Synthesizing FSMs According to Co-Büchi Properties

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 an FSM embedded in a known larger system such that the overall behavior satisfies a co-Büchi specification. The procedures for this are very similar to those used for regular (non-omega) automata, except for a special final step in which a set of FSM solutions is represented as a SAT instance. Each satisfying assignment corresponds to an FSM solution. To reduce the SAT size, a preprocessing step splits a general solution automaton into a "path" automaton and an "acceptance" automaton. Cycles in the path automaton graph are trimmed while maintaining the input-progressiveness property required for FSMs. Not all FSM solutions are represented by the SAT instance, since in theory there could be an infinite number. The computations have been implemented in the MVSIS environment and a few experiments have been done.

1 Introduction

For some applications, the objective is to find a strategy, implementable as a finite state machine, which guides a system to a given subset of states (e.g. a winning state for a game), called the accepting states. Some examples are games and some control problems. Such a situation cannot be captured by a regular automaton specification since the requirement for an FSM solution is that its language is prefix-closed implying that the initial state is accepting. ϖ-type specifications seem to be needed to describe such stronger final constraints. The motivation for considering co-Büchi specification will be clear in Section 2.

We present a proposed synthesis flow for co-Büchi specifications. The FSM synthesis problem is stated as follows: 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 synchronous composition. The most general automaton solution is given by $X = \overline{F \bullet \overline{S}}$ where the outside complementation is usually non-deterministic [9]. Thus Büchi and co-Büchi automata complementation are required, which are in general super-exponential in complexity [8]. Instead, we aim for a less general 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. The final complementation is done by simply complement-

ing the acceptance conditions to obtain a co-Büchi automaton, which is a subset of the most general solution automaton. An important subclass of co-Büchi automata is "co-looping" automata. For this class of specifications, our procedure obtains the most general solution automaton.

To derive the final FSM implementations, the co-Büchi acceptance condition is applied to trim the solution automaton by formulating a SAT [7], [8] instance, all of whose solutions correspond to particular FSM solutions. 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-final state. The SAT instance represents all FSM solutions that can be associated with sub-graphs of the automaton solution.

To help simplify the SAT instance, we use a graph pre-processing step to find a partial order based on input-progressiveness. An edge is classified as *essential* if its removal causes a state to become non-progressive. Thus this edge removal implies the removal of the corresponding state. This implies that all the states contained in any loop of only essential edges must be removed, implying the removal of other states. This pre-processing is applied until no further removals are possible; the resulting smaller graph becomes the basis for the SAT formulation. The algorithm was implemented in the MVSIS [6] environment and applied to a few examples; the only additional procedure required beyond that used for regular automata specifications was the SAT formulation and solution.

The contribution of this paper is a synthesis flow for co-Büchi specifications, which follows the exact flow for regular automata [9], and hence is simpler than for general ϖ-automata; the most complex part is a subset construction. Only in a final step, which extracts an FSM implementation, does the flow differ from that for regular automata specifications.

The paper is structured with Section 2 reviewing some preliminaries. The topology used for the unknown component problem is presented in Section 3. The proposed ϖ-property synthesis techniques are addressed in Section 4. The solution computed for a representative example is illustrated in Section 5 and the corresponding automata are shown in Appendix B. Section 6 discusses the complexity of complementing non-deterministic Büchi automata in general and contrasts this with the construction in the present paper. In Section 7, conclusions are discussed. Appendix A considers synthesizing to Büchi specifications and discusses a modification of the procedure of the present paper to make it sound for this case.

2 Preliminaries

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

A 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 $a \in \Sigma^{\nu}$, $q(a)$, 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 starts at the initial state and the set of states which appear infinitely often is a member of Acc.

Like Büchi, a **co-Büchi** automaton has also a single set (stable region) in its acceptance condition; but it 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 final states (stable region) must be a sink, i.e. there is no edge from a 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. 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].

3 Problem Statement

In this paper, we consider a synthesis problem whose topology is shown in Fig. 1. The setting is that of an unknown component, X, embedded in a larger known system where the behavior of the combined system should satisfy some external specification S. The components communicate synchronously via the channels labeled with the (multi-valued) variables, i, v, u, o. The particular topology of the communication is not really critical for the results of this paper. The synthesis problem has been studied extensively when the specification is a regular finite automaton or an FSM, and a system for efficiently computing the most general solution automaton or the most general FSM solution has been implemented [9]. In this paper, we investigate the situation where S is an ϖ-automaton.

