SCCD: SCXML Extended with Class Diagrams

Simon Van Mierlo,
Yentl Van Tendeloo,
Bart Meyers
University of Antwerp
Middelheimlaan 1, 2020
Antwerp, Belgium
firstname.lastname@uantwerpen.be

Joeri Exelmans
University of Antwerp
Middelheimlaan 1, 2020
Antwerp, Belgium
joeri.exelmans@student.uantwerpen.be

Hans Vangheluwe
University of Antwerp
Middelheimlaan 1, 2020
Antwerp, Belgium
McGill University
3480 University Street
Montréal, Québec, Canada
H3A 0E9
hv@cs.mcgill.ca

ABSTRACT
We introduce the SCCD formalism, which is a hybrid of Statecharts and Class Diagrams, and its SCCDXML representation, which is an extension of SCXML. SCCD facilitates the specification of complex timed, reactive, interactive discrete-event, dynamic-structure systems (*e.g.,* complex user interfaces, control systems), as we demonstrate using a representative example. We present an SCCD compiler that supports (a) semantic variation points for different Statecharts variants (*e.g.,* Rhapsody and Statemate), (b) code generation for different platforms (*e.g.,* Tkinter and HTML/Javascript), and (c) code generation for different families of runtimes (*e.g.,* event-based platforms and game loops). Furthermore, we discuss the history and future work of SCCD to reveal our research agenda.

INTRODUCTION
The Statecharts [3] formalism was developed to aid the specification of complex, timed, interactive discrete-event systems. Nevertheless, the exclusive use of Statecharts does not scale to the complex (and often dynamic-structure) behaviour of the software systems we want to model today. From a programming point of view, object-oriented modelling methodologies address software complexity, but are not specifically designed for modelling timed, interactive discrete-event systems [4].

We propose SCCD, a Statecharts and Class Diagrams hybrid, which combines the structural object-oriented expressiveness of Class Diagrams with the behavioural discrete-event characteristics of Statecharts. The aim is to create a minimal (but sufficiently expressive) language, providing the necessary abstractions for modelling complex, timed, interactive discrete-event systems that exhibit dynamic-structure behaviour. We first present the SCCD language and its SCCDXML representation, based on SCXML. We then explain our SCCD compiler. We review the limitations of the language and compiler, outlining future work. Finally, we explore related work and conclude the paper.

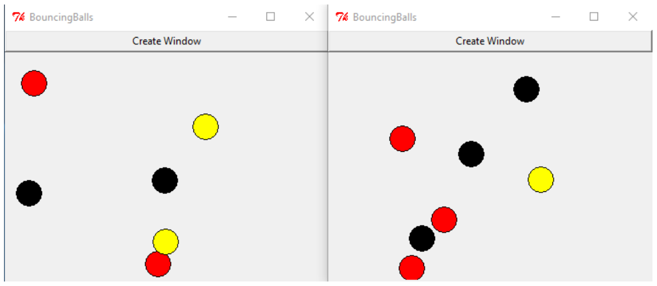

Figure 1. A screenshot of the running example. Black balls are moving, yellow balls are selected, and red balls have been moved by the user.

RUNNING EXAMPLE
To demonstrate our language, we model a timed, reactive, autonomous, dynamic-structure system, which is not easily expressed using Statecharts (and, by extension, SCXML). The system is a "bouncing balls" application, which has the following requirements:

- The application consists of a number of windows. It starts with exactly one window.
- Each window has a number of bouncing balls, and a button that spawns a new window.
- A window can be closed. If no more windows remain, the application exits.
- The user can spawn a new black ball by right-clicking in a window. The ball starts moving in a random direction.
- The user can select a ball by left-clicking on it. The ball then changes its colour to yellow, and stops moving.
- The user can move a selected ball by dragging it.
- When releasing a dragged ball, its colour changes to red, and its velocity is changed proportionally to how fast the user was moving the mouse.
- When the user presses the "delete" key, all selected balls in the current window are deleted.

