Proceedings of 35th IEEE Conference on: *Decision and Control*, Dec. 1996, Kobe, Japan

Optimal Finite State Supervisory Control

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

Abstract

Supervisory Controllers are Discrete Event Dynamic Systems (DEDSs) forming the discrete core of a Hybrid Control System.

We address the problem of automatic synthesis of Optimal Finite State Supervisory Controllers *(OSCs). We show that* Boolean First Order Logic *(BFOL) and* Binary Decision Diagrams *(BDDs) are an effective methodological and practical framework for* Optimal Finite State Supervisory Control. *Using BFOL programs (i.e. systems of boolean functional equations) and BDDs we give a symbolic (i.e. BDD based) algorithm for automatic synthesis of OSCs. Our OSC synthesis algorithm can handle arbitrary sets of final states as well as plant transition relations containing loops and uncontrollable events (e.g. failures). We report on experimental results on the use of our OSC synthesis algorithm to synthesize a C program implementing a minimum fuel OSC for two autonomous vehicles moving on a 4×4 grid.*

1 Introduction

Automatic synthesis of reactive programs is gradually becoming a reality (e.g. see [2], [23]). Here we consider a particular class of reactive programs, namely *Finite State Supervisory Controllers* (SCs). SCs are the discrete core of a *Hybrid Control System*, whereas physical plant and controllers form the continuous one (e.g. see [4], [13]). SCs are *Discrete Event Dynamic Systems* (DEDSs) and as such have been widely studied. Automatic synthesis of SCs satisfying given specifications (*Supervisory Control Problem*, SCP) was studied in [18], [24], [20] within an automata-theoretic framework; in [1], [5] within a language-theoretic framework; in [11], [17] within a process algebra framework; in [10], [14], [15], [19] within a predicate calculus framework.

In an SCP we measure performances using a 2-level cost scale: 0 (the SC satisfies the specifications), ∞ (the SC does not satisfy the specifications). However quite often in control engineering one is interested in an SC that is *as close as possible* to given specifications. This leads us to look for an SC with minimum cost on a many-level cost scale. This is the *Optimal Supervisory Control Problem*, a generalization of SCP. Optimal supervisory control has been studied e.g. in [16], [21].

In this paper we address the problem of automatic synthesis of *Optimal Finite State Supervisory Controllers* (OSCs). Informally the *Optimal Finite State Supervisory Control Problem* (OSCP) scenario is as follows. We are given a *plant* (i.e. a finite state transition system), a set of plant states (*final states*) and a *cost* for each plant transition. Plant transitions are triggered by *events* (or, using a process algebra terminology, *actions*). The set of events is finite and is divided into two disjoint subsets: *controllable* events (e.g. inputs) and *uncontrollable* events (e.g. outputs or failures). Supervisory control consists in restricting the plant behavior by disabling some (possibly all) of the controllable events in a given plant state. Note that uncontrollable events are always enabled. Given a supervisory control law an *objective function* (or *cost index*) associates to each plant state a cost on the base of plant transition costs and of the set of events enabled by the supervisor in a given plant state. The OSC is the least restrictive supervisor that drives the plant to a final state and minimizes a given objective function.

The main obstructions to OSC synthesis are *state explosion* and *nonmonotonicity* of the fixpoint computations involved in all OSC synthesis algorithms. Termination of such fixpoint computations strongly depend on OSCP data (i.e. objective function, plant transition relation, cost domain, final states, controllable events). To guarantee termination and correctness (of the computed result) all published OSC synthesis algorithms use an infinite domain ($\Re^{\geq 0} = \{x \mid x \in \Re \text{ and } x \geq 0\}$) for costs and disallow loops and uncontrollable events in the plant transition relation. When these hypotheses are not satisfied such algorithms in general do not terminate or produce wrong results.

Binary Decision Diagrams (BDDs, see [7]) are an efficient canonical representation for boolean functions that has proved very effective in contrasting state explosion in automatic verification via Model Checking (e.g. see [9], [6]) as well as in automatic synthesis of SCs satisfying given specifications (e.g. see [5]). However no symbolic (i.e. BDD based) algorithm for automatic synthesis of OSCs has been presented in the literature. BDDs can only handle finite domains. Thus to use them in an OSCP we must use a finite domain to represent costs. In an OSCP costs are only used to rank supervisory control laws. Usually only the top levels (small costs) of such ranking need to be faithfully represented. Thus restriction to a finite domain for costs is often quite reasonable.

We show that a suitable *monotonicity hypothesis* on the objective function allows us to use a finite domain for costs and to have loops and uncontrollable events in the plant transition relation. This, in turn, yields a symbolic OSC synthesis algorithm handling arbitrary plant transition relations.

To show termination (and correctness) of a symbolic OSC synthesis algorithm we need to study the effect of OSCP data on such symbolic algorithm. To this end we need to represent in the same language OSCP data and the OSC synthesis algorithm under consideration. This can be done with a symbolic (i.e. BDD based) programming language. However none of the SC or OSC synthesis approaches described in the literature suits our needs since none of them is based on a symbolic programming language. In [22] it has been shown that *Boolean First Order Logic* (BFOL) can be used as a symbolic functional programming language well suited

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

for automatic verification of Finite State Systems. Essentially a BFOL program is a system of boolean functional equations. Here we show that BFOL programs can also be used to define an OSCP and to give symbolic OSC synthesis algorithms.

The main results in this paper are the followings.

