Model-driven Development of Discrete Event Simulations

Uğur Aksu, *Graduate Student, Bilkent University,*
Cem Mergenci, *Graduate Student, Bilkent University,*
and Kaan Onarlıoğlu, *Graduate Student, Bilkent University*

Abstract—Simulation is a common tool for modeling and evaluating real-life systems. In discrete event simulation (DES) a system is represented from the perspective of its events. Modern tools for developing simulations are bound to general purpose programming concepts that cannot represent the simulation domain effectively. In this paper, we propose a model-driven approach to DES domain. We provide a meta-model from scratch based on Ecore, an alternative one that utilizes UML 2.* profiling and a domain specific language grammar. Two example models demonstrate the use of our modeling language and provided model-to-text and model-to-model transformations to Java and DOT models, respectively, show that DES can effectively be expressed by a model-driven approach.

Index Terms—MDSD, Discrete Event Simulation.

1 Introduction

Simulation is at the heart of many aspects of engineering allowing evaluation of the systems without any need for implementation. Discrete event simulation (DES) is a simulation technique which focuses on events occurring on the system which mark points of time where state of the system is updated. DES is suitable for systems that are easy to define from an operational perspective.

Since DES is a widely used method, there are many application frameworks for programming discrete event simulations. However, most of these frameworks are based on general programming paradigms and tools, and do not allow programmers to develop their simulations using the domain concepts. This makes it hard to focus on the simulation logic and business rules to be simulated since developers are required to deal with the constraints, restrictions and programming styles dictated by the programming language used.

In this study, we aim to provide a modeling language that captures domain concepts as basic abstraction elements for model-driven development of simulation systems and define the transformations for generating code and different models. We provide the transformation rules for generating code to work with SimKit, a popular Java-based simulation programming framework and also transform our models to DOT models used for defining graphs. Consequently, we alleviate the problem of using general purpose programming constructs by building executable models.

Rest of the paper is organized as follows: Section 2 presents a detailed analysis of DES domain and lists a vocabulary of common concepts. Section 3 defines a meta-model consisting of an abstract syntax and static semantics capturing domain concept relations and a visual concrete syntax based on event graphs. Two real-world example models are also provided to illustrate the use of defined modeling language. Section 4 gives an alternative meta-model extended from the UML 2.* meta-model by profiling. Section 5 describes a textual domain specific language grammar corresponding to given meta-model. Section 6 and 7 present the transformations from our DES models to Java code and to DOT models, respectively. In section 8, we discuss the lessons learned in this study and reflect on our experience and we conclude the paper in section 9.

2 Domain Analysis

In this section we present discrete event simulation domain and describe the key concepts relevant to our study.

2.1 Simulations & Discrete Event Simulation

Simulation is the process of formally describing a real-life system in order to experiment with mod-

els of the entities constituting it, observe the behavior of the system, control sources of variation and investigate their effects on system operation. Simulations make it possible to study, review and analyze a system and its components, explore alternate operation strategies and predict their effects on behavior, all without actually producing the system.

Discrete-Event Simulation (DES) is one of the many ways in the system modeling taxonomy to simulate a system, in which the operation of a system is represented as a chronological sequence of events. The main characteristic of a Discrete-Event System is that events, which mark significant changes in the system state, occur at discrete points in time; as opposed to Continuous-Time Systems where the system evolves continuously in time. Major strengths of DES over other simulation methods is its ability to stop and review a system at a particular point of time, restore the system state and model random events.[1]

Discrete-Event Systems can be described using several different ways, called *world views*. In Activity-Oriented View, time is broken into small increments and at each time increment, the system model is checked to see whether there are new events to process. Although this approach is very easy to understand and implement, it is inefficient since no events occur at most time increments. In another approach, Process-Oriented View, the system's life-cycle is represented by a sequence of activities. This approach is effective and efficient; however, it usually requires detailed planning and organization to implement. The final world view is the Event-Oriented View, where events and event transitions are the basic elements of the model and the system always advances to the occurrence of the next event. This approach offers good efficiency and it is easy to realize. Today, Process-Oriented and Event-Oriented Views are the most popular approaches, whereas Activity-Oriented View is not common.[6]

A traditional Event-Oriented modeling language is Event Graphs, where each node in the graph represents an event and a directed graph edge between a source node and a target node denotes scheduling of the target event when the source event occurs.[7] Details of the event graph notation are presented in section 3.2.

2.2 DES Lexicon

Key concepts in DES are presented and explained in detail below.

