Proc. of 2nd IEEE Int. Conf. on *Formal Engineering Methods*, Dec. 1998, Brisbane, Australia

Automatic Synthesis of Controllers from Formal Specifications

Enrico Tronci[1]
Dipartimento di Matematica Pura ed Applicata, Università di L'Aquila, Coppito 67100 L'Aquila, Italy
`tronci@univaq.it`, `http://univaq.it/~tronci`

Abstract

Many safety critical reactive systems *are indeed* embedded control systems. *Usually a control system can be partitioned into two main subsystems: a* controller *and a* plant. *Roughly speaking: the controller observes the state of the plant and sends commands (stimulus) to the plant to achieve predefined goals.*

We show that when the plant can be modeled as a deterministic Finite State System *(FSS) it is possible to* effectively *use formal methods to automatically synthesize the program implementing the controller from the plant model and the given formal specifications for the closed loop system (plant + controller). This guarantees that the controller program is correct by construction. To the best of our knowledge there is no previously published effective algorithm to extract executable code for the controller from closed loop formal specifications.*

We show practical usefulness of our techniques by giving experimental results on their use to synthesize C programs implementing optimal controllers (OCs) for plants with more than 10^9 states.

Key Words Optimal Control, Finite State Systems, Model Checking, Formal Methods, Supervisory Control, Control Systems, Reactive Systems, Embedded Systems, Discrete Event Systems, Binary Decision Diagrams.

1 Introduction

Many safety critical systems are *reactive systems*. E.g. embedded control systems, (real-time) protocols are reactive systems. Because of this safety critical characteristic correctness is a crucial issue in reactive system design. In this paper we focus our attention on reactive systems modeling (embedded) *control systems*.

A *control system* can often be partitioned into two main subsystems: a *controller* and a *plant*. Roughly speaking: (in an endless loop) the controller observes the state of the plant and on the base of such state information sends commands (stimulus) to the plant to achieve predefined goals.

The designer task is to devise a suitable controller for the given plant. There are programming languages for reactive system design that can translate a program (assuming it defines a finite state system) into a finite state automaton. Model checking (e.g. [9, 6]) techniques and tools can then be used to automatically verify correctness of the designed controller. E.g. this possibility is available for Esterel [7].

To use formal methods (namely: model checking) in this setting we need to write formal specifications for the whole system (plant and controller), a finite state model of the plant, and a program defining our controller (from which a finite state controller model is then obtained).

In this paper we show that when the plant can be modeled as a deterministic *Finite State System* (FSS) a program implementing a controller can be automatically synthesized from the given formal specifications for the closed loop system (plant + controller) and the plant model. This saves on the coding effort for the controller (thus saving on design time and cost) and allows us to focus on formal specifications (thus improving design quality).

Quite often we need to design a controller that meets some performance requirements and is able to efficiently handle (at least some) plant failures. Designing such controller by hand can be quite difficult and error prone. Automatic synthesis via formal methods in these cases not only guarantees correctness but makes controller design easier and faster.

Note that often the word synthesis refers to the process of getting a low level controller program from a high level one. E.g. from an Esterel program (high level procedural description) it is possible to get, e.g., a circuit or a C program (low level procedural descriptions) implementing the defined (with Esterel) controller. We are addressing a different problem here. Our aim is to get a procedural description, e.g. in C, of a controller starting from the *declarative* information given by the plant model and the formal specifications.

Our scenario can be described as an *Optimal Control Problem* (OCP) as follows. We are given a *plant* (i.e. a deterministic FSS), and a set of plant states (*final states*). Plant transitions are triggered by *events*. The set of events is finite. A (state feedback) controller is a function (control law) associating to each plant state an event. I.e. the controller observes the plant state and *triggers* an event accordingly to its control law. We want to find an *Optimal Controller* (OC), i.e. a controller that for each state $\mathbf{x}$ in the plant triggers an event that enables a plant transition which belongs to a shortest path from $\mathbf{x}$ to a final state. Shortly, an optimal controller drives the plant to a final state along a shortest path.

[1] This research has been partially supported by MURST funds.

From the above formulation follows that all plant transitions are considered legal. Thus in an OCP the plant models the *physical plant* (i.e. the controlled system) as well as all safety requirements. Liveness requirements are modeled by the fact that we require the controller to drive the plant to a final state along a shortest path. This is a classical and easy way to express our goals as well as the fact the such goals should be achieved (by the controller) in a finite time (indeed as soon as possible). Note that this situation is different from that of *supervisory control* (e.g. see [19]) which aim is to restrict plant behaviour by only enabling *safe* plant transitions. In fact a supervisor may allow the plant to move from a safe state to another without ever reaching a final state. A controller, on the other hand, is designed to drive the plant to a final state as soon as possible.

Quite often formal specifications for embedded control systems can easily and naturally be expressed within the OCP frame (familiar to electrical engineers).

The above given OCP frame implies that we are following the synchronous programming language paradigm (e.g. as Esterel, Lustre, Signal, StateCharts). I.e. we are assuming that, given the state observations, our controller can compute instantaneously (practically: *fast enough*) commands to send to the plant.

Needless to say control of FSSs has been widely studied in many settings in the control science as well as in the computer science communities. E.g. [25, 18, 19, 16, 17, 12, 20, 5, 10, 1, 11, 22].

In particular, our OCP is a classical problem. In fact a FSSs can be modeled as a graph which vertices represent states and edges represent transitions. Thus our OCP is the problem of finding, for each plant state, a shortest path to a final state. There are well known *classical* algorithms to solve such graph problem (e.g. Dijkstra, Dial, see [2]). Indeed in our case a suitable breadth first visit of the plant transition graph does the job in time $O(s)$ (where s is the size of the plant state space). Unfortunately the number of states of a dynamical system is typically so large (*state explosion*) that the plant transition graph cannot be explicitly represented in memory with a graph. This limits the use of the above mentioned algorithms to *small* systems.

Symbolic techniques based on BDDs (*Binary Decision Diagrams*, [8]) have been very successful in contrasting state explosion in automatic verification of FSSs via Model Checking (e.g. see [6]) as well as in automatic synthesis of *Supervisory Controllers* satisfying given specifications (e.g. see [5, 11]). A symbolic algorithm for automatic synthesis of *Optimal Supervisory Controllers* (OSCs) is given in [22]. Here, however, we are interested in finding an optimal controller in the *traditional* sense (i.e., using a supervisory control terminology, for us all events are controllable and we are allowed to enable at most one event in each state).

We could use the symbolic algorithm in [22]. However the symbolic algorithm in [22] is designed to solve an optimal supervisory control problem with uncontrollable events and (possibly) plant transitions with cost 0. Such problem is computationally harder than the one we are considering here. E.g. an explicit algorithm (given in [20]) to solve a problem (computationally simpler) than the one in [22] has run time $O(s^2 \log s)$ (where s is the size of the plant state space). Experimentally (as usual with BDDs) we found that the symbolic algorithm in [22] has run time $O(s^{1.63})$.

Unfortunately the symbolic algorithm in [22] is not able to exploit the fact that in our OCP there are no uncontrollable events nor 0 cost plant transitions. So its run time to solve our OCP is still $O(s^{1.63})$. This is, of course, not satisfactory since using an explicit breadth first search our OCP can be solved in time $O(s)$. Because of state explosion to make automatic synthesis from formal specifications practically useful we need synthesis algorithms with run time smaller (on practical cases) than $O(s)$. This motivates the design of a symbolic algorithm "tailored" to our OCP.

To use in practice the OCP frame we need to solve the following problems. 1) Find a common representation for the controlled system and for the given formal specifications. Such representation must be effectively usable to compute an optimal controller. 2) Compute an optimal controller with reasonable space and time resources. 3) The optimal controller is our synthesized program. Such program must have reasonable size and execution time.

