Efficient Development of Domain-Specific Simulation Modelling Languages and Tools

Andreas Blunk and Joachim Fischer

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

Abstract. This paper presents an approach which facilitates efficient development of domain-specific simulation modelling languages and tools for discrete-event systems. The work is motivated by a set of properties which in combination are not well supported by established frameworks. These include the provisioning of object-oriented description means, means for specifying domain-specific concepts with a distinct notation and semantics, the possibility of including general-purpose concepts into domain-specific ones, low cost tool support including an editor, a debugger, and a simulator, simulation primitives with fast execution, and extensibility means for enabling access to externally implemented simulation-specific functionality. We present a prototype that partly implements these properties. It combines established techniques derived from metamodel-based language development and extensible simulation modelling. The value is demonstrated by applying the approach to an example language from the domain of reactive systems and by comparing it to related approaches.

1 Introduction

While simulation modelling is as old as computational machinery, we are still learning how to best utilise it alongside modelling and experimentation. As the availability of low-cost computational power increases and the complexity of systems grows, the importance of simulation rises as well. The investigation of efficient, robust, explorative and problem area-adapted simulation methodologies is an important part of this activity.

Research in simulation methodology ranges from the development of efficient and effective algorithms, tools and programming languages, to the creation of new software engineering technologies, visualisation, data processing and storage methods, and even the philosophy and epistemology of simulation modelling [1].

1.1 Objective

In this paper, we argue in favor of an approach that allows to *efficiently develop domain-specific simulation modelling languages and tools* for discrete-time event-driven systems which in addition allow for fast simulations. The approach complies with two general desirable properties of simulation systems:

1. A system model should be expressed with structural and behavioural equivalence to the original system and
2. simulations should be executed efficiently.

In the next paragraph, we present accepted simulation systems which implement these two general properties and what we can learn from them.

1.2 Lessons Learned and Challenges

The introduction of model abstraction concepts by Simula [2], which later became known as *object-oriented modelling concepts*, is a major step towards achieving structural and behavioural equivalence [3]. With its class concept, Simula introduces the powerful principles of classification/exemplification and generalisation/specialisation. In addition, class instances (objects) can be divided into passive and active objects. Active objects are combinations of states and actions, which cause state changes in dependence of a model time.

However, object-orientation combined with expression means provided by a universal modelling language are not sufficient to concisely capture *domain-specific concepts*. This is because a universal modelling language defines a strict syntax in which domain-specific concepts have to expressed in. There are cases in which this preset syntax hinders application and understanding of a concept. An example is the implementation of state machines by an object-oriented pattern [4]. Applying this pattern results in a behaviour description spread over several classes which have to be created for each state of a state machine.

One possible solution to having a distinct syntax is to *extend* the syntax of a language combined with semantic foundations in the host language. The Simulation Language with Extensibility (SLX) [5] is a modern simulation language that has such possibilities, although they are limited.

Above all, we can see that a clear structure in the model alone is not causing a simulation language to be widely accepted. An assembly level language, whose many versions enjoyed great popularity since decades, is Gordon's GPSS (General Purpose Simulation System) [6]. The reason for its widespread use is (despite its structural weaknesses) the *efficient realisation of the next-event-scheduling* paradigm, which also takes so-called state events into account.

When we look at simulation languages today, we observe that both of these basic aspects can be incorporated in one language. This is achieved in languages like SLX, Modelica [7], and ODEMx [8], regardless of whether the modelled system is continuous or discrete in time.

Another aspect of simulation modelling is *experiment design*. Current research suggests that purely declarative ways of describing experiments [9] can already be sufficient to derive complete scientific experimentation workflows. These workflows are models of the experiment processes. They cannot only serve as specifications for repeated experimental procedures but also as inputs for workflow engines that trigger complex series of experiments, and that monitor and automatically evaluate their results. It is obvious that different descriptions are necessary:

(i) description of a parameterised simulation model,
(ii) description of the experiments,
(iii) description of the computational platform for execution and evaluation [10].

Equally desirable are *programmatic interfaces* to existing programming languages inside a special simulation language. They allow us to reuse statistical methods, solvers, or optimization methods that have already been implemented [11].

Self-standing solutions already exist for many of the above mentioned issues. However, there is no combined approach due to major technological differences. A recent development promising to facilitate the combination of such different aspects is object-oriented metamodelling (OOMM) [12]. In OOMM, the central part that connects all other aspects is an object-oriented metamodel. It defines the concepts of a language in an abstract way.

Other language aspects are defined with specialised language description languages, e.g. textual notation, static semantics, and execution semantics. For each such language aspect, tools can be derived automatically. Thus, OOMM is a key technology for creating domain-specific modelling languages (DSMLs or DSLs) and tools at an acceptable cost. In addition, DSLs allow models of dynamic systems to be expressed in a more concise way and with increased structural and behavioural equivalence than with object-oriented description means alone.

