Sequential Synthesis with Co-Büchi Specifications

Guoqiang Wang, Alan Mishchenko,
Robert Brayton, and Alberto Sangiovanni-Vincentelli

EECS Dept. University of California
Berkeley, California, 94720, USA
{geraldw, alanmi, brayton, alberto}@eecs.berkeley.edu

Abstract. *Computations are developed for the synthesis of a finite state machine (FSM) embedded in a known FSM such that their combined behavior satisfies a co-Büchi specification (the solution must finally enter an acceptable set of states and stay there forever). The procedures for this are shown to be very similar to those used for regular (non-omega) automata, except for a final step in which a set of FSM solutions is represented as a SAT instance of which each satisfying assignment corresponds to an FSM solution. The computations have been implemented and we discuss some results..*

Introduction

We consider sequential synthesis problems where the objective is to find a strategy, implementable as a finite state machine (FSM), which guides a system to a given subset of states (e.g. a winning state for a game, or a set of states with some desirable property), called the accepting states. Examples are games, control problems, protocol synthesis, etc. Such situations need omega automata to capture these specifications, since regular automata would require that the initial state is already an accepting state. The problems we consider are concerned with steering a system into an accepting set of states and then keeping it there. Such requirements are what can be expressed by co-Büchi automata.

We present a synthesis flow for co-Büchi specifications. The FSM synthesis problem for this is stated as: find the most general FSM X such that $F \bullet X \subseteq S$, where S is a co-Büchi automaton, F is a known FSM, and $\bullet$ represents the usual synchronous composition of two FSMs. The most general automaton solution is given by $X = \overline{F \bullet \overline{S}}$ where the outside complementation is usually non-deterministic [13]. Therefore, in general, Büchi and co-Büchi automata complementation would be required, which are super-exponential in complexity [8]. Instead, we aim for a slightly less general but more efficient solution and propose a synthesis flow, very similar to that used for regular (finite-word) automata. This uses a subset construction to obtain a deterministic Büchi over-approximation of an ND Büchi automaton. Therefore, the final complementation, done by simply complementing the acceptance conditions to obtain a co-Büchi automaton, yields a *subset*[1] of the most general solution automaton.

Until the last step, our flow does not use the co-Büchi acceptance condition for constructing the transition relations on automata structures; it merely keeps track of the resulting acceptance conditions. To derive the final FSM implementations, the acceptance condition is applied to trim the most general solution automaton by formulating a SAT [7], [8] instance, each of whose solutions corresponds to a particular FSM solution. The SAT instance contains clauses, which ensure the input-progressiveness property required for FSMs (i.e. for each input there must exist a next state and output response). Other clauses enforce the co-Büchi condition by requiring the elimination of all simple cycles that contain a non-accepting state. The SAT instance represents all FSM solutions that can be associated with sub-graphs of the automaton solution; solutions with non-simple cycles are not represented, but we argue that such solutions are impractical anyway.

To simplify the SAT instance, a graph pre-processing step derives a partial order based on input-progressiveness. In this, an edge is classified as *essential* if its removal causes a state to become non-progressive. Thus removing such an edge would imply that the corresponding state must be removed, recursively implying the removal of other states. The resulting smaller graph becomes the basis for the SAT formulation. The algorithm was implemented and we discuss some results on a few simple examples.

The contribution of this paper is a synthesis flow for co-Büchi specifications, which follows the flow for regular automata; hence it is simpler than for general omega-automata (ω-automata) and can make use of recent efficient algorithms for regular automata [9]. Only in a final step, which extracts an FSM implementation using a SAT formulation, does the flow differ substantially from that for regular automata specifications.

The paper is structured with Section 1 giving some preliminaries on ω-automata. The topology used for the unknown component problem is presented in Section 2. The proposed ω-property synthesis techniques are addressed in Section 3, including the SAT formulation. The solutions computed for two representative example problems are discussed in Section 4. Section 5 discusses the complexity of complementing non-deterministic Büchi automata in general and contrasts this with the construction in the present paper. Section 6 concludes. Appendix A considers synthesizing to Büchi specifications.