We show that *First Order Logic on a Boolean Domain* (BFOL) can be used to define the controlled system, the formal specifications and the controller. This, in turn, enables the use of symbolic (i.e. BDD based) techniques to represent the plant, to compute an OC and to generate an efficient (w.r.t. size and speed) and correct (by construction) program implementing such OC. Shortly: we show that automatic synthesis of controllers from closed loop formal specifications is feasible for nontrivial (finite state) plants.

The main results in this paper are the followings.

- We show (sect. 3) how an OCP can be defined using BFOL. This is essential if we want to use BDDs to solve an OCP.

  Indeed sect. 3 defines a BFOL based declarative programming language for (finite state) embedded control systems. The aim of such programming language is to replace the task of writing controller programs with that of writing OCPs, i.e.: (finite state) plant models and closed loop formal specifications.

- We show (4, 5) that BDDs can be used for automatic synthesis of *Optimal Controllers* for deterministic FSSs. Using our symbolic algorithm we succeeded in computing OCs for plants with more than 10^9 states (sect. 7). Experimentally we found (sect. 7) that our symbolic OC synthesis algorithm has a run time of about $O(s^{0.6})$. To the best of our knowledge there is no previously published OC synthesis algorithm with a run time better than a breadth first search (run time: $O(s)$).

Essentially sections 4 and 5 define a compiler for the declarative programming language defined in sect. 3.

- Using a symbolic representation for our solution, as opposed to an explicit state enumerative method, we get a controller executable code of reasonable size and speed (sect. 5, 7). This is essential to make automatic synthesis viable.

- One of the main source of errors in controller design is proper and efficient handling of plant failures or, more in general, exceptional situations. Thus we need to be able to model plant failures in our frame. Typically plant failures are modeled with uncontrollable events (e.g. see [5]) or with nondeterministic transitions. For efficiency reasons we disallow both of these possibilities. Nevertheless we show (sect. 6) how, for controlling purposes, plant failures can be modeled just using deterministic FSSs without uncontrollable events. As a side effect sect. 6 also shows the kind of work needed when designing controllers using our OCP frame. We think this compares favorably with the programming-verification effort needed when a controller is designed by hand using, say, a high level programming language and then verified by model checking.

- To asses performances of our symbolic algorithm we run experiments (sect. 7) on its use to synthesize a C program implementing a minimum time OC for the INTEL semiconductor manufacturing facility (fab) modeled in [3]. We allow for machine failures in our model of such plant. Our techniques can be used whenever the plant can be modeled with a FSS. The only reason why we chose this example is that it can be easily scaled (by changing buffer sizes). The usage report paper [23] gives experimental results on using our symbolic algorithm for OC synthesis for such fab without considering machine failures and lot sizes.

Our paper is organized as follows. *Section* 2 gives basic definitions. *Section* 3 defines our OCP. *Section* 4 gives a symbolic algorithm to compute the least restrictive optimal supervisory controller (OSC) solving our OCP. *Section* 5 gives an algorithm to compute an OC solving our OCP starting from the OSC computed in sect. 4. *Section* 6 shows how to model plant failures in our deterministic frame. *Section* 7 gives experimental results. More details on our experimental results are available in [24].

2 Basic Definitions

We use boolean functions to represent FSSs, i.e., in our context, plants and controllers. $\mathcal{B} = \{0, 1\}$ is the set of boolean values: 0 stands for *false* and 1 stands for *true*. A *boolean function* is a function taking boolean values and returning a boolean value. Often we represent a boolean function f with its *on-set*, i.e. the set of argument values on which f evaluates to 1.

A boolean value or a boolean variable (i.e. a variable ranging on boolean values) is a *formula* (or *boolean expression*). If f is a boolean function of arity n and $F_1 \ldots F_n$ are formulas then $f(F_1, \ldots F_n)$ is also a formula. If F, F' are formulas and x is a (boolean) variable then the following are formulas: $(\neg F)$, $(F \; op \; F')$, $\exists x F$, $\forall x F$, where op is $\vee$, $\wedge$, $\rightarrow$, $=$. A boolean variable x in a formula F is *free* if x is not under the scope of a quantifier $\exists x$, $\forall x$. Formula $F[x := G]$ is obtained from formula F by replacing all free occurrences of x in F with formula G.

Semantics is as usual, i.e.: $\vee$ (or), $\wedge$ (and), $\rightarrow$ (implication), $=$ (iff), $\neg$ (negation), $\exists$ (there exists), $\forall$ (for all).

Let F be a formula with free variables $x_0, \ldots x_{n-1}$. We say that F *holds* (notation: $\models F$) iff $(\forall x_0, \ldots \forall x_{n-1} F) = 1$. We denote with $\equiv$ syntactic equality.

2.1 Notation. We denote with boldface characters vectors of boolean functions or of formulas. We use a vectorial notation in the *expected* way. E.g. let p be an n-ary function, q be an m-ary function and $\mathbf{t} \equiv [0, 1, 1, y]$ be a vector. Then $(\forall \mathbf{x} p(\mathbf{x}, \mathbf{t}, z) \vee \exists \mathbf{x} p(\mathbf{x}) \vee \forall \mathbf{x} q(\mathbf{x}))$ stands for $(\forall x_0, \ldots x_{n-6} \; p(x_0, \ldots x_{n-6}, 0, 1, 1, y, z) \vee \exists x_0, \ldots x_{n-1} \; p(x_0, \ldots x_{n-1}) \vee \forall x_0, \ldots x_{m-1} \; q(x_0, \ldots x_{m-1}))$.

If $[F_0, \ldots F_{n-1}]$, $[G_0, \ldots G_{n-1}]$ are vectors of formulas we write $[F_0, \ldots F_{n-1}] = [G_0, \ldots G_{n-1}]$ for $((F_0 = G_0) \wedge \ldots (F_{n-1} = G_{n-1}))$. E.g. let $\mathbf{p} \equiv [p_0, \ldots p_{n-1}]$ be a vector of m-ary functions and $\mathbf{x}, \mathbf{u}$ be vectors of variables. Then $\mathbf{x} = \mathbf{p}(\mathbf{u})$ stands for $((x_0 = p_0(u_0, \ldots u_{m-1})) \wedge \ldots (x_{n-1} = p_{n-1}(u_0, \ldots u_{m-1})))$. $\Box$

States and events of a finite state plant can be represented with boolean vectors. The plant itself can be defined with a boolean function p s.t. $p(\mathbf{x}, \mathbf{u}, \mathbf{x}') = 1$ iff event $\mathbf{u}$ triggers a transition from state $\mathbf{x}$ to state $\mathbf{x}'$. Function p is the *plant transition relation*. Of course p defines a graph (*plant transition graph*) which vertices are states and with edges labeled with events. There is an edge labeled with event $\mathbf{u}$ from state (vertex) $\mathbf{x}$ to state $\mathbf{x}'$ iff $p(\mathbf{x}, \mathbf{u}, \mathbf{x}') = 1$. Using characteristic functions finite sets can also be defined with boolean functions. E.g. the set of final states of a plant can be defined

$$q^{(0)}(\mathbf{x}) = 0$$
$$q^{(h+1)}(\mathbf{x}) = end(\mathbf{x}) \lor \exists \mathbf{u}, \mathbf{x}'(p(\mathbf{x}, \mathbf{u}, \mathbf{x}') \land q^{(h)}(\mathbf{x}'))$$

Figure 1:

```
Q(p, end) {
do { q(x) = q'(x);
     q'(x) = end(x) ∨ ∃u, x'(p(x, u, x') ∧ q(x'));
   } while (q' ≠ q);       }
```

Figure 2:

with a boolean function *end* s.t. $end(\mathbf{x}) = 1$ iff $\mathbf{x}$ is a final state. Often we will define functions using (possibly fixpoint) computations on functions.

