Composing Textual Modelling Languages in Practice

Bart Meyers
Dep. of Mathematics and Computer Science
University of Antwerp
{Bart.Meyers}@ua.ac.be

Antonio Cicchetti
Malardalen Research and Technology Centre (MRTC)
Malardalen University
{antonio.cicchetti}@mdh.se

Esther Guerra, Juan de Lara
Dep. of Computer Science
Univ. Autónoma de Madrid
{Esther.Guerra, Juan.deLara}@uam.es

ABSTRACT

Complex systems require descriptions using multiple modelling languages, or languages able to express different concerns, like timing or data dependencies. In this paper, we propose techniques for the modular definition and composition of languages, including their abstract, concrete syntax and semantics. These techniques are based on (meta-)model templates, where interface elements and requirements for their connection can be established. We illustrate the ideas using the MetaDepth textual meta-modelling tool.

Categories and Subject Descriptors

I.6.5 [**Simulation and Modeling**]: Model Development—*Modeling methodologies*

General Terms

Design, Languages

Keywords

Meta-Modelling, Language Composition, Concrete Textual Syntax, MetaDepth

1. INTRODUCTION

In domain-specific modelling, the engineering of domain-specific languages (DSLs) is a vital part of the development cycle. Techniques such as meta-modelling and model transformation greatly facilitate the development of such DSLs. However, most methods for development require the language engineer to start from scratch when developing a new DSL. This is in contrast with the observation that there are similarities between DSLs: one might need a new variant of Petri nets, a different kind of automaton (as there are tens of variants of these two formalisms), or the combination of a number of formalisms. Therefore, there is a clear need for reuse of existing languages, or language modules.

In this paper we introduce an approach for the modular composition of modelling languages, based on existing language modules. Inspired by generic programming [10], we use existing, generic model composition mechanisms to achieve this [5]. This way, we aim for general applicability of our approach, provided that these composition mechanisms are supported. A language is defined by its abstract syntax (the structure of valid models, defined by a meta-model), concrete syntax (the visual/textual representation of valid models) and behavioural semantics (how its models run as a system). We support the composition of the modules of these three aspects of a language, and our implementation in the MetaDepth tool [4] supports textual concrete syntax.

Paper organization. Sec. 2 presents a running example, used throughout the paper. Sec. 3 introduces the composition mechanisms we use. Sec. 4 shows how to use these mechanisms, illustrated by the running example. Sec. 5 discusses related work and Sec. 6 concludes the paper.

MPM '12 Innsbruck, Austria

2. MOTIVATING EXAMPLE

In this section, we introduce our motivating example, i.e., Timed State Machines, and a model in this language, i.e., a model of a traffic light, to be used in the following sections.

2.1 Timed State Machines

As a running example, we will implement a language for timed automata [1]. A timed automaton is a finite-state automaton with a number of clocks. The integer value of each clock increases step-wise, along with the execution of the automaton. A transition of a timed automaton may have a guard expression over the clock values, which may enable or disable the transition. A transition may also have actions that assign a new value to a clock, after firing the transition.

Figure 1 shows a model of the timed behaviour of a traffic light, using Timed State Machines (visual syntax taken from [2]). The model shows how a traffic light goes from green to yellow, and to red, in respectively 20, 3, and 20 seconds. This is the default behaviour, when the pedestrian button is not pushed. When the traffic light is green and a pedestrian wants to cross the road, he can push a button. Whenever this happens, a *press* event occurs in the traffic light model, and after 3 seconds, the light switches to yellow, and then red, so that the pedestrian can cross. There are two clocks in this model, x and y. The clock x represents the time that the traffic light should be in the same state (green, yellow or red), while y models the delay in changing from green to yellow when the pedestrian pushes the button.