1 Preliminaries - ω-Automata

An Ω-automaton is a finite state automaton that accepts infinite strings [2], [3], [4], [5]. Although there are many different types, here we discuss only Büchi, looping, co-Büchi, co-looping and Muller automata here.

A non-deterministic (ND) **Büchi** automaton has the following form: $M = (Q, \Sigma, q_0, \Delta, Acc)$, where Q is the finite state space, Σ is the finite input alphabet, $q_0 \in Q$ is the initial state, $\Delta \subseteq Q \times \Sigma \times Q$ is the transition relation, $Acc \subseteq Q$ represents the acceptance condition. A run of M on the input word $\alpha \in \Sigma^{\varpi}$, $q(\alpha)$, is successful if it starts at the initial state and the set of states that occur infinitely often intersects Acc. For a **Muller** automaton, $Acc \subseteq 2^Q$ and a run is successful if it

[1] An important subclass of co-Büchi automata is "co-looping" automata. For this class of specifications, our procedure is exact and thus obtains the most general solution automaton.

starts at the initial state and the set of states which appear infinitely often is a member of *Acc*.

A **co-Büchi** automaton has also a single set (stable region) in its acceptance condition; but the meaning is that a run should eventually enter the stable region and stay there forever. It is a Muller-type automaton where the Muller acceptance condition consists of all subsets of the states in the stable region. Deterministic Büchi and co-Büchi automata are limited in the set of properties that can be expressed, while deterministic Muller automata, ND Büchi, and ND co-Büchi automata can express any ϖ-regular property. For an ND Büchi automaton with acceptance condition *Acc*, an input sequence is accepted if **there exists** a run that intersects *Acc* infinitely often.

A **co-looping** automaton is a co-Büchi automaton with the additional restriction that the set of final states (stable region) must be a sink, i.e. there is no edge from any final state to a non-final state. A **looping** automaton is the dual of a co-looping automaton; its non-final states are a sink. Thus an accepting run is one that always avoids a non-final state. Looping automata are useful for expressing safety properties. Looping and co-looping automata have the property that they can be determinized by the subset construction [11], which is simpler that for the general case.

Thus the difference between co-looping and co-Büchi is that the latter can have a final set from which it is possible to exit. It seems possible that in many cases with a general co-Büchi specification, the synthesis problem can be divided into two phases, the problem of steering the state of the system into a state of the final set (a co-looping problem), and the problem of keeping it there (a looping problem). These could be solved separately. We will see that the procedure used for finding the most general solution with a co-looping specification is exact, while for the general co-Büchi case, it is an approximation.

2 Problem Statement

The synthesis problems considered have a topology as shown in Figure 1.[2] An unknown component, *X*, is embedded in a larger known system, *F*, where the i/o behavior of the combined system, $X \bullet F$, should satisfy a given external specification *S*. The components communicate synchronously via the channels labeled with the (multi-valued) variables, *i, v, u, o*. This kind of synthesis problem has been studied extensively when *S* is either a regular finite automaton or an FSM [13]. In particular, efficient procedures and a program have been implemented for computing the most general solution automaton and the most general FSM solution [9]. In this paper, we investigate the situation where *S* is a co-Büchi automaton.

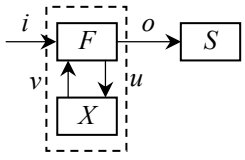

Figure 1. Topological Setup

Let *S* be a co-Büchi automaton with multi-valued input signal *o* whose values are taken from the alphabet Σ_o. *S* is represented by $S = (Q_S, \Sigma_o, q_0^S, \Delta_S, A)$, where *A* is the stable set. The fixed part *F* (or context) is assumed to be an FSM with multi-valued inputs *i* and *v* and multi-valued outputs *o* and *u*. *F* (although an FSM) can be interpreted as a special deterministic Büchi automaton, represented by $F = (Q_F, \Sigma_{i\times u\times v\times o}, q_0^F, \Delta_F, B)$, where the accepting set *B* is the set of *all* states. *X* is the unknown component, which is required to be a deterministic FSM, since in most applications, we are interested in obtaining an implementable function.