2.2 Example. The (characteristic function of the) set BR of plant states that are final or from which a final state can be reached can be defined as in fig. 1. Let ω be the least natural number h s.t. $q^{(h)} = q^{(h+1)}$. Existence of such ω (finite) follows by Tarski-Knaster fixpoint theorem (e.g. see [13, sect. 5.1]) and the fact that our state space is finite. We have: $q^{(\omega)}(\mathbf{x}) = 1$ iff $\mathbf{x} \in BR$. □

A *Boolean Symbolic Program* is a program manipulating boolean functions. We use a C-like syntax for boolean symbolic programs. In a boolean symbolic program P it is always implicitly understood that before running P all boolean functions occurring in the lhs of an assignment in P are initialized to the identically 0 boolean function. If P is a boolean symbolic program (also *program* in the following) and f is a boolean function occurring in the lhs of an assignment in P then $stdsol(P)(f)$ denotes the value of f after termination of P. If P does not terminate then $stdsol(P)(f)$ is undefined. E.g. the set BR in example 2.2 can be defined with program $Q(p, end)$ in fig. 2. In fact we have $stdsol(Q(p, end))(q) = q^{(\omega)}$.

To efficiently carry out computations on boolean functions we need an efficient representation for them. Binary Decision Diagrams (BDDs) (see [8] for details) are an efficient canonical representation for boolean functions. I.e. for each boolean function f there is (up to f argument ordering) exactly one BDD, $bdd(f)$, representing f. In the following we assume that for each function f an ordering on its arguments is given. Thus $bdd(f)$ is univocally determined. We use BDDs to carry out our computations on boolean functions. This is essential to efficiently *run* boolean symbolic programs. However in the following BDDs can be replaced by any efficient canonical representation for boolean functions.

3 Optimal Control Problem

We define our *Optimal Control Problem* (OCP) (3.3) as well as a solution to it (3.8). Informally the OCP scenario has been defined in sect. 1.

3.1 Notation. We will use (with or without subscripts) the following vectors of boolean variables. $\mathbf{x} \equiv [x_0, \ldots x_{n-1}]$ is a vector ranging over plant present states. $\mathbf{u} \equiv [u_0, \ldots u_{r-1}]$ is a vector ranging over plant events. $\mathbf{x}' \equiv [x'_0, \ldots x'_{n-1}]$ is a vector ranging over plant next states. □

A *backward reachable* state is a plant state that is final or from which a final state can be reached. Since our goal is to drive the plant to a final state backward reachable states are the only ones we need to consider. Backward reachable states play the same role as the plant *coaccessibility* hypothesis (e.g. see [18]).

3.2 Definition. Let p be an $(n + r + n)$-ary boolean function (defining our plant transition relation) and *end* be an n-ary boolean function (defining our set of final states). We define the n-ary boolean function $BR_{p,end}$ as follows: $BR_{p,end}(\mathbf{x}) = sdtdol(Q(p, end))(q)(\mathbf{x})$, where $Q(p, end)$ is defined as in fig. 2. A state $\mathbf{x}$ is said to be *backward reachable* iff $BR_{p,end}(\mathbf{x}) = 1$. □

3.3 Definition. An *Optimal Control Problem* (OCP) is a pair (p, end) s.t.:

- p is an $(n + r + n)$-ary boolean function defining our plant transition relation. Plant p defines a *deterministic* FSS, i.e. $\models (p(\mathbf{x}, \mathbf{u}, \mathbf{x}') \land p(\mathbf{x}, \mathbf{u}, \mathbf{x}'')) \rightarrow (\mathbf{x}' = \mathbf{x}'')$. From now on, unless otherwise stated, we only consider deterministic plants.
- *end* is an n-ary boolean function defining our set of final states.

E.g. $\mathcal{F}$ as in fig. 3 is an OCP. □

3.4 Definition. Let $\mathcal{F} = (p,\ end)$ be an OCP. An *admissible solution* to $\mathcal{F}$ is an $(n + r)$-ary boolean function f satisfying the following conditions.

1. $\models f(\mathbf{x}, \mathbf{u}) \rightarrow \exists \mathbf{x}'(p(\mathbf{x}, \mathbf{u}, \mathbf{x}') \land BR_{p,end}(\mathbf{x}'))$.

   I.e. an event enabled by the controller (f) can always be followed (executed) by the plant (p) and such event leads to a backward reachable state. This implies that a control action is defined only for states from which a final state can be reached. When a final state can be reached from all plant states this condition is called *plant completeness* in [5].

2. $\models \exists \mathbf{u}, \mathbf{x}'(p(\mathbf{x}, \mathbf{u}, \mathbf{x}') \land BR_{p,end}(\mathbf{x}')) \rightarrow \exists \mathbf{u}, \mathbf{x}'(p'(\mathbf{x}, \mathbf{u}, \mathbf{x}') \land BR_{p',end}(\mathbf{x}'))$, where: $p'(\mathbf{x}, \mathbf{u}, \mathbf{x}') = (p(\mathbf{x}, \mathbf{u}, \mathbf{x}') \land f(\mathbf{x}, \mathbf{u}))$.

   Informally: if in the open-loop (uncontrolled) system p (the plant) there is a transition from state $\mathbf{x}$ leading to a backward reachable state then in the

$\mathcal{F} = (p,\ end)$,
$\mathbf{x} \equiv [x_0]$, $\mathbf{u} \equiv [u_0]$, $\mathbf{s}_0 \equiv [0]$, $\mathbf{s}_1 \equiv [1]$, $\mathbf{u}_0 \equiv [0]$, $\mathbf{u}_1 \equiv [1]$,

$$\begin{array}{rcl} p(\mathbf{x},\mathbf{u},\mathbf{x}') & = & (\mathbf{x} = \mathbf{s}_0 \wedge \mathbf{u} = \mathbf{u}_0 \wedge \mathbf{x}' = \mathbf{s}_0) \vee \\ & & (\mathbf{x} = \mathbf{s}_0 \wedge \mathbf{u} = \mathbf{u}_1 \wedge \mathbf{x}' = \mathbf{s}_1) \vee \\ & & (\mathbf{x} = \mathbf{s}_1 \wedge \mathbf{u} = \mathbf{u}_0 \wedge \mathbf{x}' = \mathbf{s}_0) \vee \\ & & (\mathbf{x} = \mathbf{s}_1 \wedge \mathbf{u} = \mathbf{u}_1 \wedge \mathbf{x}' = \mathbf{s}_0), \\ end(\mathbf{x}) & = & (\mathbf{x} = \mathbf{s}_1). \end{array}$$

Open: $J_{p,end}(\mathbf{s}_0) = +\infty$, $J_{p,end}(\mathbf{s}_1) = +\infty$.
Adm: $f_1 = \{(\mathbf{s}_0, \mathbf{u}_0), (\mathbf{s}_0, \mathbf{u}_1), (\mathbf{s}_1, \mathbf{u}_0), (\mathbf{s}_1, \mathbf{u}_1)\}$;
$J_{p,end,f_1}(\mathbf{s}_0) = +\infty$, $J_{p,end,f_1}(\mathbf{s}_1) = +\infty$.
Opt: $f_2 = \{(\mathbf{s}_0, \mathbf{u}_1), (\mathbf{s}_1, \mathbf{u}_1)\}$;
$J_{p,end,f_2}(\mathbf{s}_0) = 1$, $J_{p,end,f_2}(\mathbf{s}_1) = 2$.
Mgo: $f_3 = \{(\mathbf{s}_0, \mathbf{u}_1), (\mathbf{s}_1, \mathbf{u}_0), (\mathbf{s}_1, \mathbf{u}_1)\}$;
$J_{p,end,f_3}(\mathbf{s}_0) = 1$, $J_{p,end,f_3}(\mathbf{s}_1) = 2$.

