Automated Controllability and Synthesis with Hierarchical Set Decision Diagrams

Y. Zhang B. Bérard F. Kordon Y. Thierry-Mieg

Université Pierre & Marie Curie, CNRS-UMR7606 (LIP6/MoVe),
4 place Jussieu, 75005 Paris, France
Yan.Zhang@lip6.fr, Beatrice.Berard@lip6.fr,
Fabrice.Kordon@lip6.fr, Yann.Thierry-Mieg@lip6.fr

Abstract:
Computation of a maximally permissive controller in the Ramadge-Wonham framework promises a general solution to automatically design a controller for a discrete event system, when it exists. However, like for all similar model-checking approaches, the combinatorial explosion of the state space remains a practical issue.
The work presented here investigates how to exploit both hierarchical modeling and a symbolic model-checking engine to tackle this problem. This engine is based on a powerful class of Decision Diagrams called Hierarchical Set Decision Diagrams combined with a framework called Instantiable Transition Systems, in order to describe hierarchical models. To implement the controller activity, we propose to store the set of safe states, computed offline, as a decision diagram in the controller software, allowing to take decisions on-line.
We run a prototype tool on several benchmark examples, including a problem of automated guided vehicles and a train crossing version with explicit discrete time. Results suggest good scalability, although the procedure is computationally intensive.

Keywords: Controller synthesis, discrete event system, hierarchical set decision diagrams

1. INTRODUCTION

Context. Initiated in the eighties by Ramadge and Wonham (1987) for discrete event systems, the problem of supervisory control is the following: given a model of an open system (the plant) and a specification, does there exists a controller (or supervisor) such that the behavior of the supervised system satisfies the specification? When the answer is positive, the next problem is to synthesize a non-blocking maximally permissive controller. This problem has also been seen as a two-player game, where the controller plays against the environment. The question is then reformulated as the existence of a winning strategy for the plant.

Like for all verification problems, the combinatorial explosion of the state space is one of the main practical issues making it difficult to reach performances on an industrial scale. This explains the large amount of work devoted to this question during the last twenty years.

A first research direction to deal with this problem for controllability has been concerned with the system architecture, required to be distributed or hierarchical like in Schmidt et al. (2008); Feng and Wonham (2008). Another important line of work was oriented towards symbolic computation combined with efficient state encoding techniques, for instance Binary Decision Diagrams (BDD) in Gromyko and Pistore (2006); Ma and Wonham (2008); Miremadi et al. (2008) and variants like Integer Decision Diagrams (IDD) in Zhang and Wonham (2001). In Bérard et al. (2008), another type of recently developed Decision Diagrams called Set Decision Diagrams (SDD) was reported as showing a high performance in alleviating the state explosion raised from the storage of all possible configurations of a highway system.

Contribution. This work proposes a generalizable implementation of controllability and synthesis algorithms within a more elaborated SDD-based framework called Instantiable Transition Systems (ITS) introduced in Thierry-Mieg et al. (2009). This framework, built over a symbolic model checking engine, provides an efficient way of computing state spaces for large systems obtained by composition and hierarchy. Our approach consists in restricting the system behavior within a desired specification and producing a supervisor from the largest set of safe states.

Of course, the control problem is more difficult than model-checking, where the question asked is whether some system, considered as closed, satisfies a given specification. However, when the model is given as a transition system, looking for a state-based controller can be viewed as: *(i)* generating the set of reachable states as in model checking and *(ii)* computing some particular backward reachability relation, from a subset containing failure states. For this computation, we develop a fixpoint operator on the backward transition relation, where transitions are labeled either as *environment* moves or as *controller* moves.

This answers the question of controllability. When the answer is positive, we are thus able to extract a maximal winning strategy. Our experiments on the examples of automated guided vehicles

⋆ This work has been partially supported by a Ph. D. grant from the Chinese Scholarship Council and by project DOTS (ANR-06-SETI-003).

(5AGV) and timed train crossing show the ability to describe scalable models with good efficiency.

Outline. Section 2 recalls background elements from control theory and SDD. The implementation of controllability and control synthesis is described in Section 3 and applications on case studies are presented in Section 4. In Section 5, we conclude with summary and several aspects of future work.

2. CONTEXT

This section recalls the Ramadge-Wonham framework for controllability and presents Hierarchical Set Decision Diagrams (SDD) which we use to implement our solution.

2.1 Control Algorithm

Let us now recall how to apply the state based version of Ramadge and Wonham algorithm for the Control Problem (called *CP* in the sequel) on a transition system.

