model. In a choice, only one alternative, so called the default alternative, is not protected by a guard, otherwise there would have an indeterminism in case of concrete ← abstract mapping. Such a construct is visible in Fig. 4 line 18-20. Here, in case the first alternative has to be chosen, the property `isInitial` of the considered model element has to be "true". Otherwise, the second choice (the defaut one, as not guarded) is the good one. In this case, the first choice is a keyword (`Token`), and the second choice is an empty rule (`EmptyRuleAtom`): if the considered model element has property `initial` as "true", then the representation should contain the word "initial", and only in this case. The careful reader may notice that there is no `initial` property in metaclass `State` of the metamodel of Fig. 2. Although this rule sounds ill-formed, this is an extension construct that will be explained in section 4.

Another family of rule element is loops (`MultipleRuleAtom`). Loops may be simple loops à la EBNF (`LoopRuleAtom`), or properties triggering subrule calls (`SubruleCall`). Indeed, properties may not have a [1..1] multiplicity: they can range from 0 to infinite (*): property calls are thus implicit loops. A simple loop contains an ordered sequence of rule atoms that can occur a given number of times, according to a multiplicity (`multiplicity`). Items participating in a loop may be separated by specific elements (`separators`). Loops may also be opened (`openers`) or closed (`closers`) by specific rule atoms. For instance, in Fig. 4 line 23, it is stated `(self.event opener ''on'')`. This means that the rule awaits a subrule call for `Event` 0 or 1 time, according to the `event` property multiplicity in the metamodel. If it appears once, as in Fig. 3, an "on" will appear before the event. Multiplicity may be restricted compared to the property's multiplicity. For instance, in Fig. 4 line 26, the name is absolutely required, so we restrict multiplicity to [1..1] even though in the metamodel the `name` property is declared with [0..1]. Another example is `mixed(self.state || self.transition)` in the rule for `CompositeSate`. Here, both `state` and `transition` have a [0..*] multiplicity. This means that subrules for `SimpleState`, `CompositeState`, and `Transition` may fire any number of time in any order. Property calls are composed within a choice where they can happen in any order, according to the `mixed` keyword (`ParallelRuleAtom` replaces here `AlternativeRuleAtom` to express alternatives that do not need any guard). This means that in a state machine textual specification one could find for example a simple state, then a transition, and

then another simple state, without violating the rule.

Now, according to the metamodel of Fig. 2 and to the concrete syntax expressed in Fig. 4, an automatic tool should formally be able to transform part (b) to part (c) of Fig 3 (concrete → abstract mapping), and part (c) to part (b) (concrete ← abstract mapping). In the first case, it is even possible to have text properly formatted following the scheme given in the TCSSL specification by inserting a space when there is a space in the specification, inserting a carriage return when there is a carriage return in the specification, etc.

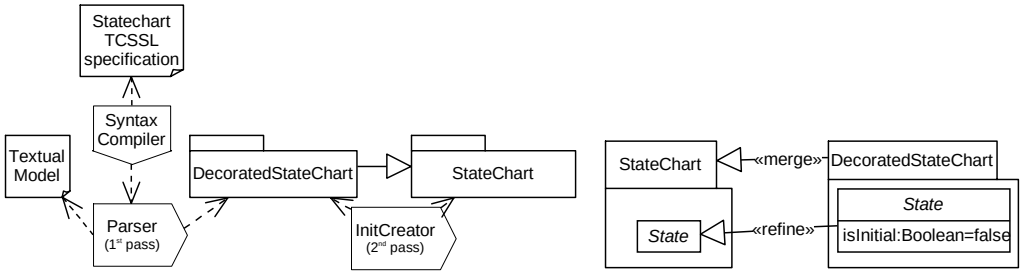
4 Towards more Complex Mappings

That paper aims at solving the abstract ↔ concrete syntax mapping by specifying rules in an EBNF-like textual concrete syntax specification language (TCSSL). However, the proposed solution does not claim to solve all the problems encountered in compiler construction. In particular, it does not avoid the necessity of multiple pass, for instance to perform type checking. This part presents a solution to integrate multiple pass analysis in the proposed solution of section 3.