- *Global Timer*: The global timer specifies the start and end time of the simulation, and keeps track of the simulation time.
- *System State* (or State): State is a collection of variables that accurately and fully describe the system simulation model at a given time.
- *Event*: Events cause significant changes in the system model at their particular scheduled occurrence times; more specifically, they modify the system states, schedule new events or cancel pending events. An event "occurs" when the simulation is at the scheduled time of the event.
- *Event Queue*: Event queue is a time-ordered list of pending events.
- *Event Scheduling*: Event scheduling means inserting a new event into the Event Queue. Events can be scheduled prior to the execution of the simulation according to the modeler's intend, scheduled when certain events occur (specified by the model) or can be generated randomly. Event scheduling may depend on certain conditions specified by the model.
- *Event Canceling*: Event canceling means deleting a particular event from the Event Queue, before it can occur. Events can be cancelled as the result of occurrence of certain other events (specified by the model).
- *Ending Condition*: Ending condition determines when the simulation terminates. The simulation can stop its execution either when the execution time or a state has a particular value. Note that, the event queue becoming empty also implies the end of the simulation since no new events can occur.

3 Metamodel From Scratch

In this section, we provide a meta-model based on Ecore for modeling DES.[4]

3.1 Abstract Syntax

Our meta-model is defined to model event graphs that are a common way to express DES. Figure 1 shows the abstract syntax based on Ecore. Model is the main entity in the diagram. It has a GlobalTimer, an EventGraph, an EventQueue and States. EventGraph consists of EventTypes and transitions which are either normal event scheduling transitions or event canceling transitions. Scheduling transitions can have a Boolean condition that determines whether or not target event is going to

Fig. 2. Basic event graph

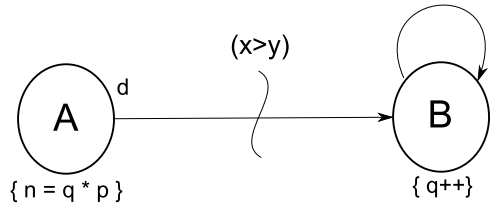

Fig. 3. Additional features of event graphs

be scheduled. Events have statements that are executed when the event is scheduled to run and these statements update the system state.

3.2 Concrete Syntax

The concrete syntax for our DES model is based on the traditional event graph model, where the nodes in the graph represent event types and the directed edges connecting them are event scheduling or cancelling transitions.

The basic form of an event graph is illustrated in Figure 2. In this construct, graph nodes representing the event types are denoted by circles, and the directed edge between two event types is depicted by an arrow. Note that each event type is identified by its name written in the node circles. In Figure 2, the edge between event types *A* and *B* means that, whenever an event of type *A* occurs, a new event of type *B* is scheduled, in other words inserted into the event queue.

We may use additional notation to enhance the basic construct in Figure 2 and make our models more expressive. The improved construct, as seen in Figure 3, demonstrates the additional capabilities of the model. A Boolean expression positioned at the top of the transition arrow from A to B between parentheses specifies a condition which must hold when A occurs, in order for B to be scheduled; if otherwise, B will not be inserted into the event queue. An integer value located next to the source end of a transition arrow specifies a delay value 'd', which means that if A occurs and all the conditions for B to be scheduled hold, then B will be scheduled to occur after a delay of 'd' simulation clock ticks. When this value is not specified, a delay of zero is assumed and B is scheduled to occur immediately.

Fig. 4. Using canceling edges

Fig. 5. Defining the start event

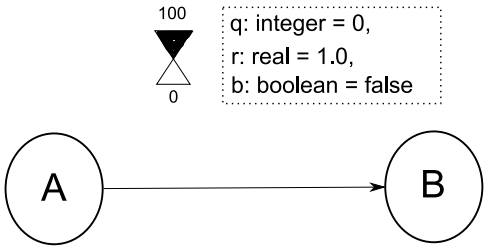

Fig. 6. Start and finish times with initial values of states

Finally, the statements written below each event node between curly parentheses represent the state changes that will take place when that event occurs. Any number of statements could be specified for each event type.

Another type of edge in the event graph model is the event canceling transition denoted by a dashed arrow as seen in Figure 4. The effect of the event canceling transition in the given model is interpreted as follows: Whenever an event of type A occurs, the first scheduled event of type B is cancelled, that is, the first occurrence of event B will be removed from the event queue. If no such event exists in the queue, then the canceling has no effect. Note that, canceling transitions cannot have a delay or a condition.

Each DES modeled by an event graph needs to specify a start event, which is implicitly scheduled to occur at the simulation start time to bootstrap the system model. In our model syntax, it is depicted by an event node with an additional smaller ring inside, as illustrated in Figure 5.