However, adding domain-specific concepts alone is not enough. One often needs to include or mix concepts of general-purpose programming languages, e.g. expressions and statements, with domain-specific ones. Therefore, we believe that an approach is required that allows to combine general-purpose as well as domain-specific concepts. These ideas are also described in [13], in which the authors propose to develop a unified approach that can be applied to combine different established concepts of modelling and programming. Our work also explores this direction of a combined approach. However, we believe that the efficient development of new language concepts and tools is a key requirement here.

1.3 Properties of Our Approach

Our approach exhibits a number of positive properties regarding the efficient development of domain-specific simulation modelling languages and tools.

It allows to create models with increased structural and behavioural equivalence by providing means for defining domain-specific concepts with a distinct notation and semantics. At the same time it allows to efficiently execute simulations, which make use of such concepts. This includes the possibility to attach already existing and efficiently implemented functionality, e.g. implementations of random number generators.

Adding domain-specific concepts to simulation modelling languages requires a frame in which they can be correctly applied. This frame can be provided by an object-oriented base simulation language. In addition, modelling concepts often make use of programming concepts. Therefore, an approach that allows to combine modelling and programming concepts is desirable.

However, for such an approach to be practical at all, a general requirement has to be fulfilled. The development of a new domain-specific simulation modelling language has to pay off, i.e. development effort including customised tools has to be low. The approach we present fulfills this requirement. It is based on a general simulation language, which can be dynamically extended by domain-specific concepts and which provides immediate support by a customised editor and simulator.

We summarise these positive properties, which our approach partly already implements, for further reference as follows. The approach exhibits

- (P1) object-oriented description means,
- (P2) means for specifying domain-specific concepts with a distinct notation and semantics,
- (P3) the possibility of including general-purpose concepts in domain-specific ones,
- (P4) low cost tool support including an editor, a debugger, and a simulator,
- (P5) simulation primitives with fast execution,
- (P6) and a programming interface enabling access to and from simulation-specific but externally implemented functionality with high efficiency.

1.4 Objective and Outline

We present a prototypical implementation of our approach by a framework named DMX (Discrete-Event Simulation Modelling Framework with Extensibility) [14] which implements properties (P1)-(P4) and partially (P5). DMX combines established techniques of metamodel-based language development and extensible languages. It consists of an extensible object-oriented base language and mechanisms for automatically deriving tools at a low cost. Property (P6) will be discussed as part of our future work. Furthermore, we confine our work to textual languages.

The paper is structured as follows. Section 2 gives an overview of related work that implements some of the properties. This motivates the presentation of our combined approach in Sect. 3. We introduce the concepts of the extensible base language and present the prototypical implementation of the framework. The value of the approach is demonstrated by applying it to an example language from the domain of reactive systems – a first use case on the path to more complex languages. In Sect. 4, we briefly discuss the fulfillment of each property and we compare our approach with two related ones in which we define the same example language. The comparison also takes into account development effort, which we identify as a general requirement of such approaches. We conclude the paper in Sect. 5 and discuss future work in Sect 6.

2 Related Work

In this section, we discuss related work which considers some of the proposed properties. Most of them have a more general focus and are not specifically designed for creating simulation languages. We give an overview of these works and point out their important characteristics.

The related work can be divided into 4 branches: extensible programming languages, extensible simulation languages, comprehensive DSL development frameworks, and further approaches for describing the execution semantics of metamodel-based languages.

Extensible Programming Languages became popular in the 1970s. The technique keeps being applied to state-of-the-art programming languages [15]. There is some recent works for the Java and the C++ language. An example is the Java Syntactic Extender (JSE) [16]. It is a pre-processor for Java allowing to add extensions of a few syntactic shapes. In addition, extensions may include certain Java constructs. Despite limitations in the syntax of extensions, customised modelling tools cannot be derived.

Extensible Simulation Languages are rare. The only such language that we know about is the Simulation Language with Extensibility (SLX) [5]. SLX is an object-oriented language that has a small but powerful C-like kernel language, in which constructs of the C language which are prone to error or primarily intended for systems programmers are excluded or restricted. On the contrary, discrete event simulation primitives for expressing concurrency, scheduling, and synchronization are added. In addition, the language can be extended by new statements and expressions with a distinct notation and semantics. The class of languages that is supported is a subset of regular languages. Semantics is defined as a mapping to SLX itself, which is the foundation for runtime efficient simulation execution. SLX has an efficient implementation of time-delays as well as state events and unconditional blockages with explicit reactivation. This is achieved by a specially developed compiler that, for simulation, has advantages regarding execution speed compared to compilers of general-purpose languages.

Comprehensive DSL Development Frameworks with the possibility of including general-purpose concepts into DSLs are a more recent trend. Outstanding representatives are Xtext [17] and the Meta Programming System (MPS) [18].

In Xtext [17], development starts with a concrete syntax from which a metamodel is derived. Semantics have to be described as a mapping to Java. This is achieved by writing a transformation which programmatically works on a Java program represented as an abstract syntax tree.