The main idea is to keep the classical way of solving the problem, that is transforming decorated abstract syntax trees. The advantage in our approach is that abstract syntax trees are models, conforming to metamodels. This offers the possibility to formally define a pass and what decorations are available for a given pass: decorations are no longer typeless key-value pairs, but attributes that are added to the metamodel. A pass is then merely a model transformation [20] "treating" attribute slots information that have a decoration role in the model. To formally add an attribute to a metamodel (i.e. a decoration), one might use higher order hierarchies [7]: this technique uses refinement of modeling elements, to add compatible additional features, such as adding attributes to a class. It works in a similar way than inheritance, but at the metamodel level: one can use a model only with the knowledge of the metamodel, or a parent of the metamodel. So, to add a new decoration attribute to a class, one should make a new metamodel "inheriting" from the official metamodel and refine some classes by adding new attributes as placeholders for decorations. An n-pass architecture includes then $n-1$ refinements of the main metamodel, a TCSSL specification for the $n-1^{th}$ refinement, and $n-1$ model transformation arranging the model so that information from refinement level x is available from level $x-1$. Note that for the approach to be valid both at concrete → abstract mapping and concrete ← abstract mapping, the participating model transformations have to be bidirectional.

An example of such decoration is the attribute `State::isInitial` that does not exist in metamodel of Fig. 2, but which is used by grammar of Fig. 4: it would be quite hard to describe the meaning of the `initial` keyword in the TCSSL specification on Fig. 4. It is actually much easier to

add an `isInitial:Boolean=false` decoration to the metamodel that is set to true in case a state is declared as initial (see macro rule `initial`). This also requires to write a second-pass model transformation that, for each state marked as initial, creates or gets the initial state of the container and creates a transition to the considered state. The example is summarized in Fig. 7: the first pass is undertaken by the parser (generated from the TCSSL specification), and the second one is a hand-written model transformation that handles the `isInitial` decoration. As `DecoratedStateChart` is compatible with `StateChart` by refinement, the latter transformation does not need to transform for example `DecoratedStateChart::Transition` instances into instances of `StateChart::Transition`.



(a) General Architecture (b) Metamodel Refinement

Fig. 7. The 2-Pass State Chart Compiler Architecture

5 Early Implementation

This section describes our prototype implementation. The TCSSL has its own metamodel and a concrete syntax; it is therefore interesting to develop and test the tool chain on the TCSSL itself. A bootstrap can then be realized. A hand-created tool parses the TCSSL specification defining the TCSSL textual concrete syntax, and creates the TCSSL model conforming to the TCSSL metamodel in the model repository. From that model, tools are generated as suggested by Fig. 1. These tools are then able to parse again the TCSSL specification for TCSSL and to create a TCSSL model, that in turn can generate the same TCSSL tools. TCSSL tools are developed in the context of the MDDi [6] open-source Eclipse project. Currently, a prototype parses the TCSSL specification, and generates the parser for its textual notation. A code generator is being developed to generate the TCSSL textual notation from the model repository. From Fig. 8, it is possible to extract the architecture of the tool chain. It is divided into two main parts : a parser and a text generator. The parser analyzes the text in the editor and fills the model representation in the repository. A second tool generates text from the model repository. This section will be divided into two parts, the first one describes the textual parser generation, and the second one will present the code generator.

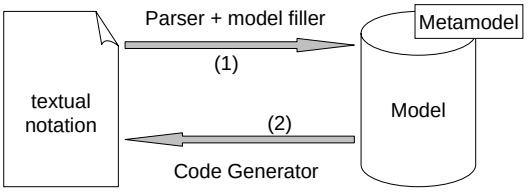


Fig. 8. Division of the Prototype into 2 Parts

5.1 From Textual Notation to Model

This part requires a text parser. The input for this parser is the content of the generic editor. The parser creates the corresponding elements into the model repository. ANTLR [19] is used to simplify the parser generation. According to its author, ANTLR is a language tool that provides a framework for constructing recognizers and translators from grammatical descriptions containing Java actions. ANTLR generates a parser from an EBNF-like grammar that describes the concrete syntax of the language. For the first developments, elements are directly created in the model repository using the actions embedded in the grammatical description.

Rules in the ANTLR grammar are mapped one-to-one to the TCSSL rules. In the state chart example, the rule `SimpleState` is directly translated into an homonym ANTLR rule. This rule returns a reference to a model element of type `SimpleState`. When this rule is called by the parser, a `SimpleState` instance is created in the model. Subrule calls returns the features or associations of the `SimpleState` element. For example, "`self.name`" returns the string corresponding to the name of the `SimpleState`. This name is then added to the `name` feature of the element.

