Solving Two Supervisory Control Benchmark Problems Using Supremica

Sajed Miremadi, Knut Åkesson, Martin Fabian, Arash Vahidi, Bengt Lennartson
Department of Signals and Systems, Chalmers University of Technology
SE-412 96 Göteborg, Sweden
{miremads,knut}@chalmers.se

***Abstract*— Two supervisory control benchmark problems for WODES'08 are solved using the tool Supremica. Supremica is a tool for formal synthesis of discrete-event control functions based on discrete event models of the uncontrolled plant and specifications of the desired closed-loop behavior. By using formal synthesis of control functions the need for formal verification is reduced since the control functions are computed to automatically fulfill the given specifications, that is, they are "correct by construction". The modeling framework in Supremica is based on finite automata. Supremica implements several techniques for being able to solve large scale problems. In this paper it is evaluated how the algorithms implemented in Supremica that are based on binary decision diagrams performs on the two benchmark problems. The two benchmark problems are generalization of two classical problems; cat and mouse, and the dining philosophers' problem. The benchmark problems are parameterized such that it is possible to create problem instances with huge state-spaces. The benchmark shows that Supremica can efficiently solve rather large problem instances.**

I. 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) [9], [10] 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 [6], [12] is an attempt to build an integrated development environment that is able to solve large scale supervisor verification and synthesis problems.

The purpose of this paper is to evaluate how the binary decision diagram based algorithms in Supremica performs on two benchmark problems that were made available for WODES'08. The experimental results shows that Supremica is able to solve rather large systems within a few minutes, including an expanded cat and mouse tower with ten levels and ten cats and ten mouses.

Supremica is constantly evolving but the latest release can always be downloaded, free for education and research, from [12].

II. 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) [9], [10] 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, [9], 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. We can also note that [11] showed that supervisory control and multi-agent planning are equivalent problems.

III. Modeling Framework in Supremica

The models are expressed using finite automata, defined as follows.

Definition 1 (Finite automaton): An *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. In this paper only determinsitic automata are used. An automaton is deterministic if for each source state p and event σ there exists at most one destination state q in the transition relation; and Q_i consists of a single state. The transition relation is 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 in the style of [5].

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

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

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

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 $\mathcal{L}(G \parallel K)\Sigma_u \cap \mathcal{L}(G) \subseteq \mathcal{L}(G \parallel K)$.

Definition 5 (Nonblocking): Let G be an automaton. G is *nonblocking* if $\mathcal{L}(G) \subseteq \overline{\mathcal{M}(G)}$.
A supervisor S is nonblocking with respect to a plant G if $G \parallel S$ 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:

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

where $S' \in \mathcal{CNB}(G, K)$. 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.

IV. Efficient Synthesis by Efficient Reachability Search

Synthesizing, as well as verifying, a supervisor generally entails enumerating the state-space of a model of the controlled system. A *monolithic* approach to this, where

the entire state-space is explicitly enumerated is typically intractable for systems of industrially interesting sizes. For instance, the monolithic model of a of a central-locking system for a modern car, [7], encompasses a global state-space of roughly 1 billion reachable states. Naturally, such a large state-space cannot be efficiently manipulated by explicit enumeration of the states.

One approach to defeat this "state-space explosion problem" for supervisor synthesis uses ideas from symbolic model checking, in particular binary decision diagrams, BDDs [3].

A. Reachability Calculations with Binary Decision Diagrams

A powerful symbolic representation for an automaton is Binary Decision Diagram (BDD) [1]. Given a set of Boolean variables V, a BDD is a Boolean function $f\colon 2^V \to \{0,1\}$ represented as a directed acyclic graph (DAG) which consists of two types of nodes: *decision nodes* and *terminal nodes*. A terminal node can either be 0-terminal or 1-terminal.

A reachability search is typically performed as an iterative fixed point calculation; The reachable set Q_k is repeatedly expanded by adding all states reachable in one step from Q_k. This is continued until no more new states are found - the global fixed point is reached.