In this topology, *X* only sees variable *u* and outputs variable *v,* so it is not able to observe variables *i* and *o* directly. The objective is to find an FSM implementation of *X* such that its synchronous composition with *F* satisfies the co-Büchi specification *S*. Solutions are obtained by solving a corresponding Ω-language containment problem: find *X* such that $F \bullet X \subseteq S$. The most general solution (not necessarily an FSM) is known to be given by $X = \overline{F \bullet \overline{S}}$. This is explained further in Section 3, along with the details of our synthesis approach.

3 Overview of the Synthesis Flow

An overview of the general synthesis flow is outlined in Figure 2, where each variable is an ω-automaton. All operations are done on Büchi or co-Büchi automata and are explained below in more detail.

1. ComplementOmega(*S*); (complement the co-Büchi automaton *S*)
2. Complete(*F*); (complete the FSM *F* as a Büchi automaton)
3. ProductOmega($P = F\overline{S}$); (Compute the product *P* of *F* and $\overline{S}$ of two Büchi automata)
4. Hide variables in *P* invisible to *X*, i.e. *i* and *o*;
5. Pseudo-determinizeOmega(*P*); (determinize the Büchi automaton *P*)
6. ComplementOmega(*P*); (dualize the acceptance conditions to produce a co-Buchi automaton)
7. Restrict to FSM solutions; (final processing to produce FSM sub-solutions)

Figure 2. Computation of $X = \overline{F \bullet \overline{S}}$ and FSM sub-solution.

The first six steps compute a general automaton solution while the last step constrains it to only FSM solutions. We have modified the general flow to avoid complementing (possibly non-deterministic) Büchi (*P*) automata, as discussed below.[3]

4.1 Computing a General Automaton Solution

We summarize the modifications of the first six steps and argue that they can be done in the same way as for regular automata with only slight modifications to keep track of the Büchi type acceptance conditions. To emphasize this difference, we have attached the term "Omega" to the name of the

[2] The particular topology of communication is not important. Our results can be adapted easily to other topologies.

[3] The general procedure would have combined Steps 5 and 6 into a single step, "complementOmega(P)".

regular transformation. All transition relations computed are the same as for the regular case. As such these operations can be handled efficiently by recent developments and implementations [9]. The co-Büchi/Büchi acceptance conditions are simply carried along and analyzed in the computations below but they do not affect the manipulations of the transition relations. The seventh step (Sections 4.2 and 4.3) applies the acceptance condition to extract a set of particular FSM solutions.

1. Complementing the Specification *S*. We assume that *S* is a deterministic co-Büchi automaton with final states *A*. It can be complemented by simply inverting its acceptance condition. Thus, $\overline{S}$ is a deterministic Büchi automaton and a run of $\overline{S}$ is accepted if it intersects $\overline{A} \equiv Q_S \setminus A$ infinitely often.

2. Completing the Fixed Part *F*. *F* is an FSM and can be interpreted as a special Büchi automaton; its accepting set of states is the set *B* of all states. Since *F,* as an automaton is incomplete, it can be completed by adding a single new state n_F, which is the only non-accepting state (it is added as a state with no-exit and a universal self-loop – a "don't care" state). For convenience, we denote the completed automaton also by *F*.

3. Creating the Product $P = F \bullet \overline{S}$. Since both *F* and $\overline{S}$ are Büchi automata, their product is conventionally done by introducing a flag as a third entry in the product state to indicate whenever an acceptance condition is met in each of the two operand Büchi automata. This is required since a product is accepting only if the acceptance condition of each of its components has been met; we need to visit *B* infinitely often as well as $\overline{A}$. In general, the flag is used to ensure that we visit infinitely often both product states of type $\{(s,t)\}$ where *s* is in *B,* as well as those of type $\{(q,r)\}$ where *r* is in $\overline{A}$. We will remove the need for this flag by using the fact that all states of *F*, except the don't care state, n_F, are accepting. The flag toggles once we have visited *B* and again once we have visited $\overline{A}$. Suppose that $\overline{A}$ has just been visited, so the current state (s,t) has $t \in \overline{A}$. There are two cases. If $s \in B$, then we have just visited *B* also, so the flag does not need to be toggled. The other case is where $s = n_F$. Since n_F is a don't care state, we can never exit from it. Thus all subsequent states of the product machine will be $(n_F, -)$. All such states are part of the non-accepting Büchi states of *P* and can never enter the accepting region. Thus, we don't need to toggle the flag, since nothing important will happen after this. Hence the product automaton $P = F \bullet \overline{S}$ is obtained by taking the regular product of the two operand automata (no extra entry in the product state is needed) to obtain the transition structure of the Büchi automaton, $P = (Q_P, \Sigma_{i,v,u,o}, q_0^P, \Delta_P, \overline{C})$. To determine $\overline{C}$, we note that *P* has the following types of states: $(b,a), (b,\overline{a}), (n_F,a), (n_F,\overline{a})$, where $a \in A$, $\overline{a} \in \overline{A}$, and $b \in B$. Thus $\overline{C} = \{(b,\overline{a})\}$ and a run of *P* is accepting if and only if it visits states of type $(b,\overline{a})$ infinitely often.