MPS [18] is more powerful than Xtext because extensions can be used jointly with DSL concepts. However, the editor is a projectional one that is unusual to operate. One does not enter the single characters that make up a construct, but instead has to choose from a set of possible constructs insertable at the current cursor position. Then the fixed textual parts of a construct are expanded and the cursor can be moved from one variable part to the next. For example, for a class construct, one can move from the name to other variable parts like attributes and operations and instantiate further constructs there.

Both frameworks automatically provide text editors. However, these are only available after a manual software generation step. The generation has to be initiated every time the language is changed. This manual step hinders rapid development of DSLs. Furthermore, both frameworks do not provide simulation primitives. These have to be made available manually by a simulation library.

There are Some Alternative Approaches for Describing the Execution Semantics of Metamodel-Based Languages. Some of them are based on operational semantics, e.g. MAS [12], M3Actions [19], and EProvide [20]. Runtime data and runtime states are described as a part of the same metamodel that also defines the abstract syntax of a language. Semantics is described by stepwise transformations of the runtime state. For such descriptions, programming languages like Java, but also UML activity diagrams or languages like Prolog and Haskell can be used. However, these approaches do not consider the necessity of runtime efficient executions of simulations. Furthermore, simulation primitives are not available and have to be added manually.

3 Approach

Our approach is based on a framework that combines an extensible object-oriented language with the immediate provisioning of essential tools at a low cost. In this section, we describe the basic concepts of the approach and its implementation. In the following, we present a use case that applies the approach to the definition of a state machine language. These explanations lay the foundation for a discussion of the approach regarding the fulfillment of the proposed properties in the next chapter.

3.1 Basic Concepts

Object-Oriented Concepts. At its core, the approach consists of a base language (BL) that includes object-oriented description means and a small set of essential simulation primitives. Its object-oriented features are confined to single inheritance between classes and multiple inheritance between interfaces combined with the well-established concept of type polymorphism. The base language and its concepts are similar to those of SLX. However, in contrast to SLX, the concepts of pointers to an object and an object as a value are not distinguished. Variables of type class are always handled as references to objects. This is a simplification derived from Simula and Java.

Simulation Concepts. The set of provided simulation primitives is short but sufficient. Concurrent processes are defined by the concept of an *active class.* Each such class defines the behaviour of its objects as a sequence of statements in an *actions part*. The behaviour starts when objects of active classes are explicitly *activated*. Further statements specify event-based process interactions: *advance* of model time, indefinite *waiting* of a process and *reactivation* by another one, *interruption* of a process and *rescheduling* it at a certain time, *yielding* control to another process, and *waiting for a certain condition* (defined as an expression accessing model structures) to become true (wait until). These are simulation primitives known to be sufficient for modelling all kinds of discrete-event systems. The provided simulation primitives are inspired by DEMOS [21] and SLX [5].

Domain-Specific Concepts. Domain-specific concepts are defined by specifying extensions to the base language. An extension specification consists of two parts. First, an extensions syntax is specified and then a mapping to concepts of the base language is defined.

Syntax Definition. The syntax is defined in an attributed BNF-like grammar language. A syntax definition consists of a set of grammar rules that extend the grammar of the BL. The first of these rules refers to the BL grammar rule that is extended by a new rule. Subsequent rules, consisting of terminals and non-terminals, define the concrete syntax of an extension. Each rule may refer to already existing BL rules, e.g. Statement and Expression, and thus reuse BL constructs. Non-terminals prefixed by a dollar sign designate references to already existing language constructs.

The class of languages that can be defined is a subset of context-free languages which can be defined by an LALR[1] grammar. Furthermore, semantic additions for specifying references between language constructs can be made by using the dollar sign in front of non-terminals.

In the definition of an execution semantics, the syntax parts have to be accessed. Therefore, syntax parts are prefixed by symbolic names. These names allow to access and evaluate the elementary or structured value of a part in semantics definitions. In addition, prefixed syntax parts define an implicit mapping to an abstract syntax. This abstract syntax is internally represented as a metamodel which extends the metamodel of the BL.

```
extension ForLoop {
   Statement -> ForLoop ;
   ForLoop -> "for" "(" it:$Variable "in" set:Expression "with"
      condition:Expression ")" "{"
         ManyStatements
      "}";
   ManyStatements -> ;
   ManyStatements -> statements:list(Statement) ManyStatements;
}
```

Listing 1. Syntax definition of a for-loop statement as an extension

[1] Look-Ahead Left to Right, Rightmost derivation.

An example is given in Listing 1. The extension defines the syntax of a for-loop as an additional kind of statement. In contrast, to the BL for-loop that iterates over all elements of a given set, the for-loop extension can be equipped with a condition expression which selects specific elements of the set. For the non-terminal $Variable, an identifier referring to an already existing Variable construct has to be supplied.

The example also shows syntax parts prefixed by symbolic names. An example is the condition expression which is prefixed by the name condition. In the semantics definition, the concrete condition can be accessed by this name.

In [22], we introduce the parts of the approach that deal with syntax extensions. We also show a prototype that implements this aspect. It allows to syntactically extend the concepts of a general-purpose language by domain-specific ones.