A screenshot of the running application is shown in Figure 1. The original Statecharts formalism has no facilities for specifying multi-agent systems whose structure changes over time. Each ball's behaviour is ideally controlled by a separate Statecharts model, with each instance able to communicate with others. The constraints on the structure of the application should additionally be specified by the application designer and checked at runtime.

THE SCCD FORMALISM

The SCCD formalism extends Statecharts with the concept of a class, which models structure, and associates with each class a definition of its behaviour (in the form of a Statecharts model). We extend SCXML for the modelling of SCCD models, thus creating the SCCDXML language. We first present the new features of our language, and then we discuss the management of objects at runtime.

Language Features

This section introduces the new features of the SCCD language and demonstrates them with our running example in the SCCDXML concrete syntax. We assume the reader is familiar with standard SCXML notation and do not repeat its definition, only highlighting our extensions.

Top-level Elements

The top-level element is a diagram. It has an input/output interface to communicate with its environment, it can optionally import library classes, and it holds a number of class definitions. One of these classes is the default, and is instantiated when the application is launched.

Listing 1. The top-level ‘diagram’ element.

```
<diagram>
  <top>
    from ui_widget import UIWidget
  </top>
  <inport name=‘input’ />
  <class name=‘MainApp’ default=‘true’>...</class>
  <class name=‘Window’>...</class>
  <class name=‘Button’>...</class>
  <class name=‘Ball’>...</class>
</diagram>
```

Listing 1 shows the top-level diagram of the example application. It imports a library class that is used to draw the graphical elements on the screen, one input port called “input” which receives events when the user interacts with the UI (for example, pressing a key), and four classes, explained in the following subsections. To comply with good OO design, each class can also be defined in its own file, with other files importing their definition.

Classes

Classes are the main addition of the SCCD language. They model both structure and behaviour. Structure is modelled in the form of attributes and relations with other classes, effectively encapsulating the runtime data of the application. Behaviour is modelled in the form of methods, which access and change the values of attributes of the class by executing statements modelled in an action language, and an SCXML model, which constitutes the “modal” part (*i.e.*, the control flow) of the class. At runtime, an object can be created and deleted, followed by the invocation of, respectively, the constructor and the destructor. The relationships modelled between classes are instantiated at runtime in the form of links. They serve as communication channels, over which objects can send and receive events.

Listing 2. The ‘Ball’ class.

```
<class name=‘Ball’>
  <relationships>...</relationships>
  <inport name=‘ball_input’/>
  <constructor>
    <parameter name=‘canvas’ />
    <parameter name=‘x’ />
    <parameter name=‘y’ />
    <super class=‘UIWidget’ />
    <body>...</body>
  </constructor>
  <destructor>
    <body>self.canvas.delete(self)</body>
  </destructor>
  <method name=‘move’>
    <parameter name=‘position’ />
    <body>...</body>
  </method>
  <scxml initial=‘bouncing’>
    <state id=‘bouncing’>...</state>
    <state id=‘dragging’>...</state>
    <state id=‘selected’>...</state>
    <state id=‘deleted’ />
  </scxml>
</class>
```

Listing 2 shows the definition of the ‘Ball’ class. It defines a number of relations (discussed in the next subsection), a constructor and destructor, a method that moves the ball to a new position, and an SCXML model that consists of four states. It can optionally also define private input ports and output ports. In this case, the ball defines a private input port, that allows the environment to send events that are only meant for a particular ball.

Relationships

Classes can have relationships with other classes. An *association relation* is defined between a source class and a target class, and has a name. An association has a multiplicity, defined as a minimal cardinality $c_{min} \in \mathbb{N}$ and a maximal cardinality $c_{max} \in \mathbb{N}_{>0} \cup \{\infty\}$. By default, $c_{min} = 0$ and $c_{max} = \infty$. They control, at runtime, how many instances of the target class have to be minimally associated to each instance of the source class, and how many instances of the target class can be maximally associated to each instance of the source class, respectively. Each time an association is instantiated, it results in a uniquely identified link between the source and target object which can be used, for example, to send events. An *inheritance relation* results in the source of the relation to inherit all attributes and methods from the target of the relation. Specialisation of the superclass’s behaviour is currently not supported.

Listing 3. Relationships of the ‘Ball’ class.

```
<class name=‘Window’>
  <relationships>
   <association name=‘parent’ class=‘MainApp’ min=‘1’ max
        =‘1’/>
   <association name=‘buttons’ class=‘Button’ />
   <association name=‘balls’ class=‘Ball’ />
   <inheritance class=‘UIWidget’ />
  </relationships>
  ...
</class>
```

Listing 3 shows the relationships of the 'Window' class. It has an association to its parent, the main application. Exactly one instance of that link has to exist between each 'Window' instance and the main application. It is additionally associated to a number of buttons and balls, and inherits from the library class 'UIWidget', allowing it to be drawn on screen.

Events

Events in SCCD are strings. They are accompanied by a number of parameter values: the sender is obliged to send the correct number of values, and the receiver declares the parameters when catching the event. Each parameter has a name, that can be used as a local variable in the action and constraint associated with the transition that catches the event.

With the addition of a public input/output interface using ports, as well as classes and associations, comes the need for event scoping. In traditional SCXML models, an event is sensed by the Statecharts model that generated it. SCCD adds the ability to transmit events to class instances and to output ports. In particular, the **raise** tag was extended with a *scope* attribute, that can take on the following values:

- `local`: The event is only visible for the sending instance (the default behaviour of SCXML).
- `broad`: The event is broadcast to all instances.
- `output`: The event is sent to an output port and is only valid in combination with the *port* attribute, which specifies the name of the output port.
- `narrow`: The event is narrow-cast to specific instances only, and is only valid in combination with the *target* attribute, which specifies the instance to send the event to. For example, an instance of the 'Window' class can narrow-cast an event by sending the event to a specific instance of the 'Ball' class, identified by a unique link identifier.
- `cd`: The event is processed by the object manager. See the next section for more details.

Listing 4. An example transition that narrow-casts an event.

```
<transition event='left−click' port='button_input' target='.'>
  <raise event='button_press' scope='narrow' target=''parent''>
    <param expr='self.event_name' />
  </raise>
</transition>
```

Listing 4 presents a transition modelled on the 'Button' class. It reacts to the user left-clicking the button (represented by an event sent on the *button_input* port). The button reacts by notifying its parent that it was clicked.

The Object Manager

At runtime, a central entity called the object manager is responsible for creating, deleting, and starting class instances, as well as managing links (instances of associations) between class instances. It also checks whether no minimal or maximal cardinalities are violated when the user deletes or instantiates an association, respectively. As mentioned previously, instances can send events to the object manager using the "cd" scope. The object manager can thus be seen as an ever-present, globally accessible object instance, although it is implicitly defined in the runtime, instead of as an SCCD class.

When the application is started, the object manager creates an instance of the default class and starts its associated Statecharts model. From then on, instances can send several events to the object manager to control the set of currently executing objects. The object manager accepts four events. We list them below, including the parameters that have to be sent as part of the event:

- **create_instance**(*association_name*, *class_name*, . . .): Creates a new instance, if it is allowed (*i.e.*, no constraints would be violated). The newly created instance is always associated to its creator (the instance that sent the event). The first parameter is the name of the association that should be instantiated to create a link between the parent and its newly created child. The second parameter is the name of the class that needs to be instantiated. This should be the class that is defined as the target of the association, or one of its subclasses. Any subsequent parameters are interpreted as arguments to the constructor of the new instance. If creation succeeds, a reply event is sent to the requester containing the unique identifier of the link created between the creator and the new object. If creation failed, an error event is sent instead.
- **delete_instance**(*link_ref*): Deletes the instances specified by the link reference. The link reference is evaluated in the context of the instance that sent the event and should result in a set of link identifiers. The target objects of these links are deleted, as well as any links for which these objects are the source or target, as long as no multiplicity constraints are violated. The object manager sends an event to the requester when deletion was successful. If deletion failed, an error event is sent instead.
- **start_instance**(*link_ref*): Starts the execution of the Statecharts model of the instances specified by the link reference.
- **associate_instance**(*source_ref*, *association_name*, *target_ref*): This event makes it possible to create associations between two existing instances. The source and target references are evaluated to two sets of instances, and each instance in the first set is connected using the specified association with the instances in the second set.

The object manager, in combination with input/output ports of the diagram, replaces the *invoke* and *send* tags of the current SCXML standard. We believe this solution to be more general and more modular. The *invoke* tag, for example, does not allow for instances (effectively, agents) to run concurrently for the whole duration of the program, does not offer a comprehensive interface for object management, and does not offer any checks on the structure of the system at runtime. Moreover, using ports instead of direct sends to a predefined location is more modular, since the Statecharts model does not need to know the actual service that it communicates with (it just needs to know its interface), which means it can be reused in different contexts.

Listing 5 shows how the 'Window' class creates an instance of the 'Ball' class as a result of the user right-clicking in-

Listing 5. Creating an instance of the 'Ball' class.

```
<state id='running'>
  <transition event='right−click' port='window_input' target='../
      creating_ball'>
    <raise scope='cd' event='create_instance'>
      <parameter expr='‘balls’' />
      <parameter expr='‘Ball’' />
      <parameter expr='self.canvas' />
      <parameter expr='self.clicked_x' />
      <parameter expr='self.clicked_y' />
    </raise>
  </transition>
</state>
<state id='creating_ball'>
  <transition event='instance_created' target='../running'>
    <parameter name='link_name' type='string'/>
    <raise event='start_instance' scope='cd'>
      <parameter expr='link_name' />
    </raise>
  </transition>
</state>
```

side of that window. The instance raises the **create_instance** event, using the *cd* scope. It specifies that a 'balls' link has to be created to refer to the new instance, and passes the appropriate constructor parameters to the 'Ball' class. It then waits for the event signalling that the instance was successfully created.

Listing 6. Deleting an instance of the 'Ball' class.

```
<state id='running'>
  <transition event='delete_ball' target='.'>
    <parameter name='link_name' type='string' />
    <raise event='delete_instance' scope='cd'>
      <parameter expr='link_name' />
    </raise>
  </transition>
</state>
```

Listing 6 shows how an instance of the 'Window' class reacts to a ball requesting to be deleted (see Listing 4). The ball sends the correct link reference, and the window then instructs the object manager to delete that ball. Currently, there is no support for objects deleting themselves.

THE SCCD COMPILER

The semantics of an SCCD model are loosely based on the agent model, where each instance of a class can be seen as an agent whose autonomous behaviour is controlled by its Statecharts model and communicates asynchronously with other agents through its input/output interface. Our compiler generates appropriate code that continuously executes the system by allowing each agent to execute a step, which optionally generates output that can be sensed by the other agents.

The compiler supports multiple programming languages, runtime platforms, and options for the Statecharts semantics. These are visually represented in the feature diagram in Figure 2, and are explained in the following subsections.

Programming Languages

The compiler can currently generate code for three programming languages: Javascript, Python, and C#. Supporting multiple languages is a major advantage, as developers can create applications in SCCD and generate code for multiple languages from the same model. The generated code then exhibits identical behaviour for each implementation language, such as a web-based application (*e.g.*, in HTML/Javascript) and a desktop application (*e.g.*, in Python).

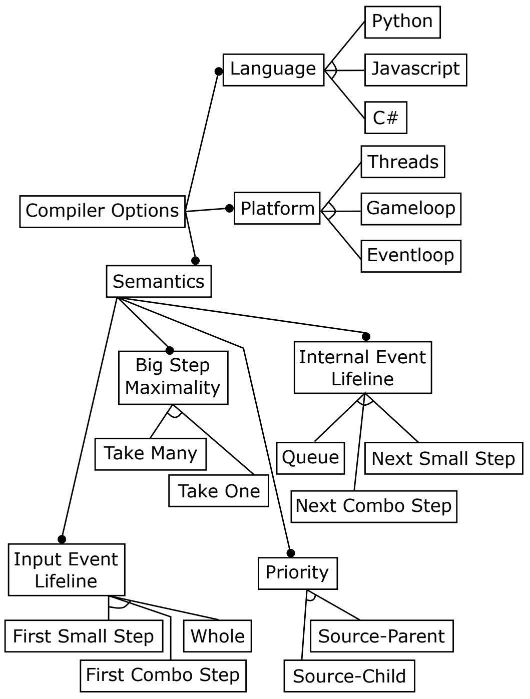

Figure 2. Feature diagram of the compiler options.

Runtime Platforms

The precisely defined semantics of SCCD are executed on a runtime platform, which provides essential functions used by the runtime kernel, such as the scheduling of (timed) events. Three runtime platforms are supported, each applicable in a different situation. The kernel attempts to run the SCCD model in real-time, meaning that the delay on timed transitions is interpreted as an amount of seconds. The raising of events and untimed transitions are executed as fast as possible. Figure 3 presents an overview of the three platforms, and how they handle events.

The most basic platform, available in most programming languages, is based on threads. Currently, the platform runs one thread, which manipulates a global event queue, made thread-safe by locks. Input from the environment is handled by obtaining this lock, which the kernel releases after every step of the execution algorithm. This allows external input to be interleaved with internally raised events. Running an application on this platform, however, can interfere with other scheduling mechanisms (*e.g.*, a UI module), or with code that is not thread-safe.

To overcome this interference problem, the event loop platform reuses the event queue managed by an existing UI platform, such as Tkinter. The UI platform provides functions for managing time-outs (for timed events), as well as pushing and

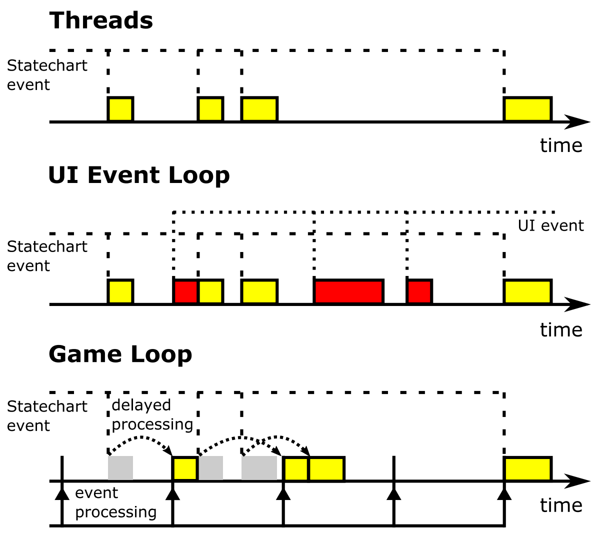

Figure 3. The three runtime platforms.

popping items from the queue. This results in a seamless integration of both Statecharts events and external UI events, such as user clicks: the UI platform is now responsible for the correct interleaving.

The game loop platform facilitates integration with game engines (such as the open-source Unity[1] engine), where objects are updated only at predefined points in time, decided upon by the game engine. In the "update" function, the kernel is responsible for checking the current time (as some time has passed since the last call to the "update" function), and process all generated events. This means that events generated in between two of these points are not processed immediately, but queued and their processing delayed until the next processing time.

Semantics

The Statecharts language has been around for a long time. In that time, its basic structures have almost not changed. In its original definition [3], Harel left many of the semantic choices undefined. Since then, many semantics have been defined, such as the one used in Statemate [6] or Rhapsody [5]. More recently, Esmaeilsabzali *et al.* [1] have performed a study of big-step modelling languages, such as Statecharts, and defined a set of semantic variation points, with which the different Statecharts execution semantics can be classified. Central to their discussion is the notion of a "big step". The execution of a Statecharts model is a sequence of big steps. A big step is a unit of interaction between a model and its environment, and corresponds to a "macrostep" in SCXML. A big step takes input from the environment (at the beginning of the big step), and produces output to the environment (after the big step has taken place). Input cannot change during the big step. A big step consists of 0 or more small steps. A small step is an unordered set of 1 or more transition executions, but in our case, a small step always consists of exactly one transition execution. Small steps are grouped in so-called combo steps. A combo step is a maximal sequence of small steps, such that it only contains transitions that are orthogonal to each other, and corresponds to a "microstep" in SCXML.

[1]`https://unity3d.com/`

The SCCD compiler allows to choose which semantics to use based on a number of semantic variation points. This gives modellers more control to fine-tune the application to their needs. The semantic variation points are listed below.

- **Big Step Maximality** specifies when a big step ends: either after one combo step executed (*Take One*), or when no more combo steps can be executed (*Take Many*).
- **Internal Event Lifeline** specifies when an internally raised event becomes available: either immediately in the next small step (*Next Small Step*), in the next combo step (*Next Combo Step*), or in the next big step, which means the event is queued and treated as an external event (*Queue*).
- **Input Event Lifeline** specifies when an input event is available during a big step: either throughout the first small step (*First Small Step*), the first combo step (*First Combo Step*), or throughout the whole big step (*Whole*).
- **Priority** specifies what to do when two transitions are enabled at the same time, where the source state of one of the transitions is the ancestor of the source state of the other transition. Either the transition of the ancestor gets priority (*Source-Parent*), or the transition of the (indirect) child gets priority (*Source-Child*).

LIMITATIONS AND FUTURE WORK

In its current state, an SCCD with several hundreds of objects runs smoothly on all platforms. Nevertheless, no extensive optimisation is incorporated in compiling an SCCD. It remains to be investigated whether the performance is adequate for an SCCD modelling a complete graphical user interface for graphical modelling involving possibly tens of thousands of objects, over multiple hierarchical levels. In the extreme case, thousands of objects might broadcast events to all objects at the same time. Ultimately, such *broadcast* scope might have to be restricted. In this sense, we intend to refine our notion of scope, by further refining the crucial role of the object manager.

The SCCD formalism supports specialisation for Class Diagrams (*i.e.,* inheritance of attributes, methods and associations). However, no specialization of the behaviour modelled in the Statecharts component of a class is implement as of today, which ideally includes specialization of Statecharts transitions and inheritance of events.

Because SCCD aims to model large, complex systems, we intend to add support for exceptions and exception handling to SCCD. Currently, when an object wants to signal an error, it has to send an error event. In the future, exceptions can be modelled as a special kind of event, and exception handling can be modelled as a dedicated SCCD. It remains to be investigated what the possibilities are when handling exceptions: can the exception handler (re)set a Statecharts's current state, or destroy objects and their Statecharts instances, and what are possible repercussions?

We also aim to introduce more object-oriented techniques to SCCD. Currently, events are strings, but modelling them as separate entities (such as classes) is useful, especially if a specialisation mechanism is implemented as mentioned above. This allows catching of events based on a supertype or on specific subtypes. This would also allow for classes to declare an interface, stating which types of events they accept on their input ports, and which types of events they will send on their output ports. This encourages re-usability, and is more general than the current prefix matching of event names in SCXML.

One major advantage of modelling using SCCD is its checks on the structure of the object diagram, to ensure that it conforms to the class diagram. Currently, however, minimal cardinality constraints are not always enforced, as they are necessarily violated in some period of time (the "initialisation phase") when an object is created. We plan to add a mechanism for classes to signal to the object manager when they finished their initialisation, after which it is safe to check minimal cardinality constraints.

SCCD currently supports low-level modelling of Statecharts interaction. In this context, we observe specific patterns in the design of complex user interfaces, for which dedicated support might decrease the complexity of SCCD models. One example of such a pattern is the modelling of hierarchical user interfaces, where resizing one window may trigger a ripple effect throughout all containers. Currently, such behaviour has to be implemented using scattered transitions, which decreases maintainability. We intend to identify typical patterns and address them individually, without overly complicating the SCCD formalism.