- BFOL and BDDs are an effective methodological and practical framework to define and study OSCPs (sec. 4) in much the same way as calculus is for dynamic systems on $\Re^n$.
- Using BFOL programs we give (5.1) a symbolic (i.e. BDD based) OSC synthesis algorithm and show (5.2) its correctness when the objective function satisfies a suitable *monotonicity* hypothesis. Note that no symbolic OSC synthesis algorithm has been presented in the literature. Using BFOL to represent both OSCP data and our OSC synthesis algorithm allows us to overcome the main difficulty in our correctness proof: showing termination of the nonmonotone fixpoint computations involved in our symbolic algorithm.
- Our OSC synthesis algorithm (5.1) can handle arbitrary sets of final states as well as plant transition relations containing loops and uncontrollable events. No previously published OSC synthesis algorithm can handle such general case. Note however that our objective function only depends on costs for enabled events. Objective functions depending also on costs for disabled events have been considered in [21] for costs ranging on $\Re^{\geq 0}$, singleton sets of final states and plants without loops or uncontrollable events.
- Our symbolic algorithm can be effectively used for automatic OSC synthesis. We show (sec. 6) experimental results on its use to synthesize a C program implementing a minimum fuel OSC for two autonomous vehicles (AVs) moving on a 4×4 grid. A problem that cannot be solved with any of the previously published OSC synthesis algorithms because of loops and uncontrollable events (AV engine failures) in the plant transition relation.

2 Basic Definitions

In this section we review standard definitions from First Order Logic (FOL) and Logic Programming (e.g. see [12], [3]) and adapt them to our case: *Boolean First Order Logic* (BFOL). BFOL is simply FOL on the boolean domain $\{0, 1\}$. BFOL syntax is the same as that for FOL with the following provisions: the only constants in BFOL alphabet are $\underline{0}, \underline{1}$ (*propositional constants*); there are no function symbols in BFOL alphabet; any BFOL term (a propositional constant or a variable) is a formula (this is because we are working on Booleans). FV(F) is the set of free variables in formula F. Formula $F[x := t]$ is obtained from formula F by replacing all free occurrences of x in F with term t. Symbol $\equiv$ denotes syntactic equality between strings.

We adopt the usual semantics for first order languages. However our universe will always be the set $Boole = \{0, 1\}$ of boolean values. Boolean value 0 stands for *false* and boolean value 1 stands for *true*. If b denotes a boolean value we also write b for $\underline{b}$. Thus, e.g., symbols 0, 1 are overloaded since they may denote propositional constants, boolean values or *just* integers. The formal context will always make clear the intended meaning. In the following it is always assumed that an alphabet is given.

2.1 Definition. • An *interpretation* I on a given alphabet is a subset of the set $\{p(t_1, \ldots t_n) \mid p$ is an n-ary predicate symbol in the alphabet and $t_1, \ldots t_n$ are propositional constants$\}$. If G is a set of predicate symbols we define $I(G) = \{p(t_1, \ldots t_n) \mid p(t_1, \ldots t_n) \ \epsilon \ I$ and $p \ \epsilon \ G\}$. We write $I(p)$ for $I(\{p\})$. Let p be an n-ary predicate symbol and $v_1, \ldots v_n$ be boolean values. We define the boolean value $I(p)(v_1, \ldots v_n)$ as follows: $I(p)(v_1, \ldots v_n) = \textbf{if } (p(v_1, \ldots v_n) \epsilon I) \textbf{ then } 1 \textbf{ else } 0$. If $n = 0$ then symbol $I(p)$ is overloaded since it may denote a set or a boolean value. The mathematical context will always make clear the intended reading for $I(p)$.

$$
\begin{array}{l}
[I,\sigma](0) = 0, \quad [I,\sigma](1) = 1, \quad [I,\sigma](x) = \sigma(x),\\
[I,\sigma](p(t_1,\ldots t_n)) = I(p)([I,\sigma](t_1),\ldots [I,\sigma](t_n)),\\
[I,\sigma](\neg F) = \textbf{if } [I,\sigma](F) \textbf{ then } 0 \textbf{ else } 1,\\
[I,\sigma](F \vee G) = \textbf{if } [I,\sigma](F) \textbf{ then } 1 \textbf{ else } [I,\sigma](G),\\
[I,\sigma](F \wedge G) = \textbf{if } [I,\sigma](F) \textbf{ then } [I,\sigma](G) \textbf{ else } 0,\\
[I,\sigma](F \rightarrow G) = \textbf{if } [I,\sigma](F) \textbf{ then } [I,\sigma](G) \textbf{ else } 1,\\
[I,\sigma](F = G) =\\
\quad \textbf{if } [I,\sigma](F) \textbf{ then } [I,\sigma](G) \textbf{ else } [I,\sigma](\neg G),\\
[I,\sigma](\exists x F) =\\
\quad \textbf{if } [I,\sigma[x := 0]](F) \textbf{ then } 1 \textbf{ else } [I,\sigma[x := 1]](F),\\
[I,\sigma](\forall x F) =\\
\quad \textbf{if } [I,\sigma[x := 0]](F) \textbf{ then } [I,\sigma[x := 1]](F) \textbf{ else } 0.
\end{array}
$$

Figure 1:

- An *assignment* is a map σ assigning a boolean value $\sigma(x)$ to each variable x. If σ is an assignment and d is a boolean value then $\sigma[x := d]$ is the assignment s.t.: $\sigma[x := d](y) = \textbf{if } (y \equiv x) \textbf{ then } d \textbf{ else } \sigma(y)$. We also write x^σ for $\sigma(x)$. An *environment* is a pair $[I, \sigma]$, where I is an interpretation and σ is an assignment. An environment $[I, \sigma]$ assigns boolean values to formulas as in fig. 1.
- Let P, S be set of formulas and I be an interpretation. We say that I is a model for S (notation: $I \models S$) iff for each formula F in S and for each assignment σ we have $[I, \sigma](F) = 1$. We say that S is a logical consequence of P (notation $P \models S$) iff for each interpretation I we have: if $I \models P$ then $I \models S$. If $S = \{F\}$ then we write: $I \models F$, $P \models F$ for, respectively, $I \models \{F\}$, $P \models \{F\}$.
- A *Model Checking Problem* (MCP) is a pair (I, F), where I is an interpretation and F is a formula. *Answer_MC* is a function from MCPs to *Boole* s.t. *Answer_MC*$(I, F) = 1$ iff $I \models F$. □

2.2 Notation. • We denote with [ ] the empty list and with $[a_0, \ldots a_{n-1}]$ the list with elements $a_0, \ldots a_{n-1}$. We say that a is in list L (notation $a \epsilon L$) iff a is an element of L. We denote with $\mid L \mid$ the set of elements in list L. If $L_1 = [a_0, \ldots a_{n-1}]$ and $L_2 = [b_0, \ldots b_{m-1}]$ then $L_1 * L_2 = [a_0, \ldots a_{n-1}, b_0, \ldots b_{m-1}]$.

- If f is a partial recursive function we write $f(x) \downarrow$ [$f(x) \uparrow$] for $f(x)$ terminates [does not terminate] on argument x. □

2.3 Definition. • A *program statement* is a formula of the form $p(x_1, \ldots x_n) = F$, where p is a predicate symbol and F is a formula s.t. $FV(F) \subseteq \{x_1, \ldots x_n\}$. Formulas $p(x_1, \ldots x_n)$ and F are called, respectively, the *head* and the *body* of the statement.

- A (BFOL) *program* [*module*] is a finite nonempty list P of program statements s.t. for each predicate symbol p occurring in a program statement in P there is *exactly* [*at most*] one program statement in P with p occurring in the head. Note that we call program what in logic programming is usually called (up to statement ordering) the *completion* of a logic program (e.g. see [12] sec. 17 or [3]). Note that a program is a module, but the converse is false. E.g. $P = [p = q]$ is a module but it is not a program since in P there is no program statement with q as head (i.e. q is not defined in P).

- Let P be a module. A predicate symbol p is *in* P iff p occurs in a statement in P. The set $Alph(P)$ of predicate symbols in P defines the alphabet of P. The *definition* of p in P is the program statement (if any) in P in which p occurs in the head. We define: $export(P) = \{p \mid p \in Alph(P)$ and there is a program statement in P where p occurs in the head$\}$; $import(P) = Alph(P) - export(P)$. Note that for a program P we have $export(P) = Alph(P)$ and $import(P) = \emptyset$. We denote with $size(P)$ the number of symbols in P. A *model* for (or a *solution* to) P is an interpretation I s.t. $I \models P$.
- *Hygiene Convention.* Let P, Q be modules. Unless otherwise stated when writing $P*Q$ it is understood that $Alph(P) \cap export(Q) = \emptyset$. This avoids us having to worry about name clashes. E.g. $[p = q] * [g = p]$ ($= [p = q, g = p]$) is allowed by our hygiene convention but $[g = p] * [p = q]$ is not. □

2.4 Notation. We denote with boldface characters vectors of predicate symbols or of formulas. We use a vectorial notation in the *expected* way. E.g. let p be an n-ary predicate symbol, q be an m-ary predicate symbol and $\mathbf{t} \equiv [0, 1, 1, y]$ a vector of terms. 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 $(\forall x_0, \ldots x_{n-6}\ p(x_0, \ldots x_{n-6}, 0, 1, 1, y, z) \vee \exists x_0, \ldots x_{n-1}\ p(x_0, \ldots x_{n-1}) \vee \forall x_0, \ldots x_{m-1}\ q(x_0, \ldots x_{m-1}))$. Let $\mathbf{x} \equiv [x_0, \ldots x_{n-1}]$ and let σ be an assignment. We write $\sigma(\mathbf{x})$ or $\mathbf{x}^\sigma$ for $[\sigma(x_0), \ldots \sigma(x_{n-1})]$. 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 predicate symbols and $\mathbf{x}, \mathbf{u}$ be vectors of variables. Then $\mathbf{x} = \mathbf{p}(\mathbf{u})$ stands for $((x_0 = p_0(u_0, \ldots u_{m-1})) \wedge \ldots (x_{n-1} = p_{n-1}(u_0, \ldots u_{m-1})))$. □

To efficiently carry out computations on BFOL interpretations we need an efficient representation for boolean functions. Binary Decision Diagrams (BDDs) (see [7] 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 predicate symbol p (in the given alphabet) an ordering on its arguments is given. Thus, given an interpretation I, $bdd(I(p))$ is univocally determined. Moreover $size\_bdd(G)$ denotes the number of vertices in BDD G. We heavily rely on BDDs. However in the following they can be replaced by any efficient canonical representation for boolean functions.

We liberally use C-like pseudo-code for our algorithms. We omit proofs because of lack of space.

3 Standard Solution

In general a program may have one, many or no solution (model). To use BFOL as a BDD based programming language we need to univocally associate a solution (if any) to a program. This gives an operational semantics to BFOL and turns it into a (functional) programming language. This is done in the present section generalizing [22].