Figure 3: An OCP

closed-loop (controlled) system p' there is a transition from state $\mathbf{x}$ leading to a backward reachable state. This liveness hypothesis ensures that the controller f can drive the plant to a final state starting from any plant state from which a final state can be reached. This hypothesis plays the same role as the *nonblocking* condition in [5], [18].

Note that an admissible solution f does not rule out loops among backward reachable states. Supervisor f only guarantees that we move to a state from which a final state can be reached.

E.g. an admissible solution to the OCP $\mathcal{F}$ in fig. 3 is f_1 in fig. 3. □

3.5 Remark. Note that we use a static (memoryless) state feedback controller (f). Memory is only in the plant model (p). As in [14], [15] this is no loss of generality since from a computational point of view memory can always be modeled as part of the plant p. □

3.6 Definition. Let $\mathcal{F} = (p,\ end)$ be an OCP. Let $Paths(p,\ end)(\mathbf{x})$ be the set of paths π (in the plant p transition graph) s.t.: π has length at least 1, the first state of π is $\mathbf{x}$, the last state of π is a final state, the only final states in π are the last state and (possibly) the first state. Note that $Paths(p,\ end)(\mathbf{x})$ may be infinite (because of loops) as well as empty (for states from which a final state cannot be reached). We define $\sup\emptyset = 0$.

- Function $J_{p,end}$ from plant states to natural numbers extended with $+\infty$ defines our *objective function* (cost index). $J_{p,end}$ is defined as follows: $J_{p,end}(\mathbf{x}) = \sup\{\text{length of } \pi \mid \pi \in Paths(p,\ end)(\mathbf{x})\}$.

  If f is an admissible solution to $\mathcal{F}$ then we write also $J_{p,end,f}$ for $J_{p',end}$ where: $p'(\mathbf{x},\mathbf{u},\mathbf{x}') = (p(\mathbf{x},\mathbf{u},\mathbf{x}') \wedge f(\mathbf{x},\mathbf{u}))$.
- An *optimal solution* to $\mathcal{F}$ is an admissible solution f to $\mathcal{F}$ s.t.: for all admissible solutions f' to $\mathcal{F}$, for all states $\mathbf{x}$, $J_{p,end,f}(\mathbf{x}) \leq J_{p,end,f'}(\mathbf{x})$.

Note that if f is an admissible solution to $\mathcal{F}$ and $\mathbf{x}$ is a state from which a final state can be reached then $J_{p,end,f}(\mathbf{x}) > 0$.

E.g. $J_{p,end}$ in fig. 3 gives the cost index for the open-loop (uncontrolled) plant p. An optimal solution to the OCP $\mathcal{F}$ in fig. 3 is f_2 in fig. 3. □

3.7 Remark. Informally an optimal solution to an OCP $\mathcal{F}$ is an admissible solution to $\mathcal{F}$ that minimizes $\mathcal{F}$ objective function and satisfies the Dynamic Programming (DP) principle.

Note that all plant states from which a final state can be reached are initial for us. This is because we are looking for a state feedback controller. This hypothesis plays the same role as the *accessibility* hypothesis in [18] or [20] and is no loss of generality. □

In general the optimal solution to an OCP is not unique. However we can define the *most general* optimal solution. As in supervisory control this is the *least restrictive* supervisor (controller) (e.g. see [18], [25], [5], [20]).

3.8 Definition. Let $\mathcal{F}$ be an OCP. A *most general optimal solution* (mgo solution) to $\mathcal{F}$ is an optimal solution f to $\mathcal{F}$ s.t. for all optimal solutions f' to $\mathcal{F}$ we have: $\models (f'(\mathbf{x},\mathbf{u}) \rightarrow f(\mathbf{x},\mathbf{u}))$. I.e. f is the least restrictive optimal solution to $\mathcal{F}$. Note that if f_1 and f_2 are mgo solutions to $\mathcal{F}$ then $f_1 = f_2$. Thus if an mgo solution exists it is unique. We also refer to the mgo solution to $\mathcal{F}$ as the *Optimal Supervisory Controller* (OSC) solving the OCP $\mathcal{F}$. E.g.: the mgo solution to $\mathcal{F}$ in fig. 3 is f_3 in fig. 3. □

3.9 Remark. Our definition of objective function measures path lengths. This means that we are assuming that for each state $\mathbf{x}$ and for each event $\mathbf{u}$ the cost of enabling event $\mathbf{u}$ in state $\mathbf{x}$ is 1. Since we are using a symbolic approach this is no loss of generality. In fact we can represent a transition of cost $k > 0$ with k transitions of cost 1. This entails adding new states to the plant. However the bits we need to add to the state representation are exactly those needed to represent transition costs. We omit the details. We cannot represent transitions with cost 0. □

4 Optimal Solution

We give (4.1) a boolean symbolic program to compute the mgo solution to an OCP.

4.1 Definition. Let $\mathcal{F} = (p,\ end)$ be an OCP. We define program $mgo(\mathcal{F})$ as in fig. 4. □

$mgo(\mathcal{F})$ {
$R(\mathbf{x}, \mathbf{x}') = \exists \mathbf{u} p(\mathbf{x}, \mathbf{u}, \mathbf{x}')$;
$a(\mathbf{x}, \mathbf{x}') = (R(\mathbf{x}, \mathbf{x}') \wedge end(\mathbf{x}'))$;
do { $b\_all(\mathbf{x}, \mathbf{x}') = b\_all'(\mathbf{x}, \mathbf{x}')$;
$g(\mathbf{x}, \mathbf{x}') = (a(\mathbf{x}, \mathbf{x}') \vee b\_all(\mathbf{x}, \mathbf{x}'))$;
$old(\mathbf{x}) = \exists \mathbf{x}' g(\mathbf{x}, \mathbf{x}')$;
$new(\mathbf{x}) = \neg old(\mathbf{x})$;
$b\_new(\mathbf{x}, \mathbf{x}') = (new(\mathbf{x}) \wedge R(\mathbf{x}, \mathbf{x}') \wedge old(\mathbf{x}'))$,
$b\_all'(\mathbf{x}, \mathbf{x}') = (b\_new(\mathbf{x}, \mathbf{x}') \vee g(\mathbf{x}, \mathbf{x}'))$,
} **while** ($b\_all' \neq b\_all$);
$f(\mathbf{x}, \mathbf{u}) = \exists \mathbf{x}'(p(\mathbf{x}, \mathbf{u}, \mathbf{x}') \wedge g(\mathbf{x}, \mathbf{x}'))$; }

Figure 4:

$ctr(f)$ {
$f_0(\mathbf{x}) = \exists u_1 \ldots u_{r-1} f(\mathbf{x}, 1, u_1, \ldots u_{r-1})$;
$f_1(\mathbf{x}) = \exists u_2 \ldots u_{r-1} f(\mathbf{x}, f_0(\mathbf{x}), 1, u_2, \ldots u_{r-1})$;
$\vdots$
$f_{r-1}(\mathbf{x}) = f(\mathbf{x}, f_0(\mathbf{x}), \ldots f_{r-2}(\mathbf{x}), 1)$; }

Figure 5:

Note that $mgo(\mathcal{F})$ is not formally monotone (e.g. see [13]) because of the negation in the definition of new. Thus, in general, the fixpoint computation involved in $mgo(\mathcal{F})$ may not terminate.

4.2 Theorem. Let $\mathcal{F}$ be an OCP. Then $stdsol(mgo(\mathcal{F}))(f)$ is the mgo solution to $\mathcal{F}$.
Proof. (*Sketch.*) Essentially program $mgo(\mathcal{F})$ performs symbolic backward reachability on the plant transition graph "recording" in g in fig. 4 all transitions that are on an optimal (i.e. shortest) path to a final state. The mgo solution f is obtained by enabling exactly the transitions in g. Note that the computation in the `do-while` loop refers only to R and never to p directly. This dramatically improves efficiency w.r.t. the algorithm in [22], however this is only possible for deterministic plants without transitions of cost 0.

