BALM User's Manual

Berkeley Language Equation Solving Group

Robert K. Brayton[1]
Jie-Hong R. Jiang[1]
Alan Mishchenko[1]
Tiziano Villa[2]
Nina Yevtushenko[3]

[1]University of California, Berkeley
[2]University of Udine, Italy
[3]Tomsk State University, Russia

Introduction to BALM

Finite-state automata and their languages are well-studied subjects since the early development of computation theory. Traditional automata manipulations are based on explicit state representation, and are limited to automata with a few thousand states. The manipulation of automata became more practical with the advent of efficient symbolic techniques based on binary decision diagrams (BDDs), satisfiability (SAT) solvers and AND-INVERTER graphs (AIGs). Based on these techniques, the BALM (Berkeley Automata and Language Manipulation) package aims at providing an experimental environment for efficient manipulation of finite automata in various application domains, e.g., synthesis, verification, control, etc. The environment features the most typical automata operations, such as determinization and state minimization, as well as some visualization capabilities, which rely on the powerful graph visualization software [Graphviz]. The applicability of BALM to finite-state machine synthesis is demonstrated by solving an unknown component problem formulated using language equations.

Unique features of BALM

Compared with other automata manipulation utilities, such as Grail [Grail], FSA [FSA], etc., BALM provides a unique environment suited for language equation solving, an important application of automata theory to component-based sequential hardware synthesis. In addition, BALM is useful in sequential system optimization because it can interface with the `BLIF-MV` or `BLIF` formats, which are commonly used in logic synthesis of Boolean networks. Using BALM, it is possible to extract sequential flexibilities in implementing a hardware finite state machine.

Computation behind BALM

In addition to traditional explicit graph enumeration algorithms for automata manipulations, BALM provides computation algorithms based on state-of-the-art symbolic techniques. It allows most manipulations of automata to be done implicitly. With these the system can handle up to millions of states. However, the complementation (determinization) of nondeterministic automata still uses explicit enumeration and hence is limited to about at most 100K states.

Describing Designs for BALM

BALM supports two file formats, `BLIF-MV` (and `BLIF`) to describe FSMs and `AUT` for describing automata. `AUT` is a special restricted form of `BLIF-MV`.

BLIF-MV

`BLIF-MV` is one of the input formats to BALM. It is a multi-valued extension of `BLIF` (Berkeley Logic Interchange Format) to efficiently specify sequential systems in the form of multi-level multi-valued (possibly non-deterministic) Boolean networks. Compared to `BLIF`, `BLIF-MV` allows non-deterministic nodes[1] and hence allows for modeling non-deterministic systems. For instance, a sequential system may contain non-determinism in its early stages of its development when some of its aspects are not completely specified. In addition, `BLIF-MV` supports multi-valued variables, which often simplify specification of the system. Although the number of values of the input and output variables of finite-state systems in the current implementation of BALM is limited to 32, the number of values of state variables is not limited.

The semantics of `BLIF-MV` is defined over flattened networks, using a combinational/sequential concurrency model. There are three basic primitives: *variables*, *tables* (nondeterministic nodes), and *latches*. A *variable* can take values from a finite domain. A relation defined over a set of variables is represented using a *table*. A table has only one output and any number of inputs. A particular variable can be designated as an output in at most one table. Several tables are conceptually inter-connected if there is a common name at the output of one table and the inputs of the other tables. Thus a named variable is conceptually a *wire*. If a table is deterministic and Boolean, it may also be thought of as a logic gate. A *latch* is a specialized element that can be placed on a wire. It divides the wire into two parts; the input to the latch, and the output of the latch. A set of initial values is associated to every latch; they must be a subset of the set of values of its wire. A state is an assignment of values to the latches of a model, where a value assigned to a latch must be in its domain. An initial state is a state where every latch takes a value from its set of initial values. A latch can have more than one initial state in general.

At every time point, the system is in some state, where each latch has a value. At every clock tick, all the latches update their values. These values then propagate through tables until all the wires have a consistent set of values. If a latch is encountered during the propagation, i.e., an output of a table is an input of a latch, the propagation process through that latch is stopped. Note that because of nondeterminism, given a single state, there may be several consistent sets of values. This semantics can be seen as a simple extension of the standard semantics of synchronous single-clocked digital circuits. In fact, if every table is deterministic and every latch has a single initial state, the two semantics are exactly equal. The only differences are in the interpretation of nondeterministic tables and latches with multiple initial states.