The start and end time for the simulation is shown besides the model with a figure representing an hour glass, in Figure 6. A simulation usually starts at time 0, but it is possible to specify a different start time. The model must also specify the state variables used in the simulation, their types and initial values, which is accomplished by listing

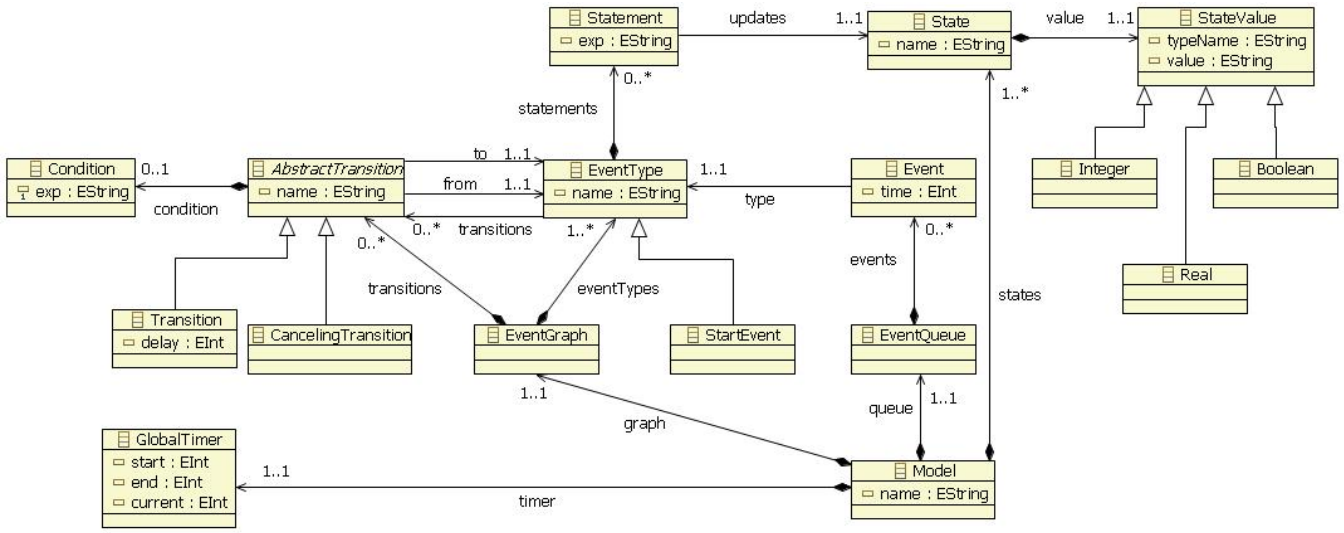

Fig. 1. DES metamodel constructed with Ecore

them in a table, also shown in Figure 6.

3.3 Static Semantics

Following well-formedness rules in OCL defines the static semantics for DES meta-model.

```
package DES

context StartEvent
inv: StartEvent::allInstances()->size()=1
inv: time = 0

context Simulation
inv: Event::allInstances()->forAll(e |
      e.time >= self.timer.current)

context Transition
inv: delay >= 0

context Event
inv: Event::allInstances()->isUnique(name)

context State
inv: State::allInstances()->isUnique(name)

context Integer::value : Integer
init: value = 0

context Real::value : Real
init: value = 0.0

context Boolean::value : Boolean
init: value = false

context GlobalTimer
inv: start <= end
inv: end >= current
inv: current >= start

endpackage
```

3.4 Example Models

In this section, we demonstrate features of our modeling language with two example cases.

3.4.1 Market Simulation Model

A market has two types of cashier's desks, regular and express. Likewise, there are two types of customers, regular and express customers, each having different service and interarrival times, with the assumption that interarrival times for regular customers are greater than interarrival times for express customers. Regular customers and express customers wait in different lines. A regular customer is served by a regular cashier's desk if there are idle regular cashier's desks. A regular customer can also be served by an express cashier's desk if all the regular cashier's desks are busy and there are available express cashier's desks. Regular customers join the end of the regular line if all the cashier's desks are busy. Similarly, an express customer is served if there are any express cashier's desk available. Otherwise, express customers join the end of the express line. (See Figure 7)

- *Qr*: Number of regular customers.
- *Qe*: Number of express customers.

- *Sr*: Number of regular cashier's desks.
- *Se*: Number of express cashier's desks.
- *Tar*: Interarrival time for regular customers.
- *Tae*: Interaarival time for express customers.
- *Tsr*: Service time at regular cashier's desks.
- *Tse*: Service time at express cashier's desks.

3.4.2 Car Washing Service Simulation Model

At a car washing facility there is one car washing machine with service time *Ts*. Customers arrive with interarrival time of *Ta* to have car washing service, waiting in one line. First customer in the line gets the service if the car washing machine is idle and its status is operating. Car washing machine fails to operate periodically with the failure period of *Tf* regardless of how long it has been working. Upon failure, the car being washed is returned back to the queue. Machine starts operating again after being fixed with the repair time of *Tr*. (See Figure 8)

- *Ta*: Interarrival times of cars.
- *Ts*: Services times.
- *Tf*: Periods between failures of machine.
- *Tr*: Periods between repair times of machine.
- *isIdle*: true/false if machine is idle or busy
- *failure*: true/false if machine is in failure status or is operating.