The Timed State Machine language proves to be an excellent example of our modular approach for language en-

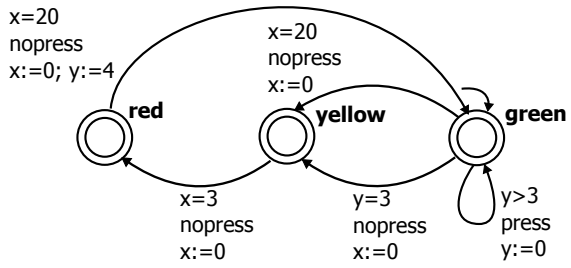

Figure 1: A Timed State Machine for a traffic light.

gineering as it can be seen as a combination of two independent language modules: (a) finite state machines, and (b) a language for expressions, with constructs for comparison, assignment and simple mathematics (for the guards and actions). These two language modules could be used autonomously, or in combination (using our approach) with different language modules. In this paper we will show how to combine them, and additional constructs such as clocks will have to be introduced.

2.2 Implementation in MetaDepth

For this paper, the concepts are illustrated using the MetaDepth tool [4]. This is a tool for textual modelling that allows the definition of the three aspects of a language: abstract, concrete syntax and semantics. Abstract syntax is modelled using the MetaDepth default textual syntax, which can be used on any meta-level. Domain-specific textual concrete syntax is modelled using a designated MetaDepth meta-model named *TextualSyntax*, which enables assigning the meta-model and each meta-class a syntactic template indicating how their instances should be represented. Models conforming to *TextualSyntax* are compiled to ANTLR grammars[1], with appropriate semantic actions to create the model in-memory [6]. MetaDepth integrates the Epsilon family of model management languages[2]. Hence, behavioural semantics (e.g., the behaviour of the traffic light) are defined with an in-place transformation using the Epsilon Object Language (EOL) [11]. This is a language similar to OCL with imperative constructs to e.g. create objects, and make assignments.

3. COMPOSITION MECHANISMS

In this section we review some (meta-)model composition mechanisms that we will use in order to compose modelling languages. These mechanisms are supported by the MetaDepth tool [4, 5], but the ideas are general, applicable to other modelling frameworks.

3.1 Extension

We use two extension mechanisms, at the class/object level and at the meta-model/model level. Inheritance is used as an extension mechanism for both types (class inheritance) and objects. In the former case, children classes inherit attribute types defined in parent classes. In the latter case, inheritance acts as value overriding. In this way, if children objects do not define a value for an attribute instance, the value is taken from the parent object.

[1] `http://www.antlr.org/`
[2] `http://www.eclipse.org/epsilon/`

Regarding models, our first modularity mechanism is (meta-)model extension, and is applicable to models and meta-models. At the meta-model level, an *extension* is a meta-model fragment that increments the elements in a base meta-model, by adding new types, new constraints, or defining new attributes of existing classes. Therefore, it is similar to *package merge* in UML [8]. This mechanism allows a modular definition of meta-models, and is used intensively, e.g., in the definition of the UML itself [13].

Our extension mechanism works at the model level as well, by using object inheritance. This is useful to define libraries of predefined models, which can be later extended with further objects, or to override the properties of existing objects.

Finally, models and meta-models can use the *import* directive in order to access other model elements.

3.2 Concepts

Some of our composition mechanisms are based on expressing *requirements* that need to be fulfilled by other (meta-)models, to enable their composition. Using terminology from generic programming [10], we call *structural concept* to a set of structural requirements to be fulfilled by a (meta-)model. They can be used to express requirements both for models and meta-models, and therefore have the form of a model or a meta-model as well[3]. However, their elements (classes, attributes, references) are interpreted as variables, which need to be *bound* to the elements of the meta-model.

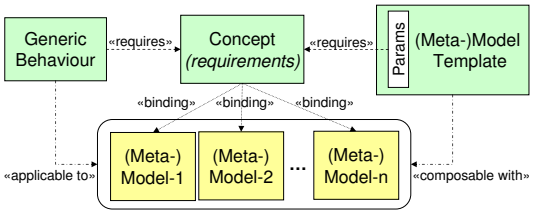

Figure 2: Expressing generic behaviour and composability criteria through concepts

Concepts are useful in several ways. First, one can define operations over a concept, so that they become reusable. When the concept gets bound to some meta-model, the operation becomes applicable to it. In this case, the concept expresses the requirements for the execution of an operation. Second, concepts can be used to express composition requirements. Hence, it is possible define meta-model fragments, where some elements of such fragment are variables. The requirements for binding such variables to concrete elements in other models can be expressed through a concept.

