Modeling and Simulation Based Design with DCharts

Thomas Huining Feng and Hans Vangheluwe
Modeling, Simulation and Design Lab
McGill University
http://msdl.cs.mcgill.ca/

Abstract

This article studies the software development based on modeling, simulation and code synthesis. DCharts, a statecharts variant with extensions, are used to model a practical application: a traffic light system. The development of this system emphasizes the use of various tools: AToM^3 is used as a visual modeling environment; SVM is the simulation engine to test the prototype of the model; SCC is the code synthesizer that generates reusable source code in multiple target languages. By successfully developing this system, this article proposes a highly automatic approach for modeling and simulation based design. This approach improves software productivity, reliability and reusability.

1 Introduction to Modeling and Simulation Based Design

As opposed to traditional software programming, modeling and simulation based design has many advantages. By modeling the behavior of the system in a formal way, the designer can then focus on more abstract issues instead of dealing with implementation details too early. The high-level design, once validated, can be automatically transformed into implementation at a much lower level with tools. Provided that the high-level design is correct and the tools involved in the transformations are also correct, the resulting low-level implementation should also be correct. This process saves a lot of labor of the designer, and it greatly improves software productivity and reliability.

1.1 The process of modeling, simulation and code synthesis

The modeling and simulation based design process is illustrated in Figure 1. A meta-model represents an executable formalism used in this development. The designer manually designs the behavior of the system as a model in this formalism. The model can then be checked with a model checking tool, if such a tool is available for the formalism. The correctness of the model can then be formally proved. The model designer (or model tester) may

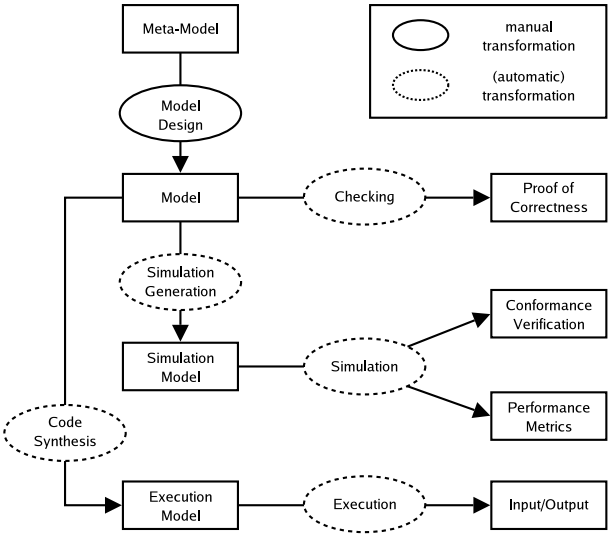

Figure 1: Modeling and simulation based design process

generate a simulation model from the original model. The simulation model is fed in a simulator to enable multiple simulations. The results of those simulations are collected. With those results, the designer verifies the conformance between the model and the initial requirements. He/She may also obtain performance metrics from those results, which evaluates the efficiency and reveals potential bottlenecks.

The simulation generation is usually automatic in that the simulation model is the same as the original model, or it can be done automatically with tools. Above this, model checking tools are used to check the validity of the original model, and simulators are used to simulate the simulation model. All these are done automatically without human intervention.

When the designer has enough confidence on the model, code synthesis tools are used to generate execution model from the original model. The execution model is directly executed. It accepts user input and produces output. Its behavior corresponds to the system requirements.

The code synthesis phase is also automatic. Code synthesizers dedicated for the formalism accepts the original

model as input, and generates execution model in a target language. The model is also optimized by the tools to improve run-time performance. The model designer need not modify the execution model, provided that the original model is correct and contains enough information for an execution. Other execution details are added by the code synthesis tools. The code is executable in its own right and is independent of simulation environments. It is the release version of the application.

1.2 The development of a TL (Traffic Light) system

A TL (Traffic Light) system is studied as an example in this article. It demonstrates the modeling and simulation based design process. This system behaves as an autonomous but also reactive traffic light. The initial requirements are as follows:

1. The traffic light has three colors: `red`, `green` and `yellow`. Initially, the light is `red`. After being `red` for 8 seconds[1], it turns `green`. It remains in this color for 5 seconds and then turns `yellow`. After another 2 seconds, it turns `red` again. The traffic light repeatedly changes colors in the same way.