4 Metamodel with UML Profiling

UML profiling is another method to define a meta-model where domain concepts are described by extending from the UML meta-classes.[8] Since we are profiling for the same domain, concepts and relations are the same as in the meta-model built from scratch.

Event, AbstractTransition, GlobalTimer and State stereotypes are extended from UML meta-class Class. Statement, StateValue and Condition stereotypes are more suitable to extend from the UML meta-class Property, because they are an attribute of a Class like a Property.

5 Domain Specific Language Grammar

In this section, we provide a language grammar that specifies the same concepts as Ecore meta-model presented in Section 3. See Figure 10 for the complete grammar in EBNF. Note that the grammar also defines a textual concrete syntax for modeling DES by providing a set of language constructs and keywords. Following example illustrates an instance of the grammar to model the event graph given in Figure 3.

```
DES sampleSim( start:0, end:100 ) {
    states{ n:real=0,
            p:real=0,
            q:integer=0
            x:integer=0,
            y:integer=0, }

    start event A {
            schedule:B if(x > y), delay 10,
            update: n=q*p; }

    event B {
            schedule:B,
            update: q++; }
}
```

6 Model-to-Text Transformation

In this section, we present the rules for transforming models created by the proposed modelling language to executable code. We map our modelling entities to Java code and create fully executable simulations using the SimKit simulation programming framework. Transformations are defined using the Xpand template language, which is a part of openArchitectureWare platform [5].

The core transformation rules from DES meta-model to SimKit Java code are presented below. For brevity, details of the generation rules are omitted in this section. For the full Xpand source code, see Figure 11.

- For a 'Model' entity, generate a Java Class extending 'SimEntityBase'.
- For each 'State' entity, generate a protected Java variable and its corresponding getter method.
- For each 'EventType' entity, generate a method 'doEventName'.
- For each 'Statement' entity of an 'EventType', generate calls to the 'firePropertyChanged' method inside the corresponding 'doEventName' method.
- For each 'Transition' entity, generate a call to the 'waitDelay' method, together with the transition delay and condition inside the 'doEventName' method of the source EventType.
- For each 'CancelingTransition' entity, generate a call to the 'interrupt' method, inside the 'doEventName' method of the source. EventType.