Written out in set-notation, the iteration step is

$$Q_{k+1} = Q_k \bigcup \{q : \exists q' \in Q_k : (q', q) \in T\}$$

By utilizing the fact that the set-notation carries over nicely to the language of logic we obtain an expression suited for use with BDDs. The first step is to replace Q_k with the characteristic function $\chi_k(q)$ defined as

$$\chi_k(q) = \begin{cases} 1 & Q \in Q_k \\ 0 & otherwise \end{cases}$$

The transition relation T is a subset of $Q \times Q$ and can be replaced by a characteristic function $\tau(q', q)$ in the same way.

Using these definitions, the equation for Q_{k+1} now becomes

$$\chi_{k+1}(q) = \chi_k(q) \vee [\exists q' : \chi_k(q') \wedge \tau(q', q)]$$

The characteristic functions χ and τ can both be directly represented by BDDs. The above expression, is a purely logical expression and therefore constitutes a well-defined operation on BDDs. This means that, in principle, all the pieces are in place for a BDD-based reachability analysis. However, to obtain efficient BDD-based algorithms, special care has to be taken to avoid producing overly large BDDs such as partitioning techniques [4].

B. Partitioning of the Full Synchronous Composition

A problem with composition of large sets of automata is that the total transition function δ becomes extremely complex. If an algorithm uses a traditional state enumeration based approach, this is usually not a problem since the set of global states Q is considered to be a larger obstacle and the real bottleneck. However, if a symbolic approach is taken, the set of states is often compressed to such degree that representing the states is not a problem anymore (to some limit, of course). This does however not apply to δ, due to its complex structure.

Similar problems have been studied in the field of (symbolic) formal verification, where the corresponding transition relation T is often too large to be represented as a single monolithic relation. It has been suggested that by partitioning methods, one may split T into a set of less complex relations with a clear connection in between, e.g. $T = T^1 \otimes \cdots \otimes T^n$. In the following, we will show how this idea can be applied to the transition function in a composite DES.

Automaton-based partitioning: An automaton-based *disjunctive partitioning* of the transition function is a series of small transition functions, each related to one automaton, whose union will form δ:

$$\delta(q, \sigma) = \bigcup_{A^i.\ \sigma \in \Sigma^i} \tilde{\delta}^i(q, \sigma)$$

Assuming that $q = \langle q^1, \cdots, q^n \rangle$, let

$$\zeta^{\hat{i},j}(q^j, \sigma) = \begin{cases} \delta^j(q^j, \sigma) & \text{if } \sigma \in \Sigma^i \cap \Sigma^j \\ q^j & \text{otherwise} \end{cases}$$

the disjunctive transition function is

$$\tilde{\delta}^i(q, \sigma) = \bigwedge_{A^j \in D(A^i)} \zeta^{\hat{i},j}(q^j, \sigma) \wedge \underbrace{\bigwedge_{A^k \notin D(A^i)} q^k \leftrightarrow \dot{q}^k}_{"keep"}$$

This rather straightforward but nontrivial reformulation of δ is discussed in [13], [14]. The main idea is to balance between time and space efficiency in a BDD based reachability search.

The "keep" clause guarantees that automata unaffected by the current transition remain in their current states. Thus, it does not contribute to the complexity of the function and in most cases be moved out of the equation (it can for example be replaced with a "replace q^k with $\dot{q}^k$" BDD operation). Notice further that it is trivial to remove all occurrences of σ from the function, transforming it into a much simpler transition relation $T^i \subseteq Q \times Q$. This representation is then used for more efficient symbolic computation.

In this benchmark the Workset algorithm [4], [13], [14] has been used. The Workset algorithms use structural information of the problem and the disjunctive transition relation to search the state space. This algorithm is designed to iteratively expand the set of reachable states in a way that is advantageous for BDD-compressibility. The workset algorithm operates by maintaining a set W_k of active partial transition relations – the workset. These transition relations are selected one at a time and a saturating "local´´ reachability search is performed by trying to reach node reduction in the region of the BDD representing states in the corresponding sub-automaton.