2. During the first 6 seconds of being `red`, a pedestrian may press a "Crosswalk" button to request the traffic light to turn `green` so that he/she can get across. If so, the light turns `green` 2 seconds after the button is pressed.

3. A policeman may pause and resume the traffic light. When being paused, the traffic light becomes `yellow` and switches between `ON` and `OFF` with the interval of 0.5 second. When the light resumes functioning, it first takes on the `red` color.

4. The policeman may also turn `OFF` the traffic light. When turned `ON` again by the policeman, the traffic light returns to the previous state exactly before it is turned `OFF`.

Tools are used to automate the development of this system. AToM^3 (A Tool for Multi-formalism and Meta-Modeling) is the visual environment in which the original model of the system is designed. SVM (Statechart Virtual Machine) is the simulator that simulates the simulation model, which is identical to the original model. Each simulation of the same model produces a trace recording its run-time behavior. The traces can then be checked and analyzed. SCC (StateChart Compiler) is then used to synthesize executable code from the model. This process highlights the use of automated tools and greatly reduces human labor.

[1]The time modeled in the system is shorter than reality for the convenience of multiple simulations and executions.

Model checking and model verification of the conformance between the model design and the initial requirements are out of the scope of this article. Formal model checking for executable formalisms, such as the DCharts formalism used here, is hard especially because they allow the specification of execution details in the models. Actions in the models may change their state in an arbitrary way and make model checking even more challenging. Model verification by means of multiple simulations and ERE (Extended Regular Expressions) is discussed in [1].

As such, this article focuses on model design, model simulation and code synthesis (and the reuse this code).

2 Model Design in the DCharts Formalism

The TL system is modeled with DCharts, a statecharts variant with extensions. The model is explicitly designed by the model designer, and this is the only creative work that the designer needs to do.

2.1 Introduction to DCharts

DCharts [2] are an executable formalism based on David Harel's statecharts [3] [4]. The DCharts syntax includes the statecharts syntax. Constructs of statecharts, such as hierarchical states, transitions between those states, orthogonal components and history, are all inherited by DCharts. Their semantics in DCharts conforms to the statecharts semantics defined by David Harel.

Extensions to statecharts are added in DCharts. They improve the modularity and reusability of the formalism. The following extensions are also discussed in [2] and [5].

- Importation. A model designed in DCharts can also be regarded as a reusable component, which can then be imported into a state of another DCharts model. All the states and transitions in the *submodel* (or, *imported model*) are copied to the inside of that state (*importation state*) at run-time. This means importation is done dynamically when the internal structure of the importation state is required. For example, the first transition to an importation state triggers an importation, and the simulator/executor imports the required submodel to obtain the default substates of the importation state. This dynamic behavior allows for recursive importation, where a model imports itself directly or indirectly, and hence creates a theoretically infinite state hierarchy.

- Transition priorities. Each state of a model has a property related to the priority of the *transitions in its scope* (transitions from that state or substates of that state). If that property is equal to *ITF* (Inner-Transition-First), transitions in its scope are inner-first (i.e., transitions from a state at a lower level

have higher priority). The opposite is *OTF* (Outer-Transition-First).

For a state that does not explicitly specify this property, it inherits this property from its parent state. However, it may always override this property by explicitly assign a value to this property.

The transition priority solves conflicts between transitions, caused by multiple transitions in a single orthogonal component competing for the same event.

- Macros. Designers may specify macros for their models. In the description of the models, they may use the names of those macros to literally represent their values.

  Macros can be redefined by the importing model when it imports a submodel into its importation state. On the one hand, this mechanism increases reusability, since the importing model can then fine-tune the behavior of the submodel; on the other hand, the modularity of the submodel is protected, because the importing model may not change its behavior in any other way.

- Ports and connections between ports. DEVS-like ports and connections [6] [7] [8] are added as yet another extension. Designers may connect multiple DCharts models via well-defined ports. Those models influence each other by sending messages via the established connections.