Definition 1. (labeled transition system). A labeled transition system over an alphabet A of actions is a tuple $\mathcal{T} = (S, s_0, \Delta, L)$, where S is the set of configurations, $s_0 \in S$ is the initial configuration, $\Delta \subseteq S \times A \times S$ is the set of transitions and L is a mapping from A into some set of labels.

In this framework, we consider the set of labels $\{e, c\}$: each action with label e corresponds to an environment transition and each action with label c represents a controlled transition. Given a configuration $s \in S$, we denote by $Succ_e(s)$ (respectively $Succ_c(s)$) the set of successor configurations of s by an environment transition (respectively a controlled transition): $Succ_e(s) = \{s' \in S \mid \langle s, a, s' \rangle \in \Delta \text{ and } L(a) = e\}$ and $Succ_c(s) = \{s' \in S \mid \langle s, a, s' \rangle \in \Delta \text{ and } L(a) = c\}$.

Definition 2. (controller strategy). Given an initial subset S_{fail} of S, of *failure* states, a controller *strategy* is a mapping f from $S \setminus S_{fail}$ into $\mathcal{P}(S)$ such that $f(s) \subseteq Succ_c(s)$. Let S_{safe} denote the set of states that will never reach the states in S_{fail}. The reachability space associated with strategy f is the set $S^f_{reach}(s_0)$ defined inductively by:

- $s_0 \in S^f_{reach}(s_0)$
- If $s \in S^f_{reach}(s_0)$ then $\forall s' \in Succ_e(s),\ s' \in S^f_{reach}(s_0)$
- If $s \in S^f_{reach}(s_0) \cap S_{safe}$ then $f(s) \subseteq S^f_{reach}(s_0)$.

Algorithm CP. The algorithm CP solves the control problem in the following sense: it terminates successfully if and only if there exists a winning strategy *i.e.* $S^f_{reach}(s_0) \cap S_{fail} = \emptyset$.
This algorithm computes a subset S_{bad} of states by applying the backward relations from S_{fail} as follows:

(1) Initially $S_{bad} = S_{fail}$, and the set is extended by the following rules:
a) any configuration s for which an environment successor is in S_{bad} (*i.e.* $\exists s' \in Succ_e(s), s' \in S_{bad}$) is added to S_{bad},
b) any configuration s such that all controller successors are in S_{bad} (*i.e.* $\forall s' \in Succ_c(s), s' \in S_{bad}$) is added to S_{bad}.
Let us note that if a controller state has no successor (the controller is blocked), the corresponding state is added to S_{bad}.
(2) The algorithm terminates either when $s_0 \in S_{bad}$ in which case the system is not controllable, or when the rules are no more applicable, in which case the system is controllable. The algorithm returns the set S_{bad}.

Note that in the temporal logic CTL, a branching time logic interpreted over labeled transition systems (see Clarke et al. (2000) for instance), a formula of the form $\mathsf{EX}\varphi$ expresses that there is a path starting from the current state, on which the next state satisfies φ. This is closely related, for the backward transition relation, to step (1.a) of the algorithm. In a similar way, formula $\mathsf{AX}\varphi$, which expresses that for all paths starting from the current state, the next state satisfies φ, corresponds to step (1.b), again for the backward transition relation.

In case of successful termination, S_{safe} is obtained by removing the saturated set of bad states from the set of reachable states: $S_{safe} = S_{reach} \setminus S_{bad}$. The controller strategy is then obtained by restricting the system moves to reach only safe states.

Of course, while the algorithm is polynomial in the size of the system, the state space of the system is very large, so that we need to use an efficient encoding for the state space.

2.2 Hierarchical Set Decision Diagrams

We now briefly describe Hierarchical SDD from Couvreur and Thierry-Mieg (2005). SDD are shared decision diagrams in which arcs are labeled by a *set* of values, instead of a single value. This set may itself be represented by an SDD, thus when labels are SDD, we think of them as hierarchical decision diagrams. This hierarchical aspect allows to directly exploit the structure of a specification, and offer significant performance improvements over most BDD variants (Thierry-Mieg et al. (2009)). In this paper we do not define SDD, the interested reader can refer to Thierry-Mieg et al. (2009) for the full definition. An *SDD* node δ represents a set of states of the system, with $\emptyset$ for the empty set, and we denote by $\mathbb{S}$ the set of all SDD nodes.