4. Hiding Variables Invisible to *X*. Hiding the variables *i* and *o* that are invisible to the unknown component *X* is simply the regular procedure of erasing such labels on the state transitions. Even though *P* is deterministic, the result $P_{\downarrow(u,v)}$ can be non-deterministic. The notation $\downarrow_{(u,v)}$ represents the normal projection operation, i.e. keeping the variables *u,v* on the transitions and erasing (existentially quantifying) everything else. The acceptance conditions are not changed.

5. Pseudo-Determinizing $P_{\downarrow(u,v)}$. Since $P_{\downarrow(u,v)}$ is a ND Büchi automaton, it can't be determinized in general (deterministic Büchi automata are not as expressive). On the other hand, complementing it is a super-exponential procedure, $2^{O(n \log n)}$ (see Section 5), which should be avoided if possible. So as a heuristic approximation[4], we apply the subset construction to the transition structure of $P_{\downarrow(u,v)}$ to obtain a Büchi automaton $\tilde{P}$, whose language *contains* that of $P_{\downarrow(u,v)}$. The final states of $\overline{C}$, which are subset states, are obtained as follows: when a subset is reached it is marked as in $\overline{C}$ if the subset contains a state of type $(b,\overline{a})$.

6. Complementing $\tilde{P}$. Since $\tilde{P}$ is deterministic, $\overline{\tilde{P}}$ can be obtained by duality, by inverting its acceptance condition; thus it keeps the same transition structure, but we interpret the result as a co-Büchi automaton with final states *C*. In general, $\overline{\tilde{P}}$ will be an under-approximation to the most general solution automaton $\overline{P_{\downarrow(u,v)}}$.

Observations. All the computations above involve only ones that are in the "normal" flow for regular automata (in terms of computing the transition relations) and which have been implemented in a program for solving problems with regular automata. The final accepting set *C* is derived as a kind of side computation as discussed above. In some cases, where the fixed part *F* is given as a logic network, the computations of the transition structures can be very efficient [9]. The implementation involves the operations of *completion*, *hiding*, *product* and *determinization*. We emphasize that so far, nothing special has been done in Section 4.1 that is associated with computing with Büchi automata except for the side computations of determining the acceptance states. Even the determinization step when deciding which of the subset states are to be put in the Büchi final set, $\overline{C}$, is a typical operation in which each subset state is classified as soon as it is generated. The only difference in the transition structures for the Büchi case will come when the interpretation of *C* (i.e. what it means for *C* to be the accepting set) is used to construct FSM solutions. This is done in the next subsection, where special non-regular methods are formulated to trim $\overline{\tilde{P}}$ to obtain FSM solutions to meet the co-Büchi condition *C*. We emphasize that all the efficient implementations done in a system for computing with regular automata can be modified easily.

Another observation is that for specifications, which are co-looping automata, the determinization step in this section is exact, i.e. $\overline{\tilde{P}} = \overline{P_{\downarrow(u,v)}}$. This follows from the fact that looping

[4] See the last paragraph of this section for a discussion of when this is not an approximation.

automata can be determinized [11]. Hence for this case, we obtain the most general solution automaton.

