Proceedings of the 18th World Congress
The International Federation of Automatic Control
Milano (Italy) August 28 - September 2, 2011

Modelling Large-Scale Discrete-Event Systems Using Modules, Aliases, and Extended Finite-State Automata

Robi Malik * **Martin Fabian** ** **Knut Åkesson** **

* *Department of Computer Science, University of Waikato, Hamilton, New Zealand (e-mail: robi@cs.waikato.ac.nz)*
** *Department of Signals and Systems, Chalmers University of Technology, Göteborg, Sweden (e-mail: {fabian|knut}@chalmers.se)*

Abstract: This paper describes the instantiation features currently implemented in the discrete-event systems modelling and analysis tool *Supremica*. Modules enable users to design reusable groups of related automata and define clear interfaces to describe their interaction. Parametrisation and repetition make it easy to design very large models of regular structure. In combination with its support for extended finite-state automata (EFA), these features enable Supremica users to develop highly complex models of discrete-event systems using a wide variety of modelling styles.

Keywords: Discrete event systems modelling and control; Automata, Petri Nets and other tools.

1. INTRODUCTION

One of the key challenges for the acceptance of discrete event system modelling tools is the development of a user-friendly modelling language that is accepted by users in industry. For several years, *hierarchical* and *modular* modelling approaches have been used a means to introduce structure into discrete event systems (Wong and Wonham, 1998; Leduc *et al.*, 2005), but such methods have only recently been supported in more general modelling tools.

Other ways of modelling reactive or control systems are based on *extended finite-state automata (EFA)* or *Statecharts* (Harel, 1987), which use guards and actions to read and update variables while executing transitions. Several concurrency models are based on these ideas, e.g., (Hoare, 1985). More recently, (Sköldstam *et al.*, 2007) combine EFA with discrete event systems, providing a semantics for guards and actions in combination with event-based automata. This approach introduces signal-based or program-code based modelling styles, making discrete event systems more accessible to industrial users.

The modelling of large-scale systems often requires parametrisation and instantiation support that is both powerful and easy to learn and use. The VALID toolset (Brandin, 2000) introduces *templates* as a means to maintain multiple copies of similar automata. The idea is to create a single automaton, called *template*, which is *instantiated* multiple times by applying different event bindings, resulting in several similar copies of the same automaton. The more recent tool IDES (Grigorov *et al.*, 2008; Grigorov and Rudie, 2010) offers a more general template modelling approach, which supports flexible event bindings and extended finite-state automata.

This paper presents the modelling and parametrisation features presently available in the discrete events systems modelling tool Supremica (Åkesson *et al.*, 2006). Supremica uses *modules*, which provide more general ways of instantiation than templates, by grouping several automata together with defined interfaces. Modules naturally provide the infrastructure needed for modular and hierarchical modelling, and in combination with other features such as event groups and extended finite-state automata, they also provide for strong parametrisation capabilities.

This paper is organised as follows. Sect. 2 briefly gives some needed definitions of finite-state automata in the context of supervisory control theory. It is followed by sect. 3, which describes the instantiation and parametrisation features implemented in Supremica, and by sect. 4, which presents two examples of moderately complex models that can be conveniently instantiated using Supremica's features. Finally, sect. 5 adds some concluding remarks.

2. PRELIMINARIES

The modelling approach used by Supremica is based on *supervisory control theory* of discrete event systems (Ramadge and Wonham, 1989; Wonham, 2009; Cassandras and Lafortune, 1999). System behaviours are represented using finite-state automata and their languages.

Definition 1. A *finite-state automaton* is a 5-tuple $G = \langle \Sigma, Q, \rightarrow, Q^i, Q^m \rangle$, where Σ is a finite *alphabet* of events, Q is a set of *states*, $\rightarrow \subseteq Q \times \Sigma \times Q$ is the *state transition relation*, $Q^i \subseteq Q$ is the set of *initial states*, and $Q^m \subseteq Q$ is the set of *marked* or *terminal states.*

When two ore more automata are executing in parallel, synchronous composition in the style of (Hoare, 1985) is used to compose them.

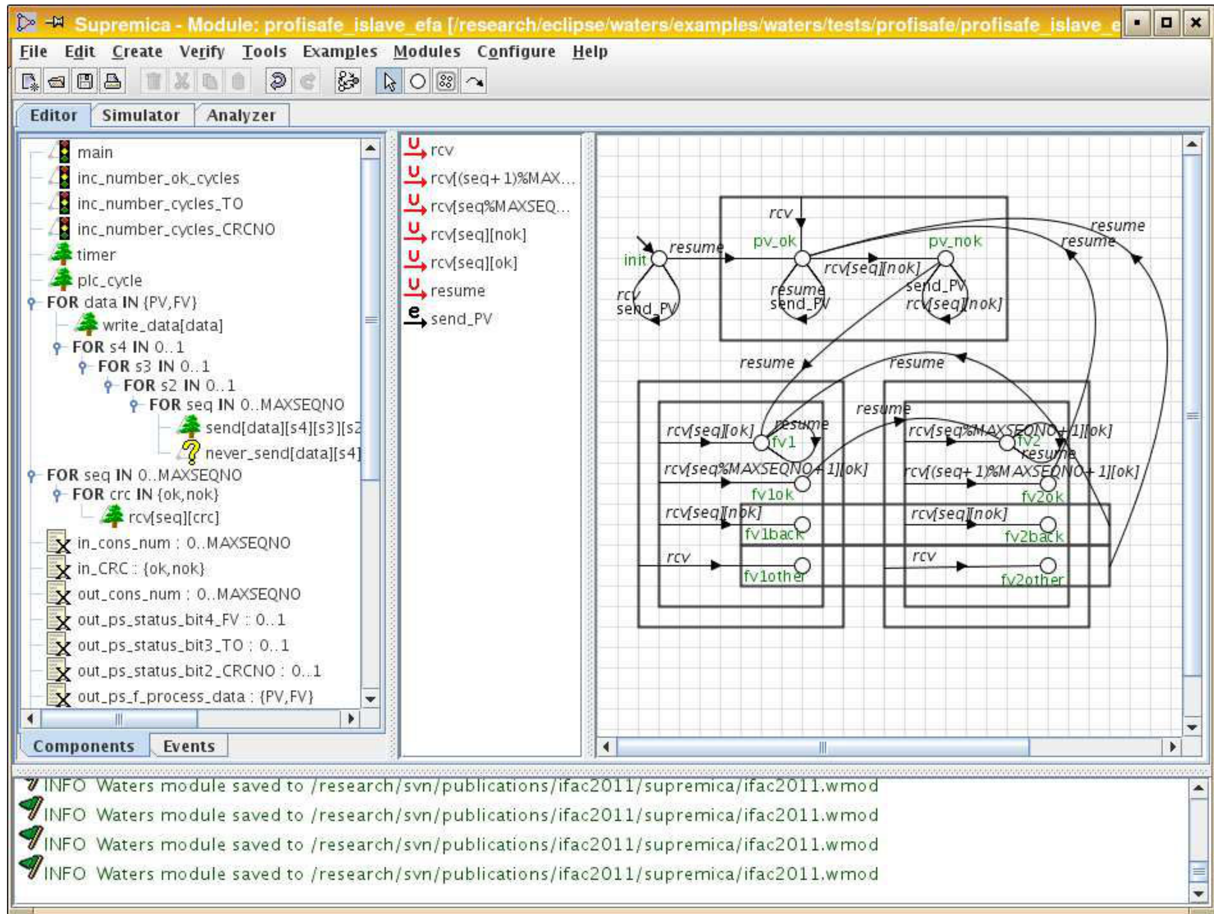


Fig. 1. Supremica IDE.

Definition 2. Let $G = \langle \Sigma_G, Q_G, \rightarrow_G, Q_G^i, Q_G^m \rangle$ and $H = \langle \Sigma_H, Q_H, \rightarrow_H, Q_H^i, Q_H^m \rangle$ be two automata. The *synchronous product* of G and H is

$$G \| H = \langle \Sigma_G \cup \Sigma_H, Q_G \times Q_H, \rightarrow, Q_G^i \times Q_H^i, Q_G^m \times Q_H^m \rangle \quad (1)$$

where

$$\begin{aligned}
(x_G, x_H) &\xrightarrow{\sigma} (y_G, y_H) && \text{if } \sigma \in \Sigma_G \cap \Sigma_H,\ x_G \xrightarrow{\sigma}_G y_G, \text{ and } x_H \xrightarrow{\sigma}_H y_H;\\
(x_G, x_H) &\xrightarrow{\sigma} (y_G, x_H) && \text{if } \sigma \in \Sigma_G \setminus \Sigma_H \text{ and } x_G \xrightarrow{\sigma}_G y_G;\\
(x_G, x_H) &\xrightarrow{\sigma} (x_G, y_H) && \text{if } \sigma \in \Sigma_H \setminus \Sigma_G \text{ and } x_H \xrightarrow{\sigma}_H y_H.
\end{aligned}$$

In supervisory control (Ramadge and Wonham, 1989), there are two types of automata, representing the *plant*, i.e., the system to be controlled, and the *specification*, i.e., the desired behaviour to be imposed on the plant through control. The event set Σ is partitioned into the set Σ_u of *uncontrollable* events and the set Σ_c of *controllable* events. Controllable events are under the control of some supervising agent or control software, and can be disabled through control, while uncontrollable events occur spontaneously in the plant.

3. MODELLING

Supremica (Åkesson *et al.*, 2006) is a tool for the modelling and analysis of finite-state machine models. Discrete event systems consisting of multiple plant and specification automata can be modelled graphically using the *editor*. Model execution can be visualised using the *simulator*, and the *analyser* provides a wide range of verification and synthesis algorithms. All these features are available in the comfortable graphical interface of Supremica's Integrated Development Environment (IDE), shown in fig. 1.

Supremica's graphical editor is designed to support the modelling of large-scale discrete event systems models using a wide variety of modelling styles. The following subsections describe the modelling approach and its features in more detail.

3.1 Modules and Instances

The main modelling unit in Supremica is a *module*, which represents a reusable collection of events and associated automata. Events can be controllable or uncontrollable, and automata can be classified as plant or specification. For example, fig. 2 shows a module that describes the behaviour of a robot, consisting of four events and two automata. Events and automata are listed to the left, and the two plant automata are shown to the right.

Each automaton has an event alphabet, which in Supremica is the set of all events that appear in the graphical representation. Within a module, automata synchronise using shared events as per the standard definition of synchronous product.

In addition to its events and automata, a module may contain *instances* of other modules. Instantiating a module means to include copies of all automata and events of

module Robot
events
  controllable parameter *enable*;
  controllable parameter *disable*;
  controllable *turn*;
  controllable *extend*;
components
  plant base;
  plant arm;
end

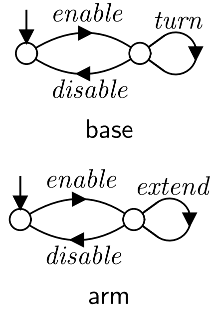


Fig. 2. Robot module.

the used (or instantiated) module within the calling (or instantiating) module.

Each module has its unique event set, and by default does not share any events with other modules. Events are local to their module and are only used for synchronisation of the automata within the module. When a module A contains two instances of another module B, this means that two separate independent copies of all the events and automata of B are created and placed in A. The two copies of B neither synchronise with each other nor with A.

To support synchronisation between modules, events can be defined to be a *parameter* of their module. When a module with parameters is instantiated, the instantiating module needs to provide an *actual* event from its own event set for each parameter of the instantiated module. After instantiation, the parameter event is replaced by the actual event, which may now synchronise with automata in other modules.

module $\mathsf{Factory}_2$
events
  controllable $start_1$;
  controllable $start_2$;
  controllable *stop*;
components
  $\mathsf{Robot}_1 = \mathsf{Robot}(enable = start_1,\ disable = stop)$;
  $\mathsf{Robot}_2 = \mathsf{Robot}(enable = start_2,\ disable = stop)$;
end

Fig. 3. Factory consisting of two robots.

In fig. 3, two instances of the Robot from fig. 2 are created to model a factory. This results in two copies of each of the automata base and arm being placed in the $\mathsf{Factory}_2$ module, which then contains automata Robot_1.base, Robot_1.arm, Robot_2.base, and Robot_2.arm. The events *turn* and *extend* are not parameters of the Robot module. These events remain *local*, and separate copies $\mathsf{Robot}_1.turn$, $\mathsf{Robot}_2.turn$, $\mathsf{Robot}_1.extend$, and $\mathsf{Robot}_2.extend$ are created, synchronising only within their Robot instance.

The parameter events *enable* are replaced by the events $start_1$ resp. $start_2$ of the $\mathsf{Factory}_2$ module, which may be shared by other automata of $\mathsf{Factory}_2$ (not shown in the example). The parameter events *disable* of both modules are replaced by the same event *stop*, effectively forcing synchronisation between the two robots. In this way, the events *enable* and *disable* in the robot module are linked to the factory's events $start_1$, $start_2$, and *stop* in a similar way as proposed in the template approach of (Grigorov *et al.*, 2008).

3.2 Arrays and Repetition

Many large discrete-event systems models have regular structure, consisting of several similar automata with similar events. Often, the number of components is large or subject to parametrisation, making it difficult to instantiate the automata or modules individually. Supremica supports such models using arrays of events and indexed components.

Events can be declared to be arrays with any number of dimensions, the only restriction being that all events in an array must have the same controllability status. Likewise, automata and module instances can be indexed, also producing an array-like collection of components.

module $\mathsf{Factory}_5$
events
  controllable $start[1..5]$;
  controllable *stop*;
components
  for $i = 1..5$ **do**
    $\mathsf{Robot}[i] = \mathsf{Robot}(enable = start[i], disable = stop)$;
  end
end

Fig. 4. Factory consisting of five robots.

In fig. 4, $start[1..5]$ is declared as an event array, which is expanded to five controllable events $start[1], \ldots, start[5]$. The array elements can be accessed by their index, which is used in the **for** construct to instantiate the Robot module from fig. 2 five times, producing five robot instances $\mathsf{Robot}[1], \ldots, \mathsf{Robot}[5]$, each parametrised by different *start* events, while synchronising on one *stop* event.

3.3 Extended Finite-State Automata

The pure finite-state machine approach is not suitable for all modelling tasks in industry, particularly where PLC programs or variables are involved. To bridge the gap between the signal-based industrial reality and the event-based supervisory control framework, (Sköldstam *et al.*, 2007) propose a form of *extended finite-state automata (EFA)*. Variables, guard expressions and action functions enable an alternative style of specification, and make it possible to parametrise not only the number of events or automata in a model, but also the number of states or the transition structure of individual automata.

Supremica supports extended finite-state automata as proposed in (Sköldstam *et al.*, 2007; Grigorov and Rudie, 2010). Modules can contain *variables* of finite range, and all transitions can be associated with guards and actions. A *guard* is a formula over the module's variables with the meaning that a transition can only be taken when the guard evaluates to true. *Actions* associated with a transition define how the variables are updated when the transition is taken.

Fig. 5 shows the use of these features to model a buffer with parametrisable size. The variable c, which can assume the values from 0 to *size*, stores the number of items currently

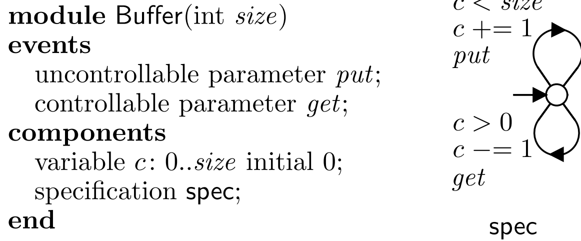


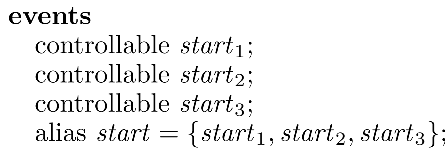
Fig. 5. Parametrisable buffer module.

in the buffer. Its initial value is 0, i.e., the buffer initially is empty. The specification automaton is only needed to associate guards and actions to the events: an item can only be put into the buffer when it is not yet full, and this action increases the number of items in the buffer by one; and an item can only be removed from the buffer when it is not empty, and this action decreases the number of items in the buffer by one.

3.4 Event Groups and Aliases

Large discrete-event systems models may contain automata using a large number of events, and often the same groups of events appear on different transitions in the same or in different automata. To facilitate the design and the readability in such cases, event groups and aliases have been introduced in Supremica.

An *event alias* is a placeholder for an event or a group of events, for example:


events
controllable $start_1$;
controllable $start_2$;
controllable $start_3$;
alias $start = \{start_1, start_2, start_3\}$;

Here, the events $start_1$, $start_2$, and $start_3$ form an event group and are bound to the alias *start*. The alias can be used anywhere where an ordinary event can be used: it can be placed on transitions of automata or it can be bound as an actual value to the parameters of an instantiated module. According to standard discrete-event systems semantics, a transition labelled with the alias *start* will be enabled when one or more of the events in the group are enabled.

3.5 Group Nodes

Group nodes are a graphical abstraction inspired by Statecharts (Harel, 1987). Their use for modelling discrete event systems was introduced in VALID (Brandin, 2000) and extended in (Ma and Wonham, 2005). In Supremica, group nodes are used to simplify the graphical representation of an automaton by making it possible to group transitions to the same target state together. For example, in fig. 6, the *reset* transition originating from the group node represents four transitions to the automaton's initial state, one from each state within the group.

4. EXAMPLES

This section presents two examples that demonstrate how Supremica's modelling and instantiation features help to model complex discrete event systems easily and concisely.

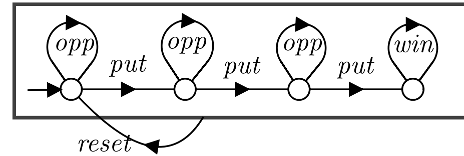


Fig. 6. Automaton with group node.

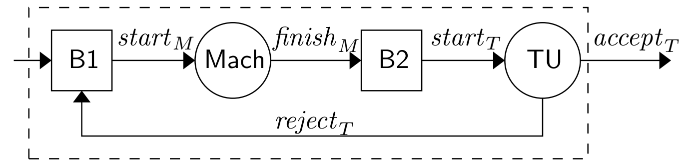


Fig. 7. Functional block of transfer line.

4.1 Transfer Line

The first example is a parametrised version of the transfer line model originally proposed in (Wonham, 2009). The model consists of n functional blocks as shown in fig. 7. Each block consists of a machine Mach and a test unit TU, linked by two buffers B1 and B2. The machine can start ($start_M$) and finish ($finish_M$) operating. Finished workpieces are placed into buffer B2, from where they are loaded into the test unit TU ($start_T$), which in turn accepts ($accept_T$) or rejects ($reject_T$) them. If accepted, a workpiece is released and transferred to the next functional block, if rejected it is returned to buffer B1 to be processed by Mach again. The buffer capacities of B1 and B2 are 3 and 1, respectively.

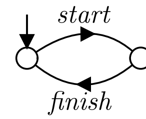
Fig. 8 shows a module representing the transfer line in Supremica. The model is parametrised by the number n of functional blocks. Event arrays are declared, so the events controlling the machines, test units, and buffers can be indexed by their functional block. The **for** construct instantiates the necessary automata for each functional block i. The modules Machine and TestUnit contain only one plant automaton as shown in the figure, with all events declared as parameters. The buffers are instantiated from the Buffer module introduced in sect. 3.3, parametrised by the size of each buffer.

When instantiating the buffers, their events are bound to the appropriate actions of the machines and test units. Work pieces can be placed into buffer B1$[i]$ in functional block i either by releasing them from the previous block $i-1$, or by rejecting them from the test unit of the current block i. Therefore, an event group is bound to the *put* event of B1$[i]$, so the corresponding transitions are linked to both $accept_T[i-1]$ and $reject_T[i]$. This also shows the use of simple arithmetic to link events from the current functional block i to the previous block $i-1$.

To complete the transfer line model, an initial loading unit Init, used to load work pieces into the first block of the transfer line, is instantiated from the Machine module.

Despite its apparent simplicity, the state space of the transfer line model increases dramatically with the number of functional blocks. While a model with one functional block has just 64 reachable states, a model with 10 functional blocks already has $2^{51} = 2,251,799,813,685,248$ reachable states. Due to their regular structure, all these

module TransferLine(int n)
events
  controllable $start_M[1..n]$;
  uncontrollable $finish_M[1..n]$;
  controllable $start_T[0..n]$;
  uncontrollable $accept_T[0..n]$;
  uncontrollable $reject_T[1..n]$;
components
  Init = Machine($start = start_T[0], finish = accept_T[0]$);
  for $i = 1..n$ **do**
    Mach[i] = Machine($start = start_M[i], finish = finish_M[i]$);
    TU[i] = TestUnit($start = start_T[i], accept = accept_T[i], reject = reject_T[i]$);
    B1[i] = Buffer($size = 3; put = \{accept_T[i-1], reject_T[i]\}, get = start_M[i]$);
    B2[i] = Buffer($size = 1; put = finish_M[i], get = start_T[i]$);
  end
end



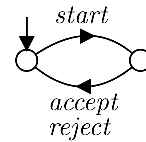
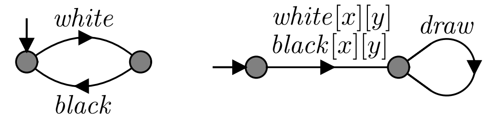
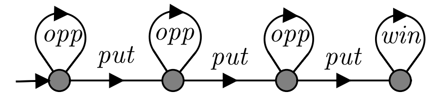


Fig. 8. Transfer line module.

module TicTacToe
events
  controllable $black[0..2][0..2]$;
  uncontrollable $winner_black_x[0..2]$;
  uncontrollable $winner_black_y[0..2]$;
  uncontrollable $winner_black_d1$;
  uncontrollable $winner_black_d2$;
  alias $winner_black = \{winner_black_x, winner_black_y,$
      $winner_black_d1, winner_black_d2\}$;
  uncontrollable $white[0..2][0..2]$;
  uncontrollable $winner_white_x[0..2]$;
  uncontrollable $winner_white_y[0..2]$;
  uncontrollable $winner_white_d1$;
  uncontrollable $winner_white_d2$;
  alias $winner_white = \{winner_white_x, winner_white_y,$
      $winner_white_d1, winner_white_d2\}$;
  uncontrollable $draw$;
components
  plant move;
  for $x = 0..2$ **do**
    for $y = 0..2$ **do**
      plant square[x][y];
    end
  end
  for $x = 0..2$ **do**
    RowWhiteX[x] = Row($put = \{white[x][0], white[x][1], white[x][2]\}, win = winner_white_x[x], opp = black$);
    RowBlackX[x] = Row($put = \{black[x][0], black[x][1], black[x][2]\}, win = winner_black_x[x], opp = white$);
  end
  for $y = 0..2$ **do**
    RowWhiteY[y] = Row($put = \{white[0][y], white[1][y], white[2][y]\}, win = winner_white_y[y], opp = black$);
    RowBlackY[y] = Row($put = \{black[0][y], black[1][y], black[2][y]\}, win = winner_black_y[y], opp = white$);
  end
  RowWhiteD1 = Row($put = \{white[0][0], white[1][1], white[2][2]\}, win = winner_white_d1, opp = black$);
  RowBlackD1 = Row($put = \{black[0][0], black[1][1], black[2][2]\}, win = winner_black_d1, opp = white$);
  RowWhiteD2 = Row($put = \{white[0][2], white[1][1], white[2][0]\}, win = winner_white_d2, opp = black$);
  RowBlackD2 = Row($put = \{black[0][2], black[1][1], black[2][0]\}, win = winner_black_d2, opp = white$);
  plant game_over;
  spec white_never_wins;
end





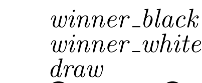


Fig. 9. Tic Tac Toe module.

transfer lines can be obtained from a single parametrisable module with three simple automata. Supremica has successfully instantiated and analysed transferlines with more than 100 functional blocks and more than 10^{150} reachable states.

4.2 Tic Tac Toe Game

The second example models the rules of the Tic Tac Toe game and can be used synthesise a strategy for this game. Two players, *white* and *black*, are taking turns to occupy the squares of a 3×3 board. The winner is the player who first occupies a horizontal, vertical, or diagonal row of three squares.

Fig. 9 shows a module representing this game. It uses two two-dimensional arrays for the moves of the two players. For example, $white[x][y]$ indicates that square (x, y) is occupied by the white player. The model is intended to synthesise a strategy for the *black* player, so *white* moves are uncontrollable and *black* moves are controllable. The plant automaton move models the fact that the two players are taking their moves alternatingly, and *white* moves first. Although *white* and *black* are event arrays, the automaton just uses the event labels *white* and *black*, which represent implicitly defined event groups containing all nine events in their array. This feature makes it possible to model the alternation of moves concisely and independently of the exact set of possible moves.