3.1 Definition. Let J be an interpretation and P be a module. Interpretation $stdsol(J, P)$ is defined as in figure 2. Note that the computation for $stdsol(J, P)$ may or may not terminate. If $stdsol(J, P)$ terminates then $stdsol(J, P) \models P$. We write $stdsol(P)$ for $stdsol(\emptyset, P)$. Let P be a program. The *standard solution* to P is $stdsol(P)$. If $stdsol(P) \downarrow$ then we write $bdd(P, p)$ for $bdd(stdsol(P)(p))$. □

interpretation $stdsol$(interpretation J, module P)
{ Let $P = [p_0(\mathbf{x}) = F_0, \ldots p_{n-1}(\mathbf{x}) = F_{n-1}]$;
  for all $k = 0, \ldots (n-1)$ **do** $I_k = \emptyset$; $i = 0$;
  while $(i < n)$ { $I' = \{p_i(\sigma(\mathbf{x})) \mid$
    $[J(import(P)) \cup I_0 \cup \ldots I_{n-1}, \sigma](F_i) = 1\}$;
    if $(I' == I_i)$ { $i = i + 1$; } **else** { $I_i = I'$;
      for all $k = 0, \ldots (i-1)$ **do** $I_k = \emptyset$; $i = 0$; } }
  return $(I_0 \cup \ldots I_{n-1})$; }

Figure 2: Standard Solution

3.2 Example. Let $P_1 = [p = \neg p]$. P_1 has no solution (model) and $stdsol(P_1) \uparrow$. Let $P_2 = [p = \neg p \vee q, q = q]$. P_2 has exactly one solution (model) $I = \{p, q\}$ and $stdsol(P_2) \uparrow$. Hence, in general, if $stdsol(P) \uparrow$ we cannot conclude that program P has no solution. Let $P_3 = [p = \neg p \vee \neg q, q = q]$. P_3 has one solution $I = \{p\}$ and $stdsol(P_3) = \{p\}$. Let $P_4 = [p = p \vee \neg q, q = q]$. P_4 has three solutions, namely $I_1 = \{p\}, I_2 = \{q\}, I_3 = \{p, q\}$, and $stdsol(P_4) = \{p\}$. □

3.3 Definition. A *query* is a pair (P, g), where P is a program and g is a predicate symbol in P. *Answer_Query* is a function from queries to *Boole* s.t.: $Answer\_Query(P, g)$ = **if** $stdsol(P) \downarrow$ **then** (**if** $stdsol(P) \models g(\mathbf{x})$ **then** 1 **else** 0) **else** $\uparrow$. □

Theorem 3.4 (essentially from [22]) shows that $stdsol(P)$ can be computed using BDDs.

3.4 Theorem. Let P be a program with $Alph(P) = \{p_0, \ldots p_{k-1}\}$. There are BDD based algorithms *bdd_compile*, *bdd_eval* s.t.:

- $bdd\_compile(P)$ = **if** $stdsol(P) \downarrow$ **then** $[bdd(P, p_0), \ldots bdd(P, p_{k-1})]$ **else** $\uparrow$.
- $Answer\_Query(P, g)$ = **if** $stdsol(P) \downarrow$ **then** $bdd\_eval(bdd\_compile(P), g)$ **else** $\uparrow$. □

4 Optimal Control

We show that BFOL can be effectively used to define (4.3) an *Optimal Finite State Supervisory Control Problem* (OSCP) as well as a solution to it (4.7). Informally the OSCP scenario has been defined in sec. 1.

4.1 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. $\mathbf{v} \equiv [v_0, \ldots v_{k-1}]$ is a vector ranging over plant transition costs. □

To each pair $(\mathbf{x}, \mathbf{u})$ we need to associate a cost $\mathbf{v}$ which is the cost to reach a final state from $\mathbf{x}$ by enabling event $\mathbf{u}$ in $\mathbf{x}$. For this reason as usual in classical supervisory control theory we restrict our attention to plants s.t. for each state $\mathbf{x}$, each transition from $\mathbf{x}$ leads to a final state or to a state from which a final state can be reached (4.2). Essentially this is the *coaccessibility* hypothesis in [18].

4.2 Definition. Let I be an interpretation, p be an $(n+r+n)$-ary predicate symbol (denoting our plant transition relation) and end be an n-ary predicate symbol (denoting our set of final states).

- $BR(p, end) = ([\ q(\mathbf{x}) = (end(\mathbf{x}) \vee \exists \mathbf{u}, \mathbf{x}'\ (p(\mathbf{x}, \mathbf{u}, \mathbf{x}') \wedge q(\mathbf{x}')))\ ], q)$. Informally q denotes the set of plant states that are final or from which it is possible to reach a final state (*backward reachable states*).
- Let $BR(p, end) = (Q, q)$. The triple (I, p, end) is said to be *backward reachable* iff $(I \cup stdsol(I, Q)) \models p(\mathbf{x}, \mathbf{u}, \mathbf{x}') \rightarrow q(\mathbf{x}')$. Informally a triple (I, p, end) is backward reachable iff each plant transition leads to a backward reachable state. □

4.3 Definition. An *Optimal Finite State Supervisory Control Problem* (just *Optimal Supervisory Control Problem*, OSCP, in the following) is a 10-tuple $(P, J, p, end, contr, leq, \mathbf{h}, \mathbf{lh}, j, f)$ s.t.:

- P is a program s.t. predicates p, end, $contr$, leq, $\mathbf{h}$, $\mathbf{lh}$ are in P, $stdsol(P) \downarrow$, j, f are not in P.
- p is an $(n + r + n)$-ary predicate symbol denoting our plant transition relation. Informally $p(\mathbf{x}, \mathbf{u}, \mathbf{x}')$ holds iff event $\mathbf{u}$ triggers a transition from state $\mathbf{x}$ to state $\mathbf{x}'$.
- end is an n-ary predicate symbol denoting our set of final states. Informally $end(\mathbf{x})$ holds iff $\mathbf{x}$ denotes a final state.
- The triple $(stdsol(P), p, end)$ is backward reachable. I.e. each plant transition leads to a backward reachable state (equivalently: each nonisolated state is backward reachable).
- $contr$ is an r-ary predicate symbol denoting our set of controllable plant events. Informally $contr(\mathbf{u})$ holds iff $\mathbf{u}$ denotes a controllable event.
- leq is a $(2k)$-ary predicate symbol denoting a total order on costs. We write $(\mathbf{v} \leq \mathbf{v}')$ for $leq(\mathbf{v}, \mathbf{v}')$. Predicate leq satisfies the following conditions (total order):
$stdsol(P) \models (\mathbf{v} \leq \mathbf{v})$,
$stdsol(P) \models ((\mathbf{v}_1 \leq \mathbf{v}_2) \wedge (\mathbf{v}_2 \leq \mathbf{v}_1)) \rightarrow (\mathbf{v}_1 = \mathbf{v}_2)$,
$stdsol(P) \models ((\mathbf{v}_1 \leq \mathbf{v}_2) \wedge (\mathbf{v}_2 \leq \mathbf{v}_3)) \rightarrow (\mathbf{v}_1 \leq \mathbf{v}_3)$,
$stdsol(P) \models (\mathbf{v}_1 \leq \mathbf{v}_2) \vee (\mathbf{v}_2 \leq \mathbf{v}_1)$.
- $\mathbf{h} \equiv [h_0, \ldots h_{k-1}]$ is a vector of $(n+r)$-ary predicate symbols. Vector $\mathbf{h}(\mathbf{x}, \mathbf{u})$ denotes the cost of enabling event $\mathbf{u}$ in plant state $\mathbf{x}$.
- $\mathbf{lh} \equiv [lh_0, \ldots lh_{k-1}]$ is a vector of $(2k)$-ary predicate symbols denoting our cost propagation function. Informally vector $\mathbf{lh}(\mathbf{v}_1, \mathbf{v}_2)$ denotes the cost resulting from incurring first cost $\mathbf{v}_1$ and then cost $\mathbf{v}_2$. Vector $\mathbf{lh}$ satisfies the following condition (leq-monotonicity):
$stdsol(P) \models ((\mathbf{v}_1 \leq \mathbf{v}_2) \wedge (\mathbf{v}_4 = \mathbf{lh}(\mathbf{v}_1, \mathbf{v}_3)) \wedge (\mathbf{v}_5 = \mathbf{lh}(\mathbf{v}_2, \mathbf{v}_3))) \rightarrow (\mathbf{v}_4 \leq \mathbf{v}_5)$.
- f is an $(n + r)$-ary predicate symbol denoting our state feedback supervisory control law. I.e. $f(\mathbf{x}, \mathbf{u})$ holds iff when the plant is in state $\mathbf{x}$ event $\mathbf{u}$ is enabled.
- j is an $(n+k)$-ary predicate symbol denoting our objective function (cost index). Informally $j(\mathbf{x}, \mathbf{v})$ holds iff when applying control law f the cost index value in $\mathbf{x}$ is $\mathbf{v}$.
- J is a module defining our objective function j. Module J is defined as follows: $J = [$
$b(\mathbf{x}, \mathbf{v}) = \exists \mathbf{u}, \mathbf{x}'\ (f(\mathbf{x}, \mathbf{u}) \wedge p(\mathbf{x}, \mathbf{u}, \mathbf{x}') \wedge end(\mathbf{x}') \wedge (\mathbf{v} = \mathbf{h}(\mathbf{x}, \mathbf{u}))) \vee \exists \mathbf{u}, \mathbf{x}'\ (f(\mathbf{x}, \mathbf{u}) \wedge p(\mathbf{x}, \mathbf{u}, \mathbf{x}') \wedge \neg end(\mathbf{x}') \wedge \exists \mathbf{v}_2\ (b(\mathbf{x}', \mathbf{v}_2) \wedge \exists \mathbf{v}_1\ ((\mathbf{v}_1 = \mathbf{h}(\mathbf{x}, \mathbf{u})) \wedge (\mathbf{v} = \mathbf{lh}(\mathbf{v}_1, \mathbf{v}_2)))))$,
$j(\mathbf{x}, \mathbf{v}) = b(\mathbf{x}, \mathbf{v}) \wedge \forall \mathbf{v}'\ (b(\mathbf{x}, \mathbf{v}') \rightarrow (\mathbf{v}' \leq \mathbf{v}))\ ]$.

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

$\mathcal{F} = (P, J, p, end, contr, leq, [h], [lh], j, f)$,
$\mathbf{x} \equiv [x_0]$, $\mathbf{u} \equiv [u_0]$, $\mathrm{s}_0 \equiv [0]$, $\mathrm{s}_1 \equiv [1]$, $\mathbf{u}_0 \equiv [0]$, $\mathbf{u}_1 \equiv [1]$.

$$
\begin{aligned}
P = [\ p(\mathbf{x}, \mathbf{u}, \mathbf{x}') &= (\mathbf{x} = \mathrm{s}_0 \wedge \mathbf{u} = \mathbf{u}_0 \wedge \mathbf{x}' = \mathrm{s}_1) \\
&\quad \vee (\mathbf{x} = \mathrm{s}_0 \wedge \mathbf{u} = \mathbf{u}_1 \wedge \mathbf{x}' = \mathrm{s}_1) \\
&\quad \vee (\mathbf{x} = \mathrm{s}_1 \wedge \mathbf{u} = \mathbf{u}_0 \wedge \mathbf{x}' = \mathrm{s}_0) \\
&\quad \vee (\mathbf{x} = \mathrm{s}_1 \wedge \mathbf{u} = \mathbf{u}_1 \wedge \mathbf{x}' = \mathrm{s}_1), \\
end(\mathbf{x}) &= (\mathbf{x} = \mathrm{s}_1), \\
contr(\mathbf{u}) &= (\mathbf{u} = \mathbf{u}_0), \\
leq(v_0, v_1) &= (v_0 \rightarrow v_1), \\
h(\mathbf{x}, \mathbf{u}) &= (\mathbf{x} = \mathrm{s}_0 \wedge \mathbf{u} = \mathbf{u}_1), \\
lh(v_0, v_1) &= (v_0 \vee v_1)\ ].
\end{aligned}
$$