The *binding* is a process by which the variables in a concept are bound to elements in a meta-model [5]. It is a one-to-one or many-to-one correspondence between the variables in the concept and elements in the meta-model, allowing some heterogeneity between the concept and the meta-model. For example, the subtyping relations in the concept must be preserved in the meta-model, but the meta-model may collapse several classes, or add intermediate ones.

In order to gain further flexibility, *hybrid concepts* can reduce to a minimum the structure required from a specific meta-model, but requiring instead the implementation

[3] For simplicity, next we just talk about *meta-model concepts*, but explanations are also applicable to *model concepts*

of some operations. Hence, in addition to a binding, the user needs to implement some operations (in EOL). This is a means to hide unessential structural requirements, which could be implemented in different ways by different meta-models, behind operations, like e.g. in Java interfaces.

3.3 Templates

While (meta-)model extension provides an extensibility mechanism, such extensions are not a reusability mechanism but a means for modularity. (Meta-)Model templates enable a more flexible definition of model and meta-model fragments, as they permit their connection (perhaps through extension) to other models, which are not specified at design time. Hence, a meta-model template is a meta-model where some elements (meta-classes, features, associations) are variables. The connection requirements for such variables are expressed through a concept.

Operation templates enable the definition of an operation independently of a concrete meta-model. For this purpose, they are defined over a concept. Hence the types used in the operation are actually variables. Then, a binding from the concept to some meta-model assigns a particular type to each variable, and induces a retyping of the operation template, which becomes applicable to the bound meta-model.

Once we have reviewed some composition mechanisms, next section shows how they can be applied to our example.

4. COMPOSING MODELLING LANGUAGES

Next we show how to create the Timed State Machine language from two existing language modules: *StateMachine* and *Expression*. Using the composition mechanisms previously presented, we tackle the three aspects of the language: abstract, concrete syntax, and semantics. The two language modules are woven in four steps:

1. *SMC*: Template instantiation of the existing *StateMachine* template with the existing *Expression* language, to obtain *SMC*, a machine with conditions and actions;
2. *TSM*: Extension of *SMC* with a *Clock* language construct, to obtain *TSM*, a state machine with time;
3. *ExpressionTemplate*: Extension of *TSM* to create a new template *ExpressionTemplate*, linking identifiers in the expressions to a particular value;
4. *TimedStateMachine*: Template instantiation of *ExpressionTemplate* to link identifiers with clocks.

Figure 3 shows these four steps. (Parts of) meta-models are shown in blue squares representing a language module, with normal arrows as associations and filled arrows as inheritance links. Template parameters are represented by red dark squares (with the name of the variable preceded by "&"), instantiations are represented by green dotted arrows, and extension is represented by red "+" symbols.

Applying these four steps results in *TimedStateMachine*, for which *Associations* have conditions and actions that can refer to clock values: e.g., a transition that can only be fired if the value of clock c is greater than 5 (see also 1).

4.1 Abstract Syntax

In order to create the abstract syntax for the language, the four steps presented above are followed:

Step 1 Listing 1 shows a template for a state machine. It has three template parameters (line 1), which are identifiers preceded by a "&". The first parameter represents a model that contains the necessary concepts for conditions and actions and is imported into the *StateMachine* model (line 3). The second and third parameter represent the condition and action construct, which are then referenced in *Transition* (lines 16–17): a *Transition* may have a condition, and multiple actions. A *Transition* also has an incoming and an outgoing *State*, and may have a symbol (lines 13–15), that should be (represented by the constraint on lines 18–20) in the alphabet (lines 4–7). A *State* can be an initial state or a final state (lines 8–11), and there should be exactly one initial state, and at least one final state (lines 22–23). Constraints (lines 6, 18–20 and 22–23) are expressed in EOL-code between dollar signs. The template requires the fulfilment of the *Evaluatable* concept (line 2). This concept (not shown due to space restrictions) states that *&Condition* and *&Action* should be node elements of a model *&ConditionActionModel* and should implement an operation *eval* (we will come back to this issue in subsection 4.3).

```
1  template <&ConditionActionModel, &Condition, &Action>
2  requires Evaluatable(&ConditionActionModel, &Condition, &Action)
3  Model StateMachine imports &ConditionActionModel {
4    Node alphabet[1] {
5      size    : int;
6      minSize : $self.size>0$
7    }
8    Node State {
9      initialflag : boolean = false;
10     finalflag   : boolean = false;
11   }
12   Node Transition {
13     incoming  : State;
14     outgoing  : State;
15     symbol    : int [0..1];
16     condition : &Condition [0..1];
17     action    : &Action[*];
18     noinvalidsymbols : $alphabet.exists( a | Transition.all().select
19       ( t | t.incoming=self).forAll
20         ( t | not t.symbol.isDefined() or t.symbol<a.size))$
21   }
22   oneInitial   : $State.allInstances().one( s | s.initialflag =true)$
23   severalFinal : $State.allInstances().exists( s | s.finalflag =true)$
24 }
25 StateMachine<Expression, Expression::Exp, Expression::Assignment> SMC;
```

Listing 1: Template for the StateMachine module

A model exists for the *Expression* language, representing simple algebraic expressions (omitted by space constraints). The language contains an abstract *Exp* class which is a superclass of all kinds of expressions, such as *Equals*, *Plus*, *Literal* (a constant value) or *Variable* (an identifier), and it also contains an *Assignment* element. Its *Exp* element can serve as a condition for a *Transition*, and *Assignment* can serve as an action for a *Transition*. In line 25 the template is instantiated with the *Expression* language, with *Exp* as the condition and *Assignment* as the action, resulting in *SMC*, a state machine with conditions and actions.

```
1 Model TSM extends SMC {
2   Node Clock { time : int; }
3 }
```

Listing 2: Adding the clock feature

Step 2 We still need an element denoting a clock, with a clock value. This is achieved by using extension as shown in Listing 2. Line 1 states that the current meta-model *TSM* extends *SMC*, which is actually the template instantiation shown in line 25 of Listing 1.

```
1 template <&ContextModel, &Context> requires IntGettable(&ContextModel, &Context)
2 Model ExpressionTemplate extends TSM, &ContextModel {
3   Node RefVar : Variable { context : &Context; }
4 }
5 ExpressionTemplate<TSM, TSM::Clock> TimedStateMachine;
```

Listing 3: Linking the *Variable* element to a clock

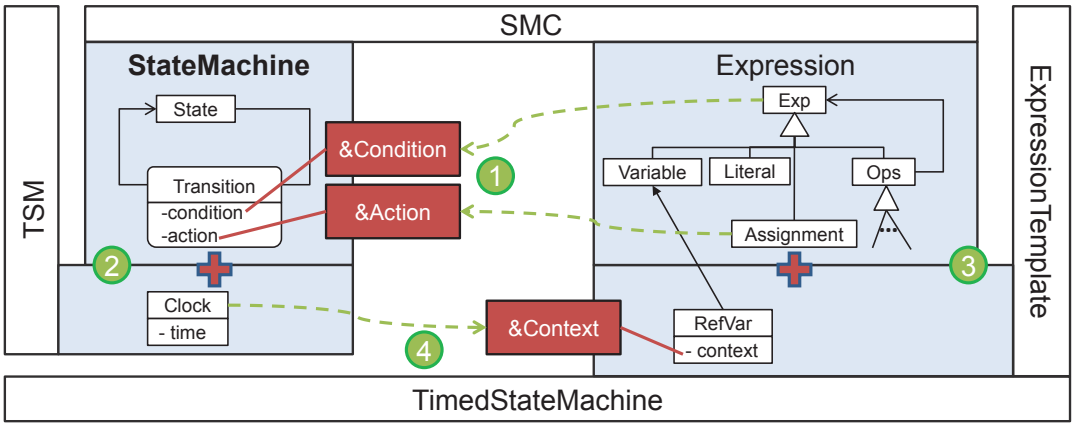

Figure 3: The four steps for weaving the two languages.

