Supremica – A Tool for Verification and Synthesis of Discrete Event Supervisors

Knut Åkesson, Martin Fabian, Hugo Flordal, and Arash Vahidi
Department of Signals and Systems
Chalmers University of Technology
SE-412 96 Göteborg, Sweden
Email: ka@s2.chalmers.se

Abstract— **A tool for automatic verification and synthesis of controllers for discrete event systems is presented. The tool, called Supremica, and it implements the main ideas in the Supervisory control theory. In addition to the verification and synthesis algorithms, Supremica can automatically generate code for C-like languages and also IEC 61131 languages. It is also possible to execute the control code against either a physical environment or a simulated environment. To handle verification and synthesis of industrial size problems Supremica implements algorithms that exploits the modular structure of the problem together with symbolic methods to efficiently represent large state-spaces.**

I. Introduction

The supervisory control theory [1] promise to be very useful to deal with the high requirements on flexibility in many new production systems. Many new production plants must be easy to reconfigure by adding and removing resources and must be able to produce many different products. For example, a modern car has many options and virtually there are no identical cars. Of course this puts a lot of requirement on the control system used to produce the car. The Supervisory control theory (SCT) may be used to analyze and also synthesize control code for the production facilities. When both the production system and products changes often it may be advantageous or even necessary to be able to automatically synthesize the control code for the current setup of production resources and products. The SCT handles this challenge by using well defined models of both the plant, i.e. the production facilities, and the specifications that models the products to be produced within the plant. A lot of theory has been developed within the SCT-framework, however the theory has to be applied in real-world applications in order to gain acceptance outside academia. The existing tools that implement the main ideas in the SCT are still lacking important features. In this paper a new tool – Supremica – is presented. This tool tries to handle some of the problems with existing tools. Supremica implements the standard supervisory control algorithms. It is possible to automatically generate standard Programmable Logic Controller (PLC) code that implements the behavior of the synthesized supervisor. It is also possible to execute the supervisor against a graphical simulation of the environment or against a physical environment. The tool is user-friendly and has been used to teach basics of the SCT to third-year automation engineering students at Chalmers University of Technology. To show the feasibility of the SCT, Supremica has been used in a number of applications. This paper presents the basic features of Supremica and also present some of the applications where Supremica has been successfully used. Supremica is free for education and research and can be downloaded from `http://www.supremica.org`.

A. Existing Tools

A number of tools implementing the main concepts in the supervisory control theory have emerged.

- **TCT** TCT is the original SCT tool developed at University of Toronto, Canada. TCT has a primitive text-based interface. A new tool called STCT, [2], with algorithms that work on IDDs, [3], a generalization of BDDs has recently been released.
- **UMDES** Developed at University of Michigan, USA, UMDES is a library of C-routines for the Supervisory control theory.
- **UKDES** Developed at University of Kentucky, USA, UKDES, [4], contains the basic algorithms and has a simple graphical user interface.
- **J-DES** Developed at Pennsylvania State University, USA, J-DES has a graphical user interface and provides similar functionality as TCT and UKDES.
- **BSP** Developed at University of L'Aquila, Italy, BSP uses BDDs to solve large scale synthesis problems. The approach is presented in [5].
- **Ver** In [6], [7] a tool for supervisory control that uses BDDs to solve large scale synthesis problems is presented. The BDD algorithms are presented in [8].
- **Valid** Valid is the only known commercial tool for supervisory control theory. Valid is developed by Siemens Corporate Research, Germany. Very little information is released about the details of this implementation. This seems to be the most complete tool among the existing supervisory tools, but the focus is on verification and not on synthesis. Valid includes efficient algorithms for verification and a recent version includes support for synthesis but experiments indicates that it has problems to handle large problems.

All tools suffer in at least one of the following areas; ease of use; features; ability to handle large scale systems; simulation;

code generation; code execution.

II. SUPREMICA

In this section Supremica is presented but first the assumptions that Supremica relies on are presented. Supremica is a tool that changes over time and the most up to date information is available at `http://www.supremica.org`. Supremica was initially built to evaluate the ideas presented in [9]. When designing Supremica we used experiences gained from the development of a prior SCT tool, Desco [10].