Adm: $I_1 = \{f(\mathrm{s}_0, \mathbf{u}_0), f(\mathrm{s}_0, \mathbf{u}_1), f(\mathrm{s}_1, \mathbf{u}_0), f(\mathrm{s}_1, \mathbf{u}_1)\}$;
$stdsol(I_1, P * J)(j) = \{j(\mathrm{s}_0, 1), j(\mathrm{s}_1, 1)\}$.
Opt: $I_2 = \{f(\mathrm{s}_0, \mathbf{u}_1), f(\mathrm{s}_1, \mathbf{u}_1)\}$;
$stdsol(I_2, P * J)(j) = \{j(\mathrm{s}_0, 1), j(\mathrm{s}_1, 0)\}$.
Mgo: $I_3 = \{f(\mathrm{s}_0, \mathbf{u}_1), f(\mathrm{s}_0, \mathbf{u}_0), f(\mathrm{s}_1, \mathbf{u}_1)\}$;
$stdsol(I_3, P * J)(j) = \{j(\mathrm{s}_0, 1), j(\mathrm{s}_1, 0)\}$.

Figure 3: An OSCP

4.4 Remark.

- Informally module J in 4.3 works as follows. Formula $b(\mathbf{x}, \mathbf{v})$ holds iff from state $\mathbf{x}$ with control law f we can reach with cost $\mathbf{v}$ a final state. Formula $j(\mathbf{x}, \mathbf{v})$ holds iff $\mathbf{v}$ is the maximum cost to reach a final state starting from state $\mathbf{x}$ when using control law f. Thus, as in [21], we take as objective function j the worst case performance of supervisor f.
- Vector $\mathbf{h}$ in 4.3 plays the same role as the Hamiltonian in a classical optimal control problem for dynamic systems on $\Re^n$ whereas $\mathbf{lh}$ plays the role of cost summation ($\int$ or $\sum$) along *state trajectories*.
- The leq-monotonicity hypothesis in 4.3 allows us to use Bellman's principle of optimality (e.g. see [8]) when constructing a solution to an OSCP (5.1, 5.2).
- Note that backward reachability for ($stdsol(P)$, p, end) as well as the conditions on leq and $\mathbf{lh}$ in 4.3 can all be automatically checked using BDDs. E.g. ($stdsol(P)$, p, end) is backward reachable iff $Answer\_Query(P* [q(\mathbf{x}) = (end(\mathbf{x}) \vee \exists \mathbf{u}, \mathbf{x}'\ (p(\mathbf{x}, \mathbf{u}, \mathbf{x}') \wedge q(\mathbf{x}'))), g(\mathbf{x}) = \forall \mathbf{u}, \mathbf{x}'\ (p(\mathbf{x}, \mathbf{u}, \mathbf{x}') \rightarrow q(\mathbf{x}'))], g) = 1$. By theorem 3.4 this can be evaluated with BDDs.
- 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 modelled as part of the plant p. Note that, unlike [18], [5], we allow nondeterministic plants. When the state is observable (as often the case) this makes our results directly applicable to hybrid control for dynamic systems on $\Re^n$ (e.g. see [4]). When the plant is deterministic our approach is equivalent to the one in [18], [5]. Note also that all plant states are initial for us. Essentially this is equivalent to the accessibility hypothesis in [18] and is no loss of generality when studying SCPs ([18]) or OSCPs ([21]). □

4.5 Definition. Let $\mathcal{F} = (P, J, p, end, contr, leq, \mathbf{h}, \mathbf{lh}, j, f)$ be an OSCP. An *admissible solution* to $\mathcal{F}$ is an interpretation I satisfying the following conditions.

1. $I = I(f)$. I.e. the only predicate symbol occurring in I is f.
2. $I \cup stdsol(P) \models (\neg contr(\mathbf{u}) \wedge \exists \mathbf{x}' p(\mathbf{x}, \mathbf{u}, \mathbf{x}')) \rightarrow f(\mathbf{x}, \mathbf{u})$. I.e. uncontrollable events cannot be disabled. This condition has been called *controller completeness* in [5], [18].
3. $I \cup stdsol(P) \models (contr(\mathbf{u}) \wedge f(\mathbf{x}, \mathbf{u})) \rightarrow \exists \mathbf{x}' p(\mathbf{x}, \mathbf{u}, \mathbf{x}')$. I.e. a controllable event enabled by

the controller can always be followed (executed) by the plant. This condition has been called *plant completeness* in [5].

4. $I' \cup stdsol(I', Q * Q') \models \exists \mathbf{u}, \mathbf{x}'(p(\mathbf{x}, \mathbf{u}, \mathbf{x}') \wedge q(\mathbf{x}')) \rightarrow \exists \mathbf{u}, \mathbf{x}'(p'(\mathbf{x}, \mathbf{u}, \mathbf{x}') \wedge q'(\mathbf{x}'))$, where: $I' = I \cup stdsol(I, P * [p'(\mathbf{x}, \mathbf{u}, \mathbf{x}') = f(\mathbf{x}, \mathbf{u}) \wedge p(\mathbf{x}, \mathbf{u}, \mathbf{x}')])$, $BR(p, end) = (Q, q)$, $BR(p', end) = (Q', q')$. 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 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 can drive the plant to a final state. This hypothesis plays the same role as the *nonblocking* condition in [5], [18].

E.g. an admissible solution to the OSCP $\mathcal{F}$ in fig. 3 is I_1 in fig. 3. □

4.6 Definition. Let $\mathcal{F} = (P, J, p, end, contr, leq, \mathbf{h}, \mathbf{lh}, j, f)$ be an OSCP. An *optimal solution* to $\mathcal{F}$ is an interpretation I satisfying the following conditions.