4.2 Computing Particular FSM Solutions *X*' - Applying the co-Büchi acceptance condition.

To obtain particular FSM implementations for the unknown component, we will generate all sub-graphs of *X,* where any loop that contains a non-stable state has been eliminated, leaving only acyclic paths from the initial state to *C*. The most difficult part is to do this while maintaining input-progressiveness of the solutions.[5] We observe that, in general, some solutions will be lost, since only sub-graphs are derived, and thus, for example, state duplication is not allowed. Therefore, solutions which circulate around a loop a finite number of times before leaving the loop would not be captured[6]. In addition, as mentioned already, in the general case, we may have lost (in case S in not co-looping) some solutions by using the subset construction to determinize $P_{\downarrow(u,v)}$.

SAT Formulation. We will focus on trimming the deterministic co-Büchi solution $\overline{\tilde{P}}$ so that the only cycles left are those entirely contained in the stable set *C*. This requires removing transitions (edges) in the graph of $\overline{\tilde{P}}$ making the non-stable part acyclic but still maintaining *u*-progressiveness (*u* is the only input for the unknown component shown in Figure 1.).

This is formulated as a SAT instance where each satisfying assignment will correspond to a way of trimming the graph to give it the desired property. For each transition, we associate a binary variable e_{jk}, which is 1 if the transition is chosen to remain. For each state *j*, the variable s_j is 1 if *j* is chosen to remain.

Let E_{ju} be the set of edges that may be traversed on input *u* in one step from *j*. $E_{ju} = e_{j1} + ... + e_{jn}$, where *n* is the cardinality of E_{ju}. The *u-progressiveness clause*, $C^u_j = E_{ju}$, says that for input *u*, there exists at least one next state. Thus the *u*-progressiveness of State *j* is $C_j = (s_j \Rightarrow \prod_u E_{ju})$, which says that if State *j* is selected, then it must be *u*-progressive, meaning that for each minterm of *u*, there exists a next state. A second type of clause, *connection clause*, says that if edge e_{ij} is selected, then both terminal states have to be selected, i.e. $C^i_{ij} = (e_{ij} \Rightarrow s_i)(e_{ij} \Rightarrow s_j)$. Finally, to eliminate every simple loop not entirely contained in the stable set, a third type of clause, *loop-breaking clause*, is constructed, one for each such loop. Suppose $L = \{e_{12}, e_{23}, e_{34}, ..., e_{l1}\}$ is such a loop. Its clause should say that at least one of these transitions should not be chosen. This is equivalent to $C_L = \overline{e_{12}e_{23}e_{34}...e_{l1}}$.

We must also require that the initial state s_0 be selected. Thus $C_0 = s_0$, i.e. $s_0 = 1$, is added to the clauses.

Since all simple unstable loops must be enumerated, there could be many such loops. To alleviate this problem, the graph is pre-processed to eliminate certain obvious transitions, using the notion of *essential* edges. This is described in Section 4.3. Hopefully, this reduction will cut down the number of loops considerably.

Theorem 1. *Any solution of our SAT instance is an FSM solution of the ϖ-language synthesis problem.*

Proof:

Suppose we have a solution of the SAT instance. The solution is a selection of a subset of states and transitions of $\overline{\tilde{P}}$. This corresponds to a sub-graph of the most general solution *X* where every state is *u*-progressive and in the graph there is no loop not entirely contained in *C*. Being *u*-progressive means that the graph represents a pseudo-non-deterministic FSM (and hence might contain several deterministic solutions). Being a sub-graph of a general solution automaton with the required properties, it is a solution of the synthesis problem, and hence all its deterministic sub-machines are solutions. QED

A SAT solver can be configured to enumerate all possible satisfying assignments. Hence the SAT instance formulated represents a **set** of FSM solutions. However, all FSM solutions may not be represented, e.g. those where a non-simple loop is traversed a finite number of times before it is exited. An associated FSM would require enough extra states to count effectively the number of times it has gone around a particular loop. In this sense, such solutions might not be of interest from a practical point of view. On the other hand, it is possible that our SAT instance is not satisfiable, but still there may exist an FSM solution. This lack could be remedied by first duplicating one or more states and then formulating a new SAT instance which is satisfiable. Finally, as noted previously, the subset determinization step in Section 4.1, Step 5, may cause some FSM solutions to be lost.