The basic data structure in Supremica is a deterministic finite automaton. Each automaton has a finite set of states, Q, one of these states is the initial state q_i. The states can be marked or non-marked and forbidden or non-forbidden. The set of marked states are denoted Q_m and the set of forbidden states Q_x. Associated with the transition between states are events where each automaton has a finite set of events, i.e. the alphabet Σ. The transition from state q_k to state q_l on the event σ is denoted as $\delta(q_k, \sigma) = q_l$. δ is assumed to be a partial function, i.e. it is not defined for all states and all events. Each event is defined to be controllable or uncontrollable. All events are assumed to be observable.

A Supremica project consists of multiple automata that together define the main problem. The behavior of all the automata is defined by the prioritized composition operator (PCO). It is important to note that the alphabets do not have to be the same but they are of course allowed to be equal. If the same event is present in multiple automata then that event must be either controllable or uncontrollable in all of the automata. The PCO is useful because it can express both full synchronous composition as defined by Hoare [11], and broadcast composition used, for example, by State diagrams in Unified Modeling Language (UML). In prioritized composition is also possible to mix full and broadcast composition. To support the PCO each event is assumed to be either prioritized or not. Note that the same event may be prioritized in one automaton and not prioritized in another. The set of prioritized event in an automaton is denoted by Σ_p. The *prioritized synchronous composition* of $n \geq 2$ finite automata is denoted by

$$\underset{p}{\|}\,(A^1, A^2, \ldots, A^n)$$

where

$$\begin{gathered} Q = Q^1 \times Q^2 \times \ldots \times Q^n, \\ \Sigma = \bigcup_{i=1}^{n} \Sigma^i, \\ q_i = \langle q_i^1, q_i^2, \ldots, q_i^n \rangle, \\ Q_m = Q_m^1 \times Q_m^2 \times \ldots \times Q_m^n \\ Q_x = \{\langle q_i^1, q_i^2, \ldots, q_i^n \rangle \mid \exists q_i^j \in Q_x^j\} \end{gathered}$$

and the transition function as

$$\delta(\langle q^1, q^2, \ldots, q^n \rangle, \sigma) =$$

$$\begin{cases} \langle \hat{\delta}^1(q^1,\sigma), \hat{\delta}^2(q^2,\sigma), \ldots, \hat{\delta}^n(q^n,\sigma) \rangle & \bigwedge_{\{i \,|\, \sigma \in \Sigma_p^i\}} \delta^i(q^i,\sigma)! \\ \text{undefined} & \text{otherwise.} \end{cases}$$

where

$$\hat{\delta}^i(q^i,\sigma) = \begin{cases} \delta^i(q^i,\sigma) & \text{if } \delta^i(q^i,\sigma)! \\ q^i & \text{otherwise.} \end{cases}$$

Informally the above definition implies that an event is present from a state in the synchronous composition if all automata that have the event as prioritized are in a state from which they can execute the event. If the event can be executed in the synchronous composition then the next state is given by executing the same event in all automata that can execute the event, i.e. both those automata that have the event as prioritized and those that do not.

Each automaton does also have type, it is one of either *plant*, *specification*, or *supervisor*. The prioritized synchronous composition of the plant automata defines the behavior of the plant. The term system used frequently in the sequel is used to denote the prioritized synchronous composition of all the automata in the project. In Supremica the user typically starts with a set of plant and specification automata. The user can then verify if the plant is controllable with respect to the specification and if the plant and specification is non-blocking when executing under the prioritized synchronous composition. If this is the case then the specifications is usable as supervisors, if the system is not controllable or not non-blocking then the user can chose to synthesize a set of supervisors (or a single supervisor) that when executing together with existing plant and specifications guarantees that the system will be both controllable and non-blocking.

Supremica has a graphical user interface where it is easy to do supervisory synthesis and verification. Supremica does not yet contain its own graphical editor for state automata; this is instead handled by another tool - DescoGUI. DescoGUI was first intended as an interface to Desco but today supports also Supremica. Supremica is able to graphically layout and display an automaton using Graphviz, from AT&T Research. This is very useful, at least for small automata, since it allows the user to synchronize or synthesize a new automaton and then graphically display the result.