SDD natively offer support for set operations like union, intersection, and set difference ($\cup, \cap, \setminus$), using caches to have complexity bounds related to the number of nodes in the decision diagram rather than the number of states (paths), like in standard BDD.

An important characteristic of SDD is the way operations (*e.g.* a transition relation) are defined. Contrary to common practice that uses a symbolic encoding of the next-state function represented by a BDD, SDD define a notion of homomorphism to capture operation semantics. A homomorphism is a mapping $\Phi : \mathbb{S} \mapsto \mathbb{S}$ satisfying $\Phi(\emptyset) = \emptyset$. We denote by Id the identity morphism ($Id(\delta) = \delta$ for any $\delta \in \mathbb{S}$), and through slight notation abuse, we consider any $\delta \in \mathbb{S}$ as a constant homomorphism. A homomorphism is *linear* if it preserves the union operation: $\forall \delta, \delta' \in \mathbb{S}, \Phi(\delta \cup \delta') = \Phi(\delta) \cup \Phi(\delta')$.

Homomorphisms can be combined by composition $(\Phi \circ \Phi')(\delta) = \Phi(\Phi'(\delta))$ or by addition $(\Phi + \Phi')(\delta) = \Phi(\delta) \cup \Phi'(\delta)$, yielding a homomorphism. These compositions are linear by construction if their operands are linear.

Non-linear compositions of homomorphisms are also possible, for instance, intersection defined by $(\Phi * \Phi')(\delta) = \Phi(\delta) \cap \Phi'(\delta)$, or set difference defined by $(\Phi - \Phi')(\delta) = \Phi(\delta) \setminus \Phi'(\delta)$. The non-linearity of these constructions means that we cannot decompose their evaluation as the union of evaluations on paths (states) of the decision diagram. However they can still be combined using $\circ, +\ldots$ with other morphisms.

The **transitive closure** * unary operator allows to perform a fixpoint computation. For any homomorphism Φ and any node $\delta \in \mathbb{S}$, $\Phi^\star(\delta)$ is evaluated by repeating $\delta \leftarrow \Phi(\delta)$ until a fixpoint is reached. In other words, $\Phi^\star(\delta) = \Phi^n(\delta)$ where n is the smallest integer such that $\Phi^n(\delta) = \Phi^{n+1}(\delta)$. This operator is often applied to $(Id + \Phi)$ instead of just Φ, allowing to accumulate newly computed states in the result (*i.e.* perform a least fixpoint computation). For instance, when $s_0 \in \mathbb{S}$ is an initial set of states, and *Succ* designates a transition relation, reachable states are computed by the expression $(Id + Succ)^\star(s_0)$.

Finally, definition of the **inverse** Φ^{-1} of a homomorphism Φ allows to obtain the predecessors by Φ of a set of states. Φ^{-1} can automatically be deduced from Φ if the user provides the set of potential states (*e.g.* usually the set of reachable states). The set of potential states is used to solve the problems due to non reversible operations, that destroy information (for instance assignment to a variable loses its previous value). This inverse mechanism is used to build the backward symbolic transition relation, which is necessary for CTL model-checking, or as proposed in this paper, to solve the control problem.

Given this high-level operation definition framework and the algorithms described in Hamez et al. (2008), the SDD library is able to use rewriting rules to change the order of event application transparently, to implement the saturation algorithm for decision diagrams originally due to Ciardo et al. (2003). This algorithm in many cases allows to bypass the "peak effect" of usual symbolic BFS fixpoint computation, where some intermediate step in the computation is untractable. It is typically two to three orders of magnitude more efficient than BFS in both time and memory, when the system admits asynchronous behaviors (*e.g.* for Petri net models, interacting processes, etc.).

Indeed, with both automatic saturation and hierarchy, SDD have been shown in Thierry-Mieg et al. (2009) to compare favorably to other existing decision diagram packages.

3. IMPLEMENTING CONTROL SYNTHESIS WITH SDD

This section defines our symbolic process to build a controller according to the definitions of Section 2.1.

Starting from a transition system $\mathcal{T} = (S, s_0, \Delta, L)$ with labels in $\{e, c\}$, we describe a control problem as a tuple $P = \langle s_0, Succ_c, Succ_e, bad \rangle$, where $s_0 \in \mathbb{S}$ designates an initial (set of) configuration(s) represented as an SDD, $Succ_c, Succ_e$ represent respectively the transitions associated with the controller and the environment, represented as homomorphisms, and *bad* is a system property expressed as a homomorphism. In simple cases, encountered in most frequent situations, *bad* will be a simple labeling function, *i.e.* a homomorphism that selects paths (states) satisfying a given state property. But *bad* could also be an arbitrarily complex expression, like a CTL property.

