Supremica in a Nutshell – Draft

Knut Åkesson, Martin Fabian and Hugo Flordal

October 9, 2007

1 Introduction

Embedded computers are often used to implement control functions for reactive systems. Formal verification techniques, like model checking, may be used to guarantee that the control functions behave as expected in all circumstances. However, an alternative approach is to automatically synthesize control functions from high-level descriptions that are correct by construction. While formal verification techniques have been developed mainly by the computer science community, formal synthesis of control functions has been developed in the control community where reactive systems are commonly referred to as *discrete event systems* and their control functions is named *supervisor*.

The supervisory control theory (SCT) is a general framework for verification and synthesis of discrete event supervisors that has shown promising results. However, in order for SCT to be accepted in industry, user friendly tools able to solve large problems are critical. Supremica is an attempt to build an integrated development environment that is able to solve large scale supervisor verification and synthesis problems. Supremica is free for education and research and may be downloaded from `www.supremica.org`.

2 Supervisory Control Theory

Reactive systems have been a research field within computer science and engineering for a long time. However, with no control theoretic background, the main focus is on *verification* of, typically already controlled, reactive systems rather than the *synthesis* of control functions for an uncontrolled system. The *Supervisory Control Theory* (SCT) took a control-theoretic model-based approach, applying formal reasoning on a model of the uncontrolled process, the *plant*, and a model of the desired behavior of the controlled system denoted the *specification*. From the plant and the specification a safety device, called a *supervisor*, can be automatically synthesized. The supervisor controls the plant to always stay within the limits set by the specification, by dynamically disallowing the plant to generate events that might otherwise have been generated.

The SCT proves that given a plant and a specification there will always exist an optimal supervisor guaranteeing that the specification will not be broken, while at the same time allowing the system to always fulfill its defined (sub-)tasks. Optimality concerns here restricting the given plant as little as absolutely necessary. Such a supervisor is said to be *maximally permissive*, since it allows the controlled system the largest possible amount of freedom, in terms of event-generation, within the constraints set by the plant and the specification.

The control theoretic contribution concerns the inclusion of a certain type of "controllability". The supervisor is mainly a safety device that hinders the plant from executing events that would take the controlled system outside the specified behavior. However, not all events can be hindered from occurring, some events are *uncontrollable*, and the supervisor must never (try to) disable any of the uncontrollable events. It is known that for a given specification and plant, a supervisor that guarantees that the entire specification can be achieved exists if and only if the specification is *controllable*. This means that the specification must be such that it can be enforced without having to (try to) disable any uncontrollable events. If the specification is not controllable, it is further known that a supervisor still exists, but this supervisor can only achieve a sub-behavior of the specification, namely what is known as a *controllable sub-language*. Even more, it is also known that a unique optimal such supervisor exists and is readily calculatable, and this supervisor will achieve the *supremal controllable sublanguage* of the specification.

In addition to controllability, which is a safety property, it is desired for the supervisor to be *non-blocking*. This is a progress property enforced by the supervisor that guarantees that at least one *marked* state is reachable from any state that it allows the controlled system to reach. Marked states typically represent (sub-)tasks that the system must always be able to finish. Typically, the initial state is a marked state, guaranteeing that under supervision of a non-blocking (and controllable) supervisor the task that the system performs can be performed again and again. As above, it is known that the *supremal controllable and non-blocking sublanguage* of a specification with respect to a given plant, exists.

Though the SCT traditionally has focused on synthesis of supervisors, verification is a natural step within synthesis. Synthesis can be viewed as a series of verification tasks, where the process model (the plant) allows the automatic alteration of the suggested, and negatively verified, supervisor. In this respect, the original specification can be viewed as a first supervisor candidate; if it is verified to be correct (controllable and non-blocking) then no further processing is necessary. Thus, by construction, a synthesized supervisor will always be verified to be correct.

To summarize, a maximally permissive, controllable and non-blocking supervisor for a given specification and plant always exists but may be expensive to calculate due to the state-space explosion problem.

3 Modeling

In Supremica models can be entered as a ordinary finite automaton or as an extended finite automaton. En extended finite automaton is an ordinary finite automaton extended with with variables, guard expressions and action functions.

3.1 Ordinary Finite Automata

Ordinary finite automata are defined as follows.

Definition 1 (Nondeterministic finite automaton) *An* nondeterministic finite automaton *is a 5-tuple* $G = \langle Q, \Sigma, \rightarrow, Q^i, Q^m \rangle$ *where* Q *is a* finite *set of* states*;* Σ*, the* alphabet*, is a nonempty finite set of* events*;* $\rightarrow \subseteq Q \times \Sigma \times Q$ *is the* transition relation*;* $Q^i \subseteq Q$ *is the set of* initial states*; and* $Q^m \subseteq Q$ *is the set of* marked states*.*

The controllability of an event is a global property and the alphabet Σ can be partitioned into the sets Σ_c and Σ_u of *controllable* and *uncontrollable* events, respectively.

The transition relation is written in infix notation, for example, $p \xrightarrow{\sigma} q$ denotes a *transition* from state p to state q associated with the event σ. This notation is further extended to strings in Σ^* in the natural way. For state sets $Q_1, Q_2 \subseteq Q$, the notation $Q_1 \xrightarrow{s} Q_2$ denotes the existence of some $q_1 \in Q_1$ and some $q_2 \in Q_2$ such that $q_1 \xrightarrow{s} q_2$.

Automata (plants, specifications and supervisors alike) running in parallel interact under lock-step synchronization.

Definition 2 *Let* $G_1 = \langle Q_1, \Sigma_1, \rightarrow_1, Q_1^i, Q_1^m \rangle$ *and* $G_2 = \langle Q_2, \Sigma_2, \rightarrow_2, Q_2^i, Q_2^m \rangle$ *be two automata. The* synchronous composition *of* G_1 *and* G_2 *is*

$$G_1 \parallel G_2 \ = \ \langle Q_1 \times Q_2, \Sigma_1 \cup \Sigma_2, \rightarrow, Q_1^i \times Q_2^i, Q_1^m \times Q_2^m \rangle \qquad (1)$$

$$\text{where} \quad \begin{array}{l} (p,q) \xrightarrow{\sigma} (p',q') \ \text{if}\ \sigma \in (\Sigma_1 \cap \Sigma_2),\ p \xrightarrow{\sigma}_1 p',\ q \xrightarrow{\sigma}_2 q' \,; \\ (p,q) \xrightarrow{\sigma} (p',q) \ \text{if}\ \sigma \in \Sigma_1 \setminus \Sigma_2,\ p \xrightarrow{\sigma}_1 p' \,; \\ (p,q) \xrightarrow{\sigma} (p,q') \ \text{if}\ \sigma \in \Sigma_2 \setminus \Sigma_1,\ q \xrightarrow{\sigma}_2 q' \,. \end{array}$$

The synchronous composition is also useful when modeling large systems because it allows the user to build multiple sub-models, and the global behavior can then be described using the sub-models and the synchronous composition operator.

The behaviour of a system may, for the purposes of this paper, be represented by its languages, i.e. the sets of strings that the system may generate.

Definition 3 (Languages) *Let* $G = \langle Q, \Sigma, \rightarrow, Q^i, Q^m \rangle$ *be an automaton. The* language *of* G*, denoted* $\mathcal{L}(G)$ *and the* marked language *of* G*, denoted* $\mathcal{M}(G)$ *are defined as*

$$\begin{aligned} \mathcal{L}(G) &= \{ s \in \Sigma^* \mid Q^i \xrightarrow{s} Q \} \,, \\ \mathcal{M}(G) &= \{ s \in \Sigma^* \mid Q^i \xrightarrow{s} Q^m \} \,. \end{aligned}$$

Now the properties controllability and nonblocking can be defined formally, in the definitions below we assume that the plant and the specification are deterministic.

Definition 4 (Controllability) *Let* G *and* K *be two automata with the same alphabet* Σ*.* K *is* controllable with respect to G *if* $\mathcal{L}(G \| K)\Sigma_u \cap \mathcal{L}(G) \subseteq \mathcal{L}(G \| K)$*.*

Definition 5 (Nonblocking) *Let* G *be an automaton.* G *is* nonblocking *if* $\mathcal{L}(G) \subseteq \mathcal{M}(G)$*.*

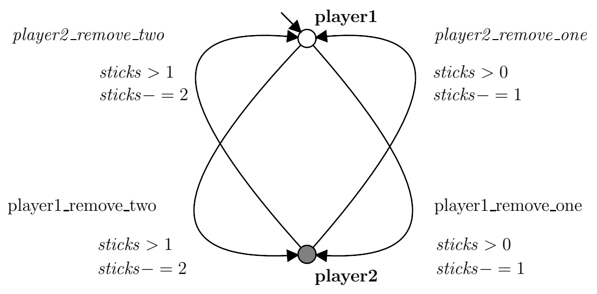


Figure 1: EFA model of a stick-picking game with two players.

A supervisor S is nonblocking with respect to a plant G if $G \,\|\, S$ is nonblocking.

The basic supervisory control problem then concerns the following. Given a plant G and a specification K, calculate a supervisor S such that:

$$\begin{array}{lrcl}
\text{i)} & \mathcal{L}(G\,\|\,S) & \subseteq & \mathcal{L}(G\,\|\,K) \\
\text{ii)} & \mathcal{M}(G\,\|\,S) & \subseteq & \mathcal{M}(G\,\|\,K) \\
\text{iii)} & \mathcal{L}(G\,\|\,S) & \subseteq & \overline{\mathcal{M}(G\,\|\,S)} \\
\text{iv)} & \mathcal{L}(G\,\|\,S)\Sigma_u \cap \mathcal{L}(G) & \subseteq & \mathcal{L}(G\,\|\,S) \\
\text{v)} & \mathcal{L}(G\,\|\,S') & \subseteq & \mathcal{L}(G\,\|\,S), \forall S' \in \mathcal{CNB}(G, K)
\end{array}$$

Conditions i) and ii) state that the controlled closed-loop behavior must be included in the specified closed-loop behavior. Condition iii) means that the closed-loop system must be non-blocking. Condition iv) states that the supervisor must be controllable with respect to the plant. Finally, condition v) states that all other controllable and non-blocking supervisor candidates must be more restrictive than S, i.e. S has to be the maximally permissive supervisor.

3.2 Extended Finite Automata

Extended Finite Automata (EFA) are an augmentation of ordinary automata, extended with variables, guard formulas and action functions. An EFA modeling a stick-picking game is shown in Figure 1. We associate the guards and actions to the transitions in the automaton. The transitions in the EFA are enabled if and only if the guard formula is true. When a transition is taken, updating actions of a set of variables may follow. For example, the event *player2_remove_two* is enabled when the guard-formula *sticks* > 1 is satisfied, and if the event is executed the action function is triggered and the value of the *sticks* variable is decreased by 2.

An Extended Finite Automaton (EFA) is an augmentation of an ordinary automaton with guard formulas and action functions. An EFA modeling a stick-picking game is presented in Fig. 1. We associate the guards and actions to the transitions in the automaton. The transitions in the EFA are enabled if and only if the guard formula is true. When a transition is taken, updating actions of a set of variables may follow. For example, the event *player2_remove_two* is enabled when the guard-formula *sticks* > 1 is satisfied, and if the event is executed the action function is triggered and the value of the *sticks* variable is decreased by 2. To define an EFA formally we introduce the characteristic function χ_W of a set W. χ_W is defined as $\chi_W(v) = 1$ if $v \in W$, 0 otherwise.

If $\chi_W(v) = 1$ the predicate "$v \in W$" is true, and if $\chi_W(v) = 0$ the predicate is false.

Definition 6 (Extended Finite Automaton)
An extended finite-state automaton E is a 6-tuple

$$E = \langle Q \times V, \Sigma, \mathcal{G}, \mathcal{A}, \rightarrow, (q_0, v_0)\rangle,$$

where:

(*i*) *$Q \times V$ is the extended finite set of states, where Q is a set of* locations *and V is the domain of definition of the variables;*

(*ii*) *Σ is a nonempty finite set of events (the alphabet);*

(*iii*) *$\mathcal{G} = \{\chi_W \mid W \in 2^V\}$ is the set of guard predicates. Each guard $g \in \mathcal{G}$ defines a set of variable values for which the statement is true.*

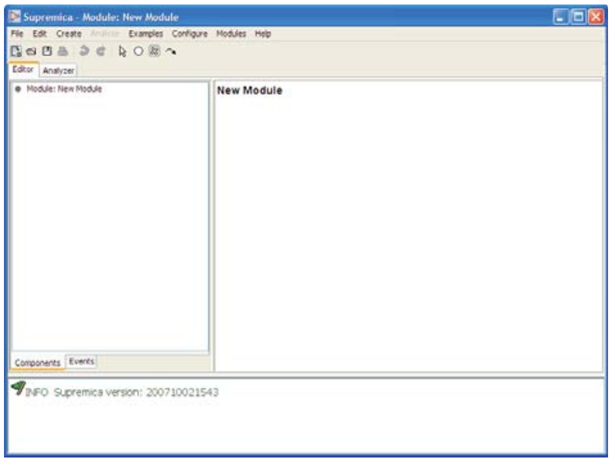
(*iv*) *$\mathcal{A} = \{a \mid a$ is a function from V to $V_\Xi\}$ is a collection of action functions.*

(*v*) *$\rightarrow \subseteq Q \times \Sigma \times \mathcal{G} \times \mathcal{A} \times Q$ is the state transition relation.*

(*vi*) *$(q_0, v_0) \in Q \times V$ is the initial state.*

We have extended the states of the ordinary automaton to $Q \times V$, where $V = V^1 \times ... \times V^n$. The finite set V is the domain of definition of an n-tuple of variables $v = (v^1, \ldots, v^n)$ with initial values $v_0 = (v_0^1, \ldots, v_0^n) \in V$. The *guards* are predicates over the variables that relate each element of V, to either *true* or *false*. We are interested in deterministic EFA, and therefore the *actions* are functions. Guards and actions are written in function notation $g = g(v)$ and $a = a(v) = (a^1(v), \ldots, a^n(v))$ where $v \in V$. The actions are functions from V to $V_\Xi = V_\Xi^1 \times ... \times V_\Xi^n$, where $V_\Xi^i = V^i \cup \{\Xi\}$, $i = 1, \ldots, n$. The symbol Ξ is used to denote a don't care updating of a variable. If $a^i(v') = \Xi$, we say that $a^i(v')$ is a *don't care updating of the variable v^i*. The transition relation is written as $p \xrightarrow{\sigma}_{g/a} q$, where $p, q \in Q$, $\sigma \in \Sigma$, $g \in \mathcal{G}$ and $a \in \mathcal{A}$. If g is absent, it is assumed that g always evaluates to true and the transition takes place when σ occurs. If a is absent, it is assumed that $a(v) = (\Xi, \Xi, \ldots, \Xi)$ and no variable is updated during the transition. The extension of EFA to include markings on locations and variable values as well as controllable and uncontrollable transitions is straightforward.

Two EFA may be composed using an operator similar to the Full Synchronous Composition (FSC) operator that is traditionally used to model the interaction between concurrently executing automata. Informally, an event in the composed system is enabled when the event is allowed from the current location in both the EFA, and both associated guard formulas are satisfied. When the event triggers, both associated action functions are executed simultaneously. For the resulting state to be well-defined it is required that the two EFA must be consistent. Informally consistency implies that the two EFA never try to assign different values to the same variable when executing a shared event. To simplify the notation when defining the synchronous product of two EFA, we assume that they share all variables. This is no restriction since we can add don't care variables that are never updated.


Supremica - Module: New Module
File Edit Create Analyze Examples Configure Modules Help
Editor Analyzer
Module: New Module
New Module
Components Events

INFO Supremica version: 200710021543

Figure 2: Supremica Main Window.

Definition 7 (Full Synchronous Composition)
Let $E_k = \langle Q_k \times V, \Sigma_k, \rightarrow_k, (q_0^k, v_0)\rangle$*,* $k = 1, 2$*, be two EFA using the shared variables* $v = (v^1, \ldots, v^n)$*. The Full Synchronous Composition (FSC) of* E_1 *and* E_2 *is*

$$E_1 \| E_2 \quad = \quad \langle Q_1 \times Q_2 \times V, \Sigma_1 \cup \Sigma_2, \rightarrow, (q_0^1, q_0^2, v_0)\rangle,$$

where the state transition relation $\rightarrow$ *is defined as*

* $(p_1, p_2) \overset{\sigma}{\rightarrow}_{g/a} (q_1, q_2)$*,* $\sigma \in \Sigma^1 \cap \Sigma^2$ *if*
  $\exists (p_1, \sigma, g_1, a_1, q_1) \in \rightarrow_1, \exists (p_2, \sigma, g_2, a_2, q_2) \in \rightarrow_2$ *such that:*

  (i) $g = g_1 \wedge g_2$,

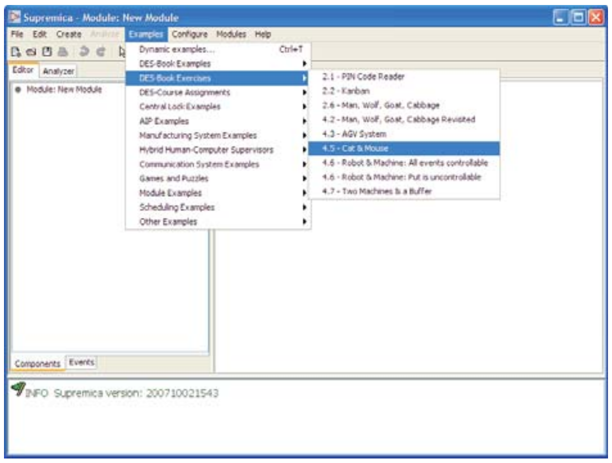
  (ii)
  $$a^i(v) = \begin{cases} a_1^i(v) & \text{if } a_1^i(v) = a_2^i(v) \\ a_1^i(v) & \text{if } a_2^i(v) = \Xi \\ a_2^i(v) & \text{if } a_1^i(v) = \Xi \end{cases} ;$$

* $(p_1, p_2) \overset{\sigma}{\rightarrow}_{g/a} (q_1, q_2)$*,* $\sigma \in \Sigma^1 \setminus \Sigma^2$ *if*
  $(p_1, \sigma, g, a, q_1) \in \rightarrow_1$ *and* $p_2 = q_2$*;*

* $(p_1, p_2) \overset{\sigma}{\rightarrow}_{g/a} (q_1, q_2)$*,* $\sigma \in \Sigma^2 \setminus \Sigma^1$ *if*
  $(p_2, \sigma, g, a, q_2) \in \rightarrow_2$ *and* $p_1 = q_1$*.*

4 Using Supremica

After starting Supremica you will see something similar to Figure 2. The easiest way to get started is to open one of the included examples. The *cat and mouse* example is a classic within the supervisory control community. To open up the example select `Examples -> DES-Book Exercises -> Cat & Mouse` from the menu, see Figure 3.


Supremica - Module: New Module
File
Edit
Create
Analyze
Examples
Configure
Modules
Help
Dynamic examples... Ctrl+T
DES-Book Examples
DES-Book Exercises
DES-Course Assignments
Central Lock Examples
AIP Examples
Manufacturing System Examples
Hybrid Human-Computer Supervisors
Communication System Examples
Games and Puzzles
Module Examples
Scheduling Examples
Other Examples
2.1 - PIN Code Reader
2.2 - Kanban
2.6 - Man, Wolf, Goat, Cabbage
4.2 - Man, Wolf, Goat, Cabbage Revisited
4.3 - AGV System
4.5 - Cat & Mouse
4.6 - Robot & Machine: All events controllable
4.6 - Robot & Machine: Put is uncontrollable
4.7 - Two Machines & a Buffer
Editor
Analyzer
Module: New Module
Components
Events

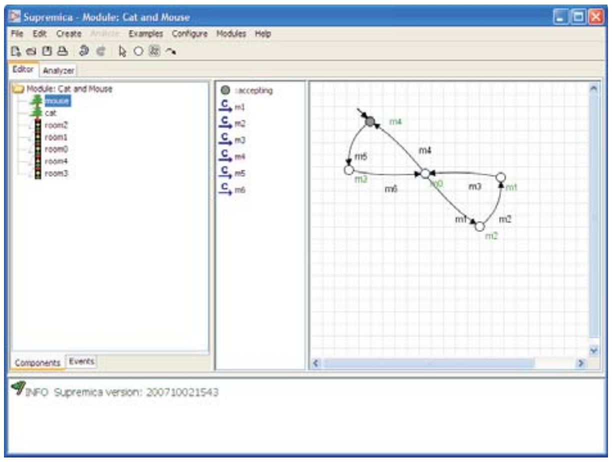
```
INFO  Supremica version: 200710021543
```

Figure 3: Open the Cat & Mouse example in Supremica.

The Cat & Mouse system consists of seven finite automata models. There is one model of the cat and model of the mouse, both are *plant*-models. Theses two plant models describes how the cat and mouse may move between five different rooms. The other five models are specifications and there is one specification for every room. The specifications models that either the cat or the mouse are allowed to be in each room at the same time instant. By clicking on a model name in the left panel a graphical representation of the automaton is displayed in the right panel, see Figure 4. Note, that plant models have a tree icon in front of their name and that specifications have a traffic light in front of their name. It is possible to change the type of a model by right-clicking with the mouse on the model name and setting a new type. Also, note how it possible to move states and events in editor panel. If you get tired of manually organizing states and transitions you could try the automatic layout function by pressing `Ctrl-L`.

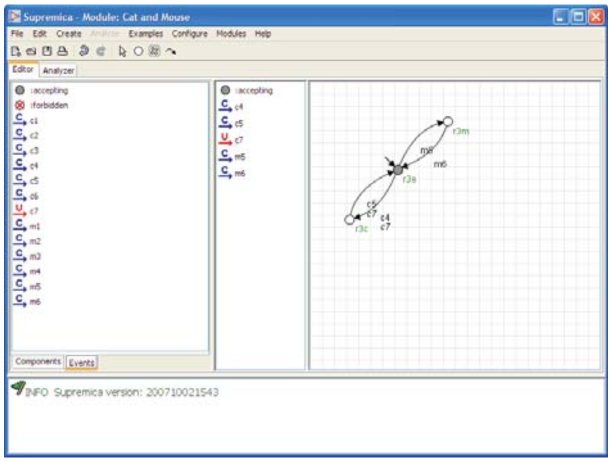
The same panel that contains all automata models (denoted as Components in Supremica) can also be changed to display all events in the module. A module consists of a set of a set of automata that belongs together. The events in the Cat & Mouse system is shown in Figure 5. *Controllable*-events has a C-icon in front of their name and *uncontrollable events* have a U-icon. You can change the controllability by right-clicking on an event name. The events in the middle panel are the events that are used by the currently edited automaton. In most cases this is equal to the alphabet of the edited automaton, but for some intricate reasons this is not always the case.

To create your own model select `File -> New` from the menu. To create a new automaton select `Create -> Component`. In the dialog give the component a name and choose the type. Double-click on the component name to start editing the component. States and transitions are created by select the corresponding tool in the toolbar, see Figure 6. New events are created by selecting `Create -> Component Event`. Events added to a transition by first choosing the select tool (the tool with the arrow) and the clicking on the event in the automation, The event name will changed to blue and the event is removed from



```
INFO  Supremica version: 200710021543
```

Figure 4: Supremica presenting a graphical representation of the mouse plant model.



```
INFO  Supremica version: 200710021543
```

Figure 5: The module events.

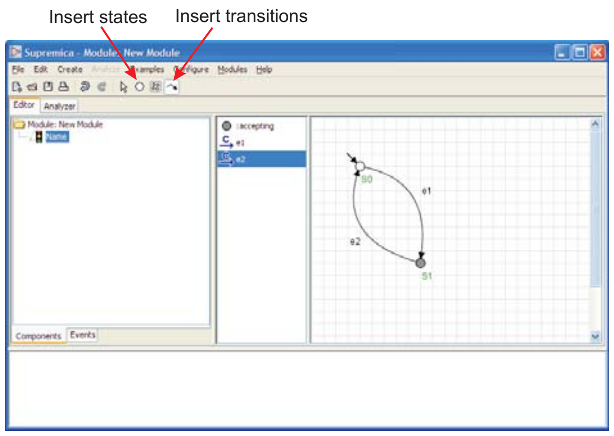


Figure 6: Select the right tool for inserting states or transitions. Drag-and-drop events from the event panel to the transitions.

the transition by pressing the `delete` button.

4.1 Analyzer

Now return to the Cat & Mouse example by select `Modules -> Cat & Mouse` in the menu.