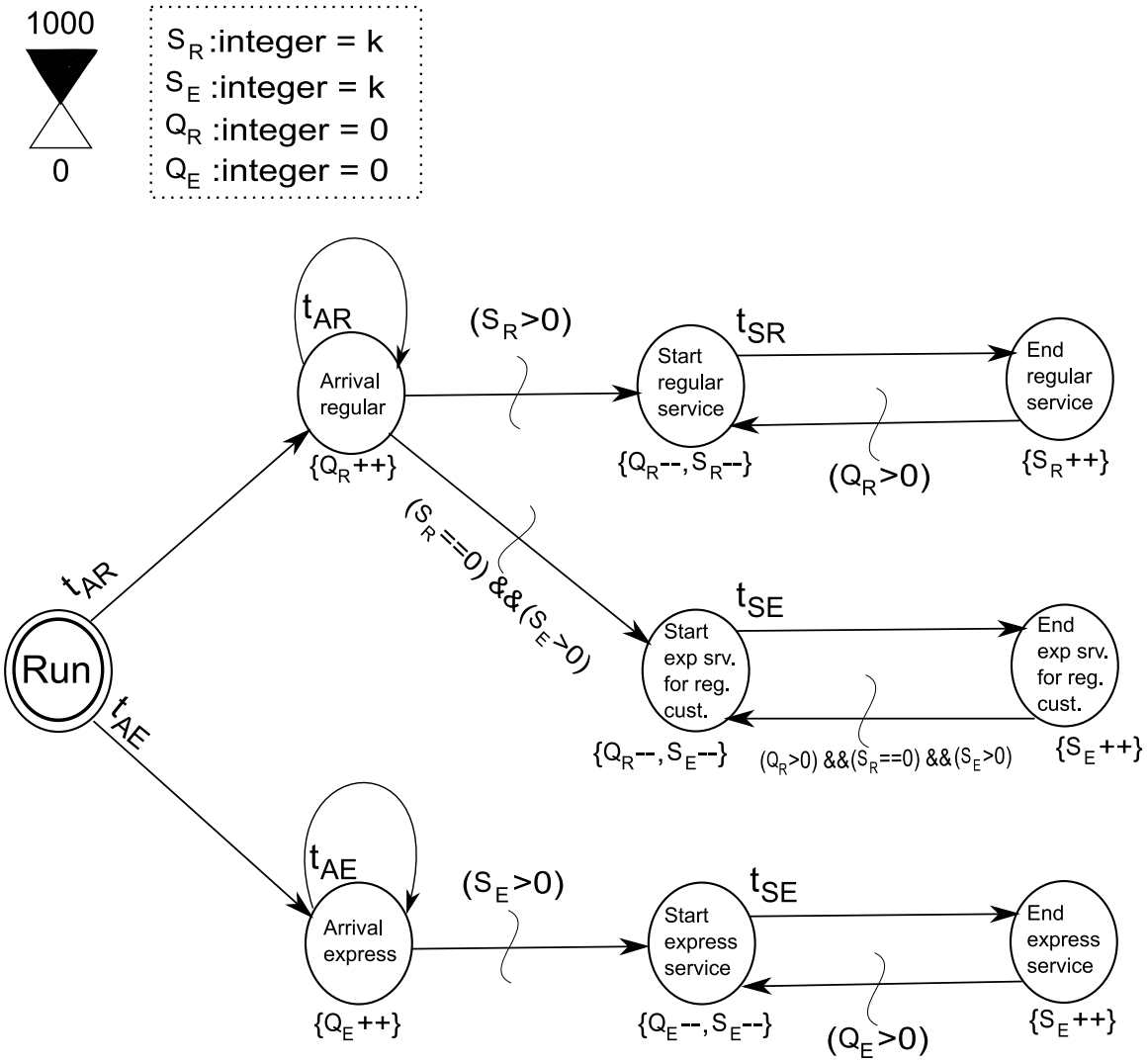

Fig. 7. Market simulation model

7 Model-to-Model Transformation

In this section we provide a transformation of our DES models to DOT models. DOT is a modelling language for defining graphs and is used by popular tools such as doxygen and Graphviz. This model-to-model transformation makes it possible to represent simulation event graph models as DOT graphs, capturing all the essential simulation data such as states and statements; so that any tool that can process the DOT metamodel can work with DES models. The transformation rules are defined using Atlas Transformation Language (ATL) [3]. Roughly, each DES EventType is mapped to a DOT Node, and the scheduling Transitions are represented by DOT Directed Arcs. State updates are transformed to DOT Node labels and Transition delays & conditions are mapped to DOT Arc Labels. Consequently, visualization of the DOT graphs are also very similar to the concrete syntax proposed for our event graph based DES models.

For the full ATL source code describing the mappings between metamodel entities, see Figure 12.

8 Discussion

Performing a good analysis of the domain and identifying the concepts clearly help building an accurate abstract syntax. We encountered little difficulties while defining a meta-model from scratch; however, it was not that easy to define the meta-model with UML profiling. There are several reasons behind this difficulty. Firstly, there is no comprehensive documentation or tutorial for profiling. Secondly, extending an existing meta-model to meet our requirements did not give us enough flexibility and expressive power to build a complete meta-model. On the one hand, UML profiling is consid-