Supremica is able to synthesize maximally permissive non-blocking and controllable supervisors. Efficient algorithms that exploit the modularity of the system, for verification and synthesis of controllable supervisors, have been developed and implemented. Efficient modular algorithms for non-blocking verification and synthesis are under development by our research group. Supremica also contains an efficient algorithm for language inclusion checking. Language inclusion checks are useful for verifying properties of a system. Supremica also implements state minimization of an automaton. State minimization computes the minimal automaton, in number of states, with the same language as the original one. Supremica also allows the user to restrict, or project out, events from the alphabet in an automaton; this implies that all the events that do not survive the restriction are removed from the strings in the language generated by the original automaton. Furthermore, Supremica implements algorithms for bounded

controllability that are useful in hybrid computer-human supervision of discrete event systems.

When the number of states is small, a graphical presentation of the automaton is advantageous to the user. However, if the automaton contains many states or contains many transitions a graphical presentation easily becomes cluttered. Supremica has an explorer for an automaton or a set of automata. The explorer displays a single state and the set of enabled events in that state. The user might click on an event to execute that event and as a result change state.

A. Synthesis

Supremica implements monolithic synthesis algorithms for solving non-blocking, controllability, and combined non-blocking and controllability problems. Supremica also implements modular algorithms for verification and synthesis of the controllability problems. The modular algorithms are presented in [12]. The modular algorithms have been evaluated on several large problems and have been able to synthesize a set of interacting supervisors with very limited time and memory requirements. For example, the modular algorithms are able to verify and synthesize supervisors for a central-locking system with 750 million reachable states in a couple of seconds. There are ongoing work on adding BDD based algorithms to Supremica, some of this work is reported in [13].

B. Verification

With "verification" we refer to proving properties of the system by help of formal methods. A survey of verification of PLC programs are presented in [14]. Supremica contains its own algorithms for verification. Currently there are efficient algorithms for verification of controllability. If the system is sufficiently modular, Supremica is able to verify problems of almost arbitrary size. There are pathological cases where the developed algorithms cannot take advantage of the modularity and the only thing that Supremica can do is to compute the complete reachability graph of the system. Experiments, on different applications, show that many applications have enough modular structure to make the modular verification and synthesis algorithms efficient. To verify non-blocking properties, Supremica uses a brute-force approach and is therefore currently unable to handle systems with more than a few million states.

C. Simulation

An interface against the Scenebeans framework, [15], for graphical simulation has been developed. This makes it possible to write graphical animations and load them together with the state automata. Supremica can then compute supervisors and execute them on-line against the graphical animation.

D. Code generation

When a set of supervisors has been synthesized it is desirable to be able to generate code that implements these supervisors. Supremica can generate code in a number of formats including IEC 61131–Instruction List and Structured Text; ABB Control Builder–Instruction List, Structured Text, and Sequential Function Chart; ANSI C; and Java bytecode.

Programmable Logic Controllers (PLCs) are used in industry to control very different kinds of devices. A standard for PLC languages has been defined in [16]. This standard is widely adopted by industry and most tools claim compliance with the standard. The standard defines five different programming languages each with its own strengths and weaknesses. The five languages are Structured Text, Function Block Diagram, Ladder Diagram, Instruction List, and Sequential Function Chart.

It is important to note that in general it is not desirable to generate a single automaton that will define the allowed behavior of the supervisor since the state-space might be huge even for rather small examples. Instead, we want to generate a more compact representation of the supervisor. The approach in Supremica is to take advantage of the possibility for modular supervisors and thus compute a set of (sub-) supervisors where the PLC is responsible for computing the set of enabled events in each state of the total supervisor. Fortunately, on-line computation of the set of enabled events in the current state is a relatively simple operation that can be computed efficiently by a PLC.

In [17] it is shown how it is possible to, given a set of automata interacting through events, compute a set of SFCs interacting through signals, where the SFCs will behave very similar to the automata. Different PLC vendors have interpreted the IEC 61131 standard differently making it harder to generate code that will have the same behavior on different platforms. The main problem is whether they first choose to evaluate all transition conditions in all SFCs before they do the transition, or if they evaluate the transition condition for a single SFC and then change the active steps before they go on to evaluate the conditions in the following SFCs. The first author of [17] has contributed to Supremica an implementation of a robust algorithm that generates code that can be executed in ABB Control Builder. Currently, this implementation only supports full synchronization and does not allow self-loops in the input automata. A more straightforward approach is to generate Structured text or Instruction List code. This approach avoids the problem with interacting SFCs and instead generates a single code block that is responsible for computing the enabled events.