3.1 SDD Homomorphisms for Controllability.

The first step is to build the set of reachable states, defined as $S_{reach} = (Succ_c + Succ_e + Id)^\star(s_0)$. This least fixpoint is efficiently computed using the built-in automatic saturation algorithms of SDD and scales relatively well. Once this set of potential states is obtained, we can compute the inverse of $Succ_c$ and $Succ_e$ denoted respectively by $Pred_c$ and $Pred_e$.

The initial set of bad states can be computed as $S_{bad} = S_{fail} = bad(S_{reach})$. The construction procedure then needs to perform a least fixpoint by adding to the set S_{bad}:

- Predecessors by an uncontrolled event of a bad state, according to step (1.a) of *CP* algorithm: $Pred_e(S_{bad})$. This is just like evaluating $\mathsf{EX}_e(bad)$ in CTL, using backward transition firing.
- States from which any move of the controller leads to a bad state, corresponding to step (1.b) of *CP* algorithm: $Pred_c(S_{bad}) \setminus Pred_c(S_{reach} \setminus S_{bad})$. This is like backward evaluation of $\mathsf{AX}_c(bad)$ in CTL, and is handled using the usual translation to $\neg\mathsf{EX}(\neg bad)$.

To allow expression of this least fixpoint using the * closure operator of homomorphisms, we slightly rewrite these terms as homomorphisms, essentially replacing S_{bad} in these definitions by Id and switching to homomorphism composition operators rather than set-theoretic SDD operations. We thus obtain:

$$S_{bad} = (Pred_e + (Pred_c - Pred_c \circ (S_{reach} - Id)) + Id)^\star \circ bad(S_{reach})$$

This slightly complex expression is passed in this form to the SDD library, which optimizes its evaluation using various rewriting rules and strategies.

The final step is to check if $s_0 \cap S_{bad} = \emptyset$. This answers the controllability problem: if s_0 is contained in S_{bad} then no winning strategy exists, *i.e.* there is no maximally permissive controller. The computation also allows us to exhibit S_{bad} and through the complement in S_{reach} the set S_{safe} of safe states.

3.2 Strategy Synthesis.

The synthesis problem then consists in building a controller that does not let the system evolve outside of the safe states. In theory, the controller could be implemented directly by building a finite automaton with one state per good state, and synchronizing it with the system, so that the system cannot evolve outside of the identified good states. However, in practice, the number of states of the system makes this approach prohibitively expensive when using explicit data structures.

We propose to implement the controller directly using the same symbolic data structures. We embed S_{bad} directly in the controller software, stored as an SDD. When the system is running, from any given state where the controller should take a decision, we build the (single path) SDD that represents this state, apply a single step $Succ_c$ to it to obtain immediate successors, then remove from the resulting states any state in S_{bad}. All the complexity of these operations are quite low, as S_{bad} can often be compactly represented as an SDD.

In fact we can simply store only the "border" of safe states, S_{border}, that is, bad states that have a predecessor in S_{safe}. Or with a slight variation of the algorithm, we can also store the safe states in the controller instead of bad states.

The choice of which of these to use is done offline, where time can be spent deciding which of these three choices has the smallest SDD representation, and heuristically trying to reduce their representation size by reordering variables in the decision diagram. Note that in SDD, as in most variants of BDD, the representation size is not directly linked to the number of elements contained in the set, and thus the "border" states

approach is not necessarily more effective than the full bad states one.

4. CASE STUDIES

This section presents two illustrative case studies, 5AGV and Train Crossing, the latter integrating explicit discrete time, to test the capability of our approach. In addition, experiments on two benchmarks from WODES'08 are shown in the end: Cat and Mouse Tower (CMT) and Dining Philosophers (DP).

The examples are given as compositions of labelled Petri nets (with discrete time in the Train example), modelled using the ITS framework, a formalism built to exploit the characteristics of SDD for model-checking problems.

ITS, introduced in Thierry-Mieg et al. (2009), is a hierarchical and compositional modelling framework, that uses label synchronization to compose behaviors. ITS define notions of type and instance: one can use elementary types such as labeled (Time) Petri nets or any kind of labeled transition system, or one of several variants of a built-in composite type that contains instances. The instances in a composite type can be of elementary or composite nature, allowing hierarchical modeling.