V. CASE STUDIES

The benchmark problems were conducted on a standard Laptop (Core 2 Duo processor, 2.2 GHz, 2GB RAM) running Windows Vista. Both problems have been solved by

exploiting a symbolic synthesis, namely Binary Decisions Diagrams (BDD). The BDD representation for a sample specification in the Dining Philosopher problem is illustrated. The BDD variable ordering for the problems is based on the Aloul's Force algorithms presented in [2]. It is a replacement heuristic algorithm for variable ordering. The main idea is to order 'connected' variables next to each other. This is conducted by "analogizing the interconnection between placeable objects with springs that exert forces according to the Hooke's law. Starting from an arbitrary, e.g., random, initial solution, we compute the forces acting on each object and displace objects in the direction of the forces" [2].

A. Cat and Mouse Tower (CMT)

This is e generalization of Consider the classical cat and mouse problem presented by Ramadge and Wonham [8] that make it possible to generate problem instances of arbitrary size.

B. Problem

Assume this five rooms maze is just the first level of a tower composed by n identical levels. A controllable bidirectional passageway connects room j of level $5 \cdot i + j$ to room j of $5 \cdot i + j + 1$ (for $i = 0$, 1, 2 ..., and $j = 1$, 2, 3, 4, 5). The first level is only connected with the second, the last level is only connected with the last-but-one. There are initially k cats in room 1 of the first level and k mice in room 5 of the last level. Design a maximally permissive nonblocking supervisor.

1) Model: There are various ways to model this problem. In the model used in the benchmark there is one automaton modeling each room, thus the model consists of $5 \cdot n$ automata, where n are the number of levels. The automaton $Spec.\ell irj$ specifies in what situations the cats and mice can enter room i of level j. For instance, Fig. 1 shows the automaton for room 1 of level 1, for $n = 2$ and $k = 2$. State ℓirj_e represents the situation when room j of level i is empty. In states ℓirj_xc and ℓirj_xm, there are x cats or x mice in room j of level i, respectively. The events are of the form $C.\ell irj_\ell i'rj'$ or $M.\ell irj_\ell i'rj'$; which means that a cat or mouse moves from room j of level i to room j' of level i'. According to the figure, when a cat (mouse) has entered the room, merely other cats (mice) can enter the room. Disregard the uncontrollable events, this model is sufficient to generate a maximally permissive supervisor considering the mutual exclusion specification. However, for rooms that include uncontrollable events, i.e. 2 and 4, in addition, we should also specify the uncontrollability in a proper manner. This is performed by adding a transition $\langle \ell irj_xm, \sigma, F\rangle$ where j is either 2 or 4, and F is a forbidden state that is given explicitly. For room 2, σ is $C.\ell ir4_\ell ir2$; and for room 4 σ is $C.\ell ir2_\ell ir4$. Fig. 2 shows the automaton for room 2 of level 2. Note, that with the given model there is no need to make a distinction between plant and specification automata. The reason is that cats that are moving uncontrollable from one room to another are always allowed to do so by the model. However, the model will enter a forbidden state if a cat enters a room with a mouse in it, this can be seen in Fig. 2.

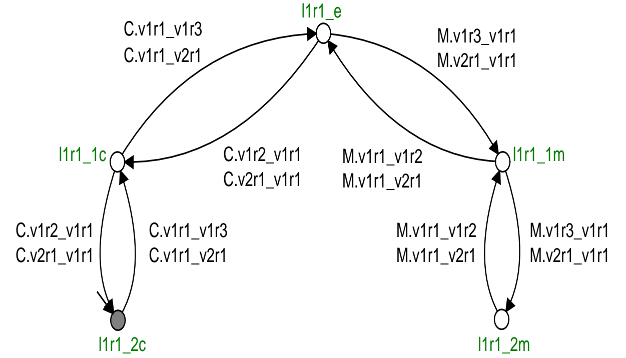

Fig. 1. Specification model for room 1 of level 1 ($n = 2$, $k = 2$).