The basic approach to code generation is to first check which events that are enabled in the automata. This is done be doing an online synchronization in the PLC. After this is checked if there are external conditions, i.e. boolean conditions on input signals, that must be true in order for the event to be enabled. Events can also be associated with internal timers that must have done a timeout before an event is enabled, so this is also checked. At this point zero, one, or more events may be enabled. If zero events are enabled then we do not have to do anything, if one event is enabled then we change state in the automata and execute actions associated with that event. If more than one event is enabled it is in general not safe to execute all events instead a selection must be done.

One approach is to execute one of enabled events randomly, another approach is to have the events ordered and execute the first event in this order. However, this is an area that need to be explored more but the paper [18] presents some properties that the automata should have in order to safe to execute with this approach.

At the time of writing it is possible to generate IEC 61131 code as Instruction List and Structured Text. Currently, it is only possible to compile and execute Instruction List code within Supremica. The execution of the generated code is discussed in more detail in the following section. Supremica is able to generate PLC code that supports a modular structure of the control code. It is important to be able to do the synchronization between multiple automata inside the PLC in order to avoid having to generate a single automaton explicitly containing all reachable states since the number of reachable states may be huge even for small problems.

E. Code Execution

Usually a special hardware device, a PLC, does execute the control code, but recently standard PCs are being increasingly used as PLCs. The PLC functions in a standard PC are then implemented exclusively by software components. The strong requirement on stability in PLC applications make Java [19] a suitable choice for implementing functionality that is not pure control functions, this might for example be network and graphical interface functionalities.

Supremica contains a soft-PLC. The compiler transforms/compiles IEC 61131 Instruction List code to Java bytecode. The runtime system uses a standard Java Virtual Machine (JVM) to execute the code. The main part of the runtime system is to enforce the PLC way of execution onto the JVM; the other part is to interact with an I/O system.

Currently, the soft-PLC is not able to handle hard real-time constraints due to limitations in the JVM. A standard for real-time Java has recently been developed, see [20]. Taking advantage of the new real-time JVM is a future project. Even if the synthesized system is non-blocking the controlled system might be blocking if a non-random policy is used to select which event, among the enabled, to execute. This problem is analyzed in [18].

In many applications and especially in soft-PLC applications, it is desirable to be able to call programming languages other than those specified in IEC 61131-3.

Figure 1 shows the steps involved in the generation of PLC code. The solid arrows correspond to currently implemented functionality. By using this design where first standardized PLC code are generated from, for example Supremica, and then compiled for a specific architecture, it becomes easy to add support for other PLC's

After the compilation there is no difference between the control code and standard Java code thus making it transparent to call standard Java methods. The major difference with a standard Java program and a PLC program is the way a PLC program is supposed to be executed.

Fig. 1. The possible PLC code generation steps. The solid arrows indicate implemented functionlity while the dashed arrows indicate possible functionality.

To allow the PLC code now in the form of Java bytecode to behave like a proper PLC, a class was written to execute the PLC-program at regular time-instants and also to do the input and output copying of variables. This approach is implemented in Supremica and allows the user to switch between running the soft-PLC, i.e. the Java Virtual Machine, against a physical I/O-card or against a software module that exhibits the same behavior as a physical I/O-card. Since events are associated with transition between states in the internal model but signals are used to communicate to the environment it is necessary to have a scheme to handle these issues. In Supremica this is handled by making it possible to associate *actions* and *conditions* with events. An action is a set of signals that should be set to high or low when the associated event is executed. A condition is a boolean expression of signals that must be evaluated to true before the associated event are allowed to execute. It is important to note that it must be known a priori that if an event is enabled in the finite automata model then the corresponding condition must sooner or later evaluate to true without executing any other events. If this is not fulfilled then it our model may say that we can execute an event but when executing it against the environment the event is enabled in the finite automata model but the corresponding condition does never evaluate to true. This will invalidate our verification and synthesis efforts, however experience show that it is generally not a problem to write code conditions that fulfills these conditions. It is also possible to have timers in Supremica that timeout after a specific time period.