Semantics Definition. Models created in the BL are used for simulation. Therefore, domain-specific concepts have to define an execution semantics. The semantics of a concept is defined by a mapping to concepts of the BL. Furthermore, the semantics of the BL concepts are informally defined by a mapping to an executable target simulation language, i.e. an existing language for which there is already a compiler. This can be an exclusive simulation language like SLX but also a general-purpose language like Java combined with a simulation library. The sole requirement for the target language is that one can write a mapping for each of the concepts of the BL.

The semantics of an extension is defined in a *semantics part* right below the syntax part. The mapping is defined by a sequence of regular BL statements combined with a special gen statement. For each concrete use of an extension, these statements are executed. The target BL code is derived from executions of the gen statements. In the next step, the resulting target BL code is included at the exact place where an extension is used. In addition, there is a special statement for changing the current substitution context to other parts of a BL model. The concept is very generic in the way that arbitrary constructs of the BL can be referred to by their abstract syntax definition. As an example, an extension of type statement could add a class definition in a BL module that is required in the substitution code of the statement extension itself.

```
extension ForLoop {  } semantics {
   gen "for (" it ":" set ") {";
   gen "if (" condition ") {";
   for (Statement stm: statements) {
      gen "" stm ";";
   }
   gen "} }";
}
```

Listing 2. Semantics definition of a for-loop statement as an extension

In Listing 2 we revisit the example of the for-loop statement again and present the definition of its semantics. Executing this definition results in a regular BL for-loop in which the conditional selection of elements is simply implemented by encapsulating the statements of the for-loop body in an if statement. Syntax parts like it and condition are implicitly replaced by their concrete syntax representation when used inside semantics parts. An example use of the for-loop extension and the corresponding target code is depicted in Listings 3 and 4.

```
list(int) is; int i; for (i in
     is with i > 0) {
   print i;
}
```

Listing 3. Example use of the for-loop extension

```
list(int) is; int i; for (i:
    is) {
   if (i > 0) {
      print i;
   }
}
```

Listing 4. Resulting BL target code for an example use of the for-loop extension

3.2 Implementation

Editor. The BL editor is implemented by using the Textual Editing Framework (TEF) [12] and the Eclipse Modeling Framework (EMF) [23]. TEF is used for deriving a BL editor by defining the concrete syntax of the BL. EMF is used for the definition of a metamodel for the BL, which is required by TEF, and also for the extensions, which make additions to the BL metamodel.

The outstanding feature of the editor is its immediate awareness of the syntax of domain-specific extensions. This feature automatically derives a DSL editor at runtime. It offers well-established editor features like syntax highlighting and content assistance. Its implementation is feasible because TEF is based on the runtime parser generator RunCC [24], which can be supplied with changing versions of a grammar at runtime. This makes the implementation of a TEF variant feasible in which extension definitions are instantly recognised by the BL editor. For each extension, the grammar rules defined by the 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.

Simulator. The BL simulator is implemented by a mapping to an executable target simulation language. The simulator is derived by compiling and executing the target language program.

BL concepts as well as simulation concepts have to be considered in the mapping description. In a first prototype, a mapping to Java in combination with the simulation library DESMO-J is defined. The mapping is described in Acceleo [26], which is a template language implementing the OMG MOF Model to Text Standard [27]. This concrete mapping is 513 LOC[2] in size, composed of template statements and target code.

DSL simulators are derived by substituting all the extensions with BL concepts as defined in their semantics parts. At the end of this process, the resulting model solely consists of BL concepts. In the final step, the model is mapped to the target simulation language that is used for execution. An overview of the whole compilation workflow is available online [14].

A major problem in the whole transformation process is the rather static nature of the used metamodeling framework EMF, as explained in [22]. In summary, it is hard to have a changing metamodel be supported by EMF at runtime. Because of this problem, the time for executing a transformation is rather long. For the simple for-loop example, it takes around 8 seconds to transform, compile, and execute a corresponding model on a high-end computer[3]. Initiating the EMF generation process for creating all the Java classes, which have to be present for metaclasses, takes most of the overall execution time.

3.3 An Example Language – State Machines

We successfully apply the approach to the definition of state machines as an example language, which we refer to as SML. The simplified use case of this language is designed as a preliminary study for the development of a language for industrial workflows in the field of supply chain management.

SML is defined by a set of domain-specific extensions. The language is an enhanced version of the one presented in [22]. It is a subset of state machines as defined by the UML [28]. The subset includes simple states, initial and final states, transitions with signal, completion, and time event triggers, and also guards and effects. The semantics of event processing is implemented as run-to-completion as defined by UML. SML state machines can be used to define behaviour inside an active class of the BL. They can also refer to properties of their enclosing class.

An example, which defines the structure and the behaviour of a simple counter, is depicted in Listing 5. After each time step, a count variable is increased until a certain limit is reached. In addition, the counter may be started, paused, and resumed by external signals. Signals are send to objects by a special send statement, which is also defined as an extension.

The syntax definition is similar to the one presented in [22]. The new semantics definition has been added. It is defined by using the core simulation constructs of the BL. An excerpt of the semantics definition is depicted in Listing 6. The complete definition of SML is 158 LOC in size. It is available online [14].

