Solving Parallel Equations with BALM-II

G. Castagnetti
M. Piccolo
T. Villa
N. Yevtushenko
A. Mishchenko
Robert K. Brayton

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2011-102
http://www.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-102.html

September 9, 2011

Solving Parallel Equations with BALM-II

G. Castagnetti[‡] M. Piccolo[‡] T. Villa[‡] N. Yevtushenko[¶]
A. Mishchenko[†] R. Brayton[†]

[‡]Dipartimento d'Informatica, University of Verona, Verona, Italy
giovanni.castagnetti,matteo.piccolo,tiziano.villa@univr.it
[¶]Tomsk State University, Tomsk, Russia
yevtushenko@elefot.tsu.ru
[†]Dept. of EECS, University of California, Berkeley, California, USA
alanmi,brayton@eecs.berkeley.edu

September 9, 2011

Abstract

In this report we describe how to solve parallel language equations over regular languages / automata and finite state machines (FSMs), using the software package BALM-II. The original version of BALM could solve equations only with respect to synchronous composition; we extended the original code to solve also equations with respect to parallel composition, adding new commands and procedures. The new version of BALM is called BALM-II, of which this document provides a user's manual. Finally, as an important application, we describe how to synthesize protocol converters with BALM-II.

1 Introduction

In [15, 19, 20, 21, 23] we presented the theory of inequalities and equations over regular languages / automata (FA) and finite state machines (FSMs), and we gave closed-form solutions for them, with respect to the synchronous composition and parallel composition operators. For instance, given the inequality $A \bullet X \subseteq C$, where $\bullet$ denotes the synchronous composition operator, its closed-form solution can be found by computing the form $X = \overline{A \bullet \overline{C}}$, where $\overline{L}$ is the complement of L.

Similarly, for the inequality $A \diamond X \subseteq C$, where $\diamond$ denotes the so-called parallel or asynchronous composition operator, we have the closed-form solution $X = \overline{A \diamond \overline{C}}$. Specialized solutions for both classes of equations, like progressiveness, were investigated in [17, 22, 3, 4].

Moreover, we implemented in the software package BALM the operations to solve synchronous inequalities and equations over automata and FSMs, and we applied them to synthesis/resynthesis of sequential circuits, pushing as fas as possible the efficiency of computations for gate-level representations of FSMs encoded by binary decision diagrams (BDD) [11]. However, the original version of BALM did not handle inequalities and equations with respect to parallel composition. In this report we describe an extension of BALM, called BALM-II, which closes this gap, and we present the new commands and procedures added for this purpose. In particular we will discuss the subtleties involved in solving this class of inequalies with respect to automata and FSMs, having to do with the semantics of parallel composition and the way to associate languages to FSMs in this case.

Finally we will apply these computational framework to the problem of protocol converter synthesis that is naturally modeled by parallel equations and received recently a lot of attention in the literature

(see [10, 7, 14, 13, 12, 18, 2, 1, 8]). We will show how we can compute the largest solution to that problem and compare with techniques to synthesize converters described in the references.

BALM-II, same as BALM, is able to work with different interconnection topologies. In Fig. 1(a) we show a general topology with two components (we can call them, the plant and the unknown component, and their composition should match a specification component) that interact through internal signals, and also have external input and output signals. A simplified topology (called also rectification topology) is

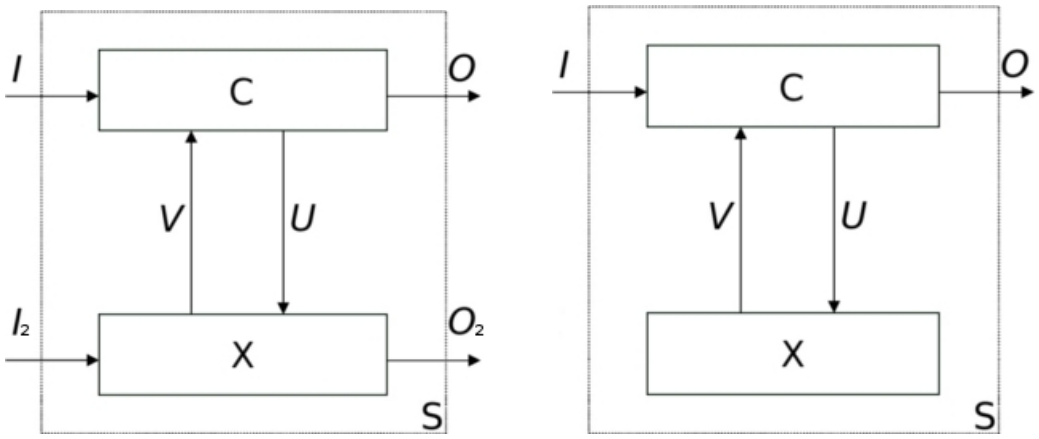

Figure 1: (a) General topology; (b) Simplified topology.

shown in Fig. 1(b), where the unknown component has no external input and output signals. For ease of exposition in the first part of the report we will refer to the topology in Fig. 1(b), whereas later we will refer to the general topology in Fig. 1(a).

2 How to Represent FSMs in BALM-II

BALM (Berkeley Automata and Language Manipulation) supports two file formats, the `BLIF-MV` (and `BLIF`) format for describing FSMs as multi-level netlists of logic, and the `AUT` format for describing automata. The `AUT` format is a restricted form of `BLIF-MV`. For a description of the formats supported by BALM and commands available we refer to [5].

The `AUT` format was designed to describe finite automata in a simple two-level form. It is essentially a restricted subset of `BLIF-MV`. The adopted restrictions allowed for a simplified version of the `BLIF-MV` parser; the restrictions are not an inherent part of any theory, and could be relaxed if future applications dictate. The unrestricted `BLIF-MV` is also used in BALM to represent the FSMs in the form of multi-valued multi-level nondeterministic networks.

In BALM, the command `read_blif_mv` can also read an `AUT` file because `AUT` is a subset of `BLIF-MV`. In such a case, the file is interpreted as a multi-valued network and not as an automaton (the number of states in this case should not exceed 32). There is no separate command to read in an automaton as an automaton, since the automata manipulation commands always read and write to `AUT` files; the input automata file name(s) are on the command line followed by the file name where the result will be written. Thus, there is no need for BALM to have separate commands to read and write automata, say `read_aut` and `write_aut`, because there is no notion of the current automaton, and instead each command that operates on automata reads and writes the automaton of interest in a specific file with extension `.aut`.

BALM-II inherits all the representation formats and commands from BALM, end extends it with features to represent and manipulate automata encoding FSMs interpreted with the semantics of parallel

composition. In the sequel we will talk of synchronous or parallel machines, according to whether they are interpreted with the semantics of synchronous or parallel composition.

All the commands implemented in BALM-II are designed for automata. When dealing with synchronous composition, FSMs are translated into automata simply merging the input and output variables of the FSM into the "inputs" of the automaton, as the Ex. 2.1 shows.

Example 2.1 *Suppose that we want to describe a counter FSM with 32 states whose description in the* `kiss` *format [16] is the following:*

```
.i 1
.o 1
.s 32
.p 64
0 st0 st0 0
1 st0 st1 1
0 st1 st1 0
1 st1 st2 1
...
0 st31 st31 0
1 st31 st0 1
.end
```

Otherwise, we could describe it in the `AUT` *format as follows (say in the file* `counter32.aut`*):*

```
.model counter32.aut
.inputs i o
.outputs Acc
.mv CS, NS 32 st0 st1 ... st31
.latch NS CS
.reset CS
0
.table Acc #all states are accepting
1
.table i o CS ->NS
0 0 st0 st0
1 1 st0 st1
0 0 st1 st1
1 1 st1 st2
...
0 0 st31 st31
1 1 st31 st0
.end
```

Dealing with the semantics of parallel composition is more difficult, the main difference being that synchronous machines and their transitions are defined over the cartesian product $A_1 \times A_2 \times \ldots \times A_n$, where $A_1, \ldots, A_n$ are the FSM's alphabets, while parallel FSMs and their transitions are defined over $A_1 \cup A_2 \cup \ldots \cup A_n$. So in a parallel transition there may be undefined values, and we need to introduce a way to "fill" these blank spaces.

In order to describe a parallel FSM using the .aut format (modified `BLIF-MV`) we introduce a new alphabet symbol `^` ("silent" symbol) that is different from any other existing alphabet symbol, and we add it as an additional value of every variable.

Given the parallel context FSM shown in Fig. 2(a) and the parallel specification FSM in Fig. 2(b), Ex. 2.2 reports their textual description in the syntax of BALM-II. These FSMs will be used as a running example in the report.

Example 2.2 *The context (or fixed) FSM,* `fixed_para.aut`*, is described as follows:*

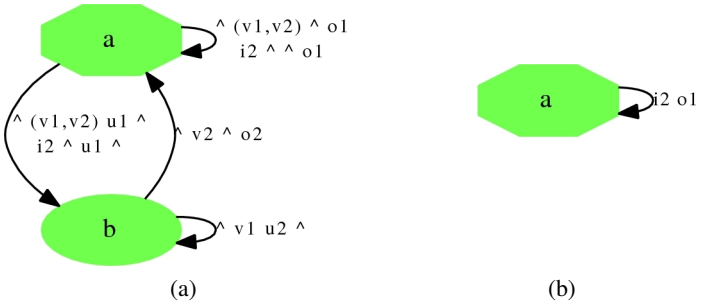

Figure 2: (a) Description of the context FSM; (b) Description of the specification FSM.

```
.model fixed_para
.inputs i v u o
.outputs Acc

.mv i 3 i1 i2 ^
.mv v 3 v1 v2 ^
.mv u 3 u1 u2 ^
.mv o 3 o1 o2 ^
.mv CS, NS 2 a b

.latch NS CS
.reset CS
a

.table CS -> Acc
.default 1

.table i v u o CS -> NS
i2 ^ ^ o1 a a
^ v1 ^ o1 a a
^ v2 ^ o1 a a
i2 ^ u1 ^ a b
^ v1 u1 ^ a b
^ v2 u1 ^ a b
^ v1 u2 ^ b b
^ v2 ^ o2 b a
.end
```

The specification FSM, `spec_para.aut`*, is described as follows:*

```
.model spec_para
.inputs i o
.outputs Acc

.mv i 3 i1 i2 ^
.mv o 3 o1 o2 ^
.mv NS, CS 1 a

.latch NS CS
.reset CS
a

.table CS ->Acc
```

```
.default 1

.table i o CS -> NS
i2 o1 a a
.end
```

3 Converting Parallel FSMs to Automata

Since BALM computes with automata, when equations defined over FSMs are given, they must be converted into the automata that recognize their languages. We already mentioned that this process is straightforward for synchronous FSMs, and is more complex for parallel FSMs. We remind first the definition of the language and related automaton associated to an FSM by the parallel semantics.

Definition 3.1 *Given an FSM $M = \langle S, I, O, T, r\rangle$, consider the finite automaton $F(M) = \langle S \cup (S \times I), I \cup O, \Delta, r, S\rangle$, where $(i, s, (s, i)) \in \Delta \wedge (o, (s, i), s') \in \Delta$ iff $(i, s, s', o) \in T$. The language accepted by $F(M)$ is denoted $L_r^{\cup}(M)$, and by definition is the* $\cup$**-language** *of M at state r. Similarly $L_s^{\cup}(M)$ denotes the language accepted by $F(M)$ when started at state s, and by definition is the $\cup$-language of M at state s. By construction, $L_s^{\cup}(M) \subseteq (IO)^{\star}$, where IO denotes the set $\{io \mid i \in I, o \in O\}$.*

We describe now the procedure to generate the automaton of a parallel FSM and its representation format in BALM-II.

The first step is to introduce a new multivalued variable (called `E` in the `.aut` files) such that each value of `E` denotes a channel that is active, whereas the other channels are inactive. A channel may contain one or more variables that have to be specified simultaneously.

Then we create as many new states as transitions (the symbolic name of a state is derived by concatenating the source and sink state names followed by the transition number numbered starting from 0) and add them to the state list of the automaton. These states are not accepting.

The next step is to split every transition into two new transitions as follows: we replace each transition $(s \xrightarrow{i/o} s')$ by the transitions $(s \xrightarrow{i} s'')$ and $(s'' \xrightarrow{o} s)$, where s'' is a new intermediate state [1]. We also annotate every transition with the information about what channel is active using the `E` variable, and we substitute every silent symbol `^` with the don't care symbol $-$. The information about the active channel, specified in every transition, is needed when we have to represent an active variable that may assume all of its possible values using a don't care symbol (i.e., `- - - - E2`).