III. Applications

Supremica has been evaluated on several different problem classes. Supremica was originally used to supervise the resource allocations in a commercial batch control system. This implementation is presented in more detail in [9]. Later the possibility for code generation and code execution was added and this made it possible to use Supremica to control several different lab processes. The first application we present is a ball process in our lab. We present this since it is an application originally controlled by a standard PLC and also used to teach students the basics of PLC programming using IEC 61131 languages. In this application we have replaced the PLC with a digital I/O in a standard PC and the uses Supremica to verify and execute state automata against both a graphical simulation of the ball process and also against the physical ball process. Secondly we present an application with

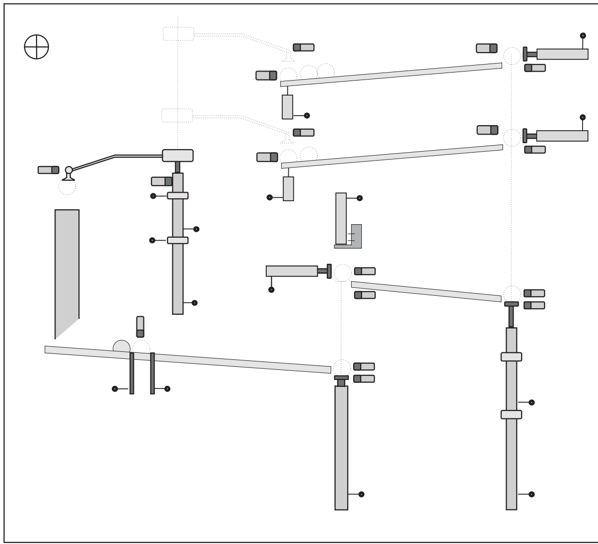

Fig. 2. The ball process.

automated guided vehicles. This is a standard example that has been used to evaluate many different control strategies. We here use the modular synthesis algorithms to synthesize a set of supervisors that together guarantees collision-free coordination of AGVs. The number of reachable states in this application is approximately 26 millions that make the use of monolithic algorithms unfeasible. The last application presents an implementation in a commercial chemical batch control system.

A. Ball Process

The Ball Process, see Figure 2, is a lab process used to teach students at Chalmers the basics of PLC programming. This implementation uses the code generation and code execution possibilities to generate IEC 61131 compatible IL code that can be compiled into Java bytecode. The soft PLC is used to execute the Java bytecode and all I/O is done with a digital I/O-card, and thus used to control the ball process.

B. Automated Guided Vehicles

A model of a flexible manufacturing cell was introduced in [21]. The cell, shown in Figure 3, consists of three work-stations, two input stations and one output station. Five Automated Guided Vehicles (AGVs) are responsible for routing the parts through the cell, see Figure 3. The AGV routes are presented in the figure. The control problem is that the routes intersect or are close to each other and thus there are zones in which no two AGVs are allowed to be at the same time. These zones are shaded light-grey in the figure. To complicate things the controller only has the possibility to prevent an AGV from proceeding at certain positions in the plant. The positions where the supervisor can prevent an AGV from proceeding are associated with the c_i events. All other events are uncontrollable. It is assumed that all events are observable to the supervisor. The input-, output-, and the work-stations are easily expressed as state automata. This applies to the five routes too. The automaton modeling Workstation 2 is shown in Figure 4 and the automaton modeling Zone 2 is shown in Figure 5. The restriction that at most one AGV can be inside a zone at a time can be viewed as a specification, while the AGV routes can be viewed as the plant. Thus, there are four specifications in this example. Supremica can now generate a set of interacting supervisors that together supervise the plant in a maximally permissive manner. In this example we will get four supervisors, one for each specification. The sizes of the supervisors are 72, 72, 116, and 68 states. The supervisor computations are completed in less than half a second on a standard PC. Note that solving this problem with a brute-force approach will take a lot of computing resources since the system has over 26 million reachable states. For this example no physical process were built, but a screenshot of Supremica running a graphical simulation of the process is shown in Figure 6.