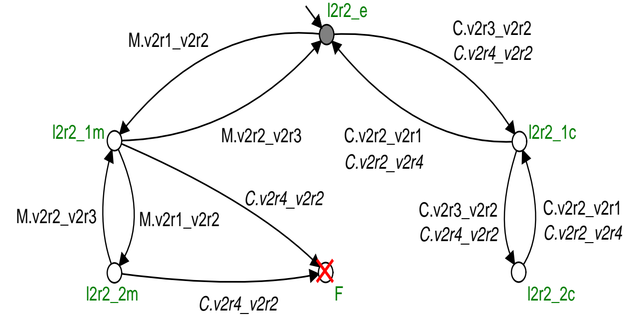

Fig. 2. Specification model for room 2 of level 2 ($n = 2$, $k = 2$). Uncontrollable events in italics.

2) Experimental Results: We have succesfully solved the problem for all instances from $(n = 1, k = 1)$ to $(n = 10, k = 10)$, i.e. 100 instances, the computing time in seconds are presented in Table I. The algorithm seems to handle problem instances with either a large number of levels are a large number of cats rather well. However, the computing time increases rapidly when both the number of levels and number of cats increases.

TABLE I

COMPUTING TIME FOR CMT

n (number of levels)	k (number of cats)	Computing time (sec)
1	1	0.6
1	5	0.5
5	1	0.4
5	5	3.2
1	10	0.6
7	7	15.5
10	1	0.4
10	7	66.0
10	10	104.0

C. Dining Philosophers (DP)

This case generalizes the classical dining philsophers problem by allowing the philosopher to go through a number

of states after picking up the left fork before he picks up the right fork.

D. Problem

Consider the dining philosophers problem where the number of intermediate states (after taking the fork on the left and before taking the fork on the right) may vary. This means that each philosopher, from the idle state takes the fork on his left reaching intermediate state 1, executes $k-1$ intermediate events reaching intermediate state k, takes his right fork entering a state where he eats, and when he is done goes back to the idle state. The uncontrollable events are "philosopher i takes the left fork" for i even. There are n philosophers around the table. Design a maximally permissive nonblocking supervisor.

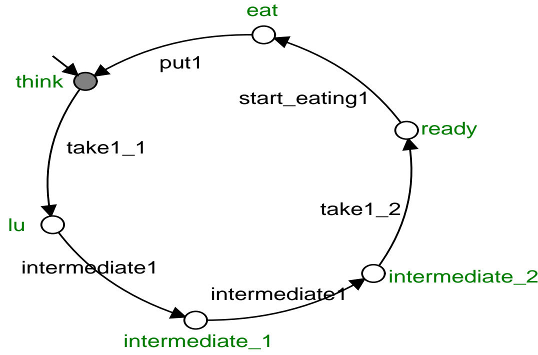

Fig. 3. Plant model for Philo:1 ($n = 3$, $k = 3$).

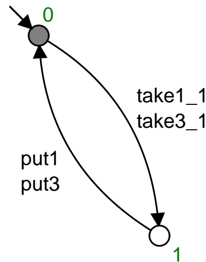

Fig. 4. Specification model for Fork:1 ($n = 3$, $k = 3$).

1) Model: For this problem, we consider an automaton (plant) for each philosopher and an automaton (specification) for each fork. Automata $Philo : 1$ and $Fork : 1$ are shown in figures 3 and 4, respectively, for three philosophers ($n = 3$) and three intermediate states ($k = 3$). Each $Philo : i$ automaton consists of the following states: $think$, lu, $intermediate_x$, $ready$, eat, where lu means that the philosopher has lifted up the left work and $intermediate_x$ means the philosopher is in intermediate state x. The interpretations for the other states are straightforward. The events for philosopher i are:

$takei_j$: Takes (lifts) fork j.

$intermediatei$: Does something meaningless.

$start_eatingi$: Starts eating.

$puti$: Puts down both of the forks.

The BDD representation for the sample specification in Fig. 4, is illustrated in Fig. 5 where a dotted line means that 0 is assigned to the variable and a solid line means that 1 is assigned to the variable . Table II shows the state and event encoding which will be used in defining the variables for the BDD. Hence, according to the table, 6 boolean variables are needed: s for the current state and s' for the next state, plus $e[0]$ and $e[1]$ for the events.