- I is an admissible solution to $\mathcal{F}$.
- For all admissible solutions I' to $\mathcal{F}$, for all assignments σ, σ' we have: if $stdsol(I, P * J) \models j(\mathbf{x}^\sigma, \mathbf{v}^\sigma)$ and $stdsol(I', P * J) \models j(\mathbf{x}^\sigma, \mathbf{v}^{\sigma'})$ then $stdsol(P) \models (\mathbf{v}^\sigma \leq \mathbf{v}^{\sigma'})$.

E.g. an optimal solution to the OSCP $\mathcal{F}$ in fig. 3 is I_2 in fig. 3. □

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

It is possible to show that the above notions (OSCP, admissible solution, optimal solution) are all well defined.

Note that for us all states are initial. As pointed out in remark 4.4 this is equivalent to have one initial state and the accessibility hypothesis which is no loss of generality. Thus our definition of optimality is equivalent to the one in [21].

In supervisory control we are interested in finding the *least restrictive* supervisor (e.g. see [18], [24], [5], [21]). This leads to the following definition.

4.7 Definition. Let $\mathcal{F}$ be an OSCP. A *most general optimal solution* (mgo solution) to $\mathcal{F}$ is an optimal solution I to $\mathcal{F}$ s.t. for all optimal solutions I' to $\mathcal{F}$ we have: $I' \subseteq I$. I.e. I is the least restrictive optimal solution to $\mathcal{F}$. Note that if I_1 and I_2 are mgo solutions to $\mathcal{F}$ then $I_1 = I_2$. 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 OSCP $\mathcal{F}$. E.g.: the mgo solution to $\mathcal{F}$ in fig. 3 is I_3 in fig. 3. □

5 Optimal Solution

We give (5.1) a BFOL program to compute the mgo solution to an OSCP. By 3.4 this gives a symbolic (i.e. BDD based) algorithm for OSC synthesis. Note how (in 5.1) BFOL programs allow a clear and succinct definition of quite complicated computations. In 5.2 we prove the correctness of our symbolic algorithm. This also shows that an OSCP has an mgo solution and thus (a fortiori) an optimal solution.

5.1 Definition. Let $\mathcal{F} = (P, J, p, end, contr, leq, \mathbf{h}, \mathbf{lh}, j, f)$ be an OSCP. We define module $mgo(\mathcal{F})$ as in fig. 4. □

Essentially module $mgo(\mathcal{F})$ defines a dynamic programming algorithm which in our BFOL framework becomes a fixpoint computation.

Informally module $mgo(\mathcal{F})$ works as follows. Formula $bo(\mathbf{x}, \mathbf{u}, \mathbf{v})$ holds iff from state $\mathbf{x}$ enabling event $\mathbf{u}$ we can reach with cost $\mathbf{v}$ a final state. Formula $adm(\mathbf{x}, \mathbf{v})$ holds iff $\mathbf{v}$ is an admissible (w.r.t. 4.5.2) cost to reach a final state from state $\mathbf{x}$. Formula $g(\mathbf{x}, \mathbf{v})$ (*cost-to-go*) holds iff $\mathbf{v}$ is the least admissible cost to reach a final state from state $\mathbf{x}$. Formula $f(\mathbf{x}, \mathbf{u})$ (defining our controller) holds iff $\mathbf{u}$ is uncontrollable or the cost for enabling $\mathbf{u}$ in $\mathbf{x}$ is less than or equal to the cost-to-go from state $\mathbf{x}$.

Note that $mgo(\mathcal{F})$ is not formally monotone. Thus, in general, the fixpoint computation involved in $mgo(\mathcal{F})$ will not terminate.

5.2 Theorem. Let $\mathcal{F}$ be an OSCP. Then $stdsol(P * mgo(\mathcal{F}))(f)$ is the mgo solution to $\mathcal{F}$.

E.g. let $\mathcal{F}$ and I_3 be as in fig. 3. Then $stdsol(P * mgo(\mathcal{F})) = I_3$. □

5.3 Remark. • By theorem 5.2 and 3.4 given $\mathcal{F}$ we can compute a BDD representation for f, namely: $bdd(P * mgo(\mathcal{F}), f)$. This gives an efficient synthesis algorithm for OSCs.

• It is possible to give counter-examples to show the following facts. (1) If we drop the leq-monotonicity hypothesis on **lh** then there may be no optimal solution to an OSCP $\mathcal{F}$ and $stdsol(P * mgo(\mathcal{F}))$ may yield wrong results. This is the case even when all events are controllable. (2) If we drop the leq-monotonicity hypothesis on **lh** then $stdsol(P * mgo(\mathcal{F}))$ may not terminate. (3) Leq-monotonicity is only a sufficient condition for termination of $stdsol(P * mgo(\mathcal{F}))$. (4) Because of loops and uncontrollable events representing costs with a *large enough* word length does not guarantee termination of $stdsol(P * mgo(\mathcal{F}))$. A "global" hypothesis on the behavior of **lh** is needed. Leq-monotonicity in 4.3 does the job. (5) When costs range on an infinite domain (the analogous of) theorem 5.2 fails even when the leq-monotonicity hypothesis is satisfied.

• Note that our plant is as general as it can be in supervisory control theory (e.g. see [18], [5]). In particular our OSC synthesis algorithm (5.1) handles arbitrary sets of final states as well as loops and uncontrollable events in the plant transition relation. No previously published OSC synthesis algorithm handles such general case.