E.g. let $\mathcal{F}$ and f_3 be as in fig. 3. Then $stdsol(mgo(\mathcal{F}))(f) = f_3$. □

4.3 Remark. Note that the problem of finding an optimal solution to an OCP entails solving two problems. First we need to find for each state a shortest path to a final state. Second we need to store for each state $\mathbf{x}$ the events enabling a transition belonging to a shortest path from $\mathbf{x}$ to a final state.

There are well known algorithms to solve the first problem, e.g. Dijkstra, Dial (see [2]). Indeed since in our case all transitions have cost 1 a breadth first visit of the plant transition graph does the job in time $O(s)$ where s is the number of states in the plant ($s \leq 2^n$). If s is not too big an array or marks on the plant transition graphs obviously solve the second problem. Unfortunately for FSSs s is so large (*state explosion*) that a running time of $O(s)$ or a storage size of $O(s)$ are unacceptable.

Using a symbolic representation for the plant transition graph, a symbolic algorithm (fig. 4) to compute an optimal controller and a symbolic representation for the controller itself enables us to synthesize OSCs for systems with more than 10^9 states (sect. 7). To the best of our knowledge no previously published algorithm can find an optimal controller for plants of such size.

Note that, once an OCP is suitably casted in BFOL, our symbolic algorithm is quite simple. Still it meets our present goal of showing that automatic synthesis of controllers from closed loop formal specifications is feasible for nontrivial plants. □

From the number of times k we go through the `do-while` loop in fig. 4 we get the length $J^*(p, end) = k - 1$ of a longest path to a final state in the optimally controlled plant.

The hypothesis that the plant is deterministic is needed. In fact for nondeterministic plants we may get wrong results. E.g. let $p(x, u, x') = ((x = 0) \wedge (u = 0) \wedge (x' = 0)) \vee ((x = 0) \wedge (u = 0) \wedge (x' = 1))$, $end(x) = (x = 0)$. We have $f^* = stdsol(mgo(p, end))(f) = \{(0, 0)\}$. Function f^* is not an admissible solution to (p, end) since f^* may drive the plant to state 1 which is not backward reachable.

Note that the algorithm in fig. 4 does not work when plant transitions with cost 0 are allowed. In fact if a state, say $\mathbf{x}$ can reach a final state with $k > 1$ transitions with cost 0 as well as with one transition with cost 1 our algorithm will (erroneously) select the path formed by one transition with cost 1.

5 Optimal Controllers

The OSC $f^s = stdsol(mgo(\mathcal{F}))(f)$ computed by the algorithm in fig. 4 follows the supervisory control paradigm. I.e. f^s enables or disables events. However our goal here is to control a system in a *traditional* sense. I.e. we want to trigger events. This entails enabling exactly one event for each state from which a final state can be reached and efficiently computing such event once a state is given. In 5.3 we show how to get a controller, i.e. a function taking states and returning events, from an OSC.

5.1 Remark. Note that, in general, enabling for each state exactly one event (arbitrarily chosen) from the set of events enabled by an admissible solution f^s (supervisor) to $\mathcal{F}$ does not yield an admissible solution to $\mathcal{F}$. E.g. let $f_4 = \{(\mathbf{s}_0, \mathbf{u}_0), (\mathbf{s}_1, \mathbf{u}_1)\} \subset f_1$ in fig. 3. I.e. f_4 is obtained from f_1 by enabling exactly one event in each state. However f_4 is not an admissible solution to $\mathcal{F}$ since f_4 drives the plant to state $\mathbf{s}_0$ instead that $\mathbf{s}_1$ as required in fig. 3.

Moreover assume that we have found an admissible solution f^s to $\mathcal{F}$ s.t. f^s enables exactly one event for each state from which a final state can be reached. Using directly f^s as a controller entails searching on-line

	M_1	M_2	M_3
Exec. time process 0	45	6	11
Exec. time process 1	51	10	2
States	96	16	13
State Vector bits	7	4	4
Lot sizes	3	1	1

Figure 6: Machine Info

for the event enabled by f^s in a given state. This may lead to a slow controller. To get a fast controller we need to compute a controller off-line rather then selecting an enabled event on-line. Note however that building off-line from f^s a table state-event is out of question because of state explosion. □

5.2 Definition. Let $\mathcal{F} = (p, \mathit{end})$ be an OCP and $\mathbf{f}$ be a vector of r n-ary boolean functions.

We define: $\mathbf{f}^{(p,end)}(\mathbf{x}, \mathbf{u}) = \mathbf{if}\ \exists \mathbf{u}, \mathbf{x}'(p(\mathbf{x}, \mathbf{u}, \mathbf{x}') \wedge BR_{p,end}(\mathbf{x}'))\ \mathbf{then}\ (\mathbf{f}(\mathbf{x}) = \mathbf{u})\ \mathbf{else}\ 0$.

Function $\mathbf{f}$ is said to be a (*optimal*) *controller* (OC) for $\mathcal{F}$ iff $\mathbf{f}^{(p,end)}$ is an (optimal) admissible solution to $\mathcal{F}$. □

A controller $\mathbf{f}$ is a vector of r n-ary functions. Thus $\mathbf{f}$ can be represented with r BDDs. Let $\mathit{size}(\mathbf{f})$ be the number of BDD vertices needed to represent the r BDDs defining $\mathbf{f}$. From the BDDs representing $\mathbf{f}$ we can automatically generate a program, say SW($\mathbf{f}$), or a combinatorial circuit, say HW($\mathbf{f}$), implementing $\mathbf{f}$. Both SW($\mathbf{f}$) and HW($\mathbf{f}$) closely follows the BDD representation of $\mathbf{f}$. Thus their size is O($\mathit{size}(\mathbf{f})$). The running time of SW($\mathbf{f}$) is O($r * n$) whereas that of HW($\mathbf{f}$) is $O(n)$.

From an optimal solution to $\mathcal{F}$ we can compute (off-line) an optimal controller using the symbolic program in fig. 5. Note that this is false in general (remark 5.1). Note also that proposition 5.3 only addresses the issue of getting a *reasonably efficient* OC from an OSC. We do not address the issue of getting (from an OSC) an OC which is optimum w.r.t. given implementation parameters, e.g. OC size or speed.

5.3 Proposition. Let $\mathcal{F} = (p, \mathit{end})$ be an OCP, ctr be as in fig. 5 and $\mathit{Ctr}(f) = [\mathit{stdsol}(\mathit{ctr}(f))(f_0), \ldots \mathit{stdsol}(\mathit{ctr}(f))(f_{r-1})\ ]$. If f is an optimal solution to $\mathcal{F}$ then $\mathit{Ctr}(f)$ is an optimal controller for $\mathcal{F}$.

Proof. (*Sketch.*) Exploiting the fact that our plant is deterministic and that the transition graph of the optimally controlled plant does not contain loops since transition costs are positive.

E.g. from f_2 in fig. 3 we get the optimal controller $\mathbf{f}(\mathbf{x}) = [f_0(\mathbf{x})] = \mathbf{u}_1 = [1]$. □

6 Manufacturing Facility

In this section we show how we can model the INTEL semiconductor manufacturing facility described in [3, figs. 2.1, 2.2]. Of course our techniques can be used whenever the plant can be modeled with a FSS. The only reason why we chose this example is that it can be easily scaled (by changing buffer sizes).

The example in this section serves the following purposes.