The hierarchical nature of the model is then mapped to a hierarchical state representation using SDD, and a homomorphism representation of the transition relation.

4.1 5AGV

Problem. The five Automated Guided Vehicles (5AGV) problem from Krogh and Holloway (1991) describes a factory floor which consists of three workstations *w1*, *w2*, and *w3* which operate on parts, two input stations, *input1* and *input2*, one output station *output*, and five AGVs, *agv A*, *agv B*, *agv D*, *agv E*, and *agv F*, which move parts from one station to another. The corresponding Petri Net model of this system is described in Fig. 1 from Lakos and Petrucci (2004). The gray rectangles denote the dangerous zones where collision may occur between multiple AGVs. Blackened transitions represent controllable events, while the other events are uncontrolled.

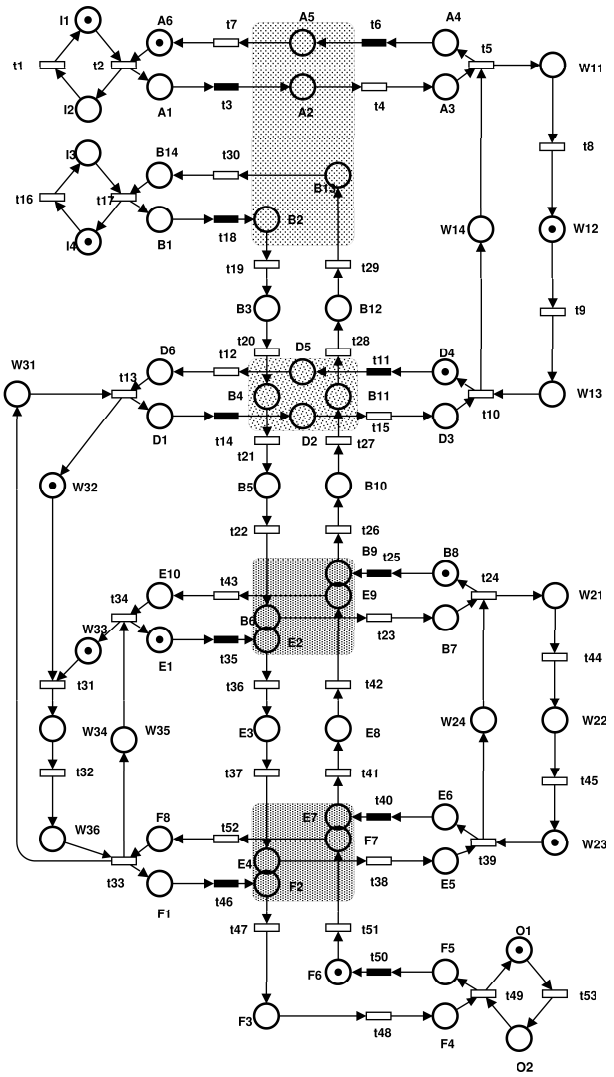

Fig. 1. Petri Net model for 5AGV

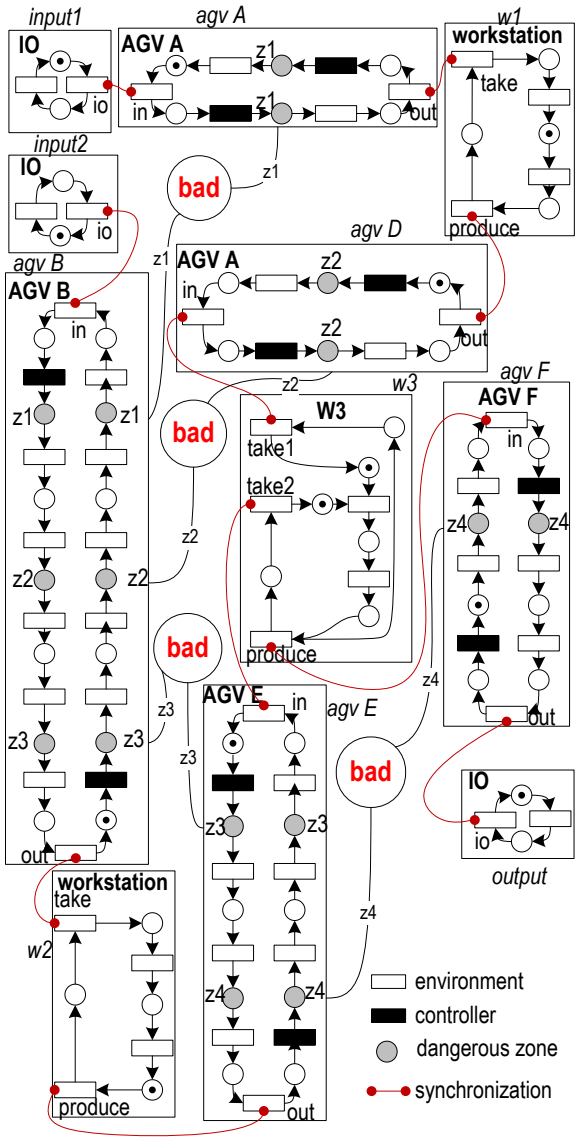

Fig. 2. ITS model for 5AGV

Model. The ITS model for 5AGV is shown in Fig. 2. This is a composition of several ITS instances for the system, where instances which have the same structure are instances of the same type. For example,*w1* and *w2* have the same type **workstation**, *input1*, *input2*, and *output* have the same type **IO**. The controlled transitions are labeled by *controller* and represented as black rectangles, while the other transitions are labeled by *environment*. Each AGV component defines its danger zone(s) as a state in which either of the two places in the zone are marked (e.g. "z1","z2"...). A bad state for the AGV system is then when two AGVs are occupying the same zone, as represented by a synchronization (circle with "bad" label in Fig. 2, like for the synchronization of "z1" in *agv A* and *agv B*).

4.2 Train Crossing