The plant automaton square$[x][y]$ models the rule that each square can be occupied only once. It is instantiated nine times, once for each square (x, y).

There are eight possible ways how a player can win, corresponding to the eight possible rows that can be formed. Each winning condition is represented by an uncontrollable event, using arrays for the three horizontal and vertical rows. Aliases *winner_white* and *winner_black* group the winning conditions for each player together. To correctly capture the winning conditions, the Row automaton needs to be instantiated differently for each row. The automaton models the fact, that a row is completed (*win*) when its three squares are all occupied (*put*). Once a row is completed, the game is over and the opponent of the winning player can now longer move (*opp*). To instantiate all the winning conditions, the Row automaton is placed in a module of its own, which is instantiated 16 times with appropriate bindings for the events *win*, *put*, and *opp*.

Two final automata complete the model. Plant game_over uses the aliases *winner_white* and *winner_black* to model the liveness requirement that it must always be possible for the game to end, either by one player winning, or by draw. Automaton white_never_wins represents a control specification that disables all winning events of the white player. It uses a *blocked events list* to ensure that all events of the *winner_white* group are in the automaton alphabet although there are no transitions associated with them. This feature of Supremica enables the modelling of globally disabled events without the overhead of adding unreachable states. The specification white_never_wins can be used to synthesise a least restrictive strategy (Ramadge and Wonham, 1989) ensuring that the black player never loses the game.

5. CONCLUSIONS

The modelling features implemented in the discrete events systems tool Supremica have been presented. The software is freely available for download [1] and includes a user-friendly graphical interface to model complex finite-state machine models. Module instantiation and event groups make it easy to create large and parametrised models by specifying only a small number of automata that are reused several times. The support for extended finite-state machines provides for variables, guards, and actions, thus supporting not only event-based models, but also signal-based and state-based modelling styles, and styles that are more closely related to program code.

REFERENCES

Åkesson, Knut, Martin Fabian, Hugo Flordal and Robi Malik (2006). Supremica—an integrated environment for verification, synthesis and simulation of discrete event systems. In: *Proc. 8th Int. Workshop on Discrete Event Systems, WODES '06*. Ann Arbor, MI, USA. pp. 384–385.

Brandin, B. (2000). VALID tool set: Validation of software applications. In: *Proc. 5th Int. Workshop on Discrete Event Systems, WODES '00*. Ghent, Belgium.

Cassandras, C. G. and S. Lafortune (1999). *Introduction to Discrete Event Systems*. Kluwer.

Grigorov, L., J. E. R. Cury and K. Rudie (2008). Design of discrete-event systems using templates. In: *Proc. American Control Conf. 2008*. Seattle, Washington, USA. pp. 499–504.

Grigorov, Lenko and Karen Rudie (2010). Techniques for the parametrization of discrete-event system templates. In: *Proc. 10th Int. Workshop on Discrete Event Systems, WODES '10*. Berlin, Germany. pp. 380–385.

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

Hoare, C. A. R. (1985). *Communicating Sequential Processes*. Prentice-Hall.

Leduc, Ryan J., Bertil A. Brandin, Mark Lawford and W. M. Wonham (2005). Hierarchical interface-based supervisory control—part I: Serial case. *IEEE Trans. Automat. Contr.* **50**(9), 1322–1335.

Ma, Chuan and W. Murray Wonham (2005). *Nonblocking Supervisory Control of State Tree Structures*. Vol. 317 of *LNCIS*. Springer.

Ramadge, Peter J. G. and W. Murray Wonham (1989). The control of discrete event systems. *Proc. IEEE* **77**(1), 81–98.

Sköldstam, M., K. Åkesson and M. Fabian (2007). Modeling of discrete event systems using finite automata with variables. In: *Proc. 46th IEEE Conf. Decision and Control, CDC '07*. pp. 3387–3392.

Wong, K. C. and W. M. Wonham (1998). Modular control and coordination of discrete-event systems. *Discrete Event Dyn. Syst.* **8**(3), 247–297.

Wonham, W. M. (2009). Supervisory control of discrete-event systems. Systems Control Group, Dept. of Electrical Engineering, University of Toronto, ON, Canada; at http://www.control.utoronto.edu/DES/.

[1] `http://www.supremica.org`