In BALM, the command `read_blif_mv` reads a `BLIF-MV` (or `BLIF`) description, and then sets up a corresponding internal data structure to represent the multi-valued network. The `write_blif_mv` command writes a `BLIF-MV` description to a file. The `BLIF-MV` format is not meant to be read or written directly by the user,

[1] These nodes generate some output from the set of pre-specified outputs.

even though simple examples in BLIF-MV may exhibit some degree of clarity. For a more detailed treatment of the BLIF-MV format and some examples, see [BLIFMV].

AUT

In this section, we describe several restrictions on the BLIF-MV format used to represent finite automata in BALM. This restricted language is called AUT format in this manual. The adopted restrictions have to do with a simplified version of the BLIF-MV parser currently implemented, and may be relaxed in the future. Note that general non-restricted BLIF-MV can be used in BALM to represent the FSMs in the form of multi-valued multi-level non-deterministic networks, as described in the previous section.

There are several restrictions on AUT, compared to BLIF-MV:

1) Only a non-hierarchical BLIF-MV specifications are allowed in AUT.
2) An AUT file should have exactly one latch and exactly two combinational nodes, one describing the next-state relation and one describing the single output called "Acc". Multi-level decompositions of the next-state node are not supported.
3) The latch's output and input variables have fixed names to be "CS" and "NS". Their values should be defined using ".mv" directive to be equal to the number of states of the automaton. The reset value of the latch is the initial state of the automaton. Currently, only automata with one initial state can be used.
4) The next-state table (output is NS) should have the automaton inputs listed on the ".names/.table" line, followed by CS variable. The last variable on the line should be "NS" variable.
5) In each cube of the next-state table, variables CS and NS can have only one specific state value, for example, "s33" or "stateABC". Multi-valued state literals of the type "(s33, s35, s37)" are not allowed in the next-state table. The don't-care literal "-" cannot be used for CS and NS variables, and we cannot simplify CS variable away. This variable should always be listed and all its values should be used in the table at least once, including the ".default" line. However, note that there are no such restrictions on how the multi-valued inputs of the automaton are specified in the table. Any literals of these variables, including the don't-care literal, can be used in the next-state table.
6) The BLIF-MV specification should have exactly one binary primary output. Its name is fixed to be "Acc" and is defined using the second combinational node of the BLIF-MV file. It has only one input, CS. The purpose of this node is to specify which of the automaton states are accepting. Thus Acc = 1 if and only if CS has a value equal to one of the accepting states.
7) The default line of the Acc table can be in one of the following forms:

   Case 1: all states are accepting

   ```
   .table -> Acc
   1
   ```

   Case 2: only one state is accepting

   ```
   .table CS -> Acc
   .default 0
   ```

   <acc_state> 1

Case 3: only one state is non-accepting

```
.table CS -> Acc
.default 1
<non_acc_state> 0
```

Case 4: several accepting states

```
.table CS -> Acc
.default 0
(comma-separated list of accepting states in parentheses) 1
```

or

```
.table CS -> Acc
.default 1
(comma-separated list of non-accepting states in parentheses) 0
```

Below is an example of a simple 3-state automaton with accepting state "DC" representing using AUT format:

```
==============================
.model spec
.inputs i o
.outputs Acc

.mv CS, NS  3  a b DC
.mv i 3

.latch NS CS
.reset CS
a

.table CS ->Acc
.default 0
DC 1

.table i o CS ->NS
.default DC
(1,2)  1   a   a
-      0   a   b
0      1   b   a

.end
==============================
```

In BALM, the same command read_blif_mv can read in an AUT file because AUT is a subset of BLIF-MV. In this case, the input is interpreted as a multi-value 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. The automata manipulation commands always take the input automata file name(s) on the command

line and write the resulting automaton into an output file name specified on the command line. This is the reason BALM does not have a separate command to write out an automaton.

Working flow of BALM

BALM supports two different flows.