The main idea in the semantics definition is to add a variable of type list of object to the class containing the state machine. This variable serves as an event

[2] Description effort is measured in lines of code (LOC), excluding comments and blank lines.

[3] Intel Core i7 2.6 GHz processor, 8 GB main memory, and a Solid State Hard Disk.

pool into which signal and time events are placed. In the next step, the behaviour is implemented by waiting for events to arrive in this set. Each event is processed one after the other, including the evaluation of guards and the execution of effects. For time events, a special active class Timer is created by using the special statement setGenContext. It allows to change the current generation context inside semantics parts. This concept allows to add constructs in other places of the enclosing model in which an extension is used. In the example, the class Timer is created in the same module in which the state machine is used. Objects of class Timer represent time events. When a time event occurs, a Timer object is placed into the corresponding event pool.

```
class Start {} class Pause {} class Resume {} class Reset {}


active class Counter {
   int count;
   int limit = 10;
   int step = 1;


   stateMachine CounterBehaviour {
      initial -> StandBy;
      state StandBy (
         Start / { count=0; } -> Active,
         Resume -> Active
      );
      state Active (
         [count >= limit] / { trace("Finished."); } -> final,
         after(step) [count < limit] / {
            count=count+1; trace("Tick " + count); } -> Active,
         Pause -> StandBy,
         Reset / { count=0; } -> Active
      );
   }
}

void main() {
   Counter c = new Counter;
   activate c;
   send new Start to c;
   advance 20;
}
```

Listing 5. Counter state machine as an example extension use

```
semantics {
      gen "list(Object) eventPool; string currentState;";
...
      gen "actions { ... while (currentState != null) {  ";
...
      gen "  empty eventPool;
         wait; ...
         while (eventPool.size > 0 and currentState != null) {
            Object ev = eventPool.first;
            remove ev from eventPool; ";

      for (State state: states) {
         gen "if (currentState == \"" state.name "\") {";
         for (Transition tr: state.outgoing) { ...
            if (tr.effect != null) {
               for (Statement stm: tr.effect.getStatements()) {
                  gen "" stm ";";
               }
            }
...

      ClassContentExtension ext = self;
      Clazz clazz = ext.eContainer() as Clazz;
      Module mod = clazz.eContainer() as Module;

      setGenContext after mod.getClassifiers().first;
      gen "active class Timer {
         " clazz.getName() " sm; ...
         actions {
            advance delay; ...
            place self into sm.eventPool; ...
         }
      } ";
}
```

Listing 6. Excerpt of the semantics definition for the state machine extension

4 Discussion

In this section, we discuss to what extent the proposed properties are present in our approach. We also compare the approach to two related ones, namely SLX [5] and Xtext [17]. We investigate to what extent each property is present in each approach. We relate this investigation to the general requirement of having low development effort for domain-specific concepts including customised tools. We measure description effort in LOC and point out characteristics of each approach.

4.1 Object-Orientation (P1)

The base language (BL), which is included in DMX, is object-oriented. Thus, property (P1) is present. Object-orientation is restricted to single class inheritance. This kind of object-orientation is used in various established modelling languages like Simula, SLX, and the System Description Language (SDL) [29]. Although single inheritance seems to be sufficient, other languages like Modelica and UML [28] provide multiple inheritance instead.

However, UML [28] does not define a precise semantics and leaves this part unresolved as a semantic variation point. Problems arise when multiple implementations of the same kinds of elements are inherited. Modelica solves this problem by merging the contents of the base class and the derived class. Thus, similar elements become one. This is a feasible solution in Modelica because operations, which can be a source of ambiguities, cannot be defined as parts of classes but only as functions on a global level. Therefore, ambiguities as a result of inheriting the same operation multiple times with different implementations cannot occur. Multiple inheritance can be a helpful feature. However, further investigation is required of how it should be supported to be a helpful instrument.

Xtext includes a base language named Xbase, which defines a large set of expressions and statements. The semantics of Xbase is defined as a mapping to Java. Therefore, Xbase also takes over Java's type system, i.e. single class and multiple interface inheritance. However, object-oriented descriptions means for defining structures like classes and relations are not present in Xbase directly. These have to be defined in Java and then referenced from languages defined with Xtext and Xbase.

Another aspect of the BL is its simplification of the type concept for class typed variables to object references only. This simplification exempts the modeller of considering runtime efficiency aspects, i.e. whether an object should be placed on the stack or on the heap. This kind of decision is intentionally left to an optimizing compiler. In SLX, there is no such simplification. In Xtext, as a result of its Java-based semantics, the same simplification is present.

4.2 Domain-Specific Additions (P2 and P3)

Modelling with increased structural and behavioural equivalence is achieved by the extension concept. It allows to define domain-specific concepts with their own notation and execution semantics (P2). The advantage of an extensions-based approach is that concepts of the base language can be included into domain-specific ones (P3). In addition, BL concepts and extensions can be used jointly within the same model.