The `State` element case is interesting: there is no default rule for the `State` element. In fact, according to Fig. 2, `State` is an abstract element. So no concrete syntax can be defined for it. In the rule `StateMachine`, "`self.top`" is a subrule call atom that refers to an element of type `State`. The parser expects subclasses of element `State`, i.e. a `SimpleState` or a `CompositeState`. A rule is defined for the element `State` in the parser. This rule is defined as a choice between each default rule corresponding to the child elements of `State`. In this case, the rule for `State` is a call to the rule `SimpleState` or `CompositeState`. `SimpleState` rule returns an element of type `SimpleState`, which is a `State` by inheritance. Fig. 9 corresponds to an excerpt of the generated grammar for the ANTLR parser. It defines the rule generated for element `State` and the rule for element `SimpleState`. The creation of elements in the model is left over in the figure.

The prototype of the tool is fully written in Java. Queries in the model are developed using the interfaces generated by EMF. More complex queries would require a dedicated language, like MTL [22] or Kermeta [13].

The ANTLR grammar file is the only element to produce for the transformation concrete → abstract syntax. This grammar file embeds the interac-

```
1: state returns [State state=null]
2:     :
3:           state = simpleState
4:      |    state = compositeState
5:   ;

6: simpleState returns [SimpleState simpleState = null]
7:     :
8:     "State" name
9:     ;
```

Fig. 9. Excerpt of the ANTLR Grammar for State Charts

tions with the model. This grammar file produces the parser for the textual notation. Fig. 10 summarizes the generation of the textual parser.

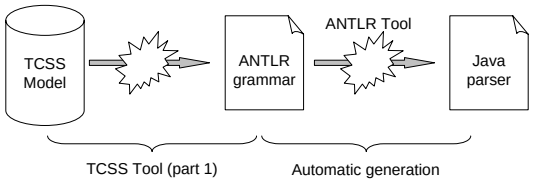


Fig. 10. Concrete → Abstract Mapping with ANTLR

5.2 From Model to Textual Notation

The second part of the tool generates code from the model. The generic editor is based on the Eclipse IDE and its EMF model repository. The EMF project contains a tool for generating source code named JET [5]. JET is a generic template engine that can be used to generate code. The templates are defined in file using a JSP-like syntax. These templates files are translated into Java classes. These classes have a method named `generate` that produces a string for a given object (the context). Fig. 11 displays the rule for the element Transition taken from the State Chart TCCSL.

```
21:     rule for Transition ::=
22:     "Transition" self.name "from" self.source<name>
23:           "to" self.target<name> (self.event opener "on")
24:     ;
```

Fig. 11. TCSSL Definition for `Transition`

A template file is associated to each rule in TCSSL. JET templates are separated into two main parts. The first part of the template is written in Java. It calls each subrule templates and stores the resulting string. "Alternatives" are basically transformed into test loops. The second part has a JSP-like syntax and simply associates the strings coming from the subrule calls to the rest of the text, like keywords and tokens. Fig. 12 displays the corresponding JET file for the element Transition.

One of the issues for this tool is the management of the indentation, in case of multi-line generated strings. For example, the `Transition` template

```
1: // Template for Transition element

2: // Java Part
3: String name = transition.getName();       // simple element

4: State source = transition.getSource();    // get the source state
5: String source_name = StateNameTemplate.generate(source); // name of the state

6: State target = transition.getTarget();    // get the target state
7: String target_name = StateNameTemplate.generate(target); // name of the state

8: Event event = transition.getEvent();
9: String event_name = "";
10: if(event != null) { event_name = "on" + EventTemplate.generate(event); }

11: // JET part
12: Transition <%= name %> from <%= source_name %>
13:      to <%= target_name %> <%= event_name %>
24: ;
```

Fig. 12. JET Code Generator for `Transition`

returns a String that is three-lines long. The TCCS language defines a relative indentation for the element transition. When text for transition is generated and used in the template for the composite state, there is an offset for the indentation. This means that the indentation must be computed in the Java part, reducing the interest of using a template engine. Fig. 13 summarizes the procedure to generate the code generator.