Problem. This benchmark from Berthomieu and Vernadat (2003) is given as a set of (Time) Petri Nets. It represents a section of a railway protected by a gate; trains approach the gate nondeterministically, with some constraints on the time they take to traverse the danger zone. When they get close to the gate, they trigger a sensor "App", detectable by the controller. Then, in 3 to 5 time units they reach the actual crossing zone which they occupy for 2 to 4 time units depending on the speed and length of the train. Finally they trigger an "Exit" sensor when they leave the danger zone.

The controller can act on the single gate of the system, itself time constrained. It can send a signal to "open" or "close" the gate any time a sensor is triggered. It then takes 1 to 2 time units to open or close the gate. A mechanism is built in the gate so it can switch from opening to closing directly, without fully opening the gate. A state is labeled as *bad* if a train is *on* the crossing while the gate is *open*.

The original model includes a controller implementation (defined for verification purposes), that essentially counts train in the danger zone, sends an order to close the gate when the first train approaches, and sends an order to open the gate only when the last train in the danger zone leaves it. The controller has been suppressed in our model since we wish to synthesize it. This benchmark model widely used in the timed model-checking literature can be scaled up by increasing the number of trains.

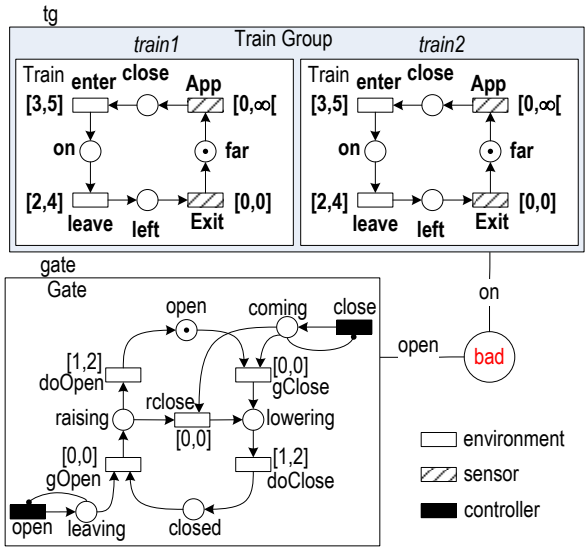

Fig. 3. ITS model for Train Crossing

Model. The ITS model of train crossing is illustrated in Fig. 3, for the case with two trains.

The time Petri net is interpreted over discrete time in the ITS model, so that it remains a discrete event system. We thus have a special action $\mathbb{1}$, which represents a duration of one time unit, and on which all components must synchronize [1] . The $\mathbb{1}$ action is considered unobservable by the controller in this experiment.