4.3 Pre-processing to Simplify the SAT Instance.

To reduce the size of the SAT instance, a preprocessing step which trims away some of the states and transitions of $\overline{\tilde{P}}$ can be done. In some cases, after this step, it is possible that no SAT solving would be needed.

Trimming the Acceptance Set. $\overline{\tilde{P}}$ is a co-Büchi automaton with accepting set *C*. We create an *acceptance* automaton as follows. A nominal initial state is created where its outgoing transitions are all the transitions from $\overline{C}$ to *C*, (the labels on these edges are irrelevant) and all states of $\overline{C}$ are eliminated. Thus all transitions from *C* to $\overline{C}$ have been eliminated. This automaton is processed in the regular way [9], which trims away some states and transitions in *C*, to make it *u*-progressive. If the result is empty, we output that no solution exists and stop, since this means that there can be no cycles entirely contained in *C*.

At this point, we modify $\overline{\tilde{P}}$ by merging all remaining nodes of *C* into a sink node *f*, having a single universal self loop. Incoming edges to *f* are only those which lead to the remaining nodes of *C*; other edges are removed. We obtain a so-

[5] Algorithms for finding minimum feedback-arc sets in directed graphs exist [1], but do not deal with input progressiveness.

[6] It might be argued that such solutions are not practical since they simply delay getting to the stable set.

called *path* (co-looping) automaton X_{path}, derived thus from $\overline{\tilde{P}}$, which has only one nominal stable state *f*.

Pre-processing the Path Automaton.

For each state in X_{path}, outgoing transitions are classified as essential or not. An *essential* edge is one that if eliminated would make that state not *u*-progressive.

1. Restrict X_{path} to only its essential transitions and their corresponding states. If this graph has a loop (of essential edges), then all states of the connected component containing the loop must be eliminated. This is because to make X_{path} acyclic at least one transition in the loop must be eliminated, requiring a corresponding state to be eliminated, which causes other transitions and states to be eliminated until the entire connected component is gone. After this, only those connected components, which have no loops of essential transitions, remain. If no states are left in the path automaton, then there is definitely no solution.
2. There may be non-essential transitions that must be eliminated because they lead to eliminated states. This can create new essential transitions (secondary essential transitions). This procedure is repeated until no further eliminations can be done.
3. Of the remaining nodes, the essential edges define a partial order of states; for each totally ordered subset of states, all backward (non-essential) edges within this subset must be eliminated because this is the only way to break such loops while still ensuring input-progressiveness. This could create additional essential transitions (tertiary essential transitions).
4. The above three steps are repeated with all the newly created essential transitions added until no further eliminations are possible. This fixed point can be considered the complex core of the problem for which the SAT instance is formulated.

After deriving any particular solution corresponding to this reduced path automaton, it is straight-forward to combine it with the solution of the acceptance automaton to get a corresponding particular solution for the original unknown component problem.

Discussion. We cannot just set the value associated with all essential transitions to 1 in the SAT clauses since one of their states may not always be in a final solution. However, knowledge of essential edges can help in satisfying the loop-breaking clauses. For example, after the preprocessing step, any simple cycle must contain at least one non-essential transition. It turns out that only a non-essential edge of any loop needs to be eliminated in order to break that loop; otherwise, assume a loop is broken by eliminating an essential edge. This implies that the source node of this edge must be eliminated. Then all edges that lead to this node must be eliminated. If any of these is an essential edge on the loop, then its source node must be eliminated. Eventually, we eliminate a node in the loop whose incoming edge on the loop is non-essential. Thus this loop could have been broken by simply eliminating that non-essential edge initially.

4 Examples

The above presented synthesis approach has interesting applications in the controller synthesis area. For example, it can nicely handle applications like the Guideway scheduling synthesis problem discussed in [12]. For ease of illustration, we discuss two small examples, the "Wolf-Goat-Cabbage" problem and an aircraft control problem, and discuss their solutions.