1. The first deals with FSMs where the fixed part *F* and specification *S* are given as FSMs. This flow is oriented towards solving for the unknown component *X* where $F \cdot X \subseteq S$ and takes advantage of the special features of *F*, *S*, and *X* to provide a very efficient solution. The flow is embodied in the command `solve_fsm_equ`.
2. The second flow deals directly with automata by reading in, manipulating and writing out automata in a file format `AUT`. In addition, an automaton can be extracted from a Boolean network in the `BLIF-MV` file format using the command `extract_aut`.

Automata, Languages and Their Manipulations

An **automaton** A is a five-tuple $(Q, \Sigma, \delta, q^0, F)$, where Q is a finite set of states, Σ is a finite alphabet, δ: $Q \times \Sigma \rightarrow 2^Q$ is the transition function (where 2^Q denotes the power set of Q), $q^0 \in Q$ is the start state, and $F \subseteq Q$ is the set of accepting states. A state is of deterministic transition if, under any input assignment, there is exactly one destination state. Otherwise, the state is nondeterministic. Moreover, a finite automaton is deterministic if all of its states are of deterministic transitions. Otherwise, it is nondeterministic.

An input **string** $\vec{\sigma} = (\sigma_1, \ldots, \sigma_n)$ with $\sigma_i \in \Sigma$, (or input sequence) is accepted by an automaton A if the set of destination states from q^0 under $\vec{\sigma}$ with respect to δ, denoted as $\delta(q^0, \vec{\sigma})$, has nonempty intersection with F. The set of all strings accepted by an automaton A forms the **language** or behavior of A, denoted as $L(A)$. (Languages of finite automata are known as **regular languages**, and can be described using **regular expressions**). In general, manipulations over languages can be accomplished in terms of manipulations over finite automata, and vice versa.

General manipulations

Given two finite automata $A_1 = (Q_1, \Sigma, \delta_1, q_1^0, F_1)$ and $A_2 = (Q_2, \Sigma, \delta_2, q_2^0, F_2)$, their **product automaton** describes their synchronized joint behavior under some input sequence. The product automaton $A_1 \cdot A_2 = (Q, \Sigma, \delta, q^0, F)$ of A_1 and A_2 can be constructed by defining

1. $Q = Q_1 \times Q_2$,
2. $\delta((q_1, q_2), \sigma) = (\delta_1(q_1, \sigma)$ for $(q_1, q_2) \in Q_1 \times Q_2$ and $\sigma \in \Sigma$,
3. $q^0 = (q_1^0, q_2^0)$, and
4. $(q_1, q_2) \in F$ if $q_1 \in F_1$ or $q_2 \in F_2$.

In BALM, a product automaton can be constructed using the command `product`.

It is a well-known fact that nondeterminism does not increase the expression power of finite automata. In fact, for any nondeterministic automaton $A = (Q, \Sigma, \delta, q^0, F)$, there always exists an equivalent deterministic one. To derive such an equivalent deterministic automaton $A' = (Q', \Sigma, \delta', q^{0\prime}, F')$ from A, one may apply the so-called **subset construction** as follows. Let

1. $Q' = 2^Q$,
2. $\delta'(q', \sigma) = \{q \in Q \mid q \in \delta(p, \sigma)$ for some $p \in q'\}$ for $q' \in Q'$ and $\sigma \in \Sigma$,
3. $q^{0\prime} = \{ q^0\}$, and
4. $F' = \{q' \in Q' \mid q' \cap \mathrm{F} \neq \varnothing\}$.

In BALM, command `check_nd` checks if an automaton is deterministic. A nondeterministic automaton can be determinized using command `determinize`.

Determinizing a nondeterministic automaton is a step usually performed before complementation since complementing a *deterministic* automaton $A = (Q, \Sigma, \delta, q^0, F)$ can be easily achieved by inverting the acceptance condition of states. That is, the complement of A is $\overline{A} = (Q, \Sigma, \delta, q^0, Q \backslash F)$. In BALM, an automaton A can be

complemented using command `complement` where a determinization is automatically performed if *A* is nondeterministic.

A state is incomplete if there is some input assignment, under which the next state transition is undefined (i.e., there is no next state under that input assignment). An automaton is said to be **incomplete** if it has at least one incomplete state. An incomplete automaton can be completed by adding a single non-accepting don't care state with a self-loop transition under any input assignment (a sink state), and directing all missing transitions from any state to this don't care state. Note that the automata before and after completion accept the same language. In BALM, an incomplete automaton can be completed using command `complete`. By default, this command adds a non-accepting don't-care state. In some applications, it is necessary to add an accepting don't-care state, which is done using command `complete -a`.