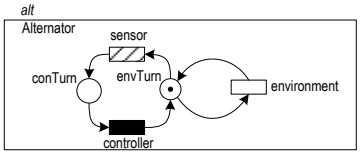

Fig. 4. Alternator for Train Crossing

Let us consider the controller can take a control action each time a train triggers one of the sensor actions "App" or "Exit". This behavior is achieved by synchronizing the system description with an alternator component depicted in Fig. 4: when in the state "envTurn", any uncontrolled action (i.e. white transition in Fig 3), including the $\mathbb{1}$ action, can be fired. Firing an observable action (i.e. sensor actions in Fig 3) switches the alternator state to "conTurn". The controller can then take a single control action (send "open" or "close" to the gate, or do nothing) and the alternator switches back to "envTurn".

4.3 Experimental Results.

Table 1. Experimental Results

n k	time *sec*	mem *MB*	S_{reach}	S_{safe}	S_{bad}	S_{border}
5 AGV Specification						
– –	0.06	2.3	3.10×10^7 (47)	1.66×10^7 (141)	1.44×10^7 (166)	1.02×10^7 (190)
Train specification						
n=5 –	14.3	17	1.54×10^7 (380)	9.09×10^6 (525)	6.27×10^6 (467)	9.19×10^5 (342)
n=10 –	257	130	8.44×10^{12} (540)	5.30×10^{12} (810)	3.14×10^{12} (712)	3.30×10^{11} (512)
n=20 –	3446	859	1.44×10^{24} (860)	9.26×10^{23} (1380)	5.09×10^{23} (1202)	3.95×10^{22} (852)
CMT Specification						
n=535 k=1	207	3099	7.2×10^6 (10704)	7.2×10^6 (12841)	3.8×10^3 (7496)	3.8×10^3 (7496)
n=11 k=11	2888	2859	8×10^{23} (7934)	6.7×10^{22} (12795)	7.3×10^{23} (18250)	2×10^{23} (313026)
n=1 k=25	16260	2066	5.6×10^8 (3408)	6.6×10^3 (162)	5.6×10^8 (3407)	1.2×10^4 (267)
DP Specification						
$n=2 \times 10^7$ k=1	561	6.5	*N/A* (305)	*N/A* (440)	*N/A* (87)	*N/A* (87)
$n=10^4$ k=1	0.52	3.34	7.7×10^{4364} (133)	7.7×10^{4364} (187)	1.4×10^{1505} (44)	1.4×10^{1505} (44)
n=200 k=3	207	1109	4.6×10^{129} (105)	4.6×10^{129} (140)	8.3×10^{107} (37)	8.3×10^{107} (84)
n=10 k=30	1119	1317	8.4×10^{14} (49)	1.4×10^{14} (62)	7×10^{14} (23)	1.1×10^{14} (48)

Table 1 reports experimentation results from our prototype tool, run on a 2.6 GHz PC with 4 *GB* of memory. The first column gives the values of parameters for the models which are scalable, i.e. the number of trains in the train example, number

[1] Provided the time bounds are closed (i.e. $[3,5]$ inclusive and not $[3,5[$), it has been shown that the reachable states (up to clock values) are the same in dense or discrete settings in Popova (1998).

of floors and cats for CMT and number of philosophers and intermediate steps for DP. Time is given in seconds and memory consumption in *MB*. We then present the size in both number of states (with quite large values) and number of SDD nodes in the final representation (between parenthesis, representative of the memory occupation) for all reachable states (S_{reach}), the set of safe states (S_{safe}), the set of bad states (S_{bad}) and the set of border states S_{border} (see section 3.2).

The AGV model is solved very easily, and is not scalable. This classic literature example is handled in 0.06 seconds using 2.3 MB of RAM. The train crossing model is more difficult, due to the time constraint management, but still scales relatively well: we can solve the control problem for up to 20 trains. The CMT benchmark example scales well in number of floors, but more poorly in number of cats. Similarly the DP example scales very well in number of philosophers, but not in number of intermediate steps. The very large number of philosophers in the first line uses a recursive encoding that exploits problem symmetry and provides logarithmic complexity to the solution. The storage requirement is small but the number of states, denoted as *N/A* (not available) could not be computed due to the overflow of the GNU big-numbers library. However, the introduction of intermediate steps breaks this symmetry when computing S_{safe}, producing a peak effect common to BDD approaches.

As discussed in section 3.2, the relative representation sizes of S_{bad}, S_{safe} and S_{border} depend on the model. For example, for AGV storing S_{safe} in the controller is the best choice, while S_{border} is the best choice for the train example.

5. CONCLUSION

This paper has shown an approach developed for solving controller synthesis for discrete event systems, possibly enriched with explicit discrete time. To cope with industrial size systems, we elaborated a general implementation based on the Instantiable Transition Systems (ITS) framework and Hierarchical Set Decision Diagrams (SDD). The two associated techniques have proven excellent scalability for model checking, which is a related problem.