Wolfe-Goat-Cabbage. The problem is a puzzle: find a strategy to transport by boat, from one side of a river to the other, all three (wolf, goat, cabbage) without having one of them eat another in the process (wolf eats goat, goat eats cabbage). The boat can hold only one of the three at one time, thus leaving the other two on a shore. We must not leave (wolf, goat) or (goat, cabbage) on the same shore unsupervised. *X* represents a transportation strategy (to be found). Since there is no input to *X*, essentiality is easy in this case; if there is only one outgoing transition associated with a state, the transition is essential; if there is more than one outgoing edge, all such edges are non-essential. The fixed part *F* is a finite state machine that describes these dynamics with an *output* = 1 if the machine enters a bad state, e.g. the wolfe and goat are left together on the same shore.

The co-Büchi specification has two states, *a* and *c*. The initial state is *a*; the stable region consists only of State *c*, which is a sink, so the specification is a co-looping automaton. A transition is made from *a* to *c* if *output* = 1.

The most general automaton solution was computed and minimized. It is a co-Büchi (actually co-looping) automaton, with only one target state in the stable set. To find a particular FSM implementation, the most general solution was split into a path and an acceptance automaton. The path automaton had 12 cycles. Some states in it cannot reach the target state and thus were removed. We also removed cycles consisting of only essential transitions; and also removed any backward non-essential edges. This led to the removal of one state and nine transitions. The resulting core structure, had 29 variables and 57 clauses of the SAT instance, among which 9 clauses are for cycles. This SAT instance was satisfiable and produced two different solutions.

Aircraft Control. The problem is to keep an airplane within a specified range of altitudes above ground, e.g. between 1 and 2 units of elevation above ground. The land elevation varies and there is a random up or down air draft causing the aircraft's rate of climb to alter. The fixed automaton, *F*, describes the dynamics of the flight in terms of the planes position, elevation and rate of climb. It has 3 inputs, 1 for the random air draft input and 2 for the auto-pilot control. The auto-pilot has two controls to adjust the rate of climb, one can increase or decrease it by 1 or 2 units and the other is a binary boost which doubles the effect of the first input. The airplane takes off at the ground elevation, so it is not initially in the accepting set of states. The up or down draft of air is limited to affect the rate of climb by at most one unit. For simplicity, there are 16 ground positions with different elevations. The plane flies in a loop over these positions. The specification automaton has two states, *a* and *c* of which *a* is the initial state and *c* is the only accepting state. Its input, called *out*, comes from the fixed part and simply indicates if the elevation is OK or not. There is a transition from *a* to *c* is *out* = OK and from *c* to *a* if *out* = notOK. This specification is clearly co-Büchi, but not co-looping since it is possible to exit the accepting set of states. Some statistics of the solution automaton were:

Fixed automaton: 7 inputs 752 states 3024 transitions.
Determinized product automaton: 5 inputs 1115 states 5643 trans.
Most general solution automaton made prefix closed and input progressive: 5 inputs 1115 states 5643 trans.
State minimized automaton: 5 inputs 13 states 140 trans.

5 Complementing Büchi Automata

After the construction of the product of two Büchi automata and the hiding of some variables (*i* and *o*) the ND Büchi automaton $P_{\downarrow u,v}$ is obtained. The last step to find the most general solution automaton solution would be to complement this. There has been much progress in complementing ND Büchi automata (see [8] for a good review and the most recent construction). A tight lower bound on the number of states in the complement Büchi automaton is $2^{O(n \log n)}$ where *n* is the number of states in the original Büchi automaton. An upper bound of $(1.06n)^n$ [8] in the number of states has been obtained recently. In general, it is known that subset constructions do not work for determinizing co-Büchi (Büchi) automata but do work for co-looping (looping) automata. The subset construction is upper bounded by 2^n in the number of states. Thus the procedure in this paper is much less expensive in complexity than the general procedure. In addition, experience with the subset construction shows that on a number of practical problems, its behavior is surprisingly well-behaved, in some cases resulting in a reduced number of states. However, the price that we pay for using this is that only a subset of the most general solution is obtained (unless of course the original specification is co-looping).