C. Resource Allocation Systems

In flexible manufacturing systems both the products to be produced and the resources used to produce the products changes over time. Typically the resources are versatile in the sense that they can be used to carry out several different production steps. A goal when designing flexible manufacturing systems is to allow the system to continue producing products when one or more of resources fail, if possible. Changing products, flexible resources, and robustness to failure sets new requirements on the control system. A control system must be adaptable to all these changes and thus it is no longer a viable alternative to have control functions that does not have full knowledge of the set of products to be produced, their resource requirements, and the resources use to produce the products. A danger with transforming a manufacturing system from a manual system to an automatic system is that the flexibility of the production process suffers. The reason for this is the high complexity involved with building a fully automatic and flexible manufacturing control system. In the work presented in [9], we show how to build efficient models of products and resources, and then how to synthesize supervisors such that all products can always be successfully produced. at least as long as any critical hardware does not fail. When multiple products are produced with in the same plant conflicts of resources might emerge. The simplest situation is two products that each hold a resource and wants access to the resource hold by the other product. In the work presented here on resource allocation in flexible manufacturing systems and chemical batch control systems we start with the assumption that it is not reasonable to expect an end-user to be able to make models, suitable for the resource allocation algorithms, of the products and resource. Instead all the models needed by the synthesis algorithms are automatically computed based on data available in a tool suitable for end-users. In our example application a commercial chemical batch control system, called SattBatch, from ABB was used. In our implementation the user only needs to interact with SattBatch. SattBatch is used

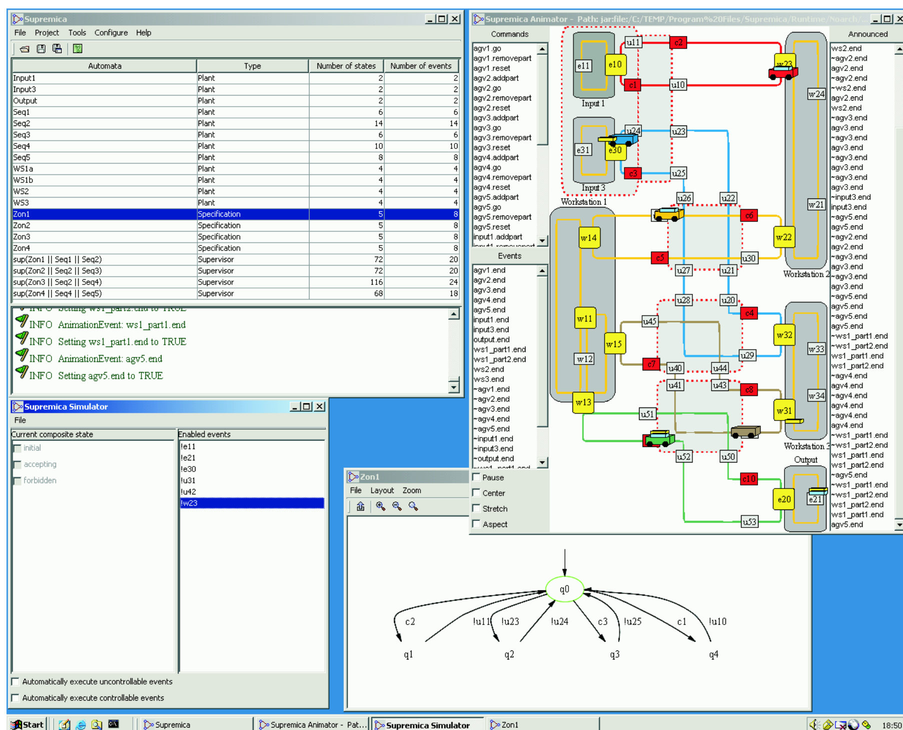

Fig. 6. Supremica doing a graphical simulation of the AGV system.

to develop recipes, which are descriptions of how to produce each product, SattBatch also holds a model of the plant, and SattBatch is then used to start and execute recipes or products. In SattBatch, the typical scenario is that the entire recipe is entered before the production of the recipe starts. However, in SattBatch it is possible to manually control the production of batches while the control system is responsible for producing the other batches. This implies that it is in general possible for the manual user to run the system into a deadlock. It is also possible to use the optimization functions in Supremica to compute the time-optimal coordination of products in the resource allocation systems. The optimization algorithms used is partially presented in [22].