• Note that our objective function only depends on costs for enabled events. Objective functions depending also on costs for disabled events have been considered in [21] for costs ranging on $\Re^{\geq 0}$, singleton sets of final states and plants without loops or uncontrollable events. □

6 Experimental Results

BDDs size strongly depends on the boolean functions that we need to represent (e.g. see [7]). Thus, as usual with BDDs, we need to run experiments to asses performances of our algorithm. We implemented (in C) a compiler for BFOL programs. In this section we report on experimental results using our compiler to synthesize OSCs using the symbolic algorithm in 5.1.

$$
\begin{aligned}
mgo(\mathcal{F}) = [\; bo(\mathbf{x},\mathbf{u},\mathbf{v}) &= (\exists \mathbf{x}'(p(\mathbf{x},\mathbf{u},\mathbf{x}') \wedge end(\mathbf{x}')) \wedge (\mathbf{v} = \mathbf{h}(\mathbf{x},\mathbf{u}))) \vee \exists \mathbf{x}'(p(\mathbf{x},\mathbf{u},\mathbf{x}') \wedge \neg end(\mathbf{x}') \wedge \\
&\qquad \exists \mathbf{v}_2(g(\mathbf{x}',\mathbf{v}_2) \wedge \exists \mathbf{v}_1((\mathbf{v}_1 = \mathbf{h}(\mathbf{x},\mathbf{u})) \wedge (\mathbf{v} = \mathbf{lh}(\mathbf{v}_1,\mathbf{v}_2))))), \\
adm(\mathbf{x},\mathbf{v}) &= \exists \mathbf{u} bo(\mathbf{x},\mathbf{u},\mathbf{v}) \wedge \forall \mathbf{u},\mathbf{v}'((bo(\mathbf{x},\mathbf{u},\mathbf{v}') \wedge \neg contr(\mathbf{u})) \rightarrow (\mathbf{v}' \leq \mathbf{v})), \\
g(\mathbf{x},\mathbf{v}) &= adm(\mathbf{x},\mathbf{v}) \wedge \forall \mathbf{v}'(adm(\mathbf{x},\mathbf{v}') \rightarrow (\mathbf{v} \leq \mathbf{v}')), \\
f(\mathbf{x},\mathbf{u}) &= \neg contr(\mathbf{u}) \vee \exists \mathbf{v}(bo(\mathbf{x},\mathbf{u},\mathbf{v}) \wedge \exists \mathbf{v}'(g(\mathbf{x},\mathbf{v}') \wedge (\mathbf{v} \leq \mathbf{v}'))\;].
\end{aligned}
$$

Figure 4:

Grid size	Cost bits (k in 4.1)	State bits (n in 4.1)	State Space Size	CPU (min)	Max BDD	OSC	C lines
2×2	3	10	1024	2:45	220,418	87	180
4×4	5	14	16384	249:21	1,400,032	339	900

Figure 5: Experimental Results on SUN Sparc Station 20 with 64MB of RAM

Our plant is formed by an $m \times m$ grid with two autonomous vehicles (AVs) moving on it. Each AV can stay where it is or can move forward, backward, left or right. Failures can occur.

Our goal is to find the OSC f s.t.: f satisfies given safety constraints, f drives each AV to a given grid region, f minimizes the fuel used to finish the job (both AVs have reached their destinations).

Fig. 5 reports our experimental results. We use 6 bits to represent events (i.e. r in 4.1 is 6). Column 'Max BDD' gives the size of the larger BDD built during the computation. Column 'OSC' gives the size of the BDD representing OSC f. From such BDD our tool can automatically generate a C program, say C(f), implementing OSC f. C(f) closely follows the BDD representation of f. Thus C(f) runs in time linear in the number of arguments of f, i.e. in this case O($n+r$). Column 'C lines' reports the number of lines of C(f).

If we assume for the running time of our OSC synthesis algorithm (5.1) an expression of form αs^{β}, where s is the size of the state space (i.e. $s = 2^n$) from the table in fig. 5 we get a running time of $35 \cdot 10^{-6} \cdot s^{1.63}$. Note that this is better than the running time of $O(s^2 \log s)$ for the algorithm in [21].

7 Conclusions

We addressed the problem (OSCP) of automatic synthesis of *Optimal Finite State Supervisory Controllers* (OSCs). Our results (summarized in sec. 1) show that, although OSC synthesis is computationally harder than SC synthesis or automatic verification (Model Checking), using BFOL programs and BDDs automatic synthesis of OSCs is possible for small size plants.

To design OSC synthesis algorithms for larger plants is 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] A. Anuchitanukul, Z. Manna, *Realizability and Synthesis of Reactive Modules*, CAV 94, LNCS 818, Springer-Verlag

[3] K. R. Apt, *Logic Programming*, Handbook of Theoretical Computer Science, Elsevier 1990

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

[8] L.C. Cooper, M. W. Cooper, *Introduction to Dynamic Programming*, Pergamon Press, 1981

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

[10] R. Kumar, V. Garg, S. I. Marcus, *Predicates and Predicate Transformers for Supervisory Control of Discrete Event Dynamical Systems*, IEEE Trans. on Automatic Control, Vol. 38, N.2, 1993

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

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

[13] M. Lemmon, J. A. Stiver, P. J. Antsaklis, *Event Identification and Intelligent Hybrid Control*, Hybrid Systems, LNCS 736, Springer-Verlag, 1993

[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 Conf. Decision and Control, 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, *Modular Feedback Logic for Discrete Event Systems*, SIAM J. Control and Optimization, Vol. 25, N. 5, pp. 1202-1218, 1987

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

[21] 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

[22] E. Tronci, *Hardware Verification, Boolean Logic Programming, Boolean Functional Programming*, LICS 1995, IEEE Computer Society, 1995

[23] M. Y. Vardi, *An Automata-Theoretic Approach to Fair Realizability and Synthesis*, CAV 95, LNCS 939, Springer-Verlag

[24] 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