- It shows how, for controlling purposes, plant failures can be modeled in our OCP frame. Note that typically plant failures are modeled with uncontrollable events (e.g. see [5]) or with nondeterministic transitions. However, for efficiency reasons, we disallow both of these possibilities in our OCP frame. Thus it is important to show how plant failures can be modeled in our OCP frame since one of the main source of errors in controller design is proper and efficient handling of plant failures or, more in general, exceptional situations.

- It shows the amount of manual work needed to design (correct by construction) controllers using our OCP frame. We think this compares favourably with the amount of work needed to design a controller by hand and then verify its correctness.

We modified the model in [3] as follows: buffers are finite (instead that infinite), machines may fail.

We represent integers with boolean vectors of suitable length on which the usual arithmetic operators (modulo vector length) can be used (as in an ALU). This is supported by our compiler for boolean symbolic programs.

Our manufacturing facility is formed by 3 machines and 6 buffers. Each machine can execute one of two (mutually exclusive) processes. Thus there are 6 processes.

Machine M_i ($i = 1, 2, 3$) can execute processes $p_{i,0}$, $p_{i,1}$. The execution time (in time units of 5 minutes) for process $p_{i,j}$ ($j = 0, 1$) is $T_{i,j}$. Process execution times are obtained from [3, table 2.1] and are given (in 5 minute time units) in fig. 6. Machine lot sizes and states are also in fig. 6.

The transition relation m_i for the FSS modeling machine M_i is given in fig. 7. We use 4 bits to represent events. The first 3 bits represent a machine action among **nop**, **start**, **done**, **working**, **failure**. The last bit is a process index. It tells which process is performing the action identified by the first 3 bits. The number of bits used to represent the state vector for each machine is given in fig. 6.

Note that since we are looking for a controller for us all events are controllable. Thus we cannot model failures with uncontrollable events as usual in supervisory control (e.g. see [5], [19]). On the other hand our synthesis algorithm only works for deterministic plants. Thus we cannot model failures using a controllable event that nondeterministically may lead to a failure state. We model a machine failure as follows.

$\mathbf{nop} = [0, 0, 0]$, $\mathbf{start} = [0, 1, 0]$, $\mathbf{done} = [0, 0, 1]$, $\mathbf{working} = [0, 1, 1]$, $\mathbf{failure} = [1, 0, 0]$.
State encoding: 0 is failure, 1 is idle/waiting, $2 \ldots T_{i,0}$ is for processing of task 0, and $(T_{i,0} + 1) \ldots (T_{i,0} + T_{i,1} - 1)$ is for processing of task 1.
$\mathbf{u}_0$ is a 3 bit vector ranging on machine events. $\mathbf{u}_1$ is a 1 bit vector ranging on machine process names.

$$
\begin{aligned}
& m_i(\mathbf{x}, \mathbf{u}_0, \mathbf{u}_1, \mathbf{x}') = \\
& ((\mathbf{x} = 0) \wedge (\mathbf{u}_0 = \mathbf{failure}) \wedge (\mathbf{x}' = \mathbf{x})) \vee \\
& ((\mathbf{x} = 1) \wedge (\mathbf{u}_0 = \mathbf{nop}) \wedge (\mathbf{x}' = \mathbf{x})) \vee \\
& ((\mathbf{x} = 1) \wedge (\mathbf{u}_0 = \mathbf{start}) \wedge (\mathbf{u}_1 = 0) \wedge \\
& \qquad (\mathbf{x}' = (\mathbf{x} + 1))) \vee \\
& ((\mathbf{x} = 1) \wedge (\mathbf{u}_0 = \mathbf{start}) \wedge (\mathbf{u}_1 = 1) \wedge \\
& \qquad (\mathbf{x}' = (T_{i,0} + 1))) \vee \\
& ((\mathbf{x} \geq 2) \wedge (\mathbf{x} < T_{i,0}) \wedge (\mathbf{u}_0 = \mathbf{working}) \wedge \\
& \qquad (\mathbf{u}_1 = 0) \wedge (\mathbf{x}' = (\mathbf{x} + 1))) \vee \\
& (\mathbf{x} \geq (T_{i,0} + 1)) \wedge (\mathbf{x} < (T_{i,0} + T_{i,1} \text{ - } 1)) \wedge \\
& \qquad (\mathbf{u}_0 = \mathbf{working}) \wedge (\mathbf{u}_1 = 1) \wedge (\mathbf{x}' = (\mathbf{x} + 1))) \vee \\
& ((\mathbf{x} = T_{i,0}) \wedge (\mathbf{u}_0 = \mathbf{done}) \wedge (\mathbf{u}_1 = 0) \wedge (\mathbf{x}' = 1)) \vee \\
& ((\mathbf{x} = (T_{i,0} + T_{i,1} \text{ - } 1)) \wedge (\mathbf{u}_0 = \mathbf{done}) \wedge (\mathbf{u}_1 = 1) \wedge \\
& \qquad (\mathbf{x}' = 1)) \vee \\
& ((\mathbf{x} = T_{i,0}) \wedge (\mathbf{u} = \mathbf{nop}) \wedge (\mathbf{u}_1 = 0) \wedge (\mathbf{x}' = \mathbf{x})) \vee \\
& ((\mathbf{x} = (T_{i,0} + T_{i,1} \text{ - } 1)) \wedge (\mathbf{u} = \mathbf{nop}) \wedge (\mathbf{u}_1 = 1) \wedge \\
& \qquad (\mathbf{x}' = \mathbf{x})).
\end{aligned}
$$

Figure 7: Machine M_i

Machine state 0 represent a failure state. A failure state cannot be reached by any state. Moreover if we are in a failure state we can only stay there (triggering event **failure**). This may seem strange. However we must keep in mind that we are building a feedback controller. The controller will never trigger a transition from a nonfailure state to a failure state. However once a failure state is reached (for whatever reason) the controller will know it (via feedback sensors). The controller cannot repair failures so it can only stay in a failure state after a failure occurs. When the machine recovers the controller will know it (via feedback sensors) and will trigger appropriate events. However we need to make state 0 backward reachable otherwise no control action will be defined for it. This is simply done by adding state 0 to the set of final states. This modeling allows us to keep our plant deterministic even when machine failures are possible and all events are controllable.

Note that this plant modeling guarantees that our OC can handle (machine) failures and/or repairs occurrences (in any combination) at any time during system (plant + controller) execution and not just at the beginning of system execution as it may seem at a first sight. This is because we are using a state feedback controller and all backward reachable states are initial for us.

Shortly: the OC does not trigger failures or repairs, but it will "know" what to do for any possible failure combination in the plant. Thus at any time our synthesized OC will exploit the working part of the plant

$$
\begin{aligned}
& buf_{inlots,outlots}(\mathbf{x}, \mathbf{u}_0, \mathbf{u}_1, \mathbf{x}') = \\
& ((\mathbf{x} \leq \mathbf{bsize}) \wedge \neg(\mathbf{u}_0 = \mathbf{done}) \wedge \neg(\mathbf{u}_1 = \mathbf{start}) \\
& \qquad \wedge (\mathbf{x}' = \mathbf{x})) \vee \\
& ((\mathbf{x} \leq (\mathbf{bsize} \text{ - } \mathbf{inlots})) \wedge (\mathbf{u}_0 = \mathbf{done}) \wedge \\
& \qquad \neg(\mathbf{u}_1 = \mathbf{start}) \wedge (\mathbf{x}' = (\mathbf{x} + \mathbf{inlots}))) \vee \\
& ((\mathbf{x} \geq \mathbf{outlots}) \wedge (\mathbf{x} \leq \mathbf{bsize}) \wedge \neg(\mathbf{u}_0 = \mathbf{done}) \wedge \\
& \qquad (\mathbf{u}_1 = \mathbf{start}) \wedge (\mathbf{x}' = (\mathbf{x} \text{ - } \mathbf{outlots}))) \vee \\
& ((\mathbf{x} \geq \mathbf{outlots}) \wedge (\mathbf{x} \leq (\mathbf{bsize} \text{ - } \mathbf{inlots})) \wedge \\
& \qquad (\mathbf{u}_0 = \mathbf{done}) \wedge (\mathbf{u}_1 = \mathbf{start}) \wedge \\
& \qquad (\mathbf{x}' = (\mathbf{x} + \mathbf{inlots} \text{ - } \mathbf{outlots})))).
\end{aligned}
$$

Figure 8: Buffer

(at that time) as much as possible without violating our given (in the plant model) safety constraints (e.g. no buffer overflow).