In SLX, domain-specific concepts of two kinds can be defined with their own notation and semantics: statements and expressions. The syntax definition of such concepts is bound to a subset of regular languages, which is not suitable for complex DSLs. Therefore, property (P2) is only present in parts.

In addition, DSL concepts can only include SLX expressions, but they cannot include other kinds of contructs like SLX statements. This is a major obstacle in defining effects in the example language SML. In SLX, one has to code them as strings. Therefore, property (P3) is only present in parts as well. The execution semantics of an extension is defined by a sequence of SLX statements in combination with a special expand statement. They define a mapping to the SLX core language. The target code is derived by executing this mapping for each extension instance and replacing it by its target code. The availability of simulation primitives make the description of an execution semantics surprisingly short. The complete definition of SML is 103 LOC in SLX.

In DMX, the set of languages that can be defined is more comprehensive. It already allows to define such complex languages as state machines. Important features making this definition feasible is the support of context-free languages in extensions. In addition, DMX includes a special statement setGenContext which allows to change the current generation context inside semantics parts. This concept allows to add constructs in other places of the enclosing model in which an extension is used. Such constructs can be helpful when defining semantics. As an example, in the semantics definition of SML, a class Timer is created in the same module in which a state machine is used. In the semantics definition the class Timer is used in order to define the semantics of time events. In contrast, SLX only allows to create constructs at the same place in which an extension is used, which limits the possibilities of defining certain semantics. The size of the SML definition is with 158 LOC in DMX compared to 103 LOC in SLX slightly larger. Yet, language definition as well as editor support are more comprehensive as well.

In Xtext, DSLs in the set of context free languages can be defined, thus property (P2) is present. In addition, general-purpose constructs like expressions and statements can be added to a language, which is property (P3). However, there is a major problem with Xtext: domain-specific additions cannot be embedded into regular Java programs and thus cannot extend the Java language. In the semantics description, expressions have to be positioned in a suitable place in the resulting Java code, where they can be correctly evaluated. As Java does not contain simulation primitives, this part of the mapping has to be defined by using a Java-based simulation library. In this comparison, DESMO-J [25] is used. The definition of SML in Xtext consists of 373 LOC (15 LOC for the syntax and 358 LOC for the semantics definition).

In DMX, domain-specific concepts can be directly embedded into a BL model. In addition, they can be used jointly with the BL. The size of the syntax definition of SML is with 25 LOC in DMX compared to 15 LOC in Xtext slightly larger. However, this can be explained by Xtext providing EBNF and our own approach providing BNF for syntax definition. With the semantics definition it is different. In Xtext the size is 358 LOC, which is more than double the size of the definition in DMX (158 LOC). There are multiple reasons for this increase in description size in Xtext: i) the programmatic construction of target code as a Java abstract syntax tree, ii) the missing integral part of simulation primitives, and iii) the need to embed BL elements like statements and expressions into a suitable evaluation context. In addition, semantics is defined in Xtend, which in comparison to Java includes many simplifications to Java as well as additions of high-level concepts like lambda expressions. This makes descriptions considerably more concise than comparable Java code.

The use of BNF notation in DMX syntax parts could be reduced by adding Extended BNF (EBNF) description means. Because EBNF is already defined as an extension of BNF, it could be made available by the very same principles of our approach. However, such an extension is of a different kind. In order for the editor to recognise EBNF-based syntax definitions, it is required to carry out extension substitution for the semantics part at runtime as well. Currently this is only implemented for the syntax part.

4.3 Low Cost Tool Support (P4)

In our approach, there is immediate tool support with an editor and a simulator. This makes the definition and the application of a domain-specific concept with a distinct notation as simple as writing an ordinary method. Editor features like syntax highlighting and content assistance, available for methods, are equally present for domain-specific extensions. All of these tools are provided at a low cost because the description effort required for defining syntax and semantics is low in comparison to SLX and Xtext. Thus, property (P4) is present.

In SLX, there is an integrated programming environment including an editor, a compiler, a launcher, and a debugger. These tools also support defined extensions. However, the syntax of extensions is not immediately recognised. Instead, the syntax is highlighted after a complete program has been compiled successfully. Expressions and statements used in extensions have to be supplied as unchecked strings.

In Xtext, a textual DSL editor can be generated from a DSL description. It features syntax highlighting and content assistance. In contrast to SLX, there is even support for expressions and statements used in extensions. There is a generic builder, which executes the mapping description and automatically compiles the resulting Java code. The Java representations of DSL constructs are available in regular Java programs. However, the development process is rather slow because tools have to be generated first before they can be applied. Especially in an iterative process, this kind of development is time-consuming. In addition, description effort is rather high.

In comparison to SLX, tool support by an editor is more comprehensive in DMX. Editor features likes syntax highlighting and content assistances are equally available as in Xtext. However, in DMX, an editor is immediately available for domain-specific concepts. In addition, the description effort required is lower than in Xtext.

In each approach, a compiler for domain-specific extensions is automatically derived from semantics descriptions. The compilers differ in compilation time. In DMX, compiling a BL program which includes extensions is rather slow at the current stage of implementation. Compiling the example counter state machine (Listing 5) takes around 10 seconds.