IV. Conclusions

A new tool, Supremica, for verification and synthesis of discrete event supervisors according to the Supervisory control theory was presented. Supremica has been used to teach students the basics of the Supervisory control theory as well as communicate the main ideas to the industry and is easy to use even for novice users. It is possible to evaluate the synthesized control code against a graphical simulation of the environment as well as generate standard compliant code that can be executed in this tool against a physical environment or by a standard compliant Programmable logic controller. Efficient algorithms and data-structures make it possible to verify and synthesize control functions for problems of industrial size. The tool has been used in a number of applications including handling the resource allocations in a commercial chemical batch control system. Ongoing work will let the tool be responsible for the optimal collision free coordination of a multi-robot system.

References

[1] P. Ramadge and W. Wonham, "The control of discrete event systems," *Proc. of IEEE*, vol. 77, no. 1, pp. 81–98, 1989.

[2] Z. Zhang and W. Wonham, "STCT: An efficient algorithm for supervisory control design," in *In B. Caillaud, P. Darondeau, L. Lavagno, X. Xie, editors, Synthesis and Control of Discrete Event Systems*, 2001, pp. 77–98.

[3] J. Gunnarsson, "Symbolic methods and tools for discrete event dynamic systems," Ph.D. dissertation, Dept. of Electrical Engineering, Linköping University, Linköping, Sweden, 1997.

[4] V. Chandra, B. Oruganti, and R. Kumar, "UKDES: A graphical software tool for the design, analysis & control of discrete event systems," 2002, submitted for publication.

[5] E. Tronci, "Automatic synthesis of controllers from formal specifications," in *Proc. of 2nd IEEE Int. Conf. on Formal Engineering Methods*, 1998.

[6] S. Balemi, G. Hoffmann, P. Gyugyi, H. Wong-Toi, and G. Franklin, "Supervisory control of a rapid thermal multiprocessor," *IEEE Transactions on Automatic Control*, vol. 38, no. 7, pp. 1040–1059, 1993.

[7] S. Balemi, "Control of discrete event systems: Theory and application," Ph.D. dissertation, Automatic Control Laboratory, Swiss Federal Insitute of Technology, Zürich, Switzerland, 1992.

[8] G. Hoffmann and H. Wong-Toi, "Symbolic synthesis of supervisory controllers," in *Proc. of the 1992 American Control Conference*, Chicago, IL, 1992, pp. 2789–2793.

[9] K. Åkesson, "Methods and tools in supervisory control theory: Operator aspects, computation efficiency and applications," PhD thesis, Chalmers University of Technology, England, 2002.

[10] M. Fabian and A. Hellgren, *Desco - a Tool for Education and Control of Discrete Event Systems*. Kluwer Academic Publishers, 2000.

[11] C. Hoare, *Communicating Sequential Processes*. Prentice-Hall International Series in Computer Science, 1985.