Specialized manipulations

An automaton *A* is **prefix closed** (so is its corresponding language) if any prefix of an accepting string in $L(A)$ is also in $L(A)$. A *deterministic* automaton can be trimmed to be prefix-closed by collapsing all non-accepting states into a single non-accepting "sink" state (no transition from this state to other accepting states) with a universal self-loop transition. BALM provides command `prefix` to trim an automaton to be prefix closed.

A state of an automaton is progressive with respect to a set of variables *U*, called *U*-progressive, if at least one of its next states under any valuation of *U* is accepting. An automaton is **progressive** with respect to *U*, if all of its states are *U*-progressive. An automaton can be trimmed to be progressive with respect to *U* by iteratively deleting states that are not *U*-progressive. BALM can trim an automaton to be progressive by command `progressive`. The number of inputs that are to be considered in U needs to be specified on the input line, e.g. `progressive -i 5`.

A finite state machine (an FSM) is a six-tuple $(Q, I, \Sigma, \Omega, \delta, \lambda)$, where Q is a finite set of states, $I \subseteq Q$ is the set of initial states, Σ and Ω are the sets of input and output alphabets, respectively, and $\delta: \Sigma \times Q \rightarrow Q$ (resp. $\lambda: \Sigma \times Q \rightarrow \Omega$) is the transition function (resp. the output function). Hence, an FSM (when converted to an automaton by combining inputs and outputs) differs from a typical automaton in that all states are accepting, input and output alphabets are differentiated. In addition, an FSM is prefix closed and input progressive. Given an automaton *A* with the specification of input and output variables, BALM is capable of trimming *A* to be an FSM. Moreover, BALM can further constrain the synthesized FSM to be of Moore type, that is, $\lambda: Q \rightarrow \Omega$ is independent of input Σ.

In composing two FSMs, it is sometimes necessary to rearrange (rename, reorder, create, or hide) input and output signals. BALM supports these rearrangements by command `support`. However, some of this is done automatically when for some of the commands. For example, `product A1 A2 A3` will automatically change the support of `A1` and `A2` to be the least common support and then create the product `A3` with that support.

Optimization

In BALM, a *deterministic* finite automaton can be state-minimized using command `minimize` based on Myhill-Nerode theorem. In addition to the above exact minimization, BALM provides a heuristic algorithm (command `dcmin`) for state minimization of a nondeterministic finite automaton, whose behavior, as a side effect, may be reduced thereafter, i.e. some of the nondeterminism may be used up in obtaining the result. A discussion of the use of `dcmin` can be found in the applications section of this manual.

Verification

Many verification problems in state-transition systems can be reduced to the checking of **language containment**, which tests if the language of one automaton is contained another. The checking can be accomplished by product and complement operations. In BALM, language containment checking is performed using command `contain` which will report if the two automata specified in the command line are related by language containment (or are equivalent), and optionally give counter-examples to non-containment.

Applications

Synthesis of Unknown Components

An important step in the design of complex systems is the decomposition of a system into a number of separate components, which interact in a well-defined way. Component-based design methodology plays an important role in facilitating design reuse. Design reuse is an essential technique in improving productivity for complex designs. A typical question is how to design an unknown component *X* that, when combined (in a way as shown in Figure 1) with a known (fixed) component *F*, satisfies specification *S* of the overall system, denoted as $F \cdot X \subseteq S$.

In [YVB+01], the solution to the above unknown component problem is formulated in language equations. In essence, the most general solution to the unknown problem can be written as the language equation $\overline{(F \cdot (\overline{S}_{\Uparrow i,v,u,o}))_{\Downarrow u,v}} \equiv \overline{F \bullet \overline{S}}$, where the overlines denote complementations, the upward $\Uparrow$ and downward $\Downarrow$ arrows denote support lifting and lowering, respectively, to the specified variables.