Essentially we are taking the so called *optimistic view*, i.e. we assume that we can deal with failures once they occur. Note however that failure interactions or "chain effects" may make such assumption false thus invalidating the above approach to failure modeling. We do not address such issues here.

In fig. 8 is given the transition relation *buf* for a buffer of size **bsize**. The state of a buffer of size **bsize** is represented with a boolean vector of $\lceil \log_2(\mathbf{bsize} + 1) \rceil$ bits. In fig. 8 **inlots** is the lot size for the machine feeding buffer *buf* and **outlots** is the lot size for the machine fed by buffer *buf*. Lot sizes are obtained from fig. 6. Buffer state ranges on nonnegative integers. Thus we require $\mathbf{bsize} \geq \mathbf{inlots}$ and $\mathbf{bsize} \geq \mathbf{outlots}$. We use 6 bits to represent buffer events. Event $\mathbf{u}_0$ (see fig. 8) is the event (disregarding process index) occurring in the machine feeding buffer *buf*. Event $\mathbf{u}_1$ (see fig. 8) is the event (disregarding process index) occurring in the machine fed by buffer *buf*.

flt in fig. 9 is a vector of 3 5-ary boolean functions. **flt** filters out process indexes from machine events.

The transition relation p for the plant modeling the manufacturing facility is given in fig. 9. Note that $\mathbf{u}_0$, $\mathbf{u}_1$, $\mathbf{u}_2$ are 4 bit vectors here.

The first buffer (*buf*) occurring in the plant definition in fig. 9 is the input buffer in the manufacturing facility. I.e. this is the buffer in which new incoming parts are stored. Thus there is no machine feeding such buffer. This is modeled by event **nop** in our plant definition. This means that the input buffer is never fed. This is correct since all (backward reachable) states are initial for us and we use a state feedback controller. Thus (as for failures) arrival of new parts in the manufacturing facility is just seen as a *disturbance* that changes the state of the input buffer. State feedback handles such disturbances.

Our goal is to find a controller that empties the plant in minimum time. Thus the set of final states is defined by function *end* in fig. 9.

Buffer Sizes	State bits	State Space Size	CPU (min)	Max BDD	OC	J^*	Parts	Avg (min.)
3,3,3,1,1,3	25	20,447,232	213:39	3200073	3476	310	19	81.579
3,3,3,2,2,3	27	46,006,272	448:16	3200073	4220	355	21	84.524
3,3,3,3,3,3	27	81,788,928	519:08	3200073	4952	361	23	78.478
4,4,4,2,2,4	31	112,320,000	766:52	3200073	6233	361	25	72.2
5,5,5,3,3,5	31	414,056,448	1222:49	3206601	6970	457	31	73.71
6,6,6,4,4,6	33	1,198,579,200	2580:44	4711283	7740	610	37	82.432
7,7,7,5,5,7	33	2,944,401,408	2938:37	5261260	7789	661	43	76.86

Figure 10: Experimental Results on SUN Sparc Station 20 with 128MB of RAM

In **flt** $\mathbf{u} \equiv [\mathbf{e}, z]$ is a 4 bit vector, $\mathbf{e}$ is a 3 bit vector, z and i are boolean variables (1 bit vectors).

$$\mathbf{flt}(i, \mathbf{u}) = \textbf{if } (i = z) \textbf{ then } \mathbf{e} \textbf{ else } \mathbf{nop}$$

$$\begin{aligned}
p(\mathbf{x}_0, \ldots \mathbf{x}_8, \mathbf{u}_0, \mathbf{u}_1, \mathbf{u}_2, \mathbf{x}'_0, \ldots \mathbf{x}'_8) = \\
& buf_{1,3}(\mathbf{x}_0, \mathbf{nop}, \mathbf{flt}(0, \mathbf{u}_0), \mathbf{x}'_0) \wedge \\
& buf_{1,3}(\mathbf{x}_1, \mathbf{flt}(1, \mathbf{u}_1), \mathbf{flt}(1, \mathbf{u}_0), \mathbf{x}'_1) \wedge \\
& m_1(\mathbf{x}_2, \mathbf{u}_0, \mathbf{x}'_2) \wedge \\
& buf_{3,1}(\mathbf{x}_3, \mathbf{flt}(0, \mathbf{u}_0), \mathbf{flt}(0, \mathbf{u}_1), \mathbf{x}'_3) \wedge \\
& buf_{1,1}(\mathbf{x}_4, \mathbf{flt}(0, \mathbf{u}_2), \mathbf{flt}(1, \mathbf{u}_1), \mathbf{x}'_4) \wedge \\
& m_2(\mathbf{x}_5, \mathbf{u}_1, \mathbf{x}'_5) \wedge \\
& buf_{1,1}(\mathbf{x}_6, \mathbf{flt}(0, \mathbf{u}_1), \mathbf{flt}(0, \mathbf{u}_2), \mathbf{x}'_6) \wedge \\
& buf_{3,1}(\mathbf{x}_7, \mathbf{flt}(1, \mathbf{u}_0), \mathbf{flt}(1, \mathbf{u}_2), \mathbf{x}'_7) \wedge \\
& m_3(\mathbf{x}_8, \mathbf{u}_2, \mathbf{x}'_8)
\end{aligned}$$

$$\begin{aligned}
& end(\mathbf{x}_0, \ldots \mathbf{x}_8) = \\
& (\mathbf{x}_0 = 0) \wedge (\mathbf{x}_1 = 0) \wedge ((\mathbf{x}_2 = 0) \vee (\mathbf{x}_2 = 1)) \wedge \\
& (\mathbf{x}_3 = 0) \wedge (\mathbf{x}_4 = 0) \wedge ((\mathbf{x}_5 = 0) \vee (\mathbf{x}_5 = 1)) \wedge \\
& (\mathbf{x}_6 = 0) \wedge (\mathbf{x}_7 = 0) \wedge ((\mathbf{x}_8 = 0) \vee (\mathbf{x}_8 = 1)).
\end{aligned}$$

Figure 9: Plant end Final States

7 Experimental Results

We implemented (in C) a BDD-based compiler for boolean symbolic programs [21]. As well known BDD size strongly depends on the boolean functions that we are representing (e.g. see [8]). Thus, as usual with BDDs, we need to run experiments to asses performance of a symbolic algorithm. In this section we report on experimental results using our compiler to synthesize OCs. I.e. given an OCP $\mathcal{F}$ we compute $\mathbf{f} = Ctr(stdsol(mgo(\mathcal{F}))(f))$ (see 4.1, 5.3).

As plant we use the INTEL semiconductor manufacturing facility modeled in sect. 6.

Fig. 10 reports our experimental results. We use 12 bits to represent events (i.e. r in 3.1 is 12). Buffer sizes are given by column 'Buffer Sizes'. Column 'State bits' gives the number of bits we used to represent the state of the all plant (i.e. n in 3.1). Column 'State Space Size' is the product of the number of backward reachable states of buffers and machines. This is less than 2^n.

Column 'Max BDD' gives the size of the larger BDD built during the computation when garbage collection is invoked at (about) 3,200,073 BDD nodes. BDD variable ordering is as follows: for each process, event variables followed by an interleaving of present-state and next-state variables.