We aim to create a graphical representation and modelling tool for SCCD based on the graphical notations of Class Diagrams and Statecharts, to further increase usability of the formalisms and readability of models. In this respect, SCCDXML would serve not only as input for the compiler, but also as the interchange format for the graphically created models. Interestingly, creating such a modelling tool is very much the intended use of SCCD, and ultimately this modelling tool will be bootstrapped using an SCCD model.

RELATED WORK

Harel and Gery [4] propose OO Statecharts, similar to SCCD, to enable precise modelling of behaviour over time, which allows full executability and automatic code synthesis. In contrast to OO Statecharts, we focus on UI modelling, and SCCD is based on SCXML. In OO Statecharts, class methods can be called by synchronous function calls, whereas in SCCD no direct function calls are made. Object creation and destruction is always handled through synchronous function calls in OO Statecharts, whereas this is done through events to an explicit mediator (the object manager) in SCCD.

Forbrig *et al.* [2] support dynamic creation of parallel subcomponents in SCXML. Their example is an email client that can handle multiple emails at once. The problem the authors address is similar to ours, but we explicitly use Class Diagrams and objects as instances, which can be seen as agents that run concurrently and communicate asynchronously (which results in an entirely different semantics). In the context of embedded real-time software systems, Selic and Rumbaugh employ the UML, which later became known as UML-RT [7]. Similar to our approach, UML-RT addresses complex, event-driven, and, potentially, distributed systems. The notation is entirely UML-compliant, as a UML profile is used. So-called capsules roughly correspond to actors, similar to objects, to which a UML State Machine is associated. Similar to our approach, ports and connectors are used for communication between actors. The State Machines, however, cannot be compositional. Rudimentary support for inheritance of State Machines is supported in the sense that the State Machine is inherited, but no further constraints are defined according to the Liskov substitution principle. The UML-RT approach comes without compiler, and strictly follows UML semantics only.

CONCLUSION

This paper presents SCCD, a combination of Statecharts and Class Diagrams, for modelling the structure as well as the behaviour of complex, timed, interactive discrete-event systems. We presented the formalism, its representation in SCCDXML (an extension of SCXML), and a versatile compiler that supports multiple semantic variation points, platforms, and runtimes. In its current form, SCCD is suitable for the modelling of, amongst others, a graphical modelling tool's GUI. We discussed the limitations of the approach, setting the stage for future work.

ACKNOWLEDGEMENTS

This work was partly funded by Flanders Make vzw, as well as with PhD fellowship grants from the Agency for Innovation by Science and Technology in Flanders (IWT) and the Research Foundation - Flanders (FWO).

REFERENCES

1. Esmaeilsabzali, S., Day, N. A., Atlee, J. M., and Niu, J. Deconstructing the semantics of big-step modelling languages. *Requirements Engineering 15*, 2 (2010), 235–265.
2. Forbrig, P., Dittmar, A., and Khn, M. Extending SCXML by a feature for creating dynamic state instances. In *2nd Workshop on Engineering Interactive Systems with SCXML* (2015).
3. Harel, D. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program. 8*, 3 (1987), 231–274.
4. Harel, D., and Gery, E. Executable object modeling with statecharts. *IEEE Computer 30*, 7 (1997), 31–42.
5. Harel, D., and Kugler, H. The rhapsody semantics of statecharts (or, on the executable core of the uml). In *Integration of Software Specification Techniques for Applications in Engineering*, vol. 3147 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2004, 325–354.
6. Harel, D., and Naamad, A. The STATEMATE semantics of Statecharts. *ACM Trans. Softw. Eng. Methodol. 5*, 4 (Oct. 1996), 293–333.
7. Selic, B., and Rumbaugh, J. Using UML for modeling complex real-time systems. Technical report, 1998.