For a more detailed description on the syntax and semantics of DCharts, the readers are referred to [2], the master's thesis that originally introduces DCharts.

2.2 AToM^3, a visual modeling environment

AToM^3 is a visual environment for modeling and metamodeling. It is created by Prof. Hans Vangheluwe in the MSDL (Modeling, Simulation and Design Lab) of McGill University. It allows its users to graphically design metamodels (the models of formalisms). By loading a metamodel in it, the AToM^3 environment is adjusted according to the allowed entities of the formalism. The users can then design models in that formalism with a visual environment dedicated to it.

A DCharts meta-model for AToM^3 is used here. It is a model of the ER (Entity-Relationship) formalism in its own right. When this DCharts meta-model is loaded, buttons for various DCharts entities appear and users can then design DCharts models in AToM^3.

2.3 The TL model in DCharts

Figure 2 shows the main component of the TL system, designed in AToM^3: the `TrafficLight` component. It is a

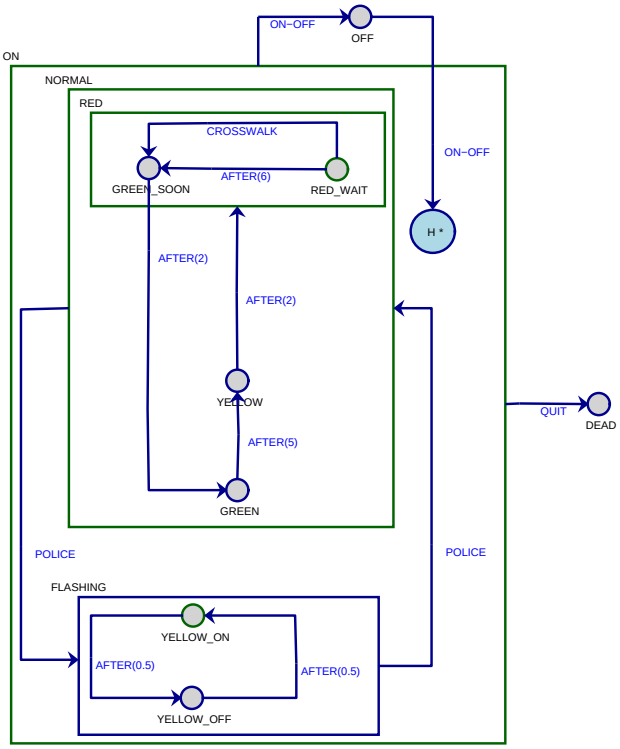

Figure 2: The `TrafficLight` component of the TL system

hierarchical DCharts model. It is also a statecharts model, since it does not use DCharts extensions. There are three top-level states: `ON`, `OFF` and `DEAD`. The `TrafficLight` is functional when it is in the `ON` state. When the policeman turns it off (by sending `ON-OFF` event to it), it goes to the `OFF` state. When the simulation/execution ends, the `QUIT` event is received and the model goes to the `DEAD` state. The `RED`, `GREEN` and `YELLOW` substates of the model represents the three possible colors. The `RED` state has two substates: `RED_WAIT` and `GREEN_SOON`. When the model is in `RED`, it stays in `RED_WAIT` for at most 6 seconds. The pedestrian may send the `CROSSWALK` event during that period to immediately change the model to the `GREEN_SOON` state. If no `CROSSWALK` event is received, the model changes to the `GREEN_SOON` automatically after 6 seconds (with an `AFTER` event, which schedules a transition after that period).

At any time, the policeman may pause the traffic light by sending a `POLICE` event. The model then goes to the `FLASHING` state, and the light flashes with an interval of 0.5 second. The model goes back to its default state with a second `POLICE` event.

The policeman may also turn off the light with an `ON-OFF` event. When this event is received the second time, the model returns to its previous state, which is recorded in a deep history.

The complete behavior of the traffic light is modeled in this component. When it is simulated or executed, it

autonomously changes colors. It also reacts to events such as `CROSSWALK`, `POLICE` and `ON-OFF`. These events are not generated by this component itself. They are input by the user from the simulation environment (such as SVM) or execution environment (such as the code synthesized by SCC). In those cases, the user acts as the pedestrian and the policeman.