TABLE II

STATE AND EVENT ENCODING FOR THE AUTOMATON IN FIG. 4

State	Encoding	Event	Encoding
$q0$	0	$take1_1$	00
$q1$	1	$take3_1$	01
		$put1$	10
		$put3$	11

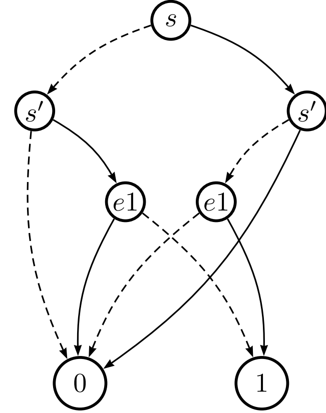

Fig. 5. The corresponding BDD for the specification automaton in Fig. 4.

TABLE III

COMPUTING TIME FOR DP

n (number of philos)	k (number of intermed.)	Computing time (sec)
5	2	3.0
100	2	6.7
200	2	40.5
300	2	149.0
5	10	2.4
5	100	2.5
5	200	2.4
5	500	2.7
5	1000	3.8

2) Experimental Results: Table 2 shows the computing time for some instances of this problem. Note, that the computing time grows much faster when the number of philospohers increases than when the number of intermediate states imncreases. The computing time is not much affected by the number of intermediate states which we believe is due to the way the Workset algorithm explores the state space.

VI. CONCLUSIONS

A tool, Supremica, for verification and synthesis of discrete event supervisors according to the Supervisory control

theory was presented. The tool implements current state-of-the-art algorithms to handle systems of industrially interesting sizes. The experimental results show that Supremica is able to solve rather large systems within a few minutes, including an expanded cat and mouse tower with ten levels, ten cats, and ten mouses.

REFERENCES

[1] S. B. Akers, "Binary decision diagrams," *IEEE Trans. Comput.*, vol. 27, pp. 509–516, Jun. 1978.

[2] F. A. Aloul, I. L. Markov, and K. A. Sakallah, "Force: a fast and easy-to-implement variable-ordering heuristic," in *ACM Great Lakes Symposium on VLSI*, 2003, pp. 116–119.

[3] R. Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Transactions on Computers*, vol. 35, no. 8, pp. 677–691, 1986.

[4] M. Byröd, B. Lennartson, A. Vahidi, and K. Akesson, "Efficient reachability analysis on modular discrete-event systems using binary decision diagrams," in *Proc. 8th Int. Workshop Discrete Event Systems, WODES '06*, Ann Arbor, MI, USA, Jul. 2006, pp. 288–293.

[5] C. A. R. Hoare, *Communicating sequential processes*, ser. Series in Computer Science. Prentice-Hall, 1985.

[6] K. Akesson, M. Fabian, H. Flordal, and R. Malik, "Supremica—an integrated environment for verification, synthesis and simulation of discrete event systems," in *Proc. 8th Int. Workshop Discrete Event Systems, WODES '06*, Jul. 2006, pp. 384–5.

[7] *KorSys homepage*, KorSys, Techniche Universität München. [Online]. Available: http://www4.in.tum.de/proj/korsys/

[8] P. Ramadge and W. Wonham, "Supervisory control of a class of discrete event processes," *Siam J. Control and Optimization*, vol. 25, no. 1, 1987. [Online]. Available: http://locus.siam.org/SICON/volume-25/art_0325013.html

[9] P. J. Ramadge and W. M. Wonham, "Supervisory control of a class of discrete event systems," *SIAM J. Control and Optimization*, vol. 25, no. 1, pp. 206–230, 1987.

[10] ——, "The control of discrete event systems," *Proc. IEEE*, vol. 77, no. 1, pp. 81–98, Jan. 1989.

[11] K.-T. Seow, C. Ma, and M. Yokoo, "Multiagent planning as control synthesis," in *AAMAS'04, New York, USA*, July 2004.

[12] "Supremica." [Online]. Available: http://www.supremica.org

[13] A. Vahidi, "Efficient analysis of discrete event systems," Ph.D. dissertation, Dept. Signals and Systems, Chalmers Univ. Technol., Göteborg, Sweden, 2004.

[14] A. Vahidi, M. Fabian, and B. Lennartson, "Efficient supervisory synthesis of large systems," *Control Eng. Practice*, vol. 14, no. 10, pp. 1157–1167, Oct. 2006.