6 Conclusions

A computational flow to synthesize an FSM according to co-Büchi specifications was derived. In general, the method is sound but incomplete since we chose for efficiency to determinize a ND Büchi automaton in the inner loop instead of complementing it. This may exclude some solutions (for co-looping specifications it is complete). The steps we proposed to compute a general solution automaton are the same as those used in regular finite-word automata synthesis, except that in last step, we derive a particular solution by formulating and solving a corresponding SAT problem. The procedure was implemented and a few simple examples were discussed to illustrate its use.

We have discussed the case where the specification is deterministic. If it is non-deterministic, it seems expeditious to complement it using the algorithms for complementation discussed in Section 5. Although these are super-exponential in complexity, the number of states in the specification usually can be kept small by putting the details in the fixed part *F*.

Synthesizing to Büchi specifications (useful for liveness properties) is discussed in Appendix A. If the specification is a looping automaton, then a procedure, derived from ours by interchanging the words Büchi and co-Büchi, is sound and complete. Otherwise, the specification can be complemented to a ND Büchi automaton after which the procedures of this paper can be followed exactly.

References

[1] C. Demetrescu and I. Finocchi, "Combinational Algorithms for Feedback Problems in Directed Graphs," Information Processing Letters, vol. 86, no. 3, pp. 129-136, May 2003.

[2] T. Henzinger, "EE219B: Lecture Notes for Computer Aided Verification," Spring 2003.

[3] S. Gurumurthy, O. Kupferman, F. Somenzi, and M. Vardi, "On Complementing Nondeterministic Buchi Automaton," CHARME'03.

[4] M. Roggenbach, "Determinization of Büchi Automata," in *Automata, Logics, and Infinite Games*, pp. 43–60, 2002.

[5] H. Jain, "Automata on Infinite Objects," B. Tech. Seminar Report., Indian Institute of Technology, April 2002.

[6] J. P. Marques-Silva and K. A. Sakallah, "GRASP: A Search Algorithm for Propositional Satisfiability," *IEEE Transactions on Computers*, vol. 48, no. 5, pp. 506–521, 1999.

[7] N. Een and N. Sorensson, "An Extensible SAT-solver," in *SAT 2003*.

[8] E. Friedgut, O. Kupferman, and M. Vardi, "Büchi Complementation Made Tighter, 2nd International Symposium on Automated Technology for Verification and Analysis, Lecture Notes in Computer Science, 2004.

[9] Reference omitted.

[10] R. P. Kurshan, "Complementing Deterministic Büchi Automata in Polynomial Time", Journal of Computer and System Sciences, 35:59-71, 1987.

[11] O. Kupferman and M.Y. Vardi, "On Bounded Specifications", In Logic for Programming, AI and Reasoning, 2001.

[12] P. J. Ramadge, W. M. Wonham, "The Control of Discrete Event Systems", Proceedings of the IEEE, vol. 77 No. 1, 1989.

[13] Reference omitted.

Appendix A – Synthesizing to Büchi Specifications

Büchi specifications provide an effective way for specifying liveness properties, i.e. always eventually something good happens. The overall flow for using Büchi specifications might seem to be similar to that already discussed for co-Büchi specifications, just by interchanging "Büchi" and "co-Büchi". However, in synthesizing to co-Büchi, we had to determinize a Büchi product (Step 5) and thus for synthesizing to Büchi, we would have to determinize a co-Büchi product. Unfortunately, the subset construction for co-Büchi produces a smaller language and so the procedure is not sound (since its complement produces a larger language).

If the specification is a looping automaton, then the basic procedure is correct. However this restriction to safety properties is not satisfactory, since the reason for considering Büchi was its ability to express liveness.

For a general deterministic Büchi specification, *S*, a sound approach would be to compute its complement using the procedure in [10]. This produces a Büchi complement, so the product and the rest of the operations are now the same as if we started out with a co-Büchi specification. In addition to this complementation procedure being linear, by adding to the description of the fixed part *F*, often the specification can be made into a simple monitor and described with only a few states. After this, the flow is the same as described for the co-Büchi case, since $\overline{S}$ is obtained as a Büchi automaton using the procedure of [10], and therefore $\overline{\tilde{P}}$ will be a co-Büchi automaton.