Column 'OC' gives the number of BDD vertices needed to represent the r BDDs defining $\mathbf{f}$ as well as the size of the automatically generated C program implementing $\mathbf{f}$ (see 5.2).

Column 'J^*' gives $k - 1$, where k is the number of iterations through the `do-while` loop in $mgo(\mathcal{F})$ (fig. 4). Thus 'J^*' is the length of a longest path to a final state in the optimally controlled plant. 'J^*' is also the maximum time (in time units of 5 minutes) needed to empty the plant using an optimal controller.

Column 'Parts' gives the maximum number of parts present in the plant.

Column 'Avg' gives 'J^*'*'time unit'/'Parts', where 'time unit' = 5 minutes. 'Avg' measures (in minutes) the average time between consecutive releases of finished lots (*outs*) from the fab (plant). Note how using the 'Avg' value we can use our formal approach to optimize buffer sizes.

Our controller is automatically synthesized from formal specifications. This guarantees its correctness (by construction) and allows us to easily handle (in an optimal way) finite buffers and machine failures. As a side effect a controller obtained in this way is also able to better exploit plant (finite) resources as compared to "traditional" heuristic scheduling policies often used in manufacturing. In fact from the simulation results in [3] we get the number of finished lots (*outs*) released by the fab in 25,000 minutes. For each scheduling policy simulated in [3] (FIFO, LBFS, FSVCT) we can compute the value $avg\_h = 25{,}000/outs$. Comparing avg_h and 'Avg' (in fig. 10) we can see that our controller improves the average release rate of about 12%. Note however that strictly speaking avg_h and 'Avg' are not obtained by carrying out the same experiment on the fab. Thus comparison between them must be considered with some caution.

Assuming a running time of form $O(s^\alpha)$ from the table in fig. 10 we get a running time for our symbolic algorithm of about $O(s^{0.6})$ where s is the size of the state space (i.e. s = 'State Space Size'). Note that this is better than the running time $O(s)$ of an explicit breadth first visit of the plant transition graph. Indeed to the best of our knowledge no previously published algorithm can synthesize optimal controllers for systems with state spaces as large as the ones we considered here.

8 Conclusions

We addressed the problem (OCP) of automatic synthesis of *Optimal Controllers* (OCs) for deterministic *Finite State Systems* (FSSs) (sect. 3). We gave a symbolic algorithm for automatic synthesis of OCs for deterministic FSSs (sect. 4, 5).

Experimentally (7) we found that our symbolic OC synthesis algorithm has a run time of about $O(s^{0.6})$. To the best of our knowledge there is no previously published OC synthesis algorithm with a run time better than a breadth first search (run time: $O(s)$).

We showed practical usefulness of our techniques by giving (sect. 6, 7) experimental results on their use to synthesize a C program implementing an OC for a semiconductor manufacturing facility (fab). We allow for machine failures in our fab model. This synthesis experiment entails computing OCs for plants with more than 10^9 states. To the best of our knowledge no previously published algorithm can handle systems of such size.

Our results show that automatic synthesis of controllers from closed loop formal specifications is feasible for nontrivial plants. Indeed, once an OCP is suitably casted in BFOL, our results are obtained using a quite simple symbolic algorithm. More sophisticated algorithms working with larger systems are the next step for our research.

References

[1] A. Aziz, F. Balarin, R. K. Brayton, M. D. DiBenedetto, A. Saldanha, A. L. Sangiovanni-Vincentelli, *Supervisory Control of Finite State Machines*, CAV 95, LNCS 939, Springer-Verlag

[2] R. K. Ahuja, T. L. Magnanti, J. B. Orlin, *Network Flows: theory, algorithms and applications*, Prentice-Hall, 1993

[3] E. Adl, M. K. Rodriquez, A.A. Tsakalis, S. Kostas, *Hierarchical Modeling and Control of Re-entrant Semiconductor Facilities*, Proc. of 35th IEEE Conf. on "Decision and Control", 1996, Kobe, Japan

[4] P. J. Antsaklis, J. A. Stiver, M. Lemmon, *Hybrid System Modeling and Autonomous Control Systems*, Hybrid Systems, LNCS 736, Springer-Verlag, 1993

[5] S. Balemi, G. J. Hoffmann, P. Gyugyi, H. Wong-Toi, G. F. Franklin, *Supervisory Control of a Rapid Thermal Multiprocessor*, IEEE Trans. on Automatic Control, Vol. 38, N. 7, July 1993

[6] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, L. J. Hwang, *Symbolic model checking:* 10^{20} *states and beyond*, Information and Computation 98, (1992)

[7] F. Boussinot, R. de Simone, *The Esterel Language. Another Look at Real Time Programming*, Proceedings of the IEEE, 79(9): 1293-1304, 1991

[8] R. Bryant, *Graph-Based Algorithms for Boolean Function Manipulation*, IEEE Trans. on C., Vol. C-35, N.8, Aug. 1986

[9] E. A. Emerson, *Temporal and Modal Logic*, Handbook of Theoretical Computer Science, Elsevier 1990

[10] R. Kumar, V. Garg, *Optimal Supervisory Control of Discrete Event Systems*, SIAM J. of "Control and Optimization", March 1995

[11] S. P. Khatri, A. Narayan, S. C. Krishnan, K. L. McMillan, R. K. Brayton, A. Sangiovanni-Vincentelli *Engineering Change in a Non-Deterministic FSM Setting*, Proc. of 33rd IEEE/ACM "Design Automation Conference", 1996

[12] B. Jonsson, K. G. Larsen, *On the complexity of Equation Solving in Process Algebra*, TAP-SOFT 91, LNCS 493, 1991, Springer-Verlag

[13] J. W. Lloyd, *Foundations of Logic Programming*, Springer-Verlag, 1987

[14] Y. Li, W. M. Wonham, *Control of Vector Discrete-Event Systems I—The Base Model*, IEEE Trans. on Automatic Control, Vol. 38, N.8, 1993

[15] Y. Li, W. M. Wonham, *Control of Vector Discrete-Event Systems II—Controller Synthesis*, IEEE Trans. on Automatic Control, Vol. 39, N.3, 1994

[16] K. M. Passino, P. J. Antsaklis, *On the Optimal Control of Discrete Event Systems*, Proc. 28th IEEE CDC, 1989

[17] J. Parrow, *Submodule Construction as Equation Solving in CCS*, Theor. Computer Science, 68, Elsevier 1989

[18] P. J. Ramadge, W. M. Wonham, *Supervisory Control of a Class of Discrete Event Processes*, SIAM J. Control and Optimization, Vol. 25, N. 1, Jan. 1987

[19] P. J. Ramadge, W. M. Wonham, *The Control of Discrete Event Systems*, Proc. of the IEEE, 77(1):81-98, Jan. 1989,

[20] R. Sengupta, S. Lafortune, *A Graph-Theoretic Optimal Control Problem for Terminating Discrete Event Processes*, Discrete Event Dynamic Systems: Theory and Applications 2, (1992): 139–172, Kluwer

[21] E. Tronci, *Hardware Verification, Boolean Logic Programming, Boolean Functional Programming*, Proc. 10th IEEE Conf. on "Logic In Computer Science" 1995, San Diego, CA, USA

[22] E. Tronci, *Optimal Finite State Supervisory Control*, Proc. of 35th IEEE Conf. on "Decision and Control", 1996, Japan

[23] E. Tronci, *On Computing Optimal Controllers for Finite State Systems*, Proc. of 36th IEEE CDC, 1997, USA

[24] Ftp to `ftp.univaq.it`, login as `anonymous`, cd to `pub/users/tronci/intelfab`

[25] W. M. Wonham, P. J. Ramadge, *On the Supremal Controllable Sublanguage of a given Language*, SIAM J. Control and Optimization, Vol. 25, N. 1, Jan. 1987