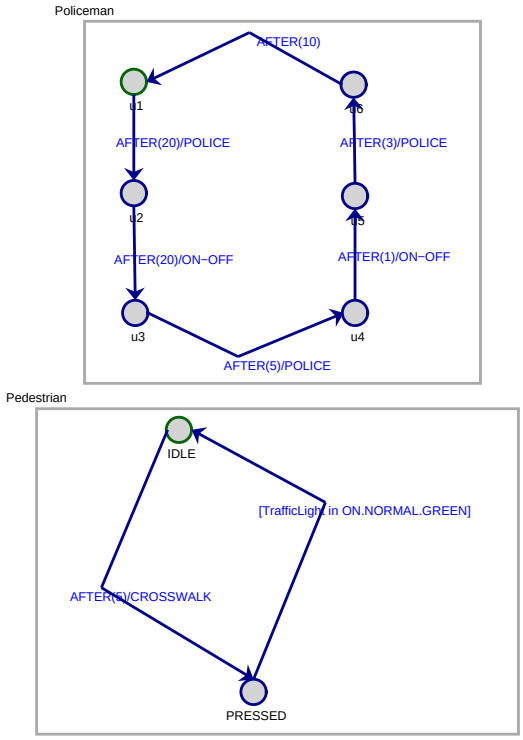

Figure 3: The police and pedestrian components

The designer may also explicitly model the pedestrian and the policeman in other components. Those components are orthogonal to the `TrafficLight` component, and they have concurrent behavior. The pedestrian component then periodically generates the `CROSSWALK` event, and the policeman component periodically generates the `POLICE` event and the `ON-OFF` event. These components explicitly model the experiment environment for the TL system.

Figure 3 shows one possible design of the `Pedestrian` orthogonal component and the `Policeman` orthogonal component.

- The `Pedestrian` is initially in its default state `IDLE`. It signals a `CROSSWALK` event after 5 seconds, and waits until the traffic light turns `GREEN`. The guard `[TrafficLight in ON.NORMAL.GREEN]` here tests this condition repeatedly.

- The `Policeman` is initially in default state `u1`. It signals a `POLICE` event after 20 seconds. This makes the traffic light flash. After another 20 seconds, an `ON-OFF` event is produced and the traffic light is turned `OFF`. The second `POLICE` event is then ignored since the traffic light is `OFF`. After that, the `ON-OFF` event sent with the transition from `u4` to `u5` turns the traffic light `ON` again. It then goes to its deep history, and it continues flashing. The third `POLICE` event after that resumes the functioning of the traffic light. It goes back to its default state `RED`.

With this simple but practical example, we focus on the following important points:

- The `TrafficLight` component demonstrates the use of several DCharts/statecharts features, such as state hierarchy, deep history and the `AFTER` special event.

- The `TrafficLight` component is at the same time autonomous and reactive. When it is treated as a stand-alone model, it accepts input from the simulation/execution environment. When it is used as a component in a larger model (by means of DCharts importation), it communicates with other parts of the same model by means of event broadcast.

- The `Pedestrian` component and the `Policeman` component explicitly model an experiment environment for the TL system. Although the traffic light itself is the part that we actually want to build, these two extra components help to test the system. If the behavior of these components is made very similar to the actual behavior of a pedestrian and a policeman, we then know whether the traffic light is working well only by means of simulations, without really putting it into practical use.

3 Simulation with SVM

After designing the model in AToM^3, the designer can then simulate it and obtain results from simulations. Simulations can be done in AToM^3 or separately with the SVM simulator.

3.1 SVM, a DCharts simulator

SVM [9] is a simulator that supports the complete DCharts syntax and semantics, a superset of the statecharts syntax and semantics. It accepts DCharts model descriptions as textual input, and outputs the simulation results. It has multiple default interfaces, including a graphical interface and a plain-text interface, from which the users interact with the models and the simulation environment.[2] The users may debug the model by looking into the internal

[2]Models may also define specific interfaces, which are different from the default interfaces internally provided by SVM.

data structures of SVM. In the debug mode, they may also modify those data structures with Python scripts.