The final step is to remove the silent symbol from the list of values of every variable because it is not needed anymore.

Ex. 3.1 shows the automata obtained, respectively, from the fixed FSM and from the specification FSM. In BALM-II, the transformation from a parallel FSM to its automaton is obtained by the new command `read_para_fsm`.

Example 3.1 *The automaton `fixed.aut` obtained from the context FSM, `fixed_para.aut`, is described as follows:*

```
.model fixed_para
.inputs i v u o E
.outputs Acc

.mv i 2 i1 i2
.mv v 2 v1 v2
.mv u 2 u1 u2
.mv o 2 o1 o2
```

[1] As an efficiency improvement, one can reduce the number of intermediate states and transitions by reuse of existing ones.

```
.mv E 4 E0 E1 E2 E3
.mv CS, NS 8 a b a_a_0 a_a_1 a_b_0 a_b_1 b_b_0 b_a_0

.latch NS CS
.reset CS
a

.table CS -> Acc
.default 0
(a,b) 1

.table i v u o E CS -> NS
- (v1,v2) - - E1 a a_a_0
- - - o1 E3 a_a_0 a
i2 - - - E0 a a_a_1
- - - o1 E3 a_a_1 a
- (v1,v2) - - E1 a a_b_0
- - u1 - E2 a_b_0 b
i2 - - - E0 a a_b_1
- - u1 - E2 a_b_1 b
- v1 - - E1 b b_b_0
- - u2 - E2 b_b_0 b
- v2 - - E1 b b_a_0
- - - o2 E3 b_a_0 a

.end
```

The automaton `spec.aut` *obtained from the specification FSM,* `spec_para.aut`*, is described as follows:*

```
.model spec_para
.inputs i o E
.outputs Acc

.mv i 2 i1 i2
.mv o 2 o1 o2
.mv E 2 E0 E1
.mv CS, NS 2 a a_a_0

.latch NS CS
.reset CS
a

.table CS -> Acc
.default 0
a 1

.table i o E CS -> NS
i2 - E0 a a_a_0
- o1 E1 a_a_0 a

.end
```

The automata in Fig. 3.1 were obtained by running the following commands:

```
read_para_fsm i|v|u|o fixed_para.aut fixed.aut
read_para_fsm i|o spec_para.aut spec.aut
```

4 New Operators on Automata

In this section we describe new commands on automata, introduced in BALM-II to solve parallel equations. One of them, `chan_synch`, renames according to a unique and same order the channels of two different automata. The syntax of `chan_synch` is:

```
chan_sync [-e] <topology1> <topology2> <input1> <input2> <out1> <out2>
```

where `<topology1>` and `<topology2>` are the descriptions of I/O channels of the two automata (comma-separated, no spaces, vertical bars separate channels, input channels come first).

Two more commands, `expansion` and `restriction`, implement the operations of expansion and restriction needed to expand and restrict the range of the channels on which an automaton is defined. The two commands share the same syntax:

```
expansion <channels> input.aut output.aut
restriction <channels> input.aut output.aut
```

where `<channels>` is a comma-separated list of channels.

All these operatios are required to compose automata and compute the solution of inequalities and equalities.

4.1 Channels synchronization

When one must operate on two (or more) automata, the `E` variable must be synchronized, i.e., each value of `E` must represent the same active channel in both automata. To that purpose we introduced a new command, `chan_sync`, that takes as inputs the topology of the channels and the description of the two files, and produces as outputs two new files with the correct mapping of `E`.

In our running example, applying `chan_synch` to `fixed.aut` and `spec.aut`, we obtain a new synchronized version of both the automata: the first one is `fixed_sync.aut` and the second one is `spec_sync.aut`. The new automaton `spec_sync.aut` is shown in Ex. 4.1, whereas `fixed_sync.aut` is the same as `fixed.aut`, because all the multi-valued variables of `spec.aut` are also in `fixed.aut`.

Example 4.1

```
.model spec_para_sync
.inputs i o E
.outputs Acc

.mv i 2 i1 i2
.mv o 2 o1 o2
.mv E 4 E0 E1 E2 E3
.mv CS, NS 2 a a_a_0

.latch NS CS
.reset CS
a

.table CS -> Acc
.default 0
a 1

.table i o E CS -> NS
i2 - E0 a a_a_0
- o1 E3 a_a_0 a

.end
```

The command used in the synchronization of the running example is:

```
chan_sync i|v|u|o|E i|o|E fixed.aut spec.aut fixed_sync.aut spec_sync.aut
```

4.2 Expansion

The expansion operation is used to extend the alphabet of the automaton to a set of specified channels.

Definition 4.1 *Given a language L over alphabet X and an alphabet V, consider the mapping $e : X \to 2^{(X \cup V)^\star}$ defined as*

$$e(x) = \{\alpha x \beta \mid \alpha, \beta \in (V \setminus X)^\star\},$$

then the language

$$L_{\Uparrow V} = \{e(\alpha) \mid \alpha \in L\}$$

over alphabet $X \cup V$ is the **expansion** *of language L to alphabet V, or V-expansion of L, i.e., words in $L_{\Uparrow V}$ are obtained from those in L by inserting anywhere in them words from $(V \setminus X)^\star$. Notice that $e(\epsilon) = \{\alpha \mid \alpha \in (V \setminus X)^\star\}$.*

Given FA F that accepts language L over X, FA F' that accepts language $L_{\Uparrow V}$ over $X \cup V$ ($X \cap V = \emptyset$) is obtained from F by adding to every state a "don't care" self-loop labeled with the specified channels.

4.3 Restriction

The restriction operation is used to restrict the alphabet of the automaton to a set of specified channels.

Definition 4.2 *Given a language L over alphabet $X \cup V$, consider the homomorphism $r : X \cup V \to V^\star$ defined as*

$$r(y) = \begin{cases} y & \text{if } y \in V \\ \epsilon & \text{if } y \in X \setminus V \end{cases},$$

then the language

$$L_{\Downarrow V} = \{r(\alpha) \mid \alpha \in L\}$$

over alphabet V is the **restriction** *of language L to alphabet V, or V-restriction of L, i.e., words in $L_{\Downarrow V}$ are obtained from those in L by deleting all the symbols in X that are not in V. Notice that $r(\epsilon) = \epsilon$.*

Given FA F that accepts language L over $X \cup V$, FA F' that accepts language $L_{\Downarrow V}$ over V is obtained from F by by replacing each edge labeled with symbols non in V by an edge labeled by the ϵ symbol. Then apply the ϵ-closure construction to obtain an equivalent deterministic finite automaton without ϵ-moves.

The flow of our restriction procedure is divided in three steps:

- In the first step we replace all the unwanted transitions by ϵ-transitions and perform an ϵ-closure for each state of the automaton. For efficiency, in practice the replacement by ϵ-transitions is made only implicitly by performing a specialized visit from each state of the automaton; the visit works as a depth-first search that moves only through the unwanted transitions. In this way, for each state, the visit stores the collection of all the states reached during the visit from the state.

- In the second step we create a new automaton with as many states as the original one, but now each state represents the ϵ-closure of the corresponding state in the original automaton. Moreover, we add transitions as follows. For every transition of the original automaton that goes from a state s_1 to a state s_2 with an action a, one or many transitions are created into the new automaton by this rule: every state that contains the original state s_1 must have a transition to the state that is the ϵ-closure of the original state s_2.

- In the last step we perform a reachability test to detect and delete all the unreachable states. When the automaton obtained by this procedure is non-deterministic (as in most cases), we perform a final determinization step.

Example 4.2 *Fig. 3 (left) shows the result after step 2 of the restriction procedure applied to the automaton in Fig. 3 (right). The original automaton is defined over channels* `E0,E1`*, and the restricted automaton is defined over channel* `E0`*. Fig. 4 shows the result of the final determinization of the automaton in Fig. 3 (left).*

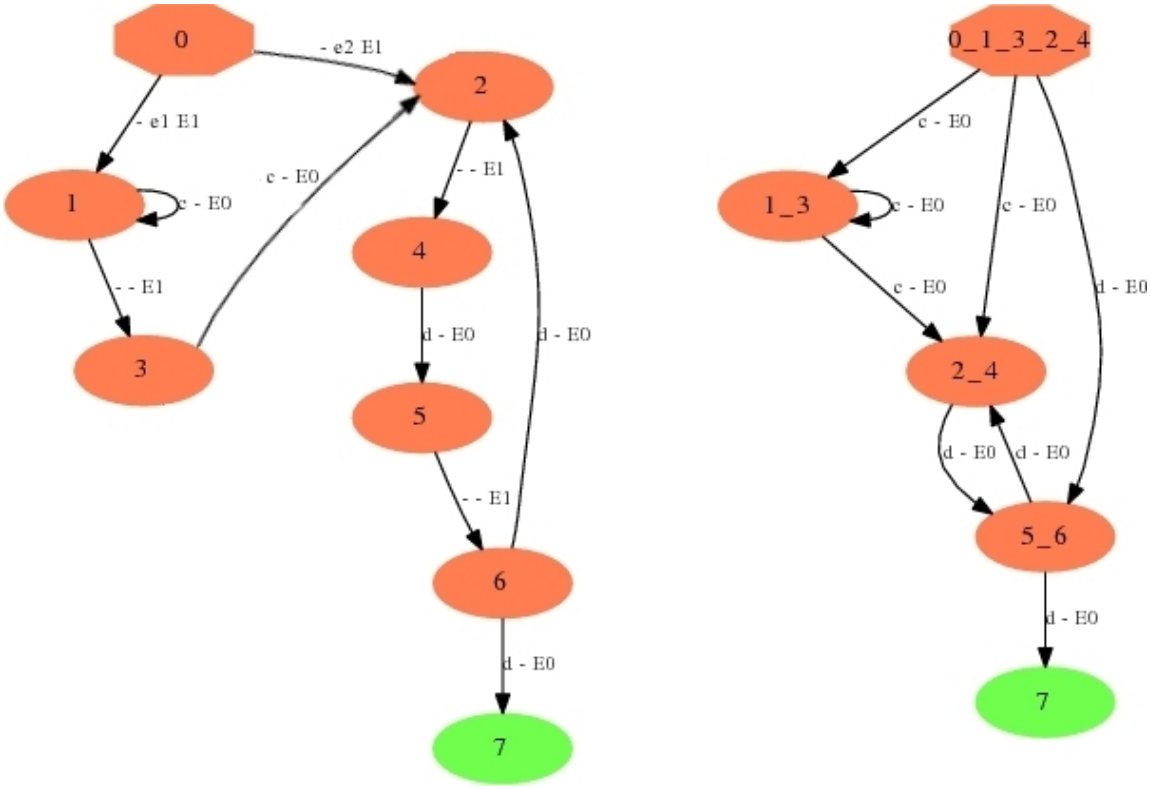

Figure 3: Example of restriction. Original automat (left); Automaton after step 2 (right).

5 Solution of Inequalities and Equations over Automata and FSMs

Consider the general topology in Fig. 1(a). Given the parallel FSM equation

$$M_A \diamond_{I_2 \cup U \cup V \cup O_2} M_X \subseteq M_C,$$

one derives the corresponding parallel automata A and C and solves the equation

$$A \diamond X_{I_2 \cup U \cup V \cup O_2} \subseteq C,$$

equivalent to

$$((A_{I_1 \cup V \cup U \cup O_1})_{\Uparrow I_2 \cup O_2} \cap X_{I_2 \cup U \cup V \cup O_2})_{I_1 \cup I_2 \cup O_1 \cup O_2} \subseteq C_{I_1 \cup I_2 \cup O_1 \cup O_2}$$

The largest automaton solution of the latter equation is given by

$$X = \overline{A \diamond_{I_2 \cup U \cup V \cup O_2} \overline{C}}$$

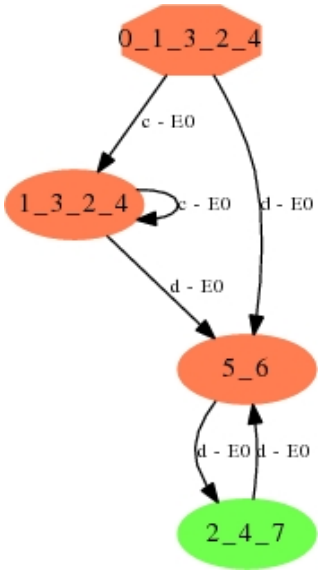

Figure 4: Determinization of the automaton in Fig.3(b) (right).

equivalent to

$$X_{I_2 \cup U \cup V \cup O_2} = \overline{((A_{I_1 \cup V \cup U \cup O_1})_{\Uparrow I_2 \cup O_2} \cap (\overline{C_{I_1 \cup I_2 \cup O_1 \cup O_2}})_{\Uparrow U \cup V})_{\Downarrow I_2 \cup U \cup V \cup O_2}}.$$

The largest FSM solution requires enforcing the hypothesis that an input must be followed by an output before another input can be produced, and it is given by (setting $I = I_1 \cup I_2$ and $O = O_1 \cup O_2$)

$$X = \overline{A \diamond_{I_2 \cup U \cup V \cup O_2} (\overline{C} \cap IO^\star)} \cap (UV)^\star,$$

equivalent to

$$X_{I_2 \cup U \cup V \cup O_2} = \overline{((A_{I_1 \cup V \cup U \cup O_1})_{\Uparrow I_2 \cup O_2} \cap (\overline{C_{I_1 \cup I_2 \cup O_1 \cup O_2}} \cap (IO)^\star)_{\Uparrow U \cup V})_{\Downarrow I_2 \cup U \cup V \cup O_2}} \cap ((UV)^\star)_{\Uparrow I_2 \cup O_2},$$

or

$$\begin{aligned} X_{I_2 \cup U \cup V \cup O_2} \quad = \quad & \overline{((A_{I_1 \cup V \cup U \cup O_1})_{\Uparrow I_2 \cup O_2} \cap (\overline{C_{I_1 \cup I_2 \cup O_1 \cup O_2}} \cap ((I_1 \cup I_2)(O_1 \cup O_2))^\star)_{\Uparrow U \cup V})_{\Downarrow I_2 \cup U \cup V \cup O_2}} \\ & \cap ((UV)^\star)_{\Uparrow I_2 \cup O_2}. \end{aligned}$$

The formulas for the topology in Fig. 1(b), or other topologies can be derived similarly introducing the appropriate supports of variables.

5.1 Script for Computing the Largest FSM Solution

We report the script to solve the FSM inequality

$$M_A \diamond_{U \cup V} M_X \subseteq M_C,$$

where M_A and M_C are, respectively, the context FSM and the specification FSM shown in Fig. 2(a) and (b), and described in Ex. 2.2. This example refers to the topology in Fig. 1(b).

The largest FSM solution is given by

$$X_{U \cup V} = \overline{(A_{I_1 \cup V \cup U \cup O_1} \cap (\overline{C_{I_1 \cup O_1}} \cap (I_1 O_1)^\star)_{\Uparrow U \cup V})_{\Downarrow U \cup V}} \ \cap \ (UV)^\star.$$

The script that implements the computation follows:

```
complement spec_sync.aut spec_sync_comp.aut
product spec_sync_comp.aut ioStar.aut spec_sync_io.aut
expansion E1,E2 spec_sync_io.aut spec_sync_io_exp.aut
product fixed_sync.aut spec_sync_io_exp.aut product.aut
restriction E1,E2 product.aut product_res.aut
support u,v,E(4) product_res.aut product_supp.aut
complement product_supp.aut product_comp_supp.aut
product product_comp_supp.aut uvStar.aut product_uv.aut
force_fsm product_uv.aut x.aut
write_para_fsm u|v|E E2|E1 x.aut x_para.aut
```

Because we are dealing with FSMs we need to execute the command `force_fsm` in order to delete those states that do not respect the alternation accepting/non-accepting that is required for the automaton to represent an FSM.

The last command `write_para_fsm` is used to extract the final FSM from the automaton that represents the largest FSM solution. The syntax of the command is:

```
write_para_fsm [-a] <topology> <channels_list> <file_in> <file_out>
```

where `<topology>` is the description of the I/O channels of the automaton (comma-separated, no spaces, vertical bars separate channels, input channels come first) and `<channels_list>` is the list of all the I/O pairs (comma-separated, no spaces, vertical bars separate I/O) that will be considered in the FSM extraction process. The resulting FSM `x_para.aut` can be manipulated again by running the command `read_para_fsm` on it to extract the automaton representing it (which can then be an input to any command of BALM-II manipulating automata).

The three final steps of the FSM solution are shown in Fig. 5; more precisely, Fig. 5(a) shows `product_uv.aut` after forcing the u, v alternation by `product`, Fig. 5(b) shows `x.aut` after forcing the accepting/non-accepting alternation by `force_fsm`, and Fig. 5(c) shows `x_para.aut` obtained by `write_para_fsm`.

For convenience of the reader, as a summary we show in Fig. 6 the state transition graphs of the context, specification and solution FSMs of our running example.

5.2 Script for Computing the Largest Automaton Solution

We report the script to solve the automaton inequality

$$A \diamond_{U \cup V} X \subseteq C,$$

where A and C are, respectively, the context and specification automata derived in Ex. 3.1 from the FSMs M_A and M_C, to which we referred already in Sec. 5.1 (originally described in Ex. 2.2).

The largest automaton solution is given by

$$X_{U \cup V} = \overline{(A_{I_1 \cup V \cup U \cup O_1} \cap (\overline{C_{I_1 \cup O_1}})_{\Uparrow U \cup V})_{\Downarrow U \cup V}}.$$

When we solve a problem on automata we do not need to ensure the alternation between inputs and outputs, so we can delete from the script in Sec. 5.1 the commands that perform the intersections with $(IO)^*$ and $(UV)^*$; moreover, we skip the command `force_fsm`.

The simplified script that implements the computation follows:

```
complement spec_sync.aut spec_sync_comp.aut
expansion E1,E2 spec_sync_comp.aut spec_sync_exp.aut
product fixed_sync.aut spec_sync_exp.aut product.aut
restriction E1,E2 product.aut product_res.aut
support u,v,E product_res.aut product_supp.aut
complement product_supp.aut x.aut
```

The resulting automaton solution `x.aut` is shown in Fig. 7.

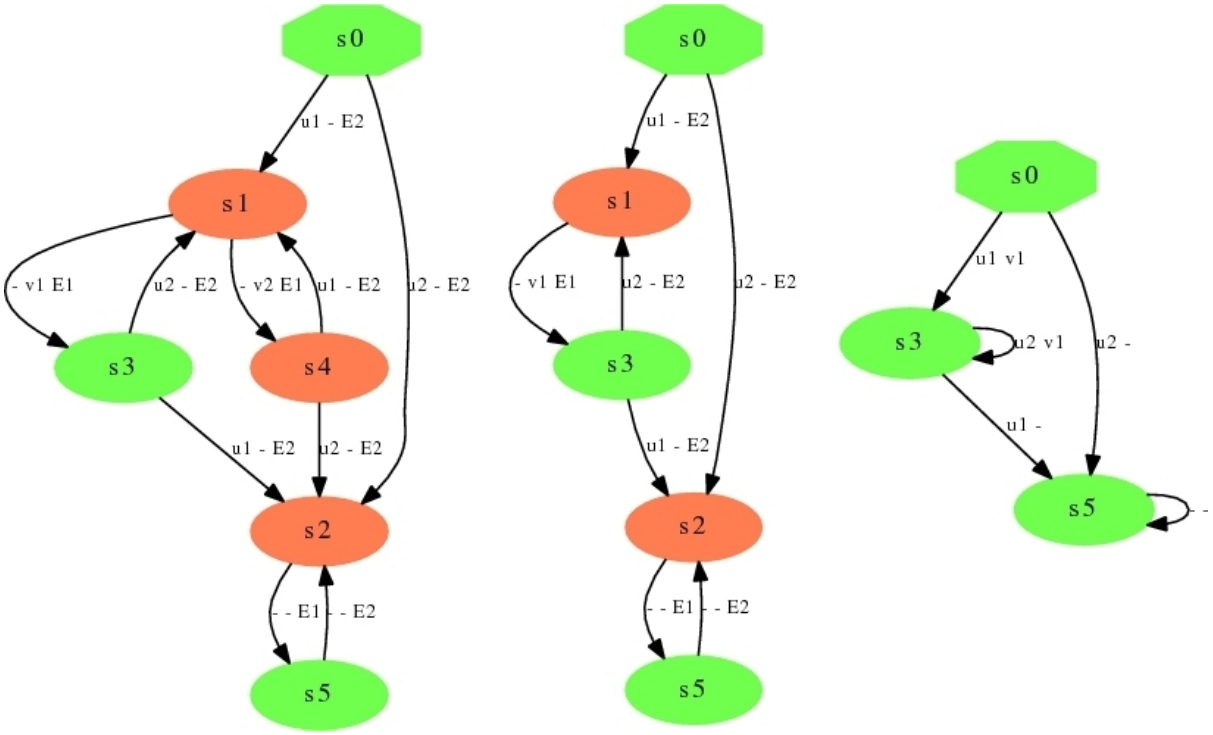

Figure 5: Largest FSM solution from script in Sec. 5.1. (a) `product_uv.aut` (after u, v alternation); (b) `x.aut` (after accepting/non-accepting alternation); (c) `x_para.aut` (after `write_para_fsm.aut`).

5.3 Verification of the Solution

Once the unknown X has been computed for the general topology of Fig. 1(a), a sanity check is to verify if its value satisfies the inequality

$$A \diamond_{I_1 \cup I_2 \cup O_1 \cup O_2} X \subseteq C,$$

equivalent to

$$((A_{I_1 \cup U \cup V \cup O_1})_{\Uparrow I_2 \cup O_2} \cap (X_{I_2 \cup U \cup V \cup O_2})_{\Uparrow I_1 \cup O_1})_{\Downarrow I_1 \cup I_2 \cup O_1 \cup O_2} \subseteq C_{I_1 \cup I_2 \cup O_1 \cup O_2},$$

For the simplified topology of Fig. 1(b) the computation reduces to:

$$(A_{I_1 \cup U \cup V \cup O_1} \cap (X_{U \cup V})_{\Uparrow I_1 \cup O_1})_{\Downarrow I_1 \cup O_1} \subseteq C_{I_1 \cup O_1}.$$

The script that implements the latter computation follows:

```
expansion E0,E3 x.aut x_exp.aut
product x_exp.aut fixed_sync.aut proof.aut
support i,v,u,o,E(4) proof.aut proof_supp.aut
restriction E0,E3 proof_supp.aut proof_supp_res.aut
force_fsm proof_supp_res.aut proof_alt.aut
contain proof_alt.aut spec_sync.aut
```

After the `product` of the context and expanded solution, we perform a `support` operation to sort the variables as in the original order. The expected result of `contain` will be (if the code is correct !): either

```
Automata are sequentially equivalent.
```

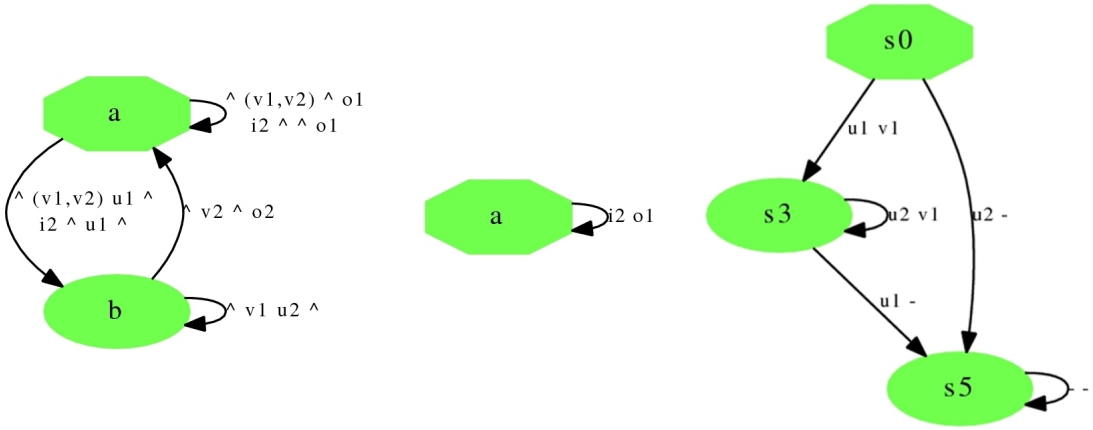

Figure 6: State transition graphs of FSMs for running example. Context FSM (left); Specification FSM (center); Solution FSM (right).

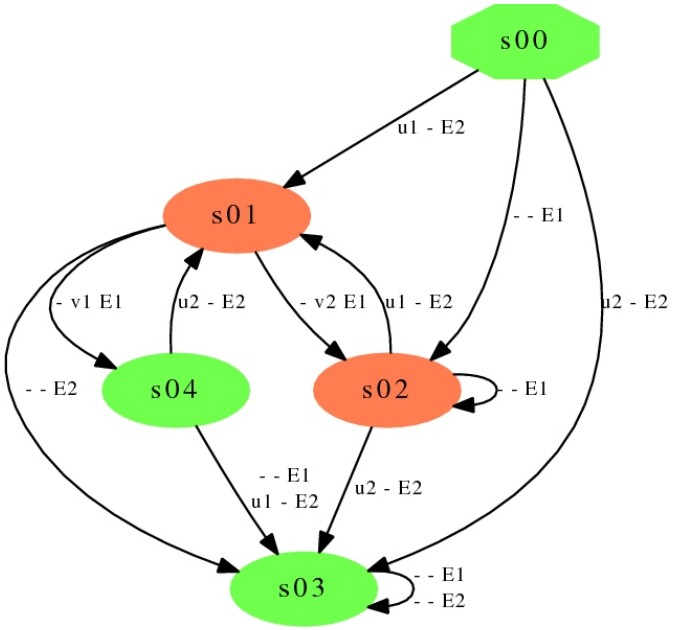

Figure 7: Largest automaton solution `x.aut` from script in Sec. 5.2.

or

```
The behavior of automaton 2 contains the behavior of automaton 1.
```

The command `force_fsm` is needed only if we are working with FSMs.

6 A Case Study from Buffalov *et al.*

In this section we will apply our approach to an example presented in a paper by Buffalov *et al.* [4]; it refers to the general topology in Fig. 1(a) (where signals I_1, I_2, O_1, O_2 are renamed, respectively, I, X, O, Y).

6.1 Complete Version of Case Study from Buffalov *et al.*

In Sec. 6.1, we discuss the full-fledged version of the example obtained from the descriptions found in [4]. For convenience of the reader, we show in Fig. 8 the state transition graphs of context and specification FSMs.

6.1.1 Descriptions of Plant and Specification of Complete Version

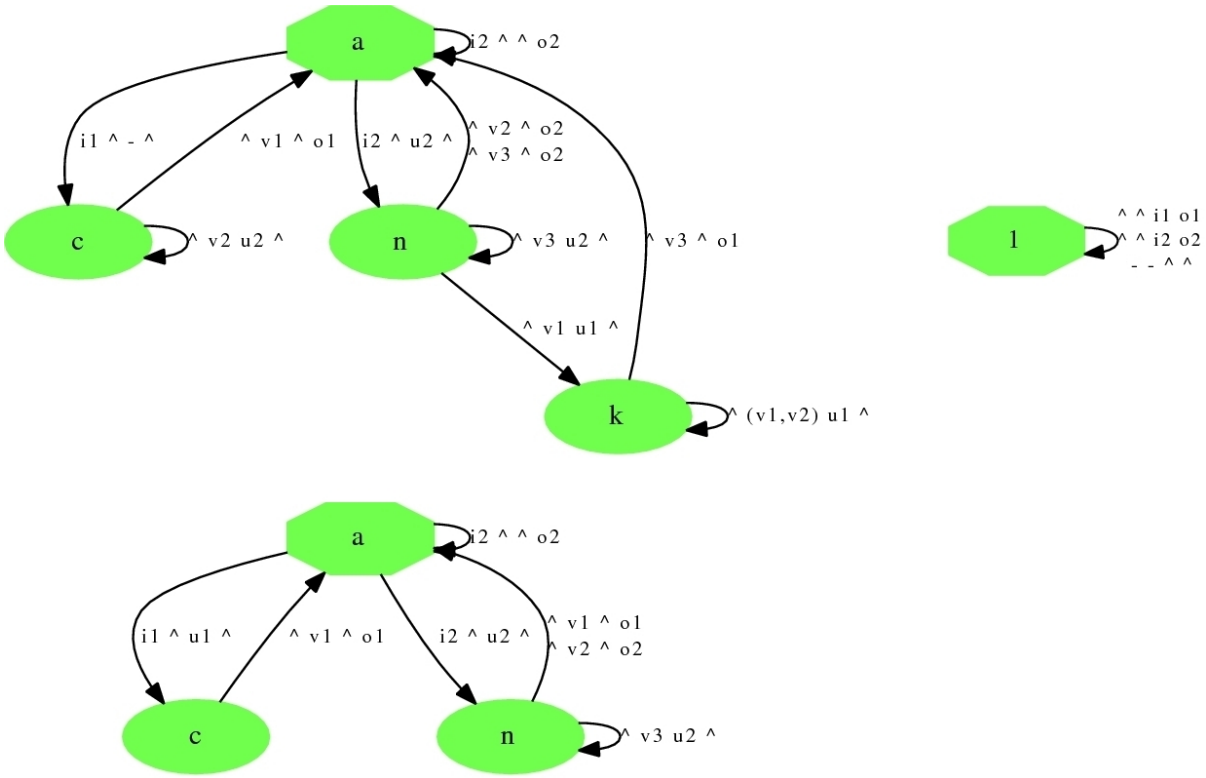

Figure 8: State transition graphs of FSMs for case study from Buffalov *et al.*. Context FSM for complete version (top left); Specification FSM for complete and simplified versions (top right); Context FSM for simplified version (bottom).

The context automaton `tcs-context.aut` (shown in Fig. 9) is:

```
.model context
.inputs i v u o E
.outputs Acc

.mv i 2 i1 i2
.mv v 3 v1 v2 v3
.mv u 2 u1 u2
.mv o 2 o1 o2
.mv E 6 E0 E1 E2 E3 E4 E5
.mv CS, NS 10 a b c d k f s m n p

.latch NS CS
.reset CS
a
```

```
.table CS -> Acc
.default 0
(a,c,n,k) 1

.table i v u o E CS -> NS
i1 - - - E2 a b
- - - - E4 b c
- v1 - - E3 c d
- - - o1 E5 d a
- v2 - - E3 c s
- - u2 - E4 s c
i2 - - - E2 a m
- - - o2 E5 m a
- - u2 - E4 m n
- v3 - - E3 n m
- v2 - - E3 n p
- - - o2 E5 p a
- v1 - - E3 n f
- - u1 - E4 f k
- (v1,v2) - - E3 k f
- v3 - - E3 k d
.end
```

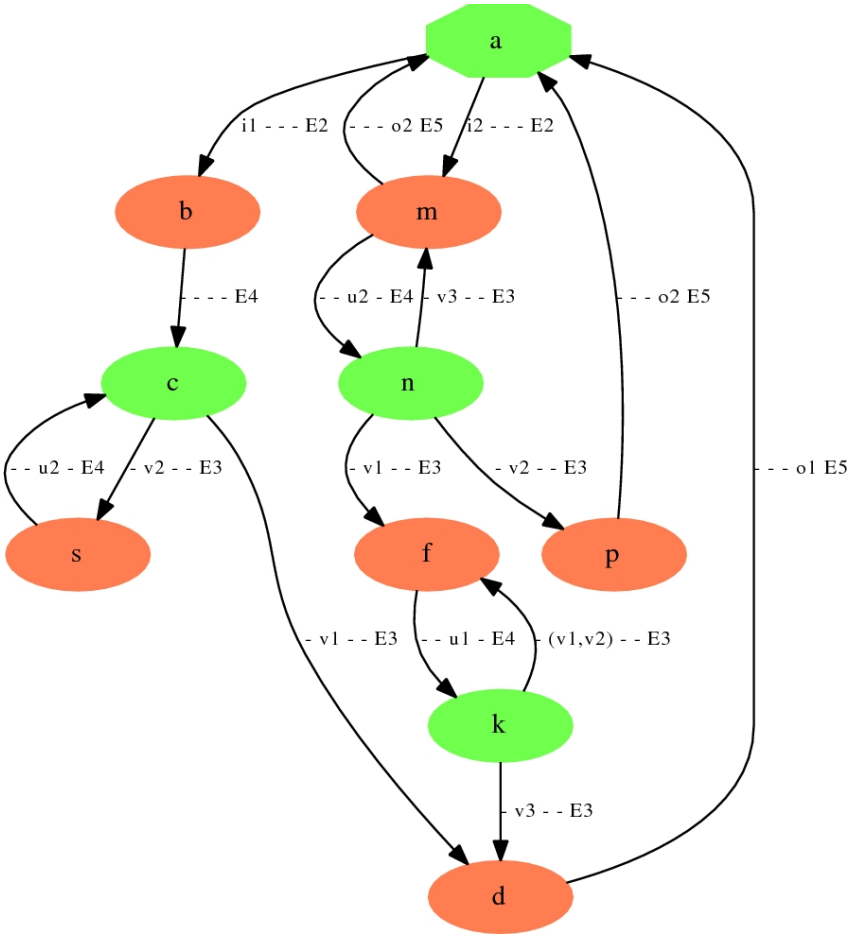

Figure 9: Context for complete version of case study from Buffalov *et al.*

The specification automaton `tcs-spec.aut` (shown in Fig. 10) is:

```
.model spec
.inputs x y i o E
.outputs Acc

.mv i 2 i1 i2
.mv o 2 o1 o2
.mv x 2 x1 x2
.mv y 2 y1 y2
.mv E 6 E0 E1 E2 E3 E4 E5
.mv CS, NS 4 1 2 3 4

.latch NS CS
.reset CS
1

.table CS -> Acc
.default 0
1 1

.table x y i o E CS -> NS
- - i1 - E2 1 2
- - - o1 E5 2 1
- - i2 - E2 1 3
- - - o2 E5 3 1
x1 - - - E0 1 4
- y1 - - E1 4 1
.end
```

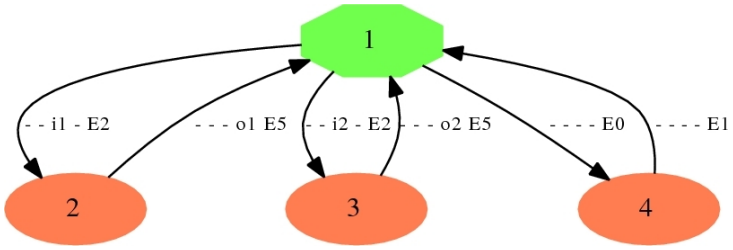

Figure 10: Specification for complete version of case study from Buffalov *et al.*

The context and specification automata can be seen as derived from the FSMs shown, respectively, in Fig. 8(a) and (b).

6.1.2 Computing the FSM Largest Solution of Complete Version

The script that computes the largest FSM solution follows:

```
complement spec_sync.aut spec_sync_comp.aut
product spec_sync_comp.aut ioStar.aut spec_sync_io.aut
expansion E3,E4 spec_sync_io.aut spec_sync_io_exp.aut
expansion E0,E1 context_sync.aut context_sync_exp.aut
product context_sync_exp.aut spec_sync_io_exp.aut product.aut
support x,y,i,v(3),u,o,E(6) product.aut product_supp.aut
restriction E0,E1,E3,E4 product_supp.aut product_res.aut
```

```
support x,y,v(3),u,E(6) product_res.aut product_res_supp.aut
complement product_res_supp.aut product_comp.aut
support x,y,u,v(3),E(6) product_comp.aut product_comp_supp.aut
product product_comp_supp.aut uvStar.aut product_uv.aut
force_fsm product_uv.aut x.aut
support x,u,y,v(3),E(6) x.aut x_supp.aut
write_para_fsm x|u|y|v|E E0|E1,E4|E3 x_supp.aut x_supp_fsm.aut
```

Notice that the `expansion` operation on the context is needed because the input and output alphabets of the specification are not a proper subset of those of the context.

Fig. 11 shows the graph of the minimized largest FSM solution `x.aut` of the complete version of the example from Buffalov *et al.*. Notice that the transition label `- ^ - ^` translates to `x/y` in the standard FSM notation with respect to the original signals; similarly, `^- ^ -` translates to `u1,u2/v1,v2,v3`.

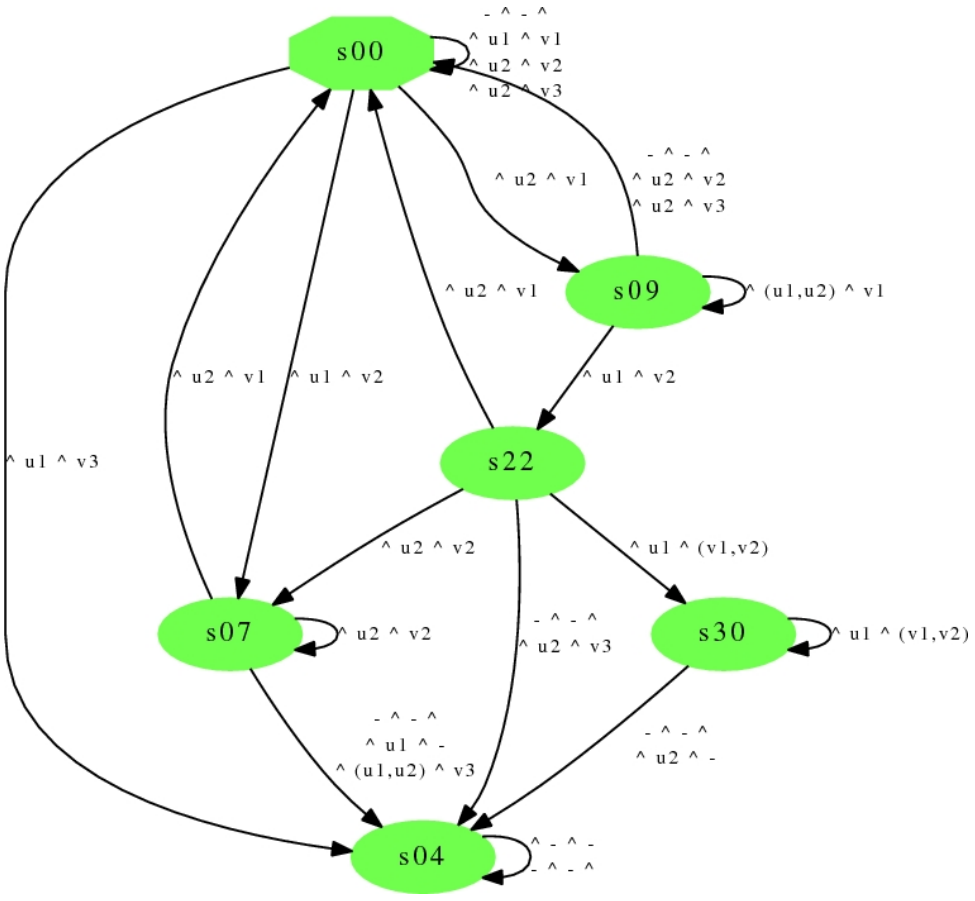

Figure 11: Minimized FSM largest solution for complete version of case study from Buffalov *et al.*

Finally, we verify whether the composition of the solution `x.aut` together with the context is contained in the specification. The script that implements the verification check follows:

```
expansion E2,E5 x.aut x_exp.aut
product x_exp.aut context_sync_exp.aut proof.aut
support x,y,i,u,v(3),o,E(6) proof.aut proof_supp.aut
restriction E0,E1,E2,E5 proof_supp.aut proof_res.aut
force_fsm proof_res.aut proof_alt.aut
contain proof_alt.aut spec_sync.aut
```

```
> Warning: Automaton 1 is completed before checking.
> Warning: Automaton 2 is completed before checking.
> Automata are sequentially equivalent
```

6.1.3 Computing the Automaton Largest Solution of Complete Version

The script that computes the largest automaton solution follows:

```
complement spec_sync.aut spec_sync_comp.aut
expansion E3,E4 spec_sync_comp.aut spec_sync_io_exp.aut
expansion E0,E1 context_sync.aut context_sync_exp.aut
product context_sync_exp.aut spec_sync_io_exp.aut product.aut
support x,y,i,v(3),u,o,E(6) product.aut product_supp.aut
restriction E0,E1,E3,E4 product_supp.aut product_res.aut
support x,y,v(3),u,E(4) product_res.aut product_res_supp.aut
complement product_res_supp.aut product_comp.aut
support x,y,u,v(3),E(4) product_comp.aut x.aut
```

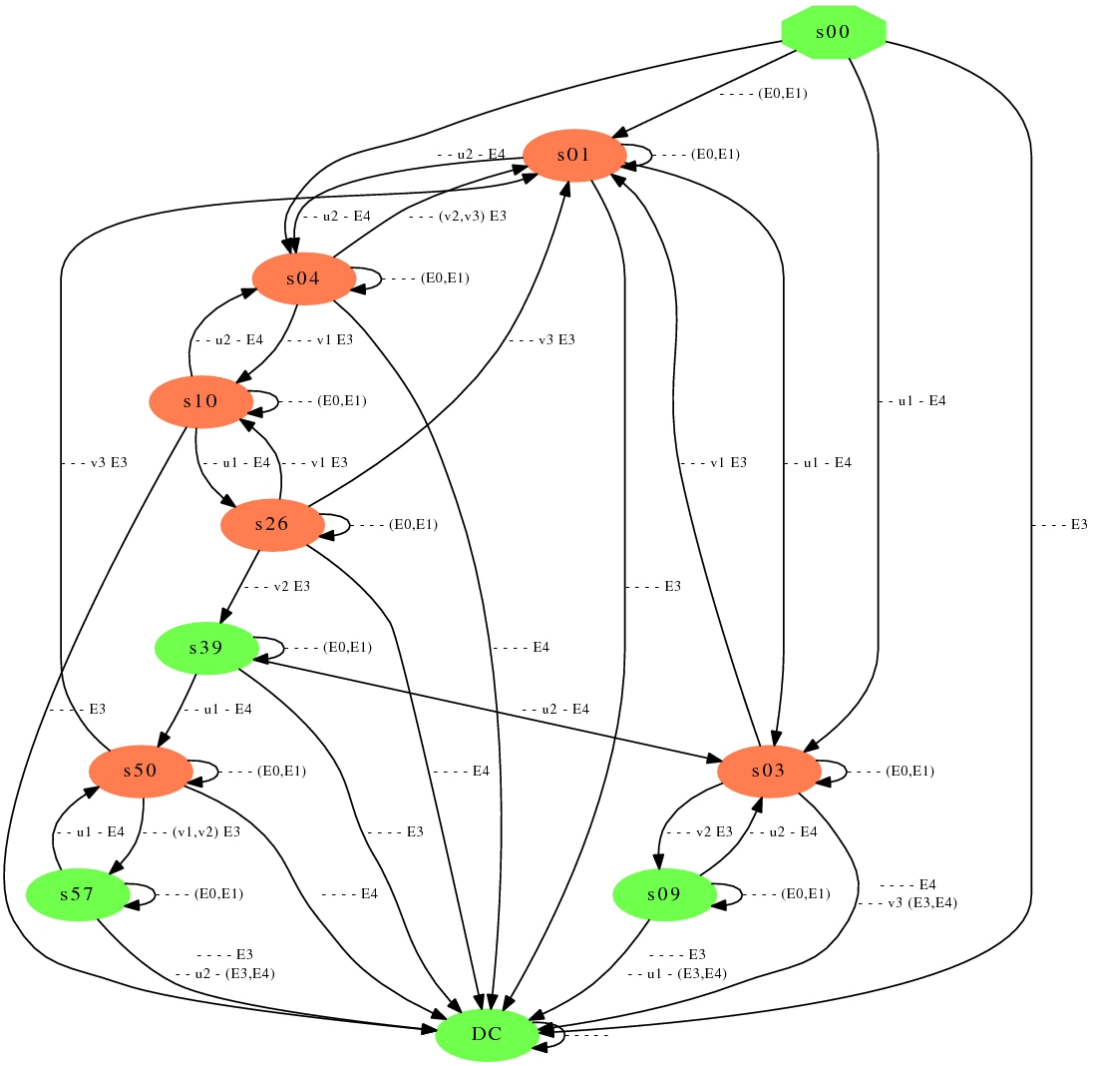

Figure 12: Minimized automaton largest solution for complete version of case study from Buffalov *et al.*

Figure 12 shows the graph of the minimized largest automaton solution `x.aut` of the complete version of the example from Buffalov *et al.*. Notice that the transition label `- ^ - ^` translates to `x/y` in the standard automaton notation with respect to the original signals; similarly, `^- ^ -` translates to `u1,u2/v1,v2,v3`.

Finally, we verify whether the composition of the solution `x.aut` together with the context is contained in the specification. The script that implements the verification check follows:

```
expansion E2,E5 x.aut x_exp.aut
product x_exp.aut context_sync_exp.aut proof.aut
support x,y,i,u,v(3),o,E(6) proof.aut proof_supp.aut
restriction E0,E1,E2,E5 proof_supp.aut proof_res.aut
contain proof_res.aut spec_sync.aut
> Warning: Automaton 2 is completed before checking.
> The behavior of automaton 2 contains the behavior of automaton 1.
```

6.2 Simplified Version of Case Study from Buffalov *et al.*

In Sec. 6.2, we discuss a simplified version of the example obtained by modifying the descriptions found in [4]. We introduce this simpler version in order to analyze carefully on a small example the relation between the FSM and automaton solutions.

6.2.1 Descriptions of Plant and Specification of Simplified Version

The context automaton `tcs_short-context.aut` (shown in Fig. 13) is:

```
.model context
.inputs i v u o E
.outputs Acc

.mv i 2 i1 i2
.mv v 2 v1 v2
.mv u 2 u1 u2
.mv o 2 o1 o2
.mv E 6 E0 E1 E2 E3 E4 E5
.mv CS, NS 7 a b c d m n p

.latch NS CS
.reset CS
a

.table CS -> Acc
.default 0
(a,c,n) 1

.table i v u o E CS -> NS
i1 - - - E2 a b
- - u1 - E4 b c
- v1 - - E3 c d
- - - o1 E5 d a
i2 - - - E2 a m
- - - o2 E5 m a
- - u2 - E4 m n
- v1 - - E3 n d
- v2 - - E3 n p
- - - o2 E5 p a
```

```
.end
```

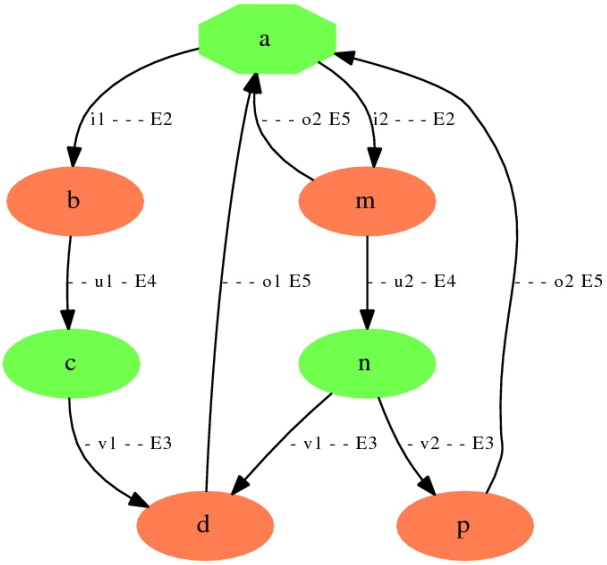

Figure 13: Context for simplified version of case study from Buffalov *et al.*

The specification automaton is the same as the one for complete version `tcs-spec.aut`. The context and specification automata can be seen as derived from the FSMs shown, respectively, in Fig. 8(c) and (b).

6.2.2 Computing the FSM Largest Solution of Simplified Version

The script that computes the largest FSM solution follows:

```
complement spec_sync.aut spec_sync_comp.aut
product spec_sync_comp.aut ioStar.aut spec_sync_io.aut
expansion E3,E4 spec_sync_io.aut spec_sync_io_exp.aut
expansion E0,E1 context_sync.aut context_sync_exp.aut
product context_sync_exp.aut spec_sync_io_exp.aut product.aut
support x,y,i,v,u,o,E(6) product.aut product_supp.aut
restriction E0,E1,E3,E4 product_supp.aut product_res.aut
support x,y,v,u,E(6) product_res.aut product_res_supp.aut
complement product_res_supp.aut product_comp.aut
support x,y,u,v,E(6) product_comp.aut product_comp_supp.aut
product product_comp_supp.aut uvStar.aut product_uv.aut
force_fsm product_uv.aut x.aut
support x,u,y,v,E(6) x.aut x_supp.aut
write_para_fsm x|u|y|v|E E0|E1,E4|E3 x_supp.aut x_supp_fsm.aut
```

Fig. 14 shows the graph of the minimized largest automaton solution `x.aut` of the simplified version of the example from Buffalov *et al.*. Notice that the transition label `- ^ - ^` translates to `x/y` in the standard FSM notation with respect to the original signals; similarly, `^- ^ -` translates to `u1,u2/v1,v2`.

Finally, we verify whether the composition of the solution `x.aut` together with the context is contained in the specification. The script that implements the verification check follows:

```
expansion E2,E5 x.aut x_exp.aut
product x_exp.aut context_sync_exp.aut proof.aut
```

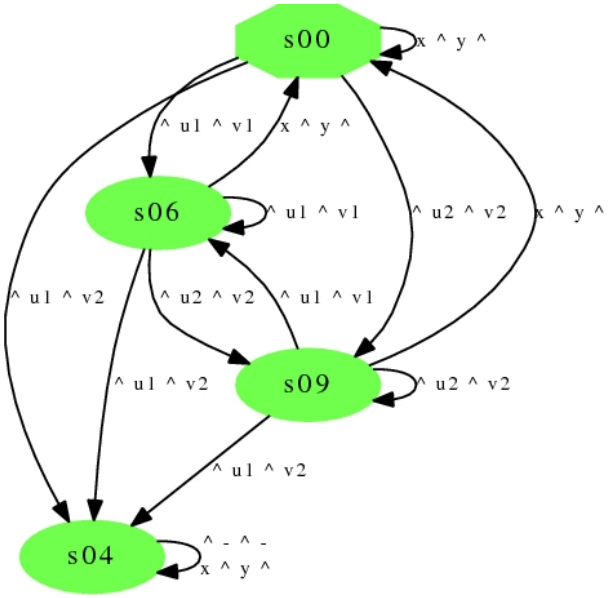

Figure 14: Minimized automaton largest solution for simplified version of case study from Buffalov *et al.*

```
support x,y,i,v(3),u,o,E(6) proof.aut proof_supp.aut
restriction E0,E1,E2,E5 proof_supp.aut proof_res.aut
force_fsm proof_res.aut proof_alt.aut
contain proof_alt.aut spec_sync.aut
> Warning: Automaton 1 is completed before checking.
> Warning: Automaton 2 is completed before checking.
> Automata are sequentially equivalent
```

6.2.3 Computing the Automaton Largest Solution of Simplified Version

The script that computes the largest automaton solution follows:

```
complement spec_sync.aut spec_sync_comp.aut
expansion E3,E4 spec_sync_comp.aut spec_sync_exp.aut
expansion E0,E1 context_sync.aut context_sync_exp.aut
product context_sync_exp.aut spec_sync_exp.aut product.aut
support x,y,i,v,u,o,E(6) product.aut product_supp.aut
restriction E0,E1,E3,E4 product_supp.aut product_res.aut
support x,y,v,u,E(6) product_res.aut product_res_supp.aut
complement product_res_supp.aut product_comp.aut
support x,y,u,v,E(6) product_comp.aut x.aut
```

The previous script differs from the one for FSMs because we deleted the lines that perform the intersections with $(IO)^*$ and $(UV)^*$, and the `force_fsm` command.

Finally, we verify whether the composition of the solution `x.aut` together with the context is contained in the specification. The script that implements the verification check follows:

```
expansion E2,E5 x.aut x_exp.aut
product x_exp.aut context_sync_exp.aut proof.aut
support x,y,i,u,v,o,E(6) proof.aut proof_supp.aut
restriction E0,E1,E2,E5 proof_supp.aut proof_res.aut
contain proof_res.aut spec_sync_all.aut
> Warning: Automaton 1 is completed before checking.
```

```
> Warning: Automaton 2 is completed before checking.
> The behavior of automaton 2 contains the behavior of automaton 1.
```

6.2.4 Comparing the FSM and Automaton Largest Solutions of Simplified Version

Comparing the two solutions using the command `contain` we obtain the following result:

```
contain xFSM.aut xAUT.aut
There is no behavior containment between the automata.
```

The `contain` command performs a containment check in both directions. The fact that FSM solutions are not a superset of automaton solutions, i.e., $FSM \not\supseteq AUT$, is not surprising because an FSM solution is obtained by forcing on automata the correct alternation between inputs and outputs, whereas an automaton solution does not have this constraint.

On the other hand, the result that automata solutions are not a superset of FSM solutions, i.e, $AUT \not\supseteq FSM$, may be surprising.

To motivate this result, we look in detail to a counterexample to the containment $AUT \supseteq FSM$.

Consider a context C and a specification S, shown respectively in Figs. 15(a) and (b); the context is defined on the external input I and the external output O, the specification is defined on the external inputs I and X and on the external outputs O and Y, whereas there are no internal signals. An FSM solution B on the external input X and on the external output Y is shown in Fig. 15(c). It is a solution, because in the composition between the context and the solution in Fig. 16, state $b1$ is deleted (by `force_fsm`) since it does not satisfy the input/output alternation. The resulting composition automaton is equal exactly to the specification.

On the other hand, B in Fig. 15(c) is not an automaton solution because its composition with the context (that coincides with the automaton in Fig. 16) is not contained in the specification. So $AUT \not\supseteq FSM$.

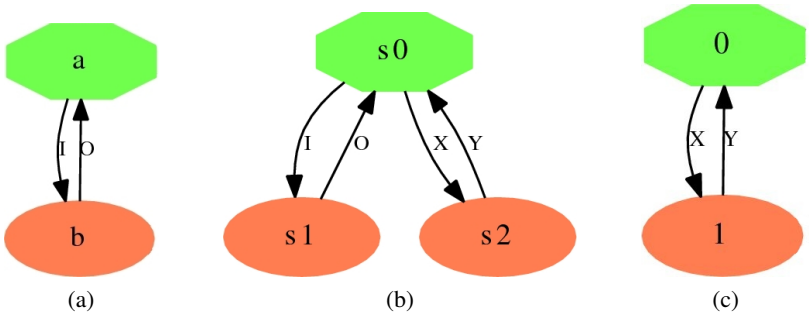

Figure 15: A counterexample to $AUT \supseteq FSM$. (a) Context C; (b) Specification S; (c) Solution B.

7 Synthesis of Protocol Converters

An application of solving parallel equations is the synthesis of protocol converters. We illustrate the problem and its solution by BALM-II by discussing two examples from the literature.

7.1 The Protocol Mismatch Problem

We define and solve an equation over finite automata to solve a problem of converter synthesis, i.e., the design of an automaton to translate between two different protocols.

A communication system has a sending part and a receiving part that exchange data through a specific protocol. A mismatch occurs when two systems with different protocols try to communicate. The

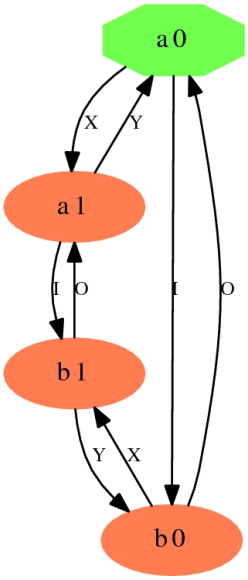

Figure 16: Composition between the context C in Fig. 15(a) and the solution B in Fig. 15(c).

mismatch problem is solved by designing a converter that translates between the receiver and the sender, while respecting the overall service specification of the behavior of the composed communication system relative to the environment. We formulate the problem as a parallel language equation: given the service specification C of a communication system, a component sender and a component receiver, find a converter X whose composition with the sender and receiver A meets the system specification after hiding the internal signals: $A \diamond X \subseteq C$.

As an example we consider the problem of designing a protocol converter to interface: an *alternating-bit* (AB) sender and a *non-sequenced* (NS) receiver. This problem is adapted from [8] and [6]. A communication system based on an alternating bit protocol is composed of two processes, a sender and a receiver, which communicate over a half duplex channel that can transfer data in either directions, but not simultaneously. Each process uses a control bit called the alternating bit, whose value is updated by each message sent over the channel in either direction. The acknowledgement is also based on the alternating bit: each message received by either process in the system corresponds to an acknowledgement message that depends on the bit value. If the acknowledgement received by a process does not correspond to the message sent originally, the message is resent until the correct acknowledgement is received. On the other hand, a communication system is non-sequenced when no distinction is made among the consecutive messages received or their corresponding acknowledgements. This means that neither messages nor their acknowledgements are distinguished by any flags such as with the alternating bit.

Fig. 17 shows the block diagram of the composed system. Each component is represented by a rectangle with incoming and outgoing labeled arrows to indicate the inputs and outputs, respectively. The sender consists of an AB protocol sender (*PS*) and of an AB protocol channel (*PC*). Meanwhile, the receiving part includes an NS protocol receiver (*PR*). The converter X must interface the two mismatched protocols and guarantee that its composition with PS, PC and PR refines the service specification (*SS*) of the composed system. The events *Acc* (*Accept*) and *Del* (*Deliver*) represent the interfaces of the communication system with the environment (the users). The converter X translates the messages delivered by the sender *PS* (using the alternating bit protocol) into a format that the receiver *PR* understands (using the non-sequenced protocol). For example, acknowledgement messages A delivered to the converter by the receiver are transformed into acknowledgements of the alternating bit protocol (*a0xc* to acknowledge a 0 bit and *a1xc* to acknowledge a 1 bit) and passed to the sender by the channel

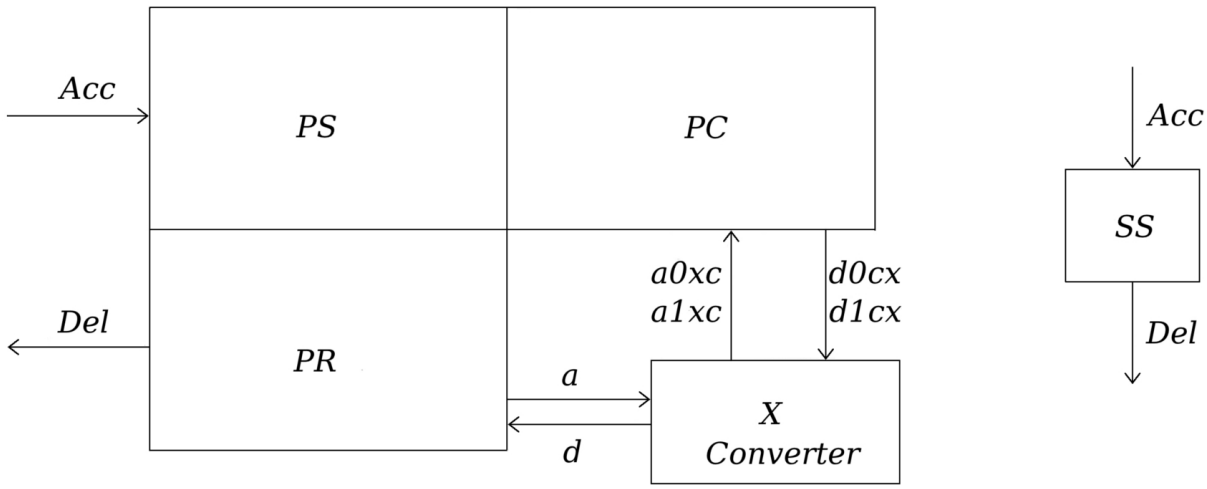

Figure 17: Communication system described in Sec. 7.1.

(*a0cs* to acknowledge a 0 bit and *a1cs* to acknowledge a 1 bit); data messages are passed from the sender to the channel (*d0sc* for a message controlled by a 0 bit and *d1sc* for a message controlled by a 1 bit) and then from the channel to the converter (*d0cx* for a message controlled by a 0 bit and *d1cx* for a message controlled by a 1 bit) to be transformed by the converter into a data message D for the receiver.

We model the components as I/O automata [9], which recognize prefix-closed regular languages, and we solve their language equations. Fig. 18 shows the automata of the components of the communication system. Missing transitions go to a trap (non-accepting) state, that loops to itself under any event.

Fig. 19 shows the largest prefix-closed solution $S = \overline{PS \diamond PC \diamond PR \diamond \overline{SS}}$ of the converter problem. All missing transitions go to an *accepting* trap state *dc* (not shown), which loops to itself under any event; e.g., the initial state has a transition to state *dc* under events $A, a0xc, a1xc, d1cx$. State *dc* can be termed the *don't care* state, because it is introduced during the determinization step to complete the automaton $PS \diamond PC \diamond PR \diamond \overline{SS}$, before the final complementation. It is reached by transitions that cannot occur due to impossible combinations of events in the composition of $PS \diamond PC \diamond PR$ and S, and so it does not matter how S behaves, once it is in state *dc* (thus the qualification *don't care* state). This makes the largest solution S non-deterministic. The solution presented in [8] and [6] does not feature this trap accepting state and so it is not complete (in [8] and [6] all missing transitions of the solution are supposed to end up in a *non-accepting* trap state, a *fail* state); without the above *dc* state, one gets only a subset of all solutions (in particular the complete solutions are missed) and this might lead to an inferior implementation.

The protocol conversion problem was addressed in [8], as an instance of supervisory control of discrete event systems, where the converter language is restricted to be a sublanguage of the context A, and in [6] with the formalism of input-output automata. In [8] the problem is modeled by the equation $A \diamond X = C$ over regular languages with the rectification topology. The solution is given as a sublanguage of A of the form $A \diamond C \setminus A \diamond \overline{C}$ (not the largest solution). An algorithm to obtain the largest compositionally progressive solution is provided that first splits the states of the automaton of the unrestricted solution (refining procedure, exponential step due to the restriction operator), and then deletes the states that violate the desired requirement of progressive composition (linear step). This algorithm does not generalize as it is to topologies where the unknown component depends also on signals that do not appear in the component A.

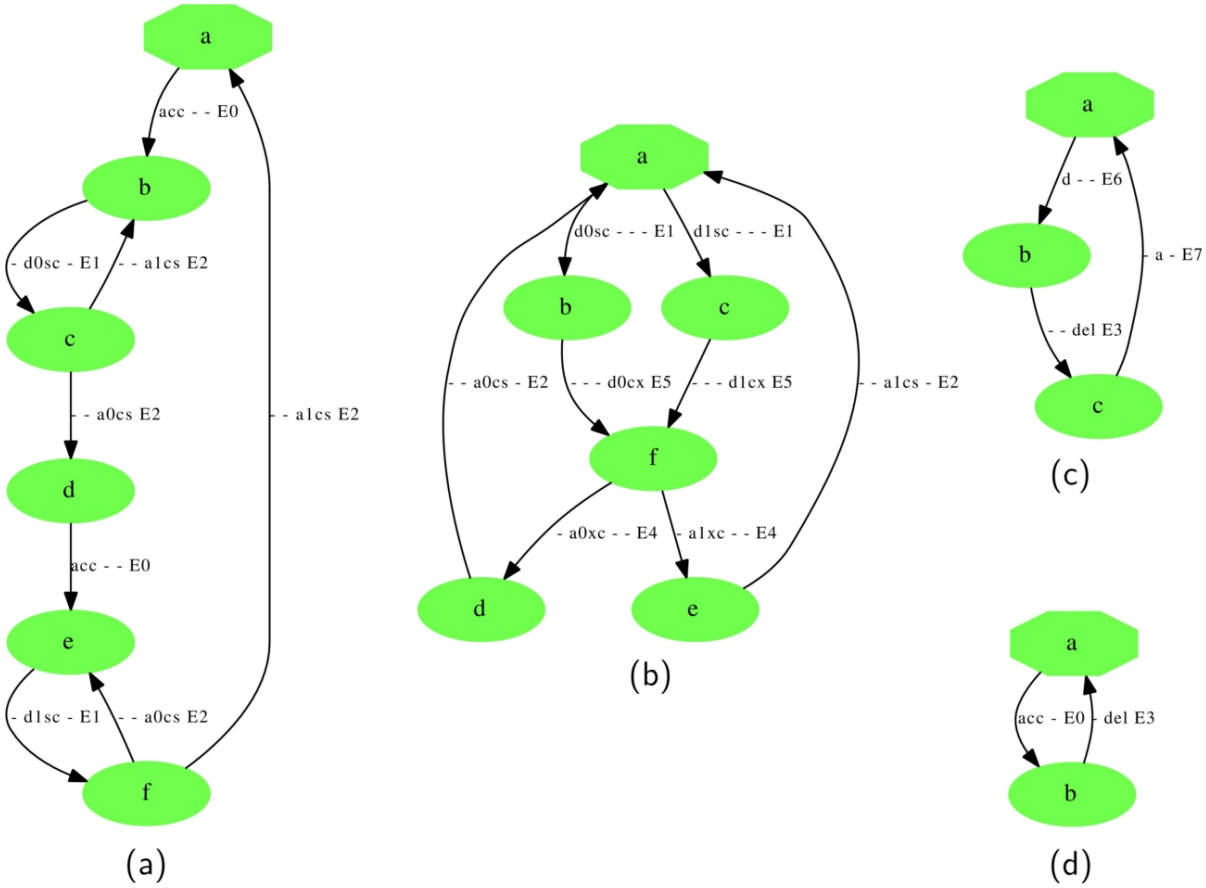

Figure 18: Automata of communication system described in Sec. 7.1 (a) Automaton of PS; (b) Automaton of PC; (c) Automaton of PR; (d) Automaton of SS.

8 Example from Martin *et al.* [10]

We show how to synthesize the protocol converter for the example from Martin *et al.* [10].

The script that computes the largest protocol converter follows:

```
complement  spec.aut  spec_comp.aut
product  fixed.aut  spec_comp.aut  product.aut
support  i(3),i1(3),E  product.aut  product_supp.aut
complement  product_supp.aut  x.aut
product  x.aut  fixed.aut  proof.aut
contain  proof.aut  spec.aut
```

Figs. 20, 21, 22 show, respectively, the fixed component `fixed_aut`, the specification `spec_aut` and solution `x_aut` of the example from Martin et al. [10]. The example refers to the simplified topology of Fig. 1(b).

9 Conclusions and Future Work

In this report we described an extension of the BALM (Berkeley Automata and Language Manipulation) package to BALM-II, which upgrades the former to handle also parallel equations over automata and FSMs of the form $A \diamond X \subseteq C$.

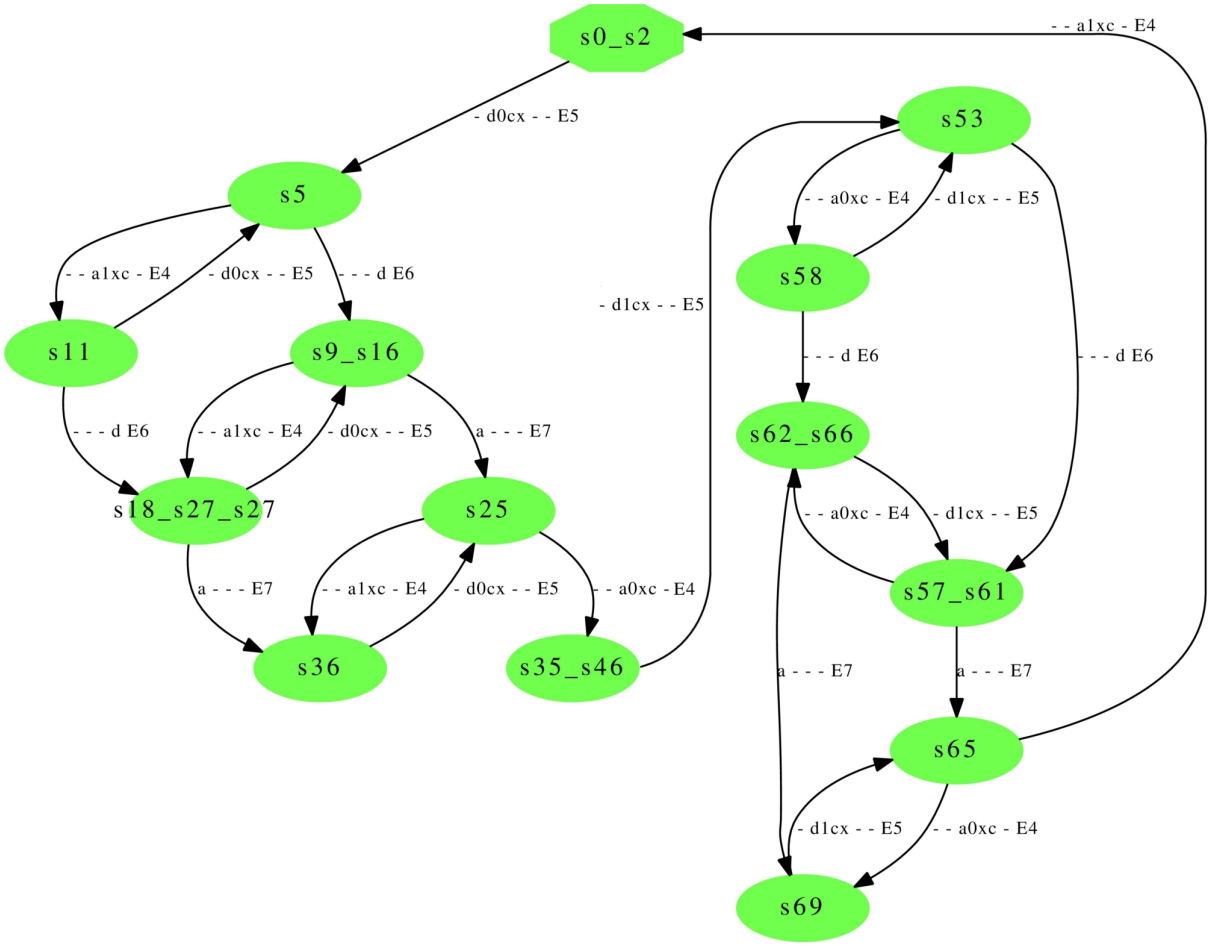

Figure 19: Largest prefix-closed solution S of the converter problem of Sec. 7.1.

We presented the different formats needed to represent FSMs and automata at the various levels of description used to perform the operations required to automatically solve the equations, verify and optimize their solutions.

We discussed in detail some examples to clarify the mechanics of the solution process, and explained at the end all the new operations of BALM-II, together with a list of the operations already existing in BALM.

Finally, we mentioned protocol converter synthesis as an application that received a lot of attention in the literature and that can be modeled and solved by BALM-II, allowing to compute all solutions, which is a feature specific to our tool compared to previous approaches reported in the literature.

Future work includes stressing the tool to assess its scalability, building a library of test cases for protocol converter synthesis and for other applications, implement operations to compute restricted solutions of parallel equations.

10 Appendix: Commands in BALM

BALM can be downloaded from the following site:
`http://embedded.eecs.berkeley.edu/mvsis/balm.html`.

The following list contains a one-line summary of all the commands available in BALM. A more detailed description is available through the online help built in the program. Please, type `help` for a

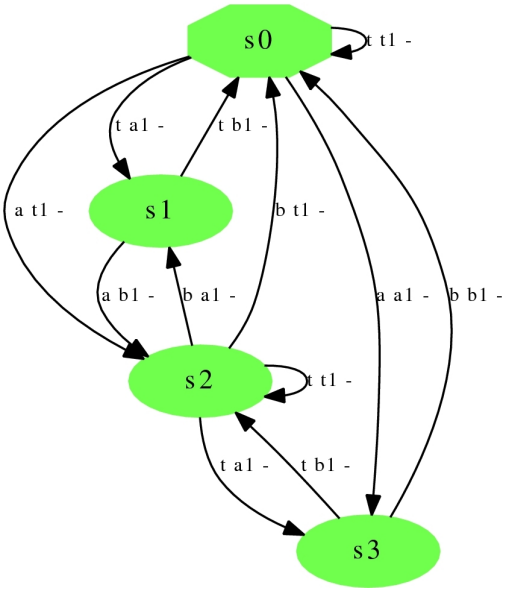

Figure 20: Fixed component `fixed_aut` of example from Martin et al. [10].

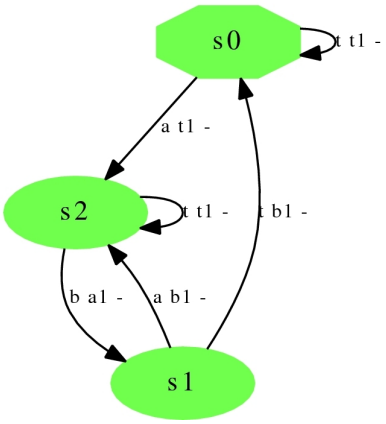

Figure 21: Specification `spec_aut` of example from Martin et al. [10].

full list of the available commands or `<command_name> -h` for a detailed description of a command including the usage.

Automata manipulation commands:

- `complement`: complement an automaton (a nondeterministic automaton will be automatically determinized first)
- `complete`: complete an automaton by adding a don't-care state
- `contain`: check language containment of two automata (checking is automatically aborted is at least one automaton is nondeterministic)
- `dcmin`: minimize the number of states by collapsing states whose transitions into care states are compatible
- `determinize`: determinize an automaton
- `minimize`: minimize the number of states of a deterministic automaton

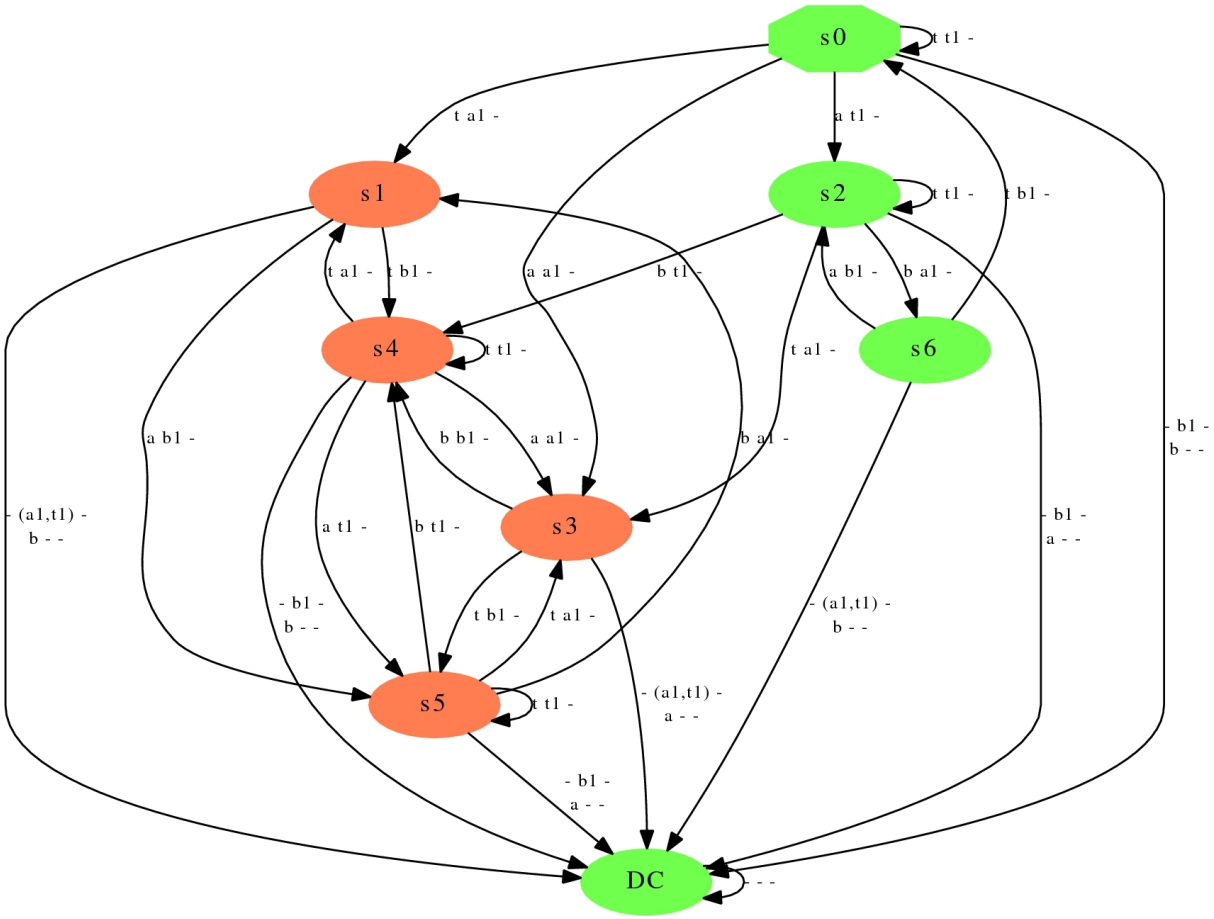

Figure 22: Solution `x_aut` of example from Martin et al. [10].

- `moore`: trim an automaton to contain Moore states only
- `prefix`: leave only accepting states that are reachable from initial states
- `product`: build the product of two automata
- `progressive`: leave only accepting and complete states that are reachable from initial states
- `support`: change the input variables of an automaton [2]

Automata viewing commands:

- `plot_aut`: visualize an automaton using DOT and GSVIEW
- `print_lang_size`: compute the number of I/O strings accepted (within the string length set by the argument `-l`) by the maximum prefix-closed sub-automaton of an automaton
- `print_nd_states`: print information about nondeterministic states of an automaton

[2] Two caveats about using `support`:

1. One must declare explicitly the number of values of input variables with more than two values.
2. It cannot handle an automaton with only one state; a work-around is to define an automaton with two equivalent states.

- `print_stats_aut`: print statistics about an automaton
- `print_support`: print the list of support variables of an automaton

I/O commands:

- `read_blif`: read the current network from the `BLIF` file
- `read_blif_mv`: read the current network from the `BLIF-MV` file
- `write_blif`: write the current network in the `BLIF` format
- `write_blif_mv`: write the current network in the `BLIF-MV` format

Miscellaneous commands:

- `alias`: provide an alias for a command
- `echo`: echo the arguments
- `help`: print the list of available commands by group
- `history`: a UNIX-like history mechanism inside the BALM shell
- `ls`: print the file names in the current directory
- `quit`: exit BALM
- `source`: execute commands from a file
- `time`: provide a simple elapsed time value
- `unalias`: removes the definition of an alias

MV network commands:

- `extract_aut`: extract the state-transition graph from the current network as an automaton
- `latch_expose`: make latch outputs visible as POs of the current network
- `latch_split`: split the current network into two networks by dividing latches and the related combinational logic; generates synthesis and verification scripts assuming that one part is fixed and another part is unknown
- `solve_fsm_equ`: solve language equation $F \bullet X \subseteq S$ using the method discussed in [11]. F and S must be given in `BLIF-MV` format and must be deterministic.

Network viewing commands:

- `print`: print multi-valued sum-of-products representation of nodes
- `print_factor`: print algebraic factored form of nodes
- `print_io`: print fanins/fanouts of nodes
- `print_latch`: print the list of latches of the current network
- `print_level`: print nodes in the current network by level

- `print_nd`: print the list of nondeterministic nodes in the current network
- `print_range`: print the numbers of values of nodes
- `print_stats`: print network statistics and report the percentage of nodes having each representation

The following list contains a description of all the commands added to BALM to upgrade it to BALM-II.

Automata manipulation commands:

- `chan_sync`: renames according to a unique and same order the channels of two different automata
- `check_nb`: tests if an automaton is nonblocking (otherwise deadlocks and livelocks can occur)
- `expansion`: performs an expansion on a parallel automaton
- `force_fsm`: forces the alternation between accepting and non accepting states as in parallel FSMs
- `prefix_close`: enforces prefix-closure, i.e., removes non-accepting states and keeps reachable accepting states
- `read_para_fsm`: reads a parallel FSM and generates its parallel automaton
- `remove_dc`: removes the don't-care states (they have only one outgoing transition with every input and output) of the automaton
- `restriction`: performs a restriction on a parallel automaton
- `sbssolve`: runs the script to solve a synchronous language equation $F \bullet X \subseteq S$ with F and S represented as automata in .aut format
- `set_all_accepting`: sets all states to accepting
- `trim`: trims an automatoni, i.e., it makes it both reachable and co-reachable
- `write_para_fsm`: transforms an automaton into a parallel FSM

References

[1] K. Avnit and A. Sowmya. A formal approach to design space exploration of protocol converters. In *The Proceedings of the Design, Automation and Test in Europe Conference*, pages 129–134, April 2009.

[2] Purandar Bhaduri and S. Ramesh. Interface synthesis and protocol conversion. *Formal Aspects of Computing*, 20:205–224, 2008. 10.1007/s00165-007-0045-4.

[3] S. Buffalov, K. El-Fakih, N. Yevtushenko, and G. v. Bochmann. Progressive solutions to a parallel automata equation. In *Proceedings of the IFIP 23rd International Conference on Formal Techniques for Networked and Distributed Systems (FORTE 2003)*, volume 2767 of *LNCS*, pages 367–382. Springer Verlag, September 2003.

[4] K. El-Fakih, S. Buffalov, N. Yevtushenko, and G. v. Bochmann. Progressive solutions to a parallel automata equation. *Theoretical Computer Science*, 362:17–32, 2006.

[5] BALM Research Group. BALM. Website and User's Manual at `http://embedded.eecs.berkeley.edu/Respep/Research/mvsis/balm.html`.

[6] H. Hallal, R. Negulescu, and A. Petrenko. Design of divergence-free protocol converters using supervisory control techniques. In *7th IEEE International Conference on Electronics, Circuits and Systems, ICECS 2000*, volume 2, pages 705–708, December 2000.

[7] Y. Jiang and Y. Jin. Protocol converter sysnthesis: an application of control synthesis. EE219C Class Project Report, December 1999.

[8] R. Kumar, S. Nelvagal, and S.I. Marcus. A discrete event systems approach for protocol conversion. *Discrete Event Dynamic Systems: Theory & Applications*, 7(3):295–315, June 1997.

[9] N. Lynch and M. Tuttle. An introduction to input/output automata. *CWI-Quarterly*, 2(3):219–246, September 1989.

[10] Grant Martin, Brian Bailey, and Andrew Piziali. *ESL Design and Verification: A Prescription for Electronic System Level Methodology*. Morgan Kaufmann, USA, 2007. 488p.

[11] A. Mishchenko, R. Brayton, J.-H. Jiang, T. Villa, and N. Yevtushenko. Efficient solution of language equations using partitioned representations. In *The Proceedings of the Design, Automation and Test in Europe Conference*, volume 01, pages 418–423, March 2005.

[12] R. Passerone. *Semantic Foundations for Heterogeneous Systems*. PhD thesis, EECS Department, University of California, Berkeley, 2004. Tech. Report No. UCB/ERL M98/30.

[13] Roberto Passerone, Luca de Alfaro, Thomas A. Henzinger, and Alberto L. Sangiovanni-Vincentelli. Convertibility verification and converter synthesis: two faces of the same coin. In *ICCAD*, pages 132–139, 2002.

[14] Roberto Passerone, James A. Rowson, and Alberto L. Sangiovanni-Vincentelli. Automatic synthesis of interfaces between incompatible protocols. In *DAC*, pages 8–13, 1998.

[15] A. Petrenko and N. Yevtushenko. Solving asynchronous equations. In S. Budkowski, A. Cavalli, and E. Najm, editors, *Formal Description Techniques and Protocol Specification, Testing and Verification - FORTE XI/PSTV XVIII '98*, pages 231–247. Kluwer Academic Publishers, November 1998.

[16] E. Sentovich, K. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. Stephan, R. Brayton, and A. Sangiovanni-Vincentelli. SIS: A system for sequential circuit synthesis. Technical report, Tech. Rep. No. UCB/ERL M92/41, Berkeley, CA, May 1992.

[17] T. Villa, N. Yevtushenko, and S. Zharikova. Characterization of progressive solutions of a synchronous FSM equation. In *Vestnik, 278, Series Physics*, pages 129–133, September 2003. (In Russian).

[18] S. Watanabe, K. Seto, Y. Ishikawa, S. Komatsu, and M. Fujita. Protocol transducer synthesis using divide and conquer approach. In *Design Automation Conference, 2007. ASP-DAC '07. Asia and South Pacific*, pages 280 –285, jan. 2007.

[19] N. Yevtushenko, T. Villa, R. Brayton, A. Petrenko, and A. Sangiovanni-Vincentelli. Solution of parallel language equations for logic synthesis. In *The Proceedings of the International Conference on Computer-Aided Design*, pages 103–110, November 2001.

[20] N. Yevtushenko, T. Villa, R. Brayton, A. Petrenko, and A. Sangiovanni-Vincentelli. Solution of synchronous language equations for logic synthesis. In *The Biannual 4th Russian Conference with Foreign Participation on Computer-Aided Technologies in Applied Mathematics*, September 2002.

[21] N. Yevtushenko, T. Villa, R. Brayton, A. Petrenko, and A. Sangiovanni-Vincentelli. Sequential synthesis by language equation solving. Technical report, Tech. Rep. No. UCB/ERL M03/9, Berkeley, CA, April 2003.

[22] N. Yevtushenko, T. Villa, R. Brayton, A. Petrenko, and A. Sangiovanni-Vincentelli. Compositionally progressive solutions of synchronous FSM equations. *Discrete Event Dynamic Systems*, 18(1):51–89, March 2008.

[23] N. Yevtushenko, T. Villa, and S. Zharikova. Solving language equations over synchronous and parallel composition operators. In M. Kunc and A. Okhotin, editors, *Proceedings of the 1st International Workshop on Theory and Applications of Language Equations, TALE 2007, Turku (Finland), 2 July 2007*, pages 14–32. Turku Centre for Computer Science, 2007.