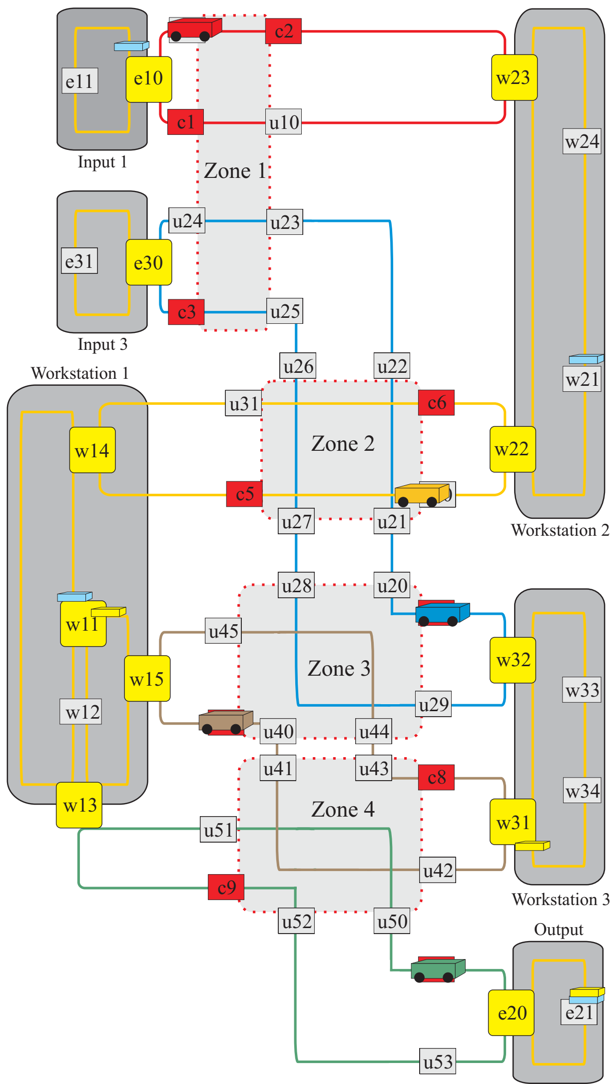

Fig. 3. A plant with Automated Guided Vehicles transporting parts between different stations.

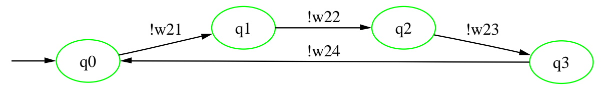

Fig. 4. The automaton modeling Workstation 2.

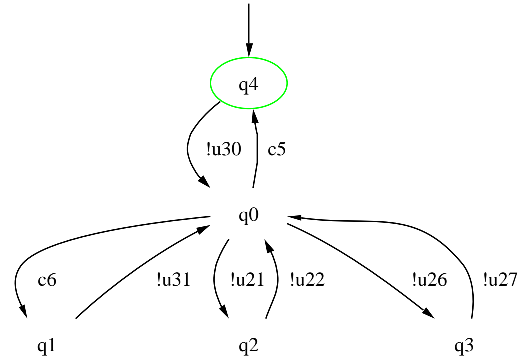

Fig. 5. The automaton modeling Zone 2.

[12] K. Åkesson, H. Flordal, and M. Fabian, "Exploiting modularity for synthesis and verification of supervisors," in *Proc. of the 15th Triennial World Congress of the International Federation of Automatic Control*, Barcelona, Spain, 2002.

[13] A. Vahidi, "A study of symbolic tools in automatic control," Department of Signals and Systems, Chalmers University of Technology, Tech. Rep., 2002, licentiate thesis.

[14] G. Frey and L. Litz, "Formal methods in PLC programming," in *Proceedings of the IEEE SMC 2000*, Nashville, USA, 2000, pp. 2431–2436.

[15] N. Pryce and J. Magee, "Scenebeans: A component-based animation framework for java," 2000, draft paper, Department of Computing, Imperial College.

[16] IEC, "IEC 61131 programmable conrollers – part 3: Programming languages," International Electrotechnical Commission, Tech. Rep., 1993.

[17] A. Hellgren, M. Fabian, and B. Lennartson, "Synchronized execution of discrete event models using sequential function charts," in *Proc. of the 1999 IEEE Conference on Decision and Control*, Phoenix AZ, USA, 1999.

[18] P. Dietrich, , R. Malik, W. Wonham, and B. Brandin, "Implementation considerations in supervisory control," in *In B. Caillaud, P. Darondeau, L. Lavagno, X. Xie, editors, Synthesis and Control of Discrete Event Systems*. Kluwer Academic Publishers, 2001, pp. 185–201.

[19] B. Joy, G. Steele, J. Gosling, and G. Bracha, *Java(TM) Language Specification*, 2nd ed. Addison-Wesley, 2000.

[20] G. Bollella, J. Gosling, B. Brosgol, P. Dibble, S. Furr, and M. Turnbull, *The Real-Time Specification for Java*. Addison-Wesley, 2000.

[21] L. Holloway and B. Krogh, "Synthesis of feedback control logic for a class of controlled Petri nets," *IEEE Transactions on Automatic Control*, vol. 35, no. 5, pp. 514–523, 1990.

[22] T. Liljenvall, "Scheduling for production systems," Department of Signals and Systems, Chalmers University of Technology, Tech. Rep., 1998, licentiate thesis.