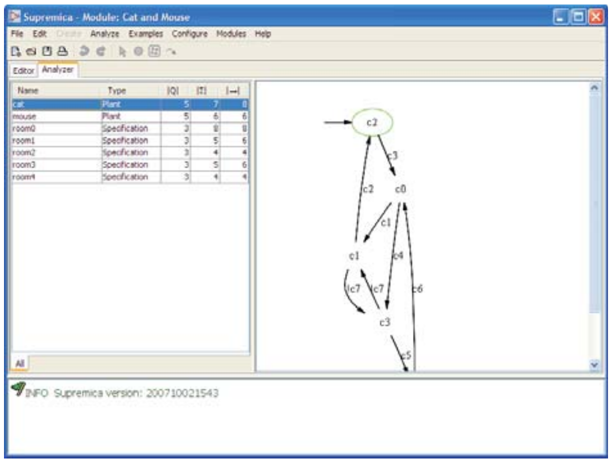
To start analyzing the Cat & Mouse example you now change from Editor mode to Analyzer mode by selecting the Analyzer tab, by double-clicking on a component name you will get something similar to Figure 7[1].

4.1.1 Synchronization

In the analyzer mode you can now start to do sophisticated analysis of your models. For small examples it is possible to compute the synchronous composition of the models. This is done by selecting all automata names and then right-clicking and selecting `Synchronization`. Select the default options in the dialog. As result you will now have a new automaton called `cat||mouse||room0||room1||room2||room3||room4`. By double-clicking on the new model name and maximizing the window you will see something similar to Figure 8.

Note, that two states in Figure 8 are surrounded by (red) rectangles, this is used to denote that these states are forbidden. These two states are forbidden because they represent states where the plant might generate an uncontrollable event that the specification tries to disable, i.e. they are uncontrollable states. It is a good exercise to examine the original models and try to explain why the plant may generate something that the specification tries to disable. Synchronizing all automata to get the complete behavior is possible to do for very simple examples but when it is possible it can give some insight to the user

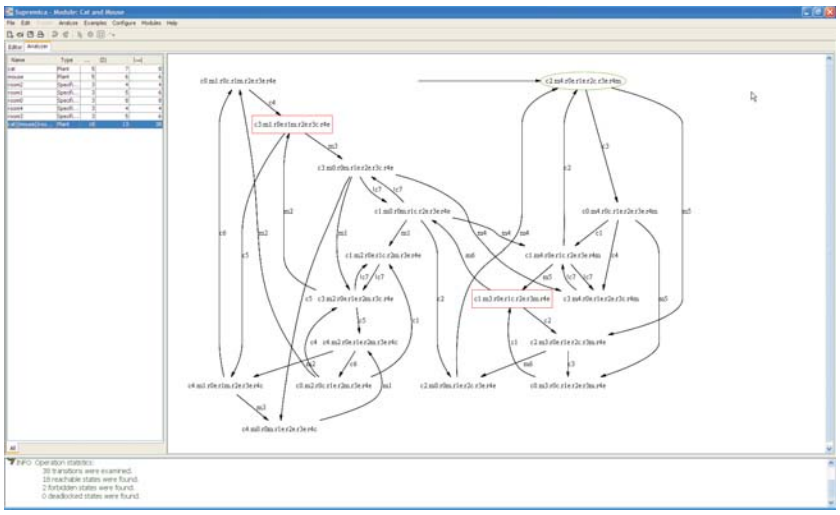
[1]The graphical representation in the analyzer is different from the editor mode. The main reason is because the analyzer in some situation has to compute a graphical layout to the models and thus an automatic layout is always generated.


Supremica - Module: Cat and Mouse
File Edit Create Analyze Examples Configure Modules Help
Editor Analyzer
All
c2
c3
c0
c1
c4
c6
c7
c5

Name	Type	|Q|	|Σ|	|→|
cat	Plant	5	7	8
mouse	Plant	5	6	6
room0	Specification	3	8	8
room1	Specification	3	5	6
room2	Specification	3	4	4
room3	Specification	3	5	6
room4	Specification	3	4	4

```
INFO Supremica version: 200710021543
```

Figure 7: Supremica in analyzer mode. Double click on the automata names to get a graphical representation to the right.


Supremica - Module: Cat and Mouse
File Edit Create Analyze Examples Configure Modules Help
Editor Analyzer
All

```
INFO Operation statistics:
        30 transitions were examined.
        18 reachable states were found.
        2 forbidden states were found.
        0 deadlocked states were found.
```

Figure 8: The synchronization between all the models in the Cat & Mouse example.

since all behavior will be included in the automaton that is the result of the synchronization procedure.

Synchronizing all automata together is something you try to avoid for all but simple toy examples. What you really would like to in Supremica is instead to figure out if a system is controllable, nonblocking and if they are not to automatically synthesize supervisors in such a way that the new system becomes controllable or nonblocking or both.

Now remove the automaton
`cat||mouse||room0||room1||room2||room3||room4` by selecting the name and pressing delete.

4.1.2 Verification

To verify if the Cat & Mouse example is controllable or nonblocking, again select all the automata, right click and select `Verify`. A dialog shows up, the first you have to choose is the property you would like to verify, for example controllability or nonblocking. The second option is the algorithm you will use to carry out the verification, for small examples you can choose `Monolithic (Explicit)`. This implies that all states will be enumerated while carrying out the verification. This is only possible for small examples and hence other algorithms have been developed that avoids explicitly enumerating all states. After having made the appropriate choices run the verification and in this case the algorithm will tell you that the system is not controllable. The reason is the same as we have for explaining why the got forbidden states when synchronizing all the automata models. To verify nonblocking do the same but just change the property to be verified.

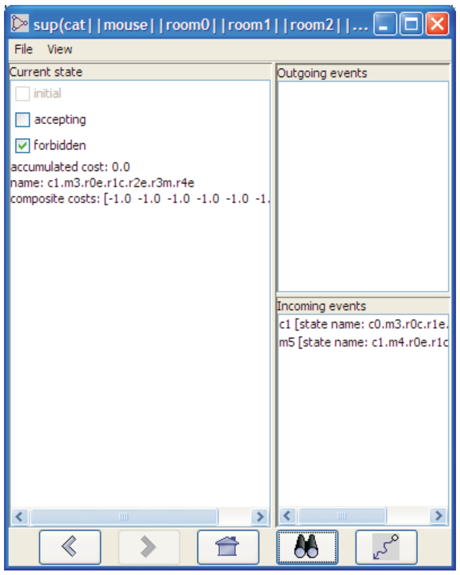
4.1.3 Synthesis

The main purpose with Supremica is to be able to do synthesis. Since the Cat & Mouse example is not controllable and supervisor has to be generated. To do this, follow the same procedure as for verification. Select all models, press the right-mouse button. In the dialog select the property that you would like the supervisor to have, in most cases you select `Nonblocking and controllable`. Again for small examples it is enough to choose the algorithm `Monolithic (Explicit)`, but for more complicated examples you could try some other algorithm. If you do not select the `Purge result` option the states that is not accepted by the supervisor will be denoted as forbidden, by selecting the `Purge result` option all states that will forbidden as a result of the synthesis procedure will also be removed.

Workbench There is also a Workbench that is suitable for newcomers to the field of SCT. The workbench allows you to go through all iterations in the synthesis procedure step-by step. The workbench is started in the same way as above.

4.2 Exploring the State Space

For small examples the state space might be explored by visually examine the automata. However, for larger examples this is not possible any longer. For



```
accumulated cost: 0.0
name: c1.m3.r0e.r1c.r2e.r3m.r4e
composite costs: [-1.0  -1.0  -1.0  -1.0  -1.0  -1.
```

```
c1 [state name: c0.m3.r0c.r1e.
m5 [state name: c1.m4.r0e.r1c
```

Figure 9: The Main Explorer Window.

these situations we have developed the *Explore States* tool, which you also invoke the same was the verification/synthesis/workbench tools. If a single automaton is selected as input the Explorer tool it is also possible to use a tool for search for states with specific properties. For example is it possible to search for a deadlock states. If such a state is found the state is selected with a double click, this will take you back to the main Explorer window, see Figure 9. The previously found state is now the current state in the Explorer Window. To compute a shortest trace from the initial state to the current state select the right most button in the bottom of the window. Now you could go to the initial state by clicking on the House-button (middle button in the window). Now it is possible to figure out how to go from the initial state to the previously found state, i.e. the deadlock state. This is done by iteratively selecting the blue events among the enabled outgoing events until the state is reached.