We encoded several case studies to show the flexibility of our framework. In particular, we selected an example with time constraints (train) to assess the possible enrichment. Experiments show good performances and scalability with minimal implementation costs, thanks to the framework and the involved techniques.

Another advantage of our approach is to provide several ways to implement the controller. Then, one can choose the most efficient one (*e.g.* in terms of memory).

Our next step is to deal with larger and more complex specifications such as the automated freeway presented in Bérard et al. (2008). To bridge the gap from the high level view of a system designer and the formal model used in the tool, we propose to rely on a Domain Specific Language (DSL) to define the system semantics in a way that allows automatic translation into SDD. A further goal would be to elaborate distributed controllers.

REFERENCES

Bérard, B., Haddad, S., Hillah, L., Kordon, F., and Thierry-Mieg, Y. (2008). Collision Avoidance in Intelligent Transport Systems: towards an Application of Control Theory. In *Proceedings of the 9th International Workshop on Discrete Event Systems (WODES'08)*, 346–351. IEEE Press, Göteborg, Sweden.

Berthomieu, B. and Vernadat, F. (2003). State class constructions for branching analysis of time petri nets. In *9th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2003)*, volume 2619 of *LNCS*, 442–457. Springer.

Ciardo, G., Marmorstein, R., and Siminiceanu, R. (2003). Saturation unbound. In *Tools and Algorithms for the Construction and Analysis of Systems*, 379–393. LNCS 2619.

Clarke, E., Grumberg, O., and Peled, D. (2000). *Model Checking*. MIT Press.

Couvreur, J.M. and Thierry-Mieg, Y. (2005). Hierarchical Decision Diagrams to Exploit Model Structure. In *Formal Techniques for Networked and Distributed Systems - FORTE 2005*, 443–457.

Feng, L. and Wonham, W.M. (2008). Supervisory control architecture for discrete-event systems. *IEEE Transactions on Automatic Control*, 53(6), 1449–1461.

Gromyko, A. and Pistore, M. (2006). A tool for controller synthesis via symbolic model checking. *Proceedings of the 8th International Workshop on Discrete Event Systems*, 475–476.

Hamez, A., Thierry-Mieg, Y., and Kordon, F. (2008). Hierarchical Set Decision Diagrams and Automatic Saturation. In *ICATPN 2008*, volume 5062 of *LNCS*.

Krogh, B. and Holloway, L. (1991). Synthesis of feedback control logic for discrete manufacturing systems. *Automatica*, 27(4), 641–651.

Lakos, C. and Petrucci, L. (2004). Modular analysis of systems composed of semiautonomous subsystems. In *ACSD'04: Proceedings of the Fourth International Conference on Application of Concurrency to System Design*, 185–194. IEEE Computer Society, Washington, DC, USA.

Ma, C. and Wonham, W.M. (2008). STSLib and Its Application to Two Benchmarks. In *Proceedings of the 9th International Workshop on Discrete Event Systems (WODES'08)*, 119–124. IEEE Press, Göteborg, Sweden.

Miremadi, S., Akesson, K., Fabian, M., Vahidi, A., and Lennartson, B. (2008). Solving two supervisory control benchmark problems using supremica. In *Proceedings of the 9th International Workshop on Discrete Event Systems (WODES'08)*, 131–136. IEEE Press, Göteborg, Sweden.

Popova, L. (1998). Essential States in Time Petri Nets. *Informatik-Berichte*, 96.

Ramadge, P. and Wonham, W. (1987). Supervisory Control of a Class of Discrete-Event Processes. *SIAM Journal of Control and Optimization*, 25(1), 206–230.

Schmidt, K., Moor, T., and Perk, S. (2008). Nonblocking hierarchical control of decentralized discrete event systems. In *IEEE Transactions on Automatic Control*, volume 53, 2252 – 2265.

Thierry-Mieg, Y., Poitrenaud, D., Hamez, A., and Kordon, F. (2009). Hierarchical set decision diagrams and regular models. In *Tools and Algorithms for the Construction and Analysis of Systems, 15th Int. Conference, TACAS 2009*, volume 5505 of *LNCS*, 1–15. Springer.

Zhang, Z. and Wonham, W. (2001). STCT: An Efficient Algorithm for Supervisory Control Design. In *Symposium on Supervisory Control of Discrete Event Systems*, 249–261.