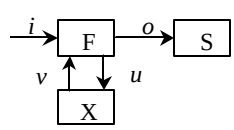

Fig. 1. Topological Setup

We consider the case where the ϖ-specification, S, is 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 an FSM with multi-valued inputs i and v and multi-valued outputs o and u. Here F

is 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 B is the set of all states. X is the unknown component, which is to be implemented as an FSM.

In this topology, the unknown component only sees variables u and v; variables i and o are hidden from it. 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 the corresponding ϖ-language containment problem, $F \bullet X \subseteq S$. The most general solution is given by $X = \overline{F \bullet \overline{S}}$. This is explained further in Section 4, along with the details of our synthesis approach.

4 Overview of the Synthesis Flow

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

```
Computation of X = F • S̄̄
    Complement S;
    Complete F as an automaton;
    Compute the product P of F and S̄;
    Hide variables invisible to X, e.g. i and o;
    Complement P;
    Restrict to FSM solutions;
```

Fig. 2. High-level algorithm for synthesis of $X = \overline{F \bullet \overline{S}}$

The first five steps compute the most general automaton solution (Section 4.1) while the last step specializes it to a large set of FSM solutions (Section 4.2). We modify this flow by avoiding complementing Büchi and co-Büchi automata, as discussed below.

4.1 Computing a General Automaton Solution

In this section, we first compute a general ϖ-automaton solution. We follow the first five steps. A sixth step is done as part of extracting particular FSM solutions.

Complementing the Specification *S*. We assume that S is a deterministic co-Büchi automaton with final states A. It is 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.

Completing the Fixed Part F. F is an FSM and can be interpreted as a special Bü chi automaton; it has a set of states B all of which are accepting. Since F as an automaton is incomplete, it is completed by adding a single new state n_F, which is the only non-accepting state (it is 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.

Creating the Product $P = F \bullet \overline{S}$. Since both F and $\overline{S}$ are Bü chi 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 operand Bü chi automaton. 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. In general, the flag is used to ensure that we visit both product states $\{(s,t)\}$ in which s is in B infinitely often as well as product states $\{(q,r)\}$ in which r is in $\overline{A}$ infinitely often. 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 it. Thus all subsequent states of the product machine will be $(n_F, -)$. All such states are part of the non-accepting Bü chi 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 to obtain the transition structure of the Bü chi automaton $P = (Q_P, \Sigma_{i,v,u,o}, q_0^P, \Delta_P, \overline{C})$. To determine $\overline{C}$, 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 is accepting if and only if it visits states of type $(b,\overline{a})$ infinitely often.

Hiding Variables Invisible to X. Hiding variables i and o that are invisible to the unknown component X is simply the normal procedure of erasing such labels on the 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.

Determinizing $P_{\downarrow(u,v)}$. Since $P_{\downarrow(u,v)}$ is a ND Bü chi automaton it can't be determinized in general. On the other hand, complementing it is a super-exponential procedure, $2^{O(n \log n)}$ (see Section 6), which should be avoided if possible. We apply the subset construction to the transition structure of $P_{\downarrow(u,v)}$ to obtain a Bü chi automaton $\tilde{P}$, whose language contains that of $P_{\downarrow(u,v)}$. The final states of $\overline{C}$ are obtained as follows. When a subset state is reached it is put in $\overline{C}$ if it contains a state of type $(b,\overline{a})$.

Complementing $\tilde{P}$. $\overline{\tilde{P}}$ can be obtained by duality, by inverting its acceptance condition; thus keeping the same transition structure, but interpreting 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 used and implemented in MVSIS for solving problems with regular automata. In particular, one can make use of a partitioned transition structure where the fixed part F is given as a multi-level circuit [9]. These computations are *completion*, *hiding*, *product* and *determinization*. Nothing special has been done in Section 4.1 that is associated with computing with Büchi automata. 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 is in the interpretation of the meaning of C when it is used to construct FSM solutions. In the next subsection, special non-regular methods are formulated to trim $\overline{\tilde{P}}$ to obtain FSM solutions to meet the co-Büchi condition C. Thus all efficient implementations done in MVSIS for computing with regular automata can be used.