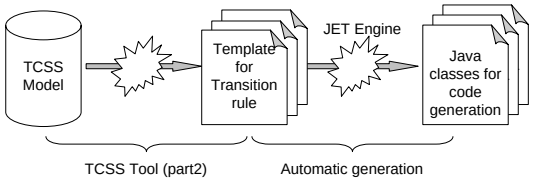


Fig. 13. Concrete ← Abstract Mapping with Jet

6 Conclusion

This work may be viewed as an experimentation in the specification of concrete syntaxes in the context of meta-modeling applied to language engineering. It is obviously far from bringing definitive answers to these complex problems. However the presented material may contribute, with many other ongoing research works on metamodeling and language engineering, to a better understanding of hard related research problems.

We have proposed a new approach, based on metamodels and EBNF-like rules, which supports a formal bi-directional mapping of both concrete → abstract, and concrete ← abstract syntaxes. An early prototype which realizes both analysis and synthesis of concrete syntax has been presented, based on existing tools such as ANTLR and the JET template engine.

Future works include the specification of a graphical concrete syntax for the TCSSL language and a better integration with executable meta-modeling languages such as Kermeta.

References

[1] Atkinson, C. and T. Kühne, *The role of meta-modeling in MDA*, in: *Workshop in Software Model Engineering (WISME@UML)*, Dresden, Germany, 2002.

[2] Clark, T., A. Evans, P. Sammut and J. Willans, *Applied metamodelling: A foundation for language-driven development* (2005).
URL http://albini.xactium.com

[3] de Lara, J. and H. Vangheluwe, *Using AToM^3 as a meta-case tool*, in: *Proceedings of the 4th International Conference on Enterprise Information Systems (ICEIS)*, 2002, pp. 642–649.

[4] Eclipse, *Eclipse Modeling Framework (EMF)* (2005).
URL http://www.eclipse.org/emf/

[5] Eclipse, *Java Emitter Templates (JET)* (2005).
URL http://www.eclispe.org/emf/

[6] Eclipse, *Model Driven Development integration (MDDi)* (2005).
URL http://www.eclispe.org/mddi/

[7] Ernst, E., *Higher-order hierarchies.*, in: L. Cardelli, editor, *ECOOP 2003 - Object-Oriented Programming, 17th European Conference on*, LNCS **2743** (2003), pp. 303–328.

[8] Fondement, F. and T. Baar, *Making Metamodels Aware of Concrete Syntax*, in: *European Conference on Model Driven Architecture (ECMDA)*, LNCS **3748** (2005), pp. 190–204.

[9] Greenfield, J., K. Short, S. Cook and S. Kent, "Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools," Wiley, 2004.

[10] Harel, D., *Statecharts: A visual formulation for complex systems*, Science of Computer Programming **8** (1987), pp. 231–274.

[11] Kent, S., *Model driven engineering*, in: M. J. Butler, L. Petre and K. Sere, editors, *Proceedings of Third International Conference on Integrated Formal Methods (IFM 2002)*, LNCS **2335** (2002), pp. 286–298.

[12] Mellor, S. J., A. N. Clark and T. Futagami, *Guest editors' introduction: Model-driven development*, IEEE Software **20** (2003), pp. 14–18.

[13] Muller, P.-A., F. Fleurey and J.-M. Jézéquel, *Weaving executability into object-oriented meta-languages*, in: *International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, LNCS **3713** (2005), pp. 264–278.

[14] Muller, P.-A., P. Studer and J.-M. Jézéquel, *Model-driven generative approach for concrete syntax composition*, in: *Workshop in Best Practices for Model Driven Software Development*, Vancouver, Canada, 2004.

[15] OMG, *Meta-Object Facility (MOF) 1.4*, OMG Document formal/02-04-03 (2002).

[16] OMG, *MOF Model to Text Transformation Language (Request For Proposal)*, OMG Document ad/2004-04-07 (2004).

[17] OMG, *OCL 2.0 Specification*, OMG Document ptc/05-06-06 (2005).

[18] OMG, *Xml Metadata Interchange (XMI 2.1)*, OMG Document formal/05-09-01 (2005).

[19] Parr, T., *ANother Tool for Language Recognition (ANTLR)* (2005).
URL http://www.antlr.org/

[20] Sendall, S. and W. Kozaczynski, *Model transformation: The heart and soul of model-driven software development*, IEEE Software **20** (2003), pp. 42–45.

[21] Sun Microsystems, *Metadata repository (MDR)* (2005).
URL http://mdr.netbeans.org/

[22] Vojtisek, D. and J.-M. Jézéquel, *MTL and umlaut NG - engine and framework for model transformation.*, ERCIM News **58** (2004), pp. 46–47.