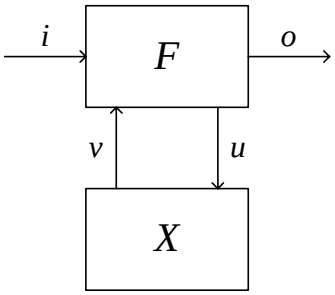

Figure 1. *The composition topology of a known component F and an unknown component X.*

As an application, we show that BALM can be used to derive the most general FSM solution to the unknown component problem by the algorithm in Figure 2, where each operation corresponds to some command in BALM.

Algorithm: ***LanguageEquationSolving***
Input: prefix closed *S*(*i*,*o*) and *F*(*i*,*v*,*u*,*o*)
Output: most general prefix closed solution *X*
begin
01 *X* := *Complete(S)*
02 *X* := *Determinize(X)*
03 *X* := *Complement(X)*
04 *X* := *Support(X,(i,v,u,o))*
05 *X* := *Product(Complete(F),X)*
06 *X* := *Support(X,(u,v))*
07 *X* := *Determinize(X)*
08 *X* := *Complete(X)*
09 *X* := *Complement(X)*
10 *X* := *PrefixClose(X)*

```
11      X := Progressive(X)
12      return X
end
```

Figure 2. *Algorithm for computing the most general prefix-closed progressive solution.*

In terms of the commands of BALM, this is illustrated assuming that the specification automaton is S.aut and the fixed automaton is initially given in the `BLIF-MV` file F.mv as a finite state machine.

```
balm> read_blif_mv F.mv
balm> extract_aut F.aut

balm> complement S.aut Sc.aut
balm> product F.aut Sc.aut P.aut
balm> support u(4),v(8) P.aut Ps.aut
balm> complement Ps.aut Pc.aut
balm> prefix Pc.aut Pp.aut
balm> progressive –i 1 Pp.aut X.aut
```

Here we have assumed that the *u* and *v* variables are single multi-valued variables with 4 and 8 values in their domains respectively. Note that in the support command, u was listed first, since in `progressive`, the input to the unknown component must come first.

Minimization of the most general solution. Command `dcmin` in BALM implements a minimization procedure DCMIN that works particularly well when the state of the fixed component *F* is communicated (through signals *u*) to the unknown component *X*. When the algorithm of language equation solving is done, there are a lot of input minterms *u* that do not exist (are unspecified) at a particular state of *X* since the state inputs from *F* usually have to agree with the internal state of *X*. This causes, during the solution process, many transitions to an accepting "don't care" state, usually named *DC1*. A don't care state is a sink and can be made equivalent to any other state (if it is accepting) by using its don't cares (see Figure 3).

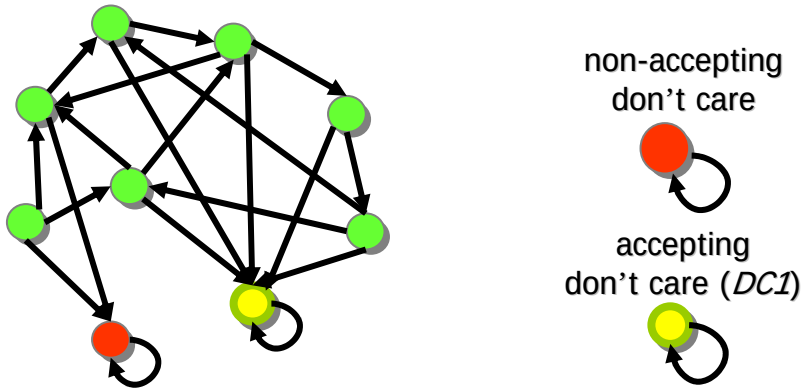

Figure 3. *The composition topology of a known component F and an unknown component X.*

Thus, any transition to *DC1* can be made to go to any state. We can use this property to make states equivalent to each other. A sufficient condition for this is that the set of transitions which are care transitions (i.e. go to some accepting state other than *DC1*) do not intersect. Thus if p_1, p_2, p_3 are the care transition predicates from states s_1, s_2, s_3 respectively, where $p_1 \wedge p_2 \wedge p_3 = \varnothing$, then s_1, s_2, s_3 can be made equivalent by using the don't cares to extend the care transitions of each state to $p_1 + p_2 + p_3$. This then makes all three of the states equivalent.