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 automata can be determinized [11]. Hence for this case, we obtain the most general solution automaton. Note that the result is the same as if we used finite word automata and made all states accepting.

4.2 Computing Particular FSM Solutions X'

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.[1] Note that, in general, we may lose some solutions, since only sub-graphs are derived, while state duplication is not allowed. Thus, for example, solutions which circulate around a loop a finite number of times before leaving the loop are not considered. In addition, we have lost some solutions by the determinization of $\overline{P_{\downarrow(u,v)}}$.

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

[1] Although algorithms for finding minimum feedback-arc sets in directed graphs are available in the literature [1], they do not deal with input progressiveness.

acyclic but still maintaining u-progressiveness (u is the only input for the unknown component shown in Fig. 1.).

This is formulated as a SAT instance. For each transition, we associate a binary variable e_{jk}, which is 1 if the transition is chosen to remain. The variable s_j is 1 if State 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*, $C_j^u = E_{ju}$, 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. $C_{ij}^i = (e_{ij} \Rightarrow s_i)(e_{ij} \Rightarrow s_j)$. 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 initially 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. *(a) An FSM solution (of the v-language synthesis problem), which corresponds to a sub-graph of $\overline{\tilde{P}}$, is a solution of our SAT instance. (b) A solution of our SAT instance is an FSM solution of the v-language synthesis problem.*

Proof:

(a) Since the derived co-Buchi automaton $\overline{\tilde{P}}$ is a general automaton solution, any deterministic solution to the FSM synthesis problem (implementation) corresponding to a sub-graph of $\overline{\tilde{P}}$ has the property that for every input string w of u symbols, there is a unique path in the automaton structure of $\overline{\tilde{P}}$. This path can be decomposed into a finite part p_1, and a final part p_2 contained entirely in C. Suppose p_1 is not simple. Then there is a state s_j that is repeated. Let w_0 be the part of the input $w = w_0 w_1 w_2$ where s_j is visited for the first time and after $w_1 = \{u_1, \cdots, u_n\}$, s_j is arrived at again. Then the input $\tilde{w} = w_0 w_1 w_1^v$ would be an allowed input string to the FSM, but the run for this never eventually remains in C. This violates the co-Büchi acceptance condition. Therefore, for any input word, the finite part of each path before finally entering

C must be simple. By construction, such a path is contained in a solution of our SAT instance.

(b) Suppose we have a solution of the SAT instance. 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 many 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 (a highly efficient one has been implemented in MVSIS). 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 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. On the other hand, it is possible that our SAT instance is not satisfiable, but still there exists an FSM solution. This 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 determinization step in Section 4.1 may cause other 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 $\overline{\tilde{P}}$ can be done. In some cases, after this step, it is possible that no SAT solving is 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 $\overline{C}$ being eliminated. This automaton is processed in the regular way [9], which trims away some states and transitions in C, to make it u-progressive. The regular progressive command in MVSIS can be used to trim down this acceptance automaton. 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-called *path* (co-looping) automaton X_{path}, based on $\overline{\tilde{P}}$, 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. Now restrict X_{path} to the essential transitions and 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 there is no way to make X_{path} acyclic since eliminating any transition in the loop requires a corresponding state must be eliminated, causing 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.

Now there may be non-essential transitions that must be eliminated because they lead to eliminated states. This can create new essential transitions (which could be called secondary essential transitions).

Of the remaining nodes, the essential edges define a partial order; 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).

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 the path automaton, it is easy to combine it with the solution of the acceptance automaton to get a corresponding particular solution for the original unknown component problem.

4.4 Discussion

Although all transitions are classified as either essential or not, we cannot just set the value associated with all essential transitions to 1 in the 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. In general, the graph consisting of essential transitions and the corresponding states could be disconnected. After the preprocessing step, any simple cycle must contain at least one non-essential transition. Only non-essential edges of any loop need to be considered for elimination; 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. Hence eliminating an essential edge always implies eliminating also a non-essential edge on any loop.

5 Example and Discussion

The above presented synthesis approach has some 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 application to the Wolf-Goat-Cabbage control problem to illustrate the cycle breaking techniques discussed in earlier sections, even though it has a trivial input progressiveness. The problem is to find a strategy to transport by boat all three (wolf, goat, cabbage) across a river 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 once. X represents a transportation strategy (to be found). There is no input to X. Thus, if there is only one outgoing transition associated with a state, the transition is essential for progressiveness; if there is more than one outgoing edge, all are non-essential. This example shows how the cycles are broken in the graph.

Figures 3 and 4 illustrate this example. The co-Büchi specification is shown in Figure 3 (left). The initial state is a; the stable region consists only of State c. The automaton of the context is not shown since too many states and transitions make it unreadable. Figure 3 (right) shows the minimized most general solution automaton. In this co-Büchi automaton, there is only one state in the stable set. To find a particular FSM implementation, the most general solution is split into two automata. The path automaton is not shown, but has 12 cycles. Some states cannot reach the target state f in the path automaton. We pre-process the path automaton to remove cycles consisting of only essential transitions; and also remove any backward non-essential edges. This leads to the removals of one state and nine transitions. The resulting core structure shown in Figure 4 (left) leads to 29 variables and 57 clauses of the SAT instance, among which 9 clauses are for cycles. It is satisfiable and Figure 4 (right) shows one particular solution of the path automaton.

6 Complexity Issues and Complementing ND 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 would be to complement this to obtain the most general solution. There has been much progress in complementing ND Büchi automata (see [8] for a good review and the latest 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. The most recent results show an upper bound of $(1.06n)^n$ [8] in the number of states. In general, it is known that subset constructions do not work for co-Büchi (Büchi) automata but do work for co-looping (looping) automata. The subset construction is upper bounded by 2^n. Thus the procedure in this paper is much less expensive than the general procedure. In addition, experience with the sub-set construction shows that on practical problems, its behavior is well-behaved, in some cases resulting in a reduced number of states. However, the cost, for the general co-Büchi case, is that only a subset of the most general solution is obtained.

7 Conclusions

A flow to synthesize an FSM according to co-Büchi specifications was derived. In general, the method is sound but not complete since the determinization step may exclude some solutions (for co-looping specifications it is complete). Indeed, our procedure might wrongly conclude that no FSM solution exists. The steps used to compute a general solution automaton are the same as those used in regular finite-word automata synthesis. A difference occurs only in the last step, which derives a particular solution by formulating and solving a corresponding SAT problem. The SAT formulation is also incomplete since not all solutions can be represented. Simple experimental results demonstrate the correctness of the proposed synthesis procedures on the examples tried.

Synthesizing to Büchi specifications (useful for liveness properties) is discussed in Appendix A, and is sound and complete if the specification is a looping automaton and if the words Büchi and co-Büchi are interchanged in the procedures.

If the specification is non-deterministic, it seems expeditious to complement it as a Büchi (co-Büchi) automaton. Algorithms for complementation are discussed in Section 6 and although super-exponential in complexity, the number of states in the specification may be small. Note that trying to treat ND specifications by simply complementing the acceptance condition does not work because this results in a **for-all** (universal) ND automaton whose product with the fixed part needs to be computed.

Acknowledgments

This research was sponsored partly under NSF contract CCR-0312676 and industrial sponsors, Fujitsu, Intel, Magma, and Synplicity. Guoqiang Wang is sponsored by the Gigascale Systems Research Center. We greatly thank Orna Kupferman for extensive reading and feedback and for suggesting the application to looping automata. Thanks also to Moshe Vardi for very useful comments.

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," The 12th Advanced Research Working Conference on Correct Hardware Design and Verification Methods, Oct. 2003 (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] Alan Mishchenko, Robert Brayton, Roland Jiang, Tiziano Villa, and Nina Yevtushenko, "Efficient Solution of Language Equations Using Partitioned Representations", EuroDAC, March 2005.
[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, Artificial Intelligence 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.

Appendix A – Büchi Specifications

Büchi specifications are natural for specifying liveness properties, i.e. always eventually something good happens. The overall flow for using Büchi specifications might seem to be very similar to that discussed for synthesizing to co-Büchi specifications, with the only difference being the words "Büchi" and "co-Büchi" interchanged. However, as pointed out earlier, the subset construction for co-Büchi produces a smaller language and so the procedure is not sound (since its complement produces a larger language).

On the other hand, if the specification is a looping automaton, then the basic procedure is correct (but properties are restricted to safety properties). The product becomes a co-looping automaton, which can be determinized. On complementing this, the most general solution becomes a looping automaton. Thus, the only modification would be in the trimming as described in Sections 4.2 and 4.3. Since $\overline{\tilde{P}}$ would be a looping automaton, the loop trimming need not be done since $\overline{C}$ is a sink; therefore $\overline{C}$ and all edges to $\overline{C}$ must be eliminated. The remaining graph is then made u-progressive. This is just like that done for finite word automata specifications [9].

For a general deterministic Büchi specification, S, a sound approach would be to compute its complement using the procedure in [10]. In addition to this procedure being linear, by adding to the description of the fixed part F, often the specification can be 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 and therefore $\overline{\tilde{P}}$ will be a co-Büchi automaton.

Appendix B–Figures for Wolf-Goat-Cabbage Example

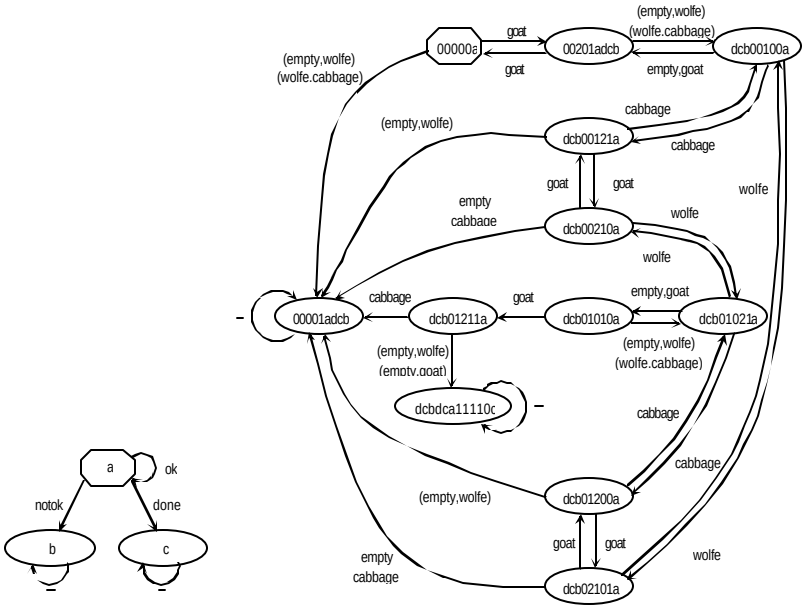

Fig. 3. Co-looping specification (left) and minimized most general automaton solution (right)

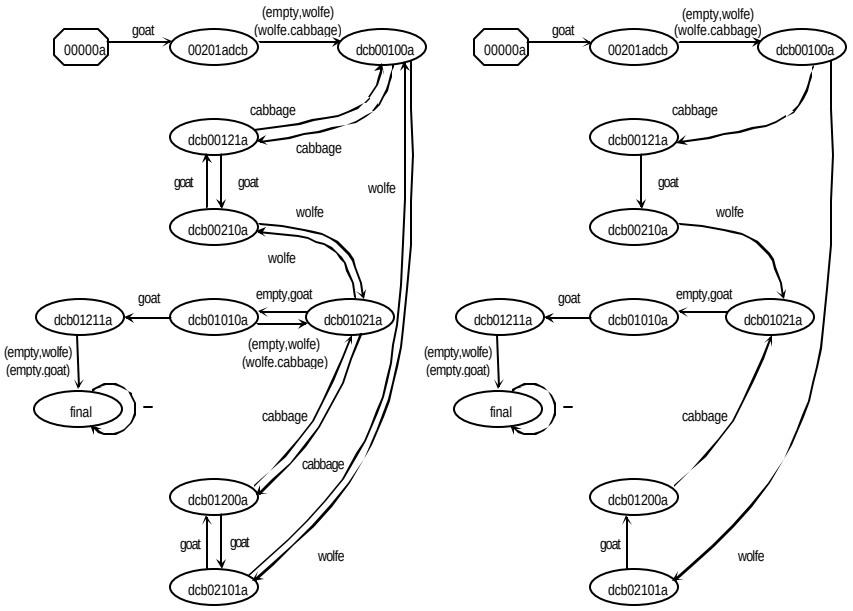

Fig. 4. Path automaton after pre-processing (left) and a particular solution of the path automa-ton with bad loops removed (right)