Supremica–An Efficient Tool for Large-Scale Discrete Event Systems

Robi Malik * **Knut Åkesson** ** **Hugo Flordal** ***
Martin Fabian **

* *The University of Waikato, Hamilton, New Zealand (robi@waikato.ac.nz)*
** *Chalmers University of Technology, Göteborg, Sweden (fabian@chalmers.se, knut@chalmers.se)*
*** *Prover Technology, Stockholm, Sweden (hugo.flordal@prover.com)*

Abstract: *Supremica* is a tool for the modelling and analysis of discrete-event control functions based on state machine models of the uncontrolled plant and specification of the desired closed-loop behaviour. The modelling framework in *Supremica* is based on finite-state machines extended with variables, guard conditions, and action functions. In order to handle large-scale problems of industrially interesting size, *Supremica* uses advanced model checking techniques such as symbolic representations and compositional abstraction. *Supremica* has been used in several industrial research projects to verify and synthesise control functions for embedded controllers, industrial robots, and flexible manufacturing systems, and to verify program code for autonomous vehicles. This paper gives an overview of the modelling features of *Supremica*, shows the verification and synthesis facilities and their performance for large problems, and presents some of the industrial applications where *Supremica* has been used.

Keywords: Discrete Event Systems, Supervisory Control, Extended Finite-State Machines, Synthesis, Verification, Formal Methods.

1. INTRODUCTION

The supervisory control theory (Ramadge and Wonham, 1989) is a general framework for modelling, verification, and synthesis of discrete event supervisors, which has shown promising results mainly within academia. In order for the supervisory control theory to be accepted in industry, user friendly tools able to solve large-scale problems of industrially interesting sizes are critical. *Supremica* (Åkesson et al., 2003, 2006) is an attempt to build an integrated development environment that can to solve large-scale supervisor verification and synthesis problems.

This paper presents *Supremica* and the main ideas of the supervisory control theory for modelling discrete event systems, as well as the technology for synthesising and verifying control functions. The presentation focuses on the tool from a user perspective. For a more theoretical description of the modelling framework with extended finite-state machines, the reader may refer to Sköldstam et al. (2007) and Mohajerani et al. (2016). For more detailed presentations of algorithms, as well as further benchmarks, see for example Åkesson et al. (2002), Mohajerani et al. (2014), and Pilbrow and Malik (2015).

Several research groups have developed tools for discrete event systems, but few combine an easily accessible user interface with high-performance algorithms. TCT (Feng and Wonham, 2006) is a popular and efficient tool with textual user interface. The Integrated Discrete-Event Systems Tool (Rudie, 2006) and DESTool (Moor et al., 2008) have nice user interfaces but their algorithms do not achieve the same performance as *Supremica*.

In the following, Section 2 gives a brief overview of the supervisory control theory. Then Section 3 describes *Supremica* and its features, Section 4 shows some experimental data to demonstrate its performance with large control problems, and Section 5 describes some recent applications of the tool. Finally, Section 6 adds concluding remarks. This paper gives a more detailed account on *Supremica* than previous publications (Åkesson et al., 2003, 2006), and describes recently added features. *Supremica* is constantly evolving, and the latest release can always be downloaded, free for education and research, from `www.supremica.org`.

2. SUPERVISORY CONTROL THEORY

Reactive systems have been a research field within computer science and engineering for a long time. *Model checking* (Baier and Katoen, 2008) provides a means to verify the correctness of, typically already controlled, reactive systems against specifications of their required properties. The *supervisory control theory* (Ramadge and Wonham, 1989) takes a control theoretic model-based approach.

In the supervisory control theory, the system model is separated into the *plant*, which represents a physical system to be controlled, and the *specification*, which represents the desired behaviour of the controlled system. The plant and

⋆ *Supremica* has been developed with financial support from the Swedish Research Council under grants 621-2001-2154 and 621-2009-3768.

specification are typically modelled as *discrete event systems*, which are finite-state machines. Control is done by a so-called *supervisor*, which is a safety device that hinders the plant from executing certain events. Some events are *controllable* and can be disabled by this supervisor, while other events are *uncontrollable* and cannot be disabled.