Step 3 and 4 The conditions and actions use named variables (e.g., x), which should refer to the clock values in our final timed state machine. This means that a *Variable* should be linked (i.e., have a context) to a *Clock* instance. This is shown in Listing 3. First a template is created (lines 1–4) where a *RefVar* with a *context* is introduced. By using template parameters, this is again a generic way of extending the *Expression* language. The concept *IntGettable* will be discussed in Section 4.3. Finally, the template is instantiated with a *Clock* (line 5).

```
1  TimedStateMachine trafficlight {
2    Clock x { time = 0; }
3    Clock y { time = 0; }
4    alphabet a{ size = 2; }
5
6    State red    { initialflag = true; finalflag = true;}
7    State green  { finalflag = true;}
8    State yellow { finalflag = true;}
9
10   Transition r2g{ incoming= red; outgoing= green; condition= x20; action= [initx , inity4 ];}
11   Transition g2y{ incoming = green; outgoing = yellow; condition = x20; action= [initx ]; }
12   //...
13   RefVar varx        { name = "x"; context = x; }
14   RefVar vary        { name = "y"; context = y; }
15   Literal zero       { value = 0; }
16   //...
17   Literal twenty     { value = 20; }
18   Assignment initx   { op1 = varx; op2 = zero; }
19   //...
20   Equals x3          { op1 = varx; op2 = three; }
21   //...
22   GreaterThan ygt3   { op1 = vary; op2 = three; }
23 }
```

Listing 4: The traffic light model as an instance of the *TimedStateMachine*.

The language *TimedStateMachine* that results from the four steps can be instantiated as in Listing 4, which shows a traffic light MetaDepth model[4]. The three states (lines 6–8), some transitions (lines 10–11) and two clocks (lines 2–3) from Figure 1 can be recognized, and the size of the alphabet is 2 (line 4): there are symbols for pressing and not pressing the button. The syntax is quite verbose, especially for the conditions and actions (lines 10–23), which become similar to abstract syntax trees. Naturally, a simpler, more intuitive syntax is desirable, which we define next.

4.2 Concrete Syntax

In MetaDepth, textual concrete syntax can be defined by creating a *TextualSyntax* model as described in detail in [6]. Similar to defining the abstract syntax, we use the four steps described above (i.e., we use the extension mechanisms) in combination with the *TextualSyntax* language to obtain the textual syntax model. Just like the abstract syntax model, the concrete syntax model is composed modularly with maximal reuse of existing models. As a difference, this time the template and composition mechanisms are used at the model level, as we are building and connecting instances of the *TextualSyntax* meta-model.

Listing 5 shows an excerpt of a template (with a slightly simplified syntax for better understandability) for the concrete syntax of *StateMachine*. It defines the syntax for a *StateMachine* model (lines 4–7), *State* instances (lines 9–15) and *Transition*s (lines 17–21). Such definitions are included in instances of *TModel* and *TNode*. In both cases, they need to reference a given meta-model (field *refModel*) or metaclass (field *refNode*), declare a syntactic template (*tempExp*) and optionally, simple semantic actions (*creationExp* field). The listing is actually a model template, so that the syntactic template for *Transition* can reference the syntactic templates defined for the template parameters (variables *&TCond* and *&TAct*, in line 19), which will become available when the model template is instantiated (lines 24–25).

```
1  template <&TConditionModel, &TCond, &TAct>
2  requires ModelTNode2(&TConditionModel, &TCond, &TAct)
3  TextualSyntax StateMachineSyntax imports StateMachine, &TConditionModel {
4    TModel TMachine {
5      refModel = StateMachine;
6      tempExp = ["StateMachine _id { ( &TState | &TTransition )* } "];
7    }
8
9    TNode TState {
10     refNode = State;
11     tempExp = ["State _id ;", " InitialState _id ;",
12                " FinalSate _id ;", " InitialFinalState _id ;"];
13     creationExp = ["", "# initialflag = true",
14                    "# finalflag = true", "# initialflag = true ; # finalflag = true"];
15   }
16
17   TNode TTransition {
18     refNode = Transition;
19     tempExp = ["#incoming -[ #symbol ]-> #outgoing [ &&TCond ] / ( [ &&TAct ] )* ;"];
20     creationExp = ["#condition = &&TCond ; #action = &&TAct"];
21   }
22   //...
23 }
24 StateMachineSyntax<ExpressionSyntax, ExpressionSyntax::Texpr,
25                    ExpressionSyntax:: Tass> SMCSyntax;
```

Listing 5: *TextualSyntax* model that describes concrete syntax for *StateMachine*

The template is instantiated using *ExpressionSyntax* (lines 24–25, analogue to Listing 1 line 25), which is the *TextualSyntax* model for the *Expression* language module (not shown due to space limitations). The textual syntax of the

[4]The syntax is explained in [4].

conditions and actions are the template parameters *&TCond* and *&TAct*, that have to be *TNode*s in *TextualSyntax* model *&TConditionModel* according to the *ModelTNode2* concept (lines 1–2, the concept itself is not shown). The *TextualSyntax* model is compiled to an ANTLR grammar with semantic actions [6], so that the parsing results in a MetaDepth model. The remaining three steps also follow the structure of the four steps, and are similar to the composition of the abstract syntax model.

Listing 6 shows the traffic light model as an instance of the TimedStateMachine, using the concrete syntax. The model is much more concise and clear, in comparison to the same model using the default abstract syntax description (see Listing 4), especially for the conditions and actions.

```
1  TimedStateMachine trafficlight {
2    Clock x 0;
3    Clock y 0;
4    InitalFinalstate green;
5   FinalState yellow;
6   FinalState red;
7    red—[0]—>green     [x=20] / [x:=0]  [y:=4];
8    green—[0]—>yellow [x=20] / [x:=0];
9    yellow—[0]—>red    [x=3] / [x:=0];
10   green—[1]—>green   [y>=4] / [y:=0];
11   green—[0]—>yellow [y=3] / [x:=0];
12 }
```

Listing 6: The traffic light model using the modularly composed concrete syntax

4.3 Semantics

Abstract and concrete syntax are composed following the same outline. This is possible, as they are both implemented as MetaDepth models (meta-models in the case of the abstract syntax), so the same composition mechanisms can be used. Semantics are implemented as EOL programs, so other mechanisms, namely concepts, have to be used.

Hence, we will define the simulator over the variable types in concept *ExecutableTimedStateMachine* defined in Listing 7. The concept is actually a hybrid concept, as it requires both a binding and an implementation of some operations. The concept requires an entity *&S* modelling states, an entity *&T* modelling transitions, and another one *&C* playing the role of clocks. For each entity in the concept, we require the implementation of some operations that will be used by the simulator. In this way, we can decouple the semantics from any specific meta-model, but we require a binding, plus the implementation of some operations. Lines 19–21 show the binding of the concept to the *TimedStateMachine* meta-model we have defined in previous sections.

```
1  concept ExecutableTimedStateMachine(&M, &S, &T, &C) {
2    Model &M {
3      Node &S {
4        operation  isInitial () : boolean;
5        operation  isFinal () : boolean;
6      }
7      Node &T {
8        operation incoming() : &S;
9        operation outgoing() : &S;
10       operation canFire(symbol : int) : boolean;
11       operation action();
12     }
13     Node &C {
14       operation getValue() : int;
15       operation setValue(value : int);
16     }
17   }
18 }
19 bind ExecutableTimedStateMachine(TimedStateMachine, TimedStateMachine::State,
20                                  TimedStateMachine::Transition, TimedStateMachine::Clock)
21 requires "TimedStateMachineOps.eol"
```

Listing 7: Concept for binding semantics to a Timed State Machine

Listing 8 shows the core of the semantics for the state machine language. The operation uses the variable types defined in the previous concepts, and their operations. First, the initial state is set as the currently active state (line 4). In each iteration of the while-loop (lines 6–17), the next symbol on the input string is fed to the state machine. This is done by first finding all enabled transitions (line 9). If there is at least one enabled transition, one is chosen randomly and is fired, all clock values are incremented (lines 10–15) and the associated actions are triggered. If there is no enabled transition, the clock values are incremented (line 16). When all input symbols are consumed, the input is accepted if the active state is final, else it is rejected (lines 18–19).

```
1  @concept(name="ExecutableTimedStateMachine")
2  operation main() {
3    //....
4    current := &S.select( s | s. isInitial () ). first ();
5    ws := 0;
6    while (ws < input.size()) {
7      symb := input.at(ws);
8      ws := ws+1;
9      var x := &T.all().select( t | t.incoming()=current and t.canFire(symb) );
10     if (x.size()>0) {
11       currenttrans := x.random();
12       current := currenttrans.outgoing();
13       for (c in &C.all()) c.setValue(c.getValue()+1);
14       for (a in currenttrans.action) a.eval();
15     }
16     else for (c in &C.all()) c.setValue(c.getValue()+1);
17   }
18   if (current. isFinal ()) ('The word '+input+' is accepted'). println ();
19   else ('The word '+input+' is rejected'). println ();
20 }
```

Listing 8: Simulator for Timed State Machines

The use of these semantics requires the implementation of a number of operations, such as *isInitial*, *canFire* or *setValue*, as stated in the concept shown in Listing 7. Listing 9 shows implementations of some of the operations for the *TimedStateMachine* meta-model. Interestingly, such operations are defined modularly, over the different meta-models involved in the composition: the *StateMachine* meta-model, and the *TSM* meta-model. Listing 10 shows some operations needed by the *Evaluatable* concept referenced in Listing 1.

```
1  operation Transition canFire(symbol : Integer) : Boolean {
2    return (not self.symbol.isDefined() or self.symbol = symbol)
3      and (not self.condition.isDefined() or self.condition.eval()); }
4  operation Clock setValue(value : Integer) { self.time := value; }
```

Listing 9: Operations needed for state machine semantics.

```
1  operation Assignment eval() { self.op1.context.setValue(self.op2.eval()); }
2  operation Equals     eval() : Boolean { return self.op1.eval() = self.op2.eval(); }
```

Listing 10: Operations for the *Assignment* and *Equals* elements of *Expression*, required by the Evaluatable concept.

5. RELATED WORK

The problems related to domain-specific modelling and its malleability are becoming more and more relevant. It can be seen as a natural consequence of the ever increasing exploitation of DSLs for embracing the MDE vision in software development [12]. Therefore, in the latest years a number of research works have been devoted to improving language design processes and in particular proposing the adoption of language modularization and composition mechanisms.

The closest approach to the idea illustrated in this paper is presented in [12], where the authors propose a language embedding technique for creating families of DSLs. The abstract syntax is defined through meta-modelling, the textual concrete syntax is specified by means of a mapping between

metaelements and keywords, and the translational semantics is expressed by means of a proper transformation language. All the methodology relies on a "host" language (in this case Ruby) that has to own enough expressiveness to allow the definition of keyword extensions, scope restrictions, type and operator extensions and overrides. The main distinction with the technique discussed in this work is the expressive power of the template/binding mechanisms, which allow us to obtain extensive forms of language combinations. In contrast, in [12] merging and importing methods are exploited to address modularity.

In [14] the author discusses mechanisms for combining DSLs with a particular focus on multiview-based modelling approaches. In this respect, the combination of several viewpoints (i.e. DSLs) is called synthesized meta-model and is reached by means of a viewpoint unification technique. In particular, correspondences are drawn among different views and the unified result is obtained relying on them. Since correspondences can be differentiated as refinements, abstractions, equivalences, and implementations, DLSs can be combined in several interesting ways. However, the approach does not offer combination facilities comparable to the template/binding mechanism illustrated in this work, and only deals with abstract and concrete syntax combinations. Moreover, in general, merging graphical concrete syntaxes discloses additional and non-trivial issues intrinsic of iconic languages and implicit semantics associated to symbols [7, 14]. That problem is alleviated in our work thanks to the exploitation of textual concrete syntaxes.

DSLs composition through model weaving is proposed in [7]: new software architecture languages can be designed by composing existing ones by means of weaving links. Similarly to [14], semantics associated to weaving links allow creating merges, extensions, refinements, and specializations among architectural languages. Additionally, in this case combinations of language semantics are supported by adopting a reference "semantic core" to which all the concepts have to refer to. The main difference with respect to our proposal is the focus on the sole software architecture languages (that simplifies the problem of semantics combination). Moreover, weaving links semantics provide less expressive power if compared to the template/binding mechanism discussed in this work.

The use of templates in modelling is not new. They are already present in the UML 2.0 specification [13], as well as in approaches like Catalysis' model frameworks [9] and package templates, and the aspect-oriented meta-modelling approach of [3]. Interestingly, while all of them consider templates for meta-models or class diagrams, none consider concepts or model templates. Moreover, they do not consider concrete syntax or semantics.

6. CONCLUSIONS

In this paper, we have demonstrated the use of different composition mechanisms to define the abstract, concrete syntax and semantics of a language in a modular way. Abstract syntax and semantics are defined through meta-models and models, and therefore template models and concepts are used. The semantics are defined through an EOL program, defined over an hybrid concept, which gathers the entities an operations needed by the simulator, which should then be implemented for the types of the different meta-models involved in the composition.

In the future, we plan to apply this approach to further examples. We are also working in supporting more advanced textual concrete syntax and more advanced composition mechanisms allowing, e.g. a bidirectional binding for two templates or a more flexible binding.

7. REFERENCES

[1] R. Alur and D. L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235, Apr. 1994.

[2] J. Bengtsson and W. Yi. Timed automata: Semantics, algorithms and tools. In *Lectures on Concurrency and Petri Nets*, volume 3098 of *LNCS*, pages 87–124. Springer, 2004.

[3] T. Clark, A. Evans, and S. Kent. Aspect-oriented metamodelling. *The Computer Journal*, 46:566–577, 2003.

[4] J. de Lara and E. Guerra. Deep meta-modelling with MetaDepth. In *TOOLS'10*, volume 6141 of *LNCS*, pages 1–20. Springer, 2010.

[5] J. de Lara and E. Guerra. From types to type requirements: Genericity for model-driven engineering. *Software and System Modeling*, in press, 2011.

[6] J. de Lara and E. Guerra. Domain-specific textual meta-modelling languages for model driven engineering. In *ECMFA'12*, volume 7349 of *LNCS*, pages 259–274. Springer, 2012.

[7] D. Di Ruscio, I. Malavolta, H. Muccini, P. Pelliccione, and A. Pierantonio. Developing next generation adls through mde techniques. In *ICSE'10 (1)*, pages 85–94. ACM, 2010.

[8] J. Dingel, Z. Diskin, and A. Zito. Understanding and improving uml package merge. *Software and System Modeling*, 7(4):443–467, 2008.

[9] D. F. D'Souza and A. C. Wills. *Objects, components, and frameworks with UML: the catalysis approach.* Addison-Wesley Longman Publishing Co., Inc., 1999.

[10] D. Gregor, J. Järvi, J. G. Siek, B. Stroustrup, G. D. Reis, and A. Lumsdaine. Concepts: linguistic support for generic programming in C++. In *OOPSLA*, pages 291–310. ACM, 2006.

[11] D. S. Kolovos, R. F. Paige, and F. Polack. The Epsilon Object Language (EOL). In *ECMDA-FA'06*, volume 4066 of *LNCS*, pages 128–142. Springer, 2006.

[12] J. Sanchez Cuadrado and J. G. Molina. A model-based approach to families of embedded domain-specific languages. *IEEE Trans. Softw. Eng.*, 35(6):825–840, Nov. 2009.

[13] UML. `http://www.uml.org/`.

[14] A. Vallecillo. On the combination of domain specific modeling languages. In *ECMFA'10*, volume 6138 of *LNCS*, pages 305–320. Springer, 2010.