SVM can be used as a plugin to enable the simulation in AToM^3. The current states and enabled transitions are highlighted during a simulation. It can also be invoked from the command-line, with the file name of a model description given as a parameter.

3.2 Simulation trace of the TL model

The `TrafficLight` component is defined in text file `TrafficLight.des`, and the `Pedestrian` component and the `Policeman` component are defined in `TLExperiment.des`. The former model description is also imported into the latter one to create a third orthogonal component. When we start a simulation with `TrafficLight.des`, we can only see the traffic light. From the SVM interface we may input events to interrupt its autonomous behavior. Such events include `CROSSWALK`, `POLICE` and `ON-OFF`. When we start a simulation with `TLExperiment.des`, all the three components in the experiment environment are created and they interact with each other by sending events.

To obtain textual simulation traces, a series of `DUMP` statements are added to the model. Those statements have no effect on the model behavior, but they print out useful information for verification and analysis purpose. For example, the following is one possible output trace obtained from a single simulation:

```
(05.00) Pedestrian: CROSSWALK sent
(05.01) TrafficLight: CROSSWALK received
(07.01) TrafficLight: turn green
(07.02) Pedestrian: crossing the intersection
(12.01) TrafficLight: turn yellow
(12.02) Pedestrian: CROSSWALK sent
(14.01) TrafficLight: turn red
(20.00) Policeman: POLICE sent
(20.01) TrafficLight: start flashing
(40.01) Policeman: ON-OFF sent
(40.02) TrafficLight: turn OFF
(45.04) Policeman: POLICE sent
(46.04) Policeman: ON-OFF sent
(46.05) TrafficLight: turn ON
(49.11) Policeman: POLICE sent
(49.16) TrafficLight: stop flashing
(57.24) TrafficLight: turn green
(57.25) Pedestrian: crossing the intersection
(62.25) TrafficLight: turn yellow
(62.26) Pedestrian: CROSSWALK sent
(64.26) TrafficLight: turn red
(72.27) TrafficLight: turn green
(72.27) Pedestrian: crossing the intersection
(77.27) TrafficLight: turn yellow
(77.28) Pedestrian: CROSSWALK sent
(79.12) Policeman: POLICE sent
(79.13) TrafficLight: start flashing
(99.13) Policeman: ON-OFF sent
(99.14) TrafficLight: turn OFF
```

This output trace consists of a sequence of messages, each of which conforms to format "`(time) sender: body`". `time` is a float number representing the time when the message is generated[3]; `sender` is the name of the component that sends the message; and `body` is the message body.

This output trace gives us information on the model behavior. One can see from the trace that about 5 seconds after startup (when the traffic light is red), the pedestrian sends a `CROSSWALK` event. As expected, the light turns green after 2 seconds, and the pedestrian immediately crosses the intersection. 5 seconds later, the light turns yellow, and at almost the same time but later, the pedestrian sends `CROSSWALK` again. This event is ignored since no transition from the `YELLOW` state of the `TrafficLight` responds to this event. Then the light turns red as expected. The policeman pauses the light at time 20. The light starts flashing until another `POLICE` event is received at time 40. Etc..

The model tester should remember that the pedestrian is still waiting for the light, because, unfortunately, he/she sends the `CROSSWALK` event right after the light turns yellow. That event is thus ignored. The tester should then suggest the designer to improve the model in two possible ways: either the `TrafficLight` responds to the `CROSSWALK` event while it is in the `YELLOW` state, or the pedestrian is allowed to cross the intersection while the light is yellow (though dangerous).

4 Code synthesis with SCC and code reuse

After the model has passed a set of tests by means of simulations, the designer becomes confident enough of the model. He/She may then synthesize code in a target language, and release the executable code.

4.1 SCC, a code synthesizer for DCharts

SCC is able to synthesize executable code from textual DCharts model descriptions. It is invoked from the command-line with the file name of a model description as a parameter. The user may choose from one of the supported target languages, such as Python, C++, Java and C#. The code has exactly the same behavior as the DCharts model in simulations. It also has a simple textual interface.