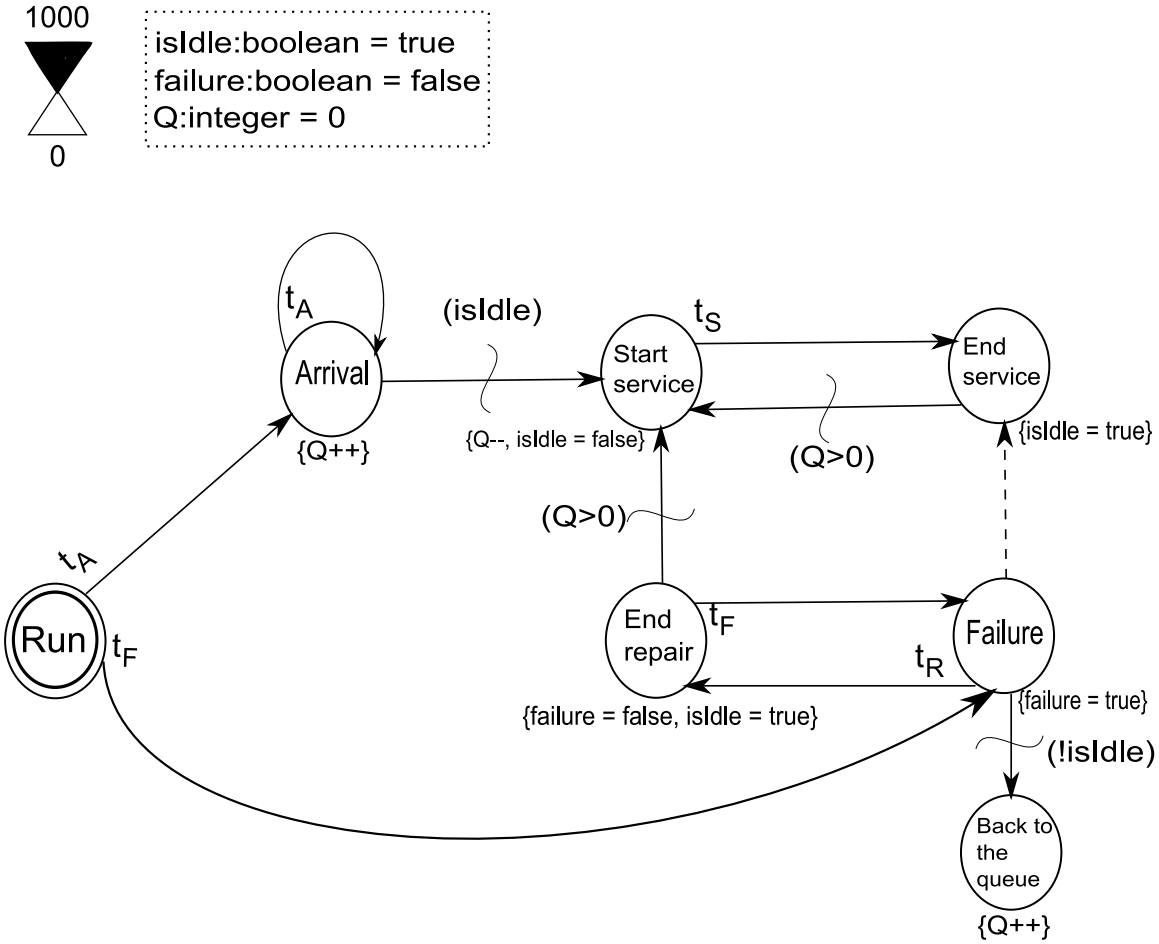

Fig. 8. Car washing service simulation model

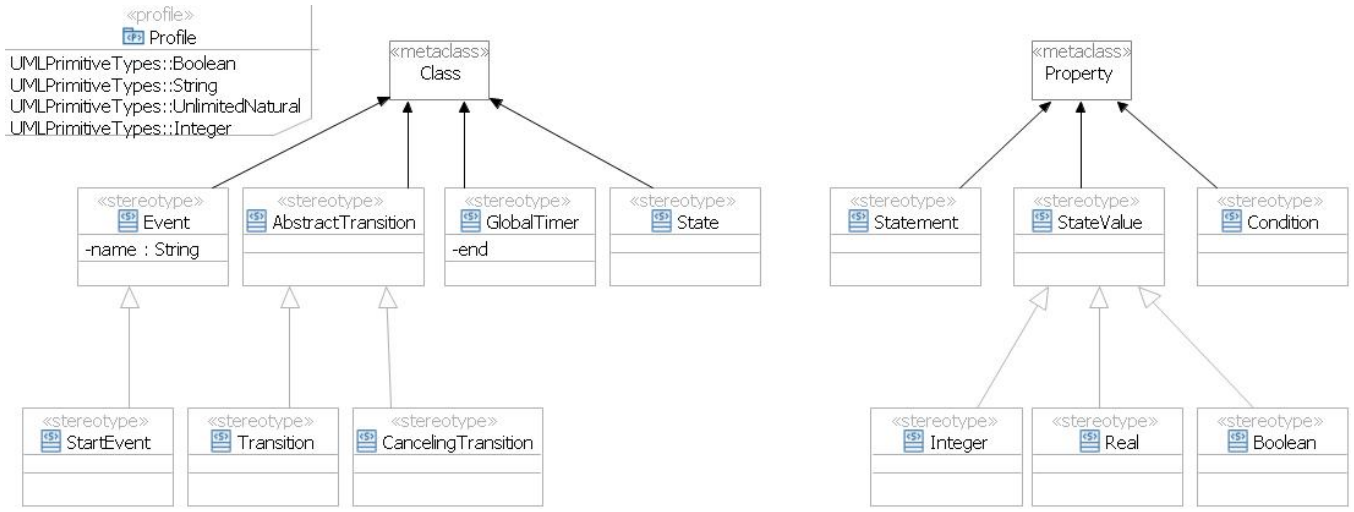

Fig. 9. UML meta-class extensions

ered useful as it provides a well-defined concrete syntax without effort; on the other hand, UML concrete syntax is not suitable for representing DES; event graphs are a more natural choice.

Despite the fact that it was easy to define the meta-models, computer aided design tools are not capable enough to capture the meta-modelling process as a whole. Moreover, Eclipse Modeling Framework (EMF) seems incomplete and poorly designed. Many simple tasks require too much

effort to perform and features are buggy.

Normally, grammars are thought to be harder to construct than the graph based abstract syntax; on the contrary, we have found that defining a grammar for our domain was relatively easy. By considering event graph representations, we were even able to define a textual concrete syntax.

Transforming the DES models to SimKit modules was one of the primary goals of this study. Since both the proposed DES models and the SimKit framework is based on the event graph representation of DES, constructing the mapping rules was a straightforward task. Working with openArchitectureWare and Xpand language, we were surprised to see that code generation process was very accurate and there were only minor bugs with the tools. However, it should be noted that, working with a pre-existing framework, SimKit, made it possible for us to generate very high level code and we did not have to worry about the inner working mechanisms of the framework. We can conclude that MDSD, with the current model-to-text transformation tools, is suitable for mapping models to high level programming frameworks; but the flexibility of Xpand for generating code from scratch needs further considerations.

Despite the relative maturity of model-to-code transformation process, model-to-model transformations were not an easy task to perform. During our studies, we were confronted with many integration problems between EMF platform and ATL; moreover, ATL transformations usually resulted in models that were reported to be corrupted. All in all, model-to-model tranformation process does not seem to be very refined and needs more maturity before it can be used in professional software developement.

9 Conclusion

In this paper, we presented a model-driven approach for developing discrete event simulation systems. We have provided the analysis of the domain, described the meta-model for creating models using Ecore, UML profiling and grammars and given a domain specific language to visually represent our models, with some examples. We also present a Model-to-Model transformation to DOT, and a Model-to-Text transformation to Java code from our models.

To conclude, DES systems can benefit from a model-driven software development approach since they can effectively be represented by event graphs that conform to our meta-model. This makes it possible to define an easy to understand simulation model without using simulation programming frameworks. Model-to-Text transformations can be applied to DES models in order to have an executable code that runs on a simulation programming framework; such as SimKit[2]. Furthermore, by using Model-to-Model transformations DES models can be transformed into other models for various purposes like visual representation.

Acknowledgments

We would like to thank Asst. Prof. Dr. Bedir Tekinerdoğan, for his efforts and for organizing the Turkish Model-Driven Software Development Workshop, 2009.

References

[1] P. Ball. Introduction to discrete event simulation. Technical report, University of Strathclyde, 1996.
[2] Arnold Buss. Discrete event programming with simkit. Technical report, Simulation News Europe, 2002.
[3] ATLAS Project Eclipse Modeling Framework. Atlas transformation language. http://www.eclipse.org/m2m/atl/, 2008.
[4] *Eclipse Modeling Framework (EMF)*. http://www.eclipse.org/modeling/emf/?project=emf, April 2009.
[5] openArchitectureWare. Xpand template language. http://www.openarchitectureware.org/, 2008.
[6] M. Pidd. *Computer Simulation in Management Science*. John Wiley & Sons, Inc., 1992.
[7] Lee Schruben. Simulation modeling with event graphs. *Commun. ACM*, 26(11):957–963, 1983.
[8] *Unified Modeling Language (UML)*. http://www.uml.org/, April 2009.

```
DESDeclaration = "DES", "varName", "(", "start", ":", integer, "end", ":", integer, ")", "{",
                 stateList, eventList, "}";

stateDeclaration = varName, ":", (("int", "=", integer) | ("real", "=", real) | ("bool", "=",
                 boolean));

stateList = "States", "{", stateDeclaration, {",", stateDeclaration}, "}";

eventList = startEvent, {event};

startEventDeclaration = "start", event;

eventDeclaration = "event", varName, "{" , [stateUpDate, ","], [eventScheduling,  ","],
                 [eventCancelling], "}";

stateUpDate = "update", ":", statement , {statement};

targetEvent = varName, {",", ifStatement}, {",",delay};

eventScheduling = "schedule", ":", targetEvent, {",", targetEvent};

delay = "delay", ":", integer;

eventCancelling = "cancel", ":", varName, {",", varName};

statement = varName, "=", exp, ";";

varName = varName, ( alpha | digit | "_") | alpha;

boolean = "true" | "false";

integer = digit | integer, digit;

digit = "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9";

alpha = "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k" | "l" | "m" | "n" | "o" |
      "p" | "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z" | "A" | "B" | "C" | "D" | "E"
      | "F" | "G" | "H" | "I" | "J" | "K" | "L" | "M" | "N" | "O" | "P" | "Q" | "R" | "S" | "T" |
      "U" | "V" | "W" | "X" | "Y" | "Z"

real = integer, "." integer | ".", integer;

exp = exp , "||",  aboveOr | aboveOr;

aboveOr = aboveOr,  "&&", aboveAnd | aboveAnd;

aboveAnd = aboveAnd, equalityOp, aboveEqualityOp | aboveEqualityOp;

aboveEqualityOp = aboveEqualityOp, comparisonOp, aboveComparisonOp | aboveComparisonOp;

aboveComparisonOp = aboveComparisonOp,  "+", abovePM | aboveComparisonOp, "-", abovePM | abovePM;

abovePM = abovePM,  "*",  aboveMDM | abovePM, "/", aboveMDM | abovePM, "%", aboveMDM | aboveMDM;

aboveMDM = aboveUM | "-",  aboveUM | "++",  varName | "--", varName | "!", aboveUM;

aboveUM = "(", exp, ")" | literalExp | varName;

literalExp = integer | real| boolean;

ifStatement = "if", "(", exp, ")";

comparisonOp = "==" | ">" | "<" | ">=" | "<=" | "!=";
```

Fig. 10. EBNF domain specific language grammar

```
«IMPORT metamodel»

«EXTENSION template::GeneratorExtensions»

«DEFINE main FOR Model»
      «FILE name+".java"»
            import simkit.*;

            public class «name» extends SimEntityBase {
                  «FOREACH states AS s»
                        protected «s.value.typeName» «s.name»;

                        public «s.value.typeName» «s.getter()»() {
                              return «s.name»;
                        }
                  «ENDFOREACH»

                  «FOREACH graph.eventTypes AS e»
                        public void do«e.name.toFirstUpper()»() {
                              «IF e.name == "Run"»
                                    reset();
                              «ENDIF»

                              «FOREACH e.statements AS s»
                              firePropertyChange( "«s.updates.name»", «s.updates.name», «s.exp» );
                              «ENDFOREACH»

                              «EXPAND transitionClass FOREACH e.transitions»
                        }
                  «ENDFOREACH»

                  public void reset() {
                        super.reset();
                        «FOREACH states AS s»
                              «s.name» = «s.value.value»;
                        «ENDFOREACH»
                  }
            }
      «ENDFILE»
«ENDDEFINE»

«DEFINE transitionClass FOR AbstractTransition»
«ENDDEFINE»

«DEFINE transitionClass FOR Transition»
      «EXPAND conditionalTransition FOR this»
      waitDelay( "«to.name»", «delay» );
«ENDDEFINE»

«DEFINE transitionClass FOR CancelingTransition»
      «EXPAND conditionalTransition FOR this»
      interrupt( "«to.name»" );
«ENDDEFINE»

«DEFINE conditionalTransition FOR AbstractTransition»
      «IF condition != null»
            if ( «condition.exp» )
      «ENDIF»
«ENDDEFINE»
```

Fig. 11. Xpand rules for Model-to-Text transformation.

```
module transform; -- Module Template
create dotfile : DOT from Model : metamodel;

rule GraphToDOTGraph
{
      from
            g: metamodel!EventGraph
      to
            out: DOT!Graph (
                  type <- 'digraph',
                  name <- 'DES Model in DOT',
                  rankDir <- 'BT',
                  labeljust <- '|',
                  labelloc <- 't',
                  compound <- true,
                  nodeSeparation <- 0.75,
                  nodes <- g.eventTypes
            )
}

rule EventTypeToNode
{
      from
            e: metamodel!EventType
      to
            out: DOT!Node (
                  name <- e.name,
                  shape <- NodeShape
            ),

            NodeShape: DOT!RecordNodeShape (
                  name <- 'circle',
                  label <- EventLabel
            ),

            EventLabel: DOT!ComplexLabel (
                  compartments <- EventLabelCompartment
            ),

            EventLabelCompartment: DOT!HorizontalCompartment (
                  complexLabel <- nameComplexLabel,
                  compartments <- simpleStatementCompartment
            ),

            nameComplexLabel: DOT!ComplexLabel (
                  compartments <- simpleNameCompartment
            ),

            simpleNameCompartment: DOT!SimpleCompartment (
                  content <- e.name
            ),

            simpleStatementCompartment: DOT!SimpleCompartment (
                  content <- e.statements->iterate( s; acc: String = '' | acc + s.exp )
            )
}


rule TransitionToDirectedArc
{
      from
            t: metamodel!Transition
      to
            out: DOT!DirectedArc (
                  fromNode <- t.from,
                  toNode <- t.to,
                  arrowHead <- arrowHeadShape,
                  arrowTail <- arrowTailShape,
                  taillabel <- arcTailLabel,
                  label <- conditionLabel
            ),

            arrowHeadShape: DOT!ArrowShape (
                  name <- 'vee',
                  isPlain <- false,
                  clipping <- 'none'
            ),

            arrowTailShape: DOT!ArrowShape (
                  name <- 'none',
                  isPlain <- false,
                  clipping <- 'none'
            ),

            arcTailLabel : DOT!SimpleLabel (
                  content <- if t.delay > 0 then ''+t.delay else '' endif
            ),

            conditionLabel : DOT!SimpleLabel (
                  content <- if t.condition.oclIsUndefined() then '' else t.condition.exp endif
            )
}
```

Fig. 12. ATL rules for Model-to-Model transformation.