4.4 Simulation Primitives with Fast Execution (P5)

In DMX, runtime efficient executions are preserved by efficient implementations of the BL simulation primitives. This part is delegated to the selected simulation target language. For some of these languages efficient implementations of compilers already exist. An impressive example is SLX. By writing a mapping of the BL to SLX, one can benefit from fast executions combined with an increased expressive power in defining domain-specific extensions. Also, extensions benefit from fast executions because their semantics are defined using the very same concepts of the BL.

The same argument holds for simulation libraries written in Java, although they may not be as runtime efficient. An example with poor execution speed is DESMO-J. That is because its coroutine implementation is based on Java threads. However, there are more efficient libraries. A library which is prominent in the network simulation community is JiST [30]. It implements coroutines by Java Byte Code rewriting which makes it faster than DESMO-J. A mapping to JiST, which would be similar to the one already implemented for DESMO-J, can result in a viable simulator as well.

Execution time of SLX and Xtext/DESMO-J is measured in an experiment[4] with an example model which includes two counters (as presented in Sect. 3.3), limited to 10^6 counts. In SLX, the simulation is finished after 0.06 seconds. In Xtext/DESMO-J, it takes 186 seconds to complete. The corresponding DMX model executes in 144 seconds when using DESMO-J. There is a slight increase in execution time in the Xtext-based SML. This might be because the state machine semantics for processing events are implemented in an object-oriented way by following a state machine pattern [4]. In contrast, the semantics of the DMX-based SML is defined by determining the current state with a number of simple if statements.

Furthermore, the execution time of DMX could easily be increased to the same time as pure SLX by defining a BL-to-SLX transformation. Thus, DMX could benefit from the very efficiently implemented SLX simulation core. In addition, DMX models already created can be used without any changes.

This is only a first measurement with a simple example which focusses on process switching times. Depending on the concrete model, other aspects might be more important. Nevertheless, the example can serve as a first indicator of execution times.

[4] Intel Core i7 2.6 GHz processor, 8 GB main memory, and a Solid State Hard Disk.

4.5 Externally Implemented Functionality (P6)

The possibility of connecting different target simulation languages (mentioned in Sect. 4.4) is important for several reasons. It is important i) for creating models which are independent of some current state-of-the-art simulator platform, ii) for running simulations with the most efficient simulator platform available, and iii) for integrating external functionality provided by libraries or tools without too much effort. The last remark can be an implementation of property (P6).

Although we have not yet investigated this aspect in depth, we believe that the BL to target language mapping already offers a good solution. It should be feasible to access external functionality implemented by tools written in the target language with not much effort. One can declare a BL function as native in which calls to this function are forwarded to their implementation in the target language. A prerequisite is that an external tool has to offer an interface to its functions accessible in the chosen target language.

In SLX, external functions implemented in C/C++ can be invoked. This is achieved in a number of steps. A model has to 1) declare a function as natively available via a Dynamic Link Library (DLL), 2) generate a C/C++ header and implementation file, 3) implement the function in C/C++, and 4) compile it as a DLL so that it can be accessed from a SLX model.

In Xtext, external functions implemented in Java are instantly accessible because its semantics are already defined as a mapping to Java.

5 Conclusions

We present an approach exhibiting a number of properties which are important in order to develop domain-specific languages used in simulation in a more efficient way. This includes the development of the language as well as its tools. We measure description effort of our approach for defining a state machine language and present the derivation of tools like an editor at a low cost. In comparison to related approaches, description effort and the cost for having tools is reduced while maintaining the expressive power and execution efficiency required for domain-specific simulation languages. Further research has to show if languages that are even more complex can be developed analogously.

6 Future Work

The provisioning of debuggers is an area of special interest to us. Currently, debuggers still have to be implemented by hand, e.g. for simulation languages like SLX. Furthermore, dedicated debuggers do not even exist for simulation libraries written in programming languages. We believe that our approach can be extended to an immediate provisioning of DSL debuggers (part of P4). In [31], we already describe the debugging aspect of a DSL, so that a debugger can be derived automatically. The work is based on EProvide, which is a framework for defining the execution semantics in an operational way. However, other approaches [32] show that such tools could also be derived for execution semantics described as transformations.

The second area of interest is investigating the integration of externally available functionality as described in Sect. 4.5.