If an interpreted language (such as Python) is chosen, the user can execute the code immediately. Otherwise, the user needs to compile the code with a compiler for the target language to obtain the binary.

[3]Because SVM is a real-time simulator, its time accuracy highly depends on the operating system and is in many cases not exact. Virtual-time simulations are also possible with an extra `Clock` component, as discussed in [2].

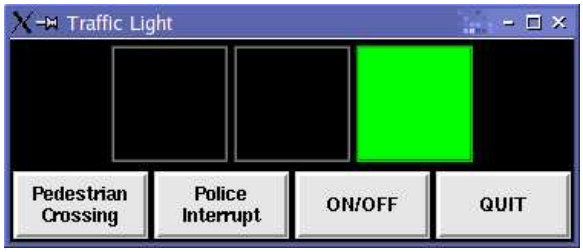

Figure 4: The traffic light application

It is possible to incorporate the code in a larger (and possibly hand-written) application. The code reuse of the `TrafficLight` component is an example discussed below.

4.2 Code synthesis and code reuse of the TL model

The designer may synthesize code with SCC for the whole TL system, which includes the three components: `TrafficLight`, `Pedestrian` and `Policeman`. This code has the same behavior as the model simulation, and it produces traces similar to the simulation trace shown above.

However, the user may only want the code for the `TrafficLight` component only. He/She can then reuse this code in a practical application, which reacts to real policeman and real pedestrian. It is convenient to synthesize code for the `TrafficLight` component only, because in the design of the system, the model is separated into two parts: `TrafficLight.des` and `TLExperiment.des`. `TrafficLight.des` contains the `TrafficLight` component, and it is imported into the latter file. The designer now synthesizes code from `TrafficLight.des` instead of `TLExperiment.des`.

When code for the `TrafficLight` component is obtained, it can thus be reused in a larger application. The application accesses methods of the model class through a well-defined interface. An example of such an application is shown in Figure 4. In this case, a GUI is defined in Python. It imports the Python code synthesized from the `TrafficLight` component, and uses that part of code to maintain the current state and react to external stimuli.

5 Conclusion

Modeling and simulation based design is studied in this article with a concrete example of the TL system. Three steps in this highly automatic development process are emphasized: model design, simulation and code synthesis.

DCharts, a variant of David Harel's statecharts with extensions, are introduced and used to model the example system. Simulation of the DCharts model is supported by SVM. The simulation traces produced by multiple simulations reveal potential problems in the model, and also provide information for the analysis of a performance metrics. When thoroughly tested, a useful part of the model (excluding the components of the experiment environment) is transformed into executable code with SCC. The code can then be reused in user applications.

Compared to traditional software programming, this development process reduces human labor and increases productivity, reliability and reusability.

References

[1] Thomas Huining Feng and Hans Vangheluwe. Case study: Consistency problems in a UML model of a chat room. In *Sixth International Conference on the Unified Modelling Language (UML 2003), Workshop on Consistency Problems in UML-based Software Development II*, October 2003. San Francisco, USA.

[2] Thomas Huining Feng. DCharts, a formalism for modeling and simulation based design of reactive software systems. Master's thesis, School of Computer Science, McGill University, Montréal, Canada, May 2004.

[3] David Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[4] David Harel and Amnon Naamad. The STATEMATE semantics of statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, 1996.

[5] Thomas Feng. An extended semantics for a Statechart Virtual Machine. In *Summer Computer Simulation Conference. Student Workshop*, pages S147 – S166, July 2003. Montréal, Canada.

[6] Bernard P. Zeigler. *Multifacetted modelling and discrete event simulation*. Academic Press Professional, Inc., 1984.

[7] Bernard P. Zeigler. *Theory of Modelling and Simulation*. Krieger Publishing Co., Inc., 1984.

[8] Bernard P. Zeigler and Sankait Vahie. DEVS formalism and methodology: Unity of conception/diversity of application. In *Proceedings of the 1993 Winter Simulation Conference*, pages 573–579, 1993.

[9] Thomas Huining Feng. Statechart Virtual Machine (SVM), 2003. MSDL, McGill University, `http://msdl.cs.mcgill.ca/people/tfeng/?research=svm`.