Given a specification and plant, a supervisor that guarantees that the entire specification can be achieved, exists if and only if the specification is *controllable.* Controllability is a safety property that states that all uncontrollable events possible in the controlled system must be permitted by the specification. If the specification is not controllable, then it may still be possible to construct a supervisor, but this supervisor only achieves a sub-behaviour of the specification. Even more, it is known that a *supremal controllable sublanguage* of the given plant and specification language exists and is readily calculable. This language may be empty, if there is no way to controllably enforce the specification. Otherwise, it represents a *least restrictive* supervisor which implements an optimal solution to the control problem (Ramadge and Wonham, 1989).

In addition to controllability, it is desired for the supervisor to be *nonblocking.* This is a progress property that guarantees that at least one *marked* state is reachable from any state that can be reached in the controlled system. Marked states typically represent states where all pending tasks have been completed. As above, the *supremal controllable and nonblocking sublanguage* of a specification with respect to a given plant exists and can be calculated.

Controllability and nonblocking can be checked automatically using model checking techniques, and if they are satisfied, then the specification itself can be used as an optimal supervisor. If not satisfied, then model checking produces a *counterexample* (Baier and Katoen, 2008), which is a trace of behaviour that takes the system to an undesired state. By inspecting counterexamples, engineers can gain a deeper understanding of the system, enabling them to discover and fix problems. *Synthesis* (Ramadge and Wonham, 1989) goes one step further and automatically produces an optimal controllable and nonblocking supervisor.

While the supervisory control theory has shown promising results within academia, industrial adoption is hitherto scarce. There seem to be two main reasons for this: the modelling formalism, which is not straightforward for the average control engineer typically used to Matlab/Simulink; and the lack of useful tools able to handle the complexity of real-life industrial applications. *Supremica*'s main focus has so far been on the second problem. Now, being able to handle systems of industrial sizes, the focus shifts to incorporate modelling, verification, and synthesis into the everyday life of engineers.

3. THE SUPREMICA IDE

Supremica is designed as an Integrated Development Environment (IDE) that allows the creation, verification, and synthesis of discrete event system models. Fig. 1 shows a screenshot. The main components of the IDE are the *Editor* that creates state machines graphically, the *Simulator* that animates the system behaviour, and the *Analyser* that supports a variety of finite-state machine algorithms.

3.1 Editor

The *Supremica* editor provides a graphical user interface to create and modify discrete event system models. Models consisting of several finite-state machines can be saved in a single file. They are displayed and edited graphically, with states and transitions labelled by easy-to-use drag & drop operations. As common in supervisory control theory, state machines can be designated to be *plants* or *specifications.* Events can be designated to be *controllable* or *uncontrollable* as well as *observable* or *unobservable.*

Supremica supports ordinary finite-state machines (FSM) and extended finite-state machines (EFSM). The transitions of ordinary FSMs are labelled by events only, while EFSMs are a generalisation and have transitions labelled with *guards* and *actions* in addition to events (Chen and Lin, 2000). The guards and actions reference *variables*, which can be declared over finite integer ranges or as enumerated type. A transition in an EFSM is enabled if the guard formula is true, and if it is taken, the variables are updated in accordance with the actions on the transition.

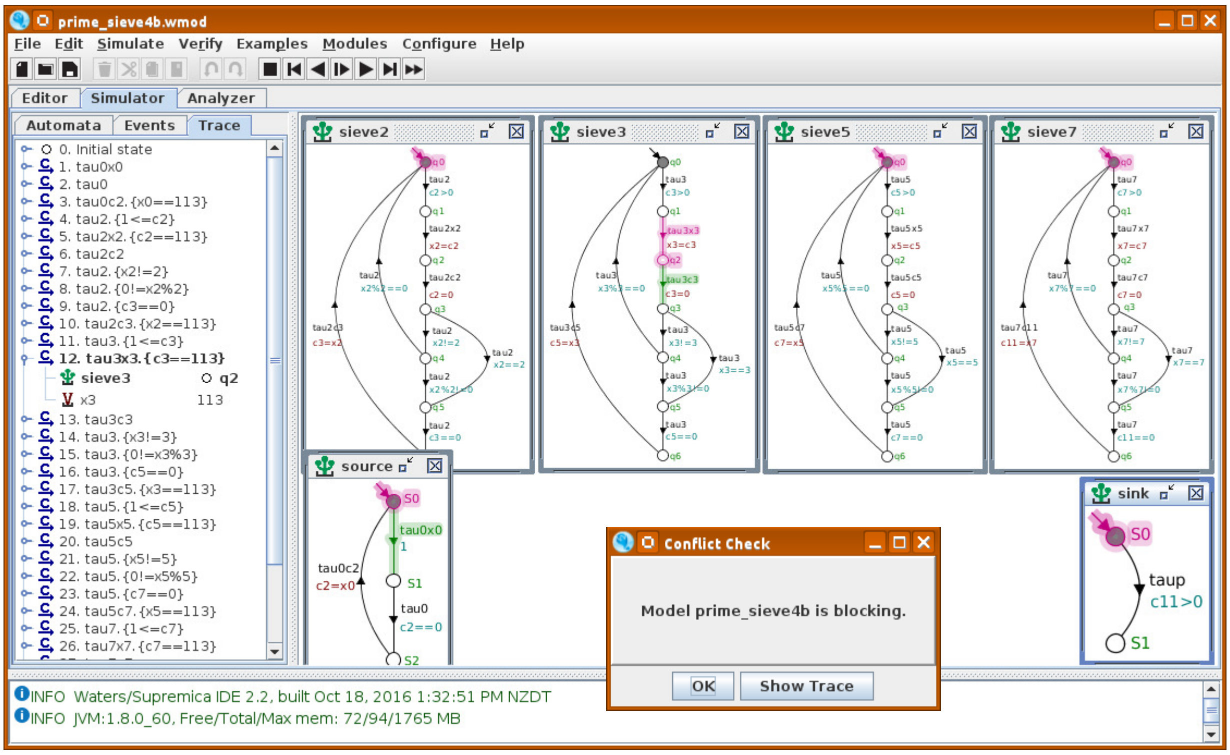
Fig. 2 shows such an EFSM, which is part of a model of a distributed prime sieve algorithm. Its variables `c3`, `c5`, and `x3` are defined over the integer range $0, \ldots, 24$. The transition from `q0` to `q1` is labelled with event `tau3` and has a guard $\texttt{c3} > 0$, which means that the transition can only be taken when `c3` is greater than zero. The transition from `q1` to `q2` has an action $\texttt{x3} = \texttt{c3}$, which means that the variable `x3` will take as new value the value that `c3` currently has. Guards and actions can include simple arithmetics, for example the guard of the transition from `q4` to `q0` uses the modulo operator (%) to enable the transition only when the value of `x3` is divisible by 3.

Additionally, *Supremica* supports parametrisation to facilitate the creation of large models (Malik et al., 2011). It is possible to create several similar state machines (FSMs or EFSMs) by substitution of parameter variables in a template, or to create transitions with parametrised sets of events. State machines can also be grouped into modules and re-used with different parameter instantiations. As an example, these features make it possible to create a model of a manufacturing line with a variable number n of identical machines, and then instantiate it to get the correct model for any value of n.

3.2 Simulator

The *Supremica* simulator can display several state machines in a model and show their synchronised behaviour. At any time, the simulator displays the current state of all state machines and the currently eligible events. The user can choose which event to execute and observe the resulting state changes. It is possible to step forwards and backwards through the history, and it is easy to determine for each event whether it is enabled, and if not, which state machine is responsible for its disablement.

Synchronisation between FSMs is done by handshaking based on common events (Hoare, 1985). That is, if two



```
0. Initial state
1. tau0x0
2. tau0
3. tau0c2.{x0==113}
4. tau2.{1<=c2}
5. tau2x2.{c2==113}
6. tau2c2
7. tau2.{x2!=2}
8. tau2.{0!=x2%2}
9. tau2.{c3==0}
10. tau2c3.{x2==113}
11. tau3.{1<=c3}
12. tau3x3.{c3==113}
    sieve3      q2
    x3          113
13. tau3c3
14. tau3.{x3!=3}
15. tau3.{0!=x3%3}
16. tau3.{c5==0}
17. tau3c5.{x3==113}
18. tau5.{1<=c5}
19. tau5x5.{c5==113}
20. tau5c5
21. tau5.{x5!=5}
22. tau5.{0!=x5%5}
23. tau5.{c7==0}
24. tau5c7.{x5==113}
25. tau7.{1<=c7}
26. tau7x7.{c7==113}
```

Model prime_sieve4b is blocking.

```
INFO  Waters/Supremica IDE 2.2, built Oct 18, 2016 1:32:51 PM NZDT
INFO  JVM:1.8.0_60, Free/Total/Max mem: 72/94/1765 MB
```

Fig. 1. Screenshot of counterexample simulation.

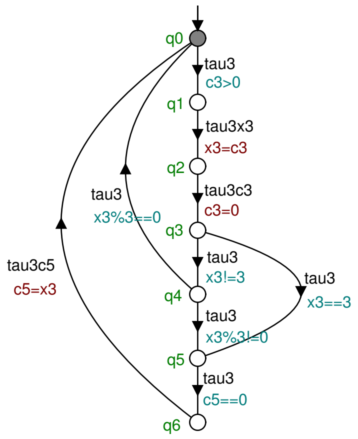


Fig. 2. Extended finite-state machine (EFSM).

or more state machines use the same event, then this event can only be executed if all these state machines are in a state where the event is enabled, and if executed, then all these state machines change their state together. EFSMs are also synchronised based on shared events, and in addition guards and actions are treated as logical formulas and combined by conjunction.

Internally, the simulator is based on ordinary FSMs, as are most analysis algorithms in *Supremica*. EFSMs are translated into an equivalent groups of FSMs for computation and simulation. As there are different semantics (Sköldstam et al., 2007; Mohajerani et al., 2016) for transforming EFSMs into ordinary automata, a few user-settable alternatives are available in *Supremica*.

One problem with the translation from EFSM into FSM models is the possibly exponential blow-up in the number of events, which can cause performance problems for large models. This problem has been recognised, and recent algorithms (Miremadi et al., 2011; Mohajerani et al., 2016) bypass the translation into FSMs and work directly on the EFSM model.

3.3 Analyser

The *Supremica* analyser provides access to a wide range of FSM analysis and simplification algorithms. It is used as a workbench to experiment with FSMs and the settings of the various verification and synthesis algorithms available.

3.4 Verification

Both the editor and analyser include verification functionality to check the model automatically and detect possible errors. *Supremica* supports the standard discrete event system properties of *controllability* and *nonblocking* (Ramadge and Wonham, 1989). It is also possible to check whether the model contains *control loops* (Malik and Malik, 2006). These are *universal* properties that can be checked automatically for every model without further user interaction. Their verification is invoked conveniently from the menu, and in many cases quickly returns an answer whether or not the model passes the check.

Supremica also allows for the verification of application specific safety properties. These properties are specified by finite-state machines designed by the user to describe a maximum allowable behaviour, and checked by means of a *language inclusion check* (Brandin et al., 2004). This makes it possible to ask specific questions about the application at hand.

If verification fails, *Supremica* also computes a *counterexample*, which is a trace of events that takes the system to a problematic state. The counterexample trace can be displayed and explored interactively in the simulator. Fig. 1 shows an example of this. The user can replay the counterexample step-by-step and inspect the states of all components at each stage. This feature helps the user to understand the cause of a problem and fix it.

3.5 Synthesis

The *Supremica* analyser includes the functionality to synthesise a least restrictive controllable and nonblocking supervisor for a collection of plant and specification FSMs (Ramadge and Wonham, 1989). The synthesised supervisor can take the form of a single FSM, or it can be a modular supervisor, consisting of several components, obtained by *compositional synthesis* (Mohajerani et al., 2014) or *supervisor localisation* (Cai and Wonham, 2015). It can also be simplified using supervisor reduction (Su and Wonham, 2004). Small supervisor FSMs can be displayed and edited in the graphical user interface. If a supervisor component has too many states to be displayed, it can still be stored in files or interacted with through the simulator.

As an alternative, the *Supremica* editor includes the *seamless synthesis* feature, which performs a symbolic synthesis based on *Binary Decision Diagrams* (BDDs) (Miremadi et al., 2011; Fei et al., 2014). In this case, the synthesised supervisor is added to the graphical plant model in the form of EFSM transition guards. This gives the user the possibility to implement only the guards representing the supervisor in a controller, such as a PLC.

4. PERFORMANCE

Verifying or synthesising a supervisor generally entails enumerating the state-space of the controlled system, the size of which typically grows exponentially with the number of state machines in the model. To deal with the large state numbers, *Supremica* includes highly optimised explicit verification algorithms, programmed in C++, that can explore more than 100 million states on contemporary computers (Malik, 2016). This is still insufficient for many practical control problems. Several groups have conducted research to improve the performance of discrete event systems algorithms.

One approach to mitigate the state-space explosion problem uses ideas from *symbolic model checking* (Baier and Katoen, 2008), in particular BDDs (Akers, 1978; Bryant, 1986). BDDs can efficiently store and manipulate Boolean functions representing for example supervisory control problems. BDD-based algorithms can handle huge state-spaces while using little memory, and have been applied successfully to large discrete event systems (Vahidi, 2004; Song, 2006).

Another approach to handle large discrete event systems is to exploit the inherent *modularity* common in these models. Well-designed discrete event system models typically consist of a large number of small state machines. This makes it possible to perform synthesis and verification by considering only parts of the model at a time, without ever constructing the full synchronous composition. Safety properties such as controllability can be analysed using *incremental* or *modular* verification, where the system is split into groups of state machines that are analysed separately (Åkesson et al., 2002; Brandin et al., 2004). Nonblocking verification and synthesis can be addressed through the *compositional* approach, where the state machines are composed gradually and simplified after each step (Flordal and Malik, 2009; Su et al., 2010; Mohajerani et al., 2014).

Supremica includes explicit, modular, BDD, compositional, and other verification and synthesis algorithms for solving controllability and nonblocking verification, and synthesis problems. Table 1 shows the results of some benchmark computations. The table shows the runtimes and memory usage for synthesis, nonblocking verification, and controllability verification using different algorithms. The test cases include complex industrial models and case studies from different application areas such as manufacturing systems and automotive body electronics, most of which are also used as benchmarks by Mohajerani et al. (2014). The "Aut" column shows the number of state machines in the model, and "State-space" is the number of reachable states of the uncontrolled system. These experiments were run on a standard desktop PC using a single 3.3 GHz microprocessor and not more than 2 GiB of RAM.

The data for "**Comp. Syn.**" refers to the compositional synthesis algorithm (Mohajerani et al., 2014). This algorithm calculates a least restrictive, controllable, and nonblocking supervisor for the given plants and specification, in the form of a set of synchronised FSMs. The data for "**Comp. Nbl.**" refers to compositional nonblocking verification with special events (Pilbrow and Malik, 2015), which is an improved version of the earlier algorithm (Flordal and Malik, 2009). "**Mod. Cont.**" is incremental controllability verification (Brandin et al., 2004), which uses counterexamples to identify suitable subsystems to verify controllability. The timeout for the **tbed_uncont** model occurred after 20 minutes. Finally, "**BDD Nbl.**" and "**BDD Cont.**" are BDD-based nonblocking and controllability verification. They are based on a BDD search algorithm with improvements for better performance on discrete event systems models. They use a variable ordering based on the FORCE heuristics (Aloul et al., 2003) and a disjunctive partitioning and search strategy (Song, 2006).

The experiments show that *Supremica* can handle the verification and synthesis of discrete event systems with up to 10^{17} states within a few seconds, in some cases minutes. All the algorithms can be tuned with various parameters for better performance, however this data was obtained with default settings to give a general impression of the performance of *Supremica*.

5. APPLICATIONS

Supremica is used at Chalmers and other universities to communicate the ideas of supervisory control to students. In addition, *Supremica* has been used in several research projects with industrial partners to verify and synthesise supervisors and controllers in various applications. This section describes four of these projects.

Table 1. Benchmark examples from *Supremica*.

Model			Comp. Syn.		Comp. Nbl.		BDD Nbl.		Mod. Cont.		BDD Cont.	
Name	Aut	State-space	Time [s]	Mem. [MiB]	Time [s]	Mem. [MiB]	Time [s]	Mem. [MiB]	Time [s]	Mem. [MiB]	Time [s]	Mem. [MiB]
agv	16	$2.57 \cdot 10^7$	0.8	37.6	0.2	108.7	0.1	32.0	0.0	31.5	0.1	31.4
agvb	17	$2.29 \cdot 10^7$	0.6	38.2	0.2	107.6	0.1	31.4	0.0	31.4	0.0	32.0
aip0alps	35	$3.00 \cdot 10^8$	0.4	57.3	0.2	101.3	0.2	11.7	0.0	11.7	0.1	11.6
fencaiwon09b	31	$8.93 \cdot 10^7$	0.4	46.5	0.4	129.4	0.9	9.4	0.1	8.9	0.3	8.8
fencaiwon09s	29	$3.00 \cdot 10^8$	0.4	37.8	0.3	38.1	0.1	31.4	0.0	31.4	0.0	32.0
psl_big	37	$3.87 \cdot 10^7$	0.5	45.9	0.3	39.2	1.3	31.6	0.0	28.2	0.0	28.9
psl_partleft	39	$7.69 \cdot 10^7$	1014.0	878.7	0.2	38.3	0.1	35.9	0.0	33.0	0.0	32.7
psl_restart	37	$3.87 \cdot 10^7$	550.2	396.4	0.4	39.9	0.9	32.9	0.0	29.9	0.0	30.1
tbed_hisc1	184	$2.87 \cdot 10^{17}$	11.1	662.7	0.2	38.1	469.3	28.4	0.1	15.8	436.5	27.7
tbed_noderailb	84	$3.20 \cdot 10^{12}$	5.2	419.3	3.2	140.6	91.8	38.2	9.7	32.9	102.2	28.5
tbed_uncont	84	$3.66 \cdot 10^{12}$	5.5	528.0	3.3	260.0	166.2	27.7	Timeout		7.0	33.5
verriegel3b	52	$1.32 \cdot 10^9$	1.2	83.4	0.3	39.0	2.3	14.6	0.1	13.2	1.2	15.0
verriegel4b	64	$6.26 \cdot 10^{10}$	1.6	128.1	0.3	38.4	8.5	18.1	0.1	15.7	4.5	18.2
6linka	53	$2.45 \cdot 10^{14}$	2.8	154.7	0.3	38.4	4.0	30.8	0.0	31.4	0.5	30.8
6linki	53	$2.75 \cdot 10^{14}$	1.8	63.9	0.2	37.7	4.6	31.5	0.0	31.4	2.0	31.4
6linkp	48	$4.43 \cdot 10^{14}$	2.7	249.7	0.2	38.2	1.5	30.8	0.0	31.4	0.5	31.5
6linkre	59	$6.21 \cdot 10^{13}$	0.6	56.5	0.3	39.2	2.3	31.4	0.0	32.0	1.7	31.4

5.1 Industrial Robot Interlocking

In a modern car factory, several industrial robots work concurrently in a shared space to complete their tasks, for example welding together the frame of a car. In this setting, there is a risk for collisions between the robots. Manually programming the coordination logic to avoid collisions is both time-consuming and error prone due to the number of possible situations that must be taken into account. Using 3D-simulation models of the robots, it is possible to automatically extract finite-state models of the tasks and of the mutual exclusion *zones* where only a single robot is allowed to be at a time. This approach has been implemented as an add-on for a commercial robot programming and simulation environment, the coordination logic synthesis part is done using the *Supremica* kernel. The approach is presented by Flordal et al. (2007).

5.2 Virtual Manufacturing

The development and validation of a manufacturing systems against a simulation model is attracting increasing interest in industry. The main motivation is that the systems can be integrated and tested before constructing any physical system. In addition to this, the ability to continuously test leads to increased reliability and enables better coping with late changes. Having a computable model has the added benefit that it can be used as a base for preparation and implementation assisted by formal methods. In a case study (Bengtsson et al., 2012; Dahl et al., 2016), *Supremica* was used as an aid in generating operation sequences for validation during production preparation. The result showed significant promise, and currently this approach is further developed together with a European manufacturing company.

5.3 Restart in Manufacturing Systems

In manufacturing systems, restart after an unforeseen error is typically a complicated task, where the operator has to manually position the devices in known poses before production can continue. This is further complicated by a need to use the devices to help remove heavy work pieces. Bergagård and Fabian (2013) describe an approach based on the supervisory control theory to calculate *restart states* in a manufacturing control system. This supports the operator to position the devices in a state from where it is guaranteed that nominal production can continue. As shown by Bergagård and Fabian (2013), this can be formulated as a supervisor synthesis problem, solved offline, where re-execution specifications can be included in the calculation to obtain a correct behaviour for the restarted system.

5.4 Autonomous Vehicles

In collaboration with a European car manufacturer, *Supremica* was used to formally verify a part of the code for an autonomous vehicle (Petersson and Zita, 2016). This code manages the lateral state of the vehicle, effectuating lane change and related tasks. The manually written code implements a state machine, which was modelled as an EFSM in *Supremica*. Specifications were then modelled and some of these were verified not to be fulfilled; one serious bug was revealed. Verification with *Supremica*'s abstraction-based compositional conflict check examined 601168 states in 0.45 seconds and produced a 38-event long counterexample. The error was then through a unit test confirmed to also exist in the actual code.

6. CONCLUSIONS

A tool, *Supremica*, for the modelling and analysis of discrete event supervisors according to the supervisory control theory was presented. Extended Finite-State Machines are used as modelling formalism. The tool implements current state-of-the-art algorithms to handle systems of industrially interesting sizes. *Supremica* is currently used in a number of research projects together with industrial partners to facilitate the development of control functions. The use of formal methods for synthesising control functions that are correct by construction, addresses an important problem with implications for both how software and hardware control functions are developed.

ACKNOWLEDGEMENTS

Supremica has been developed by many people over many years and with support from several national and international projects, all too numerous to mention here.

REFERENCES

Akers, S.B. (1978). Binary decision diagrams. *IEEE Trans. Comput.*, 27(6), 509–516.

Åkesson, K., Fabian, M., Flordal, H., and Malik, R. (2006). Supremica—an integrated environment for verification, synthesis and simulation of discrete event systems. In *8th Int. Workshop on Discrete Event Systems*, 384–385. IEEE.

Åkesson, K., Fabian, M., Flordal, H., and Vahidi, A. (2003). Supremica—a tool for verification and synthesis of discrete event supervisors. In *11th Mediterranean Conf. Control and Automation*.

Åkesson, K., Flordal, H., and Fabian, M. (2002). Exploiting modularity for synthesis and verification of supervisors. In *15th IFAC World Congress*. Barcelona, Spain.

Aloul, F.A., Markov, I.L., and Sakallah, K.A. (2003). FORCE: A fast & easy-to-implement variable-ordering heuristic. In *13th ACM Great Lakes Symp. VLSI*, 116–119.

Baier, C. and Katoen, J.P. (2008). *Principles of Model Checking.* MIT Press.

Bengtsson, K., Bergagård, P., Thorstensson, C., Lennartson, B., Åkesson, K., Yuan, C., Miremadi, S., and Falkman, P. (2012). Sequence planning using multiple and coordinated sequences of operations. *IEEE Trans. Autom. Sci. Eng.*, 9(2), 308–319.

Bergagård, P. and Fabian, M. (2013). Calculating restart states for systems modeled by operations using supervisory control theory. *Machines*, 1(3), 116–141.

Brandin, B.A., Malik, R., and Malik, P. (2004). Incremental verification and synthesis of discrete-event systems guided by counter-examples. *IEEE Trans. Control Syst. Technol.*, 12(3), 387–401.

Bryant, R.E. (1986). Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.*, 35(8), 677–691.

Cai, K. and Wonham, W.M. (2015). New results on supervisor localization, with case studies. *Discrete Event Dyn. Syst.*, 25(1), 203–226.

Chen, Y. and Lin, F. (2000). Modeling of discrete event systems using finite state machines with parameters. In *IEEE Int. Conf. Control Applications*, 941–946.

Dahl, M., Bengtsson, K., Bergagård, P., Fabian, M., and Falkman, P. (2016). Integrated virtual preparation and commissioning: supporting formal methods during automation systems development. *IFAC-PapersOnLine*, 49(12), 1939–1944.

Fei, Z., Miremadi, S., Åkesson, K., and Lennartson, B. (2014). Efficient symbolic supervisor synthesis for extended finite automata. *IEEE Trans. Control Syst. Technol.*, 22, 2368–2375.

Feng, L. and Wonham, W.M. (2006). TCT: A computation tool for supervisory control synthesis. In *8th Int. Workshop on Discrete Event Systems*, 388–389. IEEE.

Flordal, H., Fabian, M., Åkesson, K., and Spensieri, D. (2007). Automatic model generation and PLC-code implementation for interlocking policies in industrial robot cells. *Control Engineering Practice*, 15(11), 1416–1426.

Flordal, H. and Malik, R. (2009). Compositional verification in supervisory control. *SIAM J. Control Optim.*, 48(3), 1914–1938.

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

Malik, P. and Malik, R. (2006). Modular control-loop detection. In *8th Int. Workshop on Discrete Event Systems*, 119–124. IEEE.

Malik, R. (2016). Programming a fast explicit conflict checker. In *13th Int. Workshop on Discrete Event Systems*, 464–469. IEEE.

Malik, R., Fabian, M., and Åkesson, K. (2011). Modelling large-scale discrete-event systems using modules, aliases, and extended finite-state automata. In *18th IFAC World Congress*, 7000–7005. Milano, Italy.

Miremadi, S., Åkesson, K., and Lennartson, B. (2011). Symbolic computation of reduced guards in supervisory control. *IEEE Trans. Autom. Sci. Eng.*, 8(4), 754–765.

Mohajerani, S., Malik, R., and Fabian, M. (2014). A framework for compositional synthesis of modular nonblocking supervisors. *IEEE Trans. Autom. Control*, 59(1), 150–162.

Mohajerani, S., Malik, R., and Fabian, M. (2016). A framework for compositional nonblocking verification of extended finite-state machines. *Discrete Event Dyn. Syst.*, 26(1), 33–84.

Moor, T., Schmidt, K., and Perk, S. (2008). libFaudes — an open source C++ library for discrete event systems. In *9th Int. Workshop on Discrete Event Systems*, 125–130. IEEE.

Petersson, P. and Zita, A. (2016). *Logical modelling and formal verification of decision and control functions for autonomous vehicles.* Master's thesis, Chalmers University of Technology, Göteborg, Sweden.

Pilbrow, C. and Malik, R. (2015). An algorithm for compositional nonblocking verification using special events. *Sci. Comput. Programming*, 113(2), 119–148.

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

Rudie, K. (2006). The integrated discrete-event systems tool. In *8th Int. Workshop on Discrete Event Systems*, 394–395. IEEE.

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

Song, R. (2006). *Symbolic Synthesis and Verification of Hierarchical Interface-based Supervisory Control.* Master's thesis, Department of Computing and Software, McMaster University, Hamilton, ON, Canada.

Su, R. and Wonham, W.M. (2004). Supervisor reduction for discrete-event systems. *Discrete Event Dyn. Syst.*, 14(1), 31–53.

Su, R., van Schuppen, J.H., Rooda, J.E., and Hofkamp, A.T. (2010). Nonconflict check by using sequential automaton abstractions based on weak observation equivalence. *Automatica*, 46(6), 968–978.

Vahidi, A. (2004). *Efficient Analysis of Discrete Event Systems—Supervisor Synthesis with Binary Decision Diagrams.* Ph.D. thesis, Chalmers University of Technology, Göteborg, Sweden.