References

1. Law, A.M.: Simulation Modeling and Analysis. McGraw-Hill (2007)
2. Dahl, O.J., Nygaard, K.: SIMULA: an ALGOL-based simulation language. Communications of the ACM 9(9), 671–678 (1966)
3. Møller-Pedersen, B.: Scandinavian Contributions to Object-Oriented Modeling Languages. In: Impagliazzo, J., Lundin, P., Wangler, B. (eds.) HiNC3. IFIP AICT, vol. 350, pp. 339–349. Springer, Heidelberg (2011)
4. Shalyto, A., Shamgunov, N., Korneev, G.: State Machine Design Pattern. In: NET Technologies 2006 Short Communication Papers Proceedings, pp. 51–58 (2006), `http://dotnet.zcu.cz/NET_2006/Papers_2006/!Proceedings_Short_Papers_2006.pdf`
5. Henriksen, J.O.: SLX – The X is for Extensibility. In: Proceedings of the 32nd Conference on Winter Simulation (WSC 2000). Society for Computer Simulation International (2000), `http://informs-sim.org/wsc00papers/027.PDF`
6. Gordon, G.: The development of the General Purpose Simulation System (GPSS). ACM SIGPLAN Notices 13(8), 183–198 (1978)
7. Fritzson, P.A.: Principles of Object-Oriented Modeling and Simulation with Modelica 2.1. John Wiley & Sons (2004)
8. Fischer, J., Ahrens, K.: Objektorientierte Prozeßsimulation in C++. Addison-Wesley (1996)
9. Bowers, S., McPhillips, T., Ludäscher, B., Cohen, S., Davidson, S.B.: A Model for User-Oriented Data Provenance in Pipelined Scientific Workflows. In: Moreau, L., Foster, I. (eds.) IPAW 2006. LNCS, vol. 4145, pp. 133–147. Springer, Heidelberg (2006)
10. Kühnlenz, F., Fischer, J.: A Language-centered Approach for Transparent Experimentation Workflows. In: Proceedings of the CSSim 2011-Conference on Computer Modelling and Simulation (2011), `http://metrik.informatik.hu-berlin.de/grk-wiki/index.php/Expwf`
11. Zeigler, B.P.: Theory of Modeling and Simulation, 2nd edn. Academic Press (2000)
12. Scheidgen, M.: Description of Computer Languages Based on Object-Oriented Meta-Modelling. Doctoral Thesis, Humboldt University Berlin (2008), `http://www2.informatik.hu-berlin.de/~scheidge/downloads/thesis.pdf`
13. Madsen, O.L., Møller-Pedersen, B.: A Unified Approach to Modeling and Programming. In: Petriu, D.C., Rouquette, N., Haugen, Ø. (eds.) MODELS 2010, Part I. LNCS, vol. 6394, pp. 1–15. Springer, Heidelberg (2010)
14. Blunk, A.: Discrete-Event Simulation Modelling Framework with Extensibility (DMX), `http://ablunk.github.com/dmx`
15. Zingaro, D.: Modern Extensible Languages. McMaster University (2007), `http://www.cas.mcmaster.ca/sqrl/papers/SQRLreport47.pdf`
16. 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 (2001)

17. Itemis AG: Xtext, http://www.eclipse.org/Xtext/
18. JetBrains: Meta Programming System (MPS), http://www.jetbrains.com/mps
19. Soden, M., Eichler, H.: An Approach to use Executable Models for Testing. In: Enterprise Modelling and Information Systems Architectures Concepts and Applications (EMISA 2007). Lecture Notes in Informatics, vol. P-117, pp. 75–86. Gesellschaft für Informatik (2007), http://subs.emis.de/LNI/Proceedings/Proceedings119/gi-proc-119-006.pdf
20. Sadilek, D.A., Wachsmuth, G.: Prototyping Visual Interpreters and Debuggers for Domain-Specific Modelling Languages. In: Schieferdecker, I., Hartman, A. (eds.) ECMDA-FA 2008. LNCS, vol. 5095, pp. 63–78. Springer, Heidelberg (2008)
21. Birtwistle, G.: DEMOS – a System for Discrete Event Modelling on Simula. Springer (1987)
22. Blunk, A., Fischer, J.: Prototyping Domain Specific Languages as Extensions of a General Purpose Language. In: Haugen, Ø., Reed, R., Gotzhein, R. (eds.) SAM 2012. LNCS, vol. 7744, pp. 72–87. Springer, Heidelberg (2013)
23. Eclipse Foundation: Eclipse Modeling Framework, http://www.eclipse.org/modeling/emf
24. Ritzberger, F.: RunCC – Java Runtime Compiler Compiler, http://runcc.sourceforge.net
25. Page, B., Kreutzer, W.: The Java Simulation Handbook: Simulating Discrete Event Systems with UML and Java. Shaker Verlag (2005)
26. Acceleo: Transforming Models into Code, http://www.eclipse.org/acceleo
27. Object Management Group: MOF Model To Text Transformation Language (MOFM2T), 1.0 (2008), http://www.omg.org/spec/MOFM2T/1.0/
28. Object Management Group: Documents Associated With Unified Modeling Language (UML), V2.4.1 (2011), http://www.omg.org/spec/UML/2.4.1/
29. International Telecommunication Union: Z.100 series, Specification and Description Language, http://www.itu.int/rec/T-REC-Z.100/en
30. Barr, R.: An efficient, unifying Approach to Simulation using Virtual Machines. Doctoral Dissertation, Cornell University (2004), http://jist.ece.cornell.edu/docs/040517-thesis.pdf
31. 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)
32. Wu, H., Gray, J., Mernik, M.: Grammar-driven Generation of Domain-specific Language Debuggers. Software – Practice & Experience 38(10), 1073–1103 (2008)