DCMIN works by building an incompatibility graph among the states. There is an edge $s_1 \rightarrow s_2$ if and only if $p_1 \wedge p_2 \neq \varnothing$. The choice, of which states to merge together into a single representative equivalent state, is made by finding a minimum coloring of the incompatibility graph.

The more state information communicated by *F* to *X*, the better DCMIN works. However, note that this is just one way of using the flexibility provided by the don't cares in the solution *X* and certainly may not be the best way. On the other hand, DCMIN is very fast and in practice seems to be quite effective. Other good methods of state minimization are either unknown or are very computationally intensive.

One way to understand why DCMIN works is the following. When the product machine is created during the solution phase to determine *X*, each state is a pair, one state from *F* and one state from the specification *S*. Although this product automaton is determinized and complemented to get *X*, the genesis of *X* starts out with an image of the states of the product. In general, we do not need to make this correspondence between states, and communicating the internal states of *F* does not make any difference here in *X*. However, when we use DCMIN, this information (correspondence between states) is used, hopefully to an advantage. We will provide an illustration of the use of DCMIN in the next application.

Synthesis of Winning Strategies for Combinatorial Games

Finding winning strategies of some combinatorial games, such as the NIM game, tic-tae-toe, the wolf-goat-cabbage puzzle, etc., can be formulated as solving the unknown component problem. Therefore, BALM can be used to synthesize winning strategies of these combinatorial games. There are several examples of these formulations in the accompanying directory of examples that is provided with BALM.

Computing Sequential Flexibilities

We illustrate this with an example where we start with a given FSM, *S*. We then divide it into two parts, calling the first part *F* and the second part *A*. We will then compute the maximum sequential flexibility for *A*. In command `latch_split` of BALM, we require that the language of the FSM $F \cdot X$ is contained in the language of the FSM *S* where *F* is a part of *S* containing a subset of its latches. We will use a running example, `planet.blif`, to illustrate this application. In *S* (which is `planet.blif`), the latches are `(v7,v8,v9,v10,v11,v12)`. Using the command, `latch_split 0-3`, the

first four latches are removed leaving the latches of F as `(v11,v12)`. When we compute the product of $F \cdot \overline{S}$ in solving for the solution of $F \cdot X \subseteq S$, the state space variables of the product corresponds to

`(v11,v12,v7,v8,v9,v10,v11,v12)`.

In general, values in the first, `(v11,v12)`, need not agree with those in the last, `(v11,v12)`. However, suppose we force them to agree. This sets up a simulation relation between the states of $F \cdot X$ and S, namely a state of $F \cdot X$, say `(0,0,sx)` is related to a state of *S*,

`sx =(-,-,-,-,0,0)`,

i.e. the values of `(v11,v12)` in *F* are forced to take the same values as `(r4,r5,r6)` in *S*. All other product states are not allowed. For example, a product state `(0,0,1,1,0,1,0,1)` can't exist because of this forced correspondence since the two copies of `(v11,v12)` disagree.

One way to effect this is to

1) expose the latches in *S* that correspond to the latches of *F*, i.e. make these latches primary outputs of *S*, and
2) make all the latches of *F* primary outputs of *F*.

In the computation for the solution *X*, the primary outputs of *F* and *S* are always forced to have the same values since they have the same name. These two steps can be done by a simple manual procedure. BALM has the command `latch_expose` which makes all latches in a file, primary outputs, in addition to the normal ones. For Step 1) this can be applied to *F*. Unfortunately, in Step 2) applying `latch_expose` to *S* exposes too many latches of *S* and we have to eliminate some. We give an example to illustrate the procedure to be used. In the example, `planet.blif` is *S*. The six latches of `planet.blif` are `(v6, …,v11)`.

```
read_blif_mv planet.blif
latch_split 0-3
latch_expose
write_blif_mv planet.mv

read_blif_mv planetf.blif
latch_expose
write_blif_mv planetf.blif
```

The command `latch_split` creates files `planetf.blif`, `planeta.blif`, and `planets.script`. These files reflect that all the latches of planet were made primary outputs. The fourth command writes the "exposed" planet network into a new file which we name `planet.mv`. The last three commands expose the latches of *F* as required by Step 2). Unfortunately, in file `planet.mv`, all the latches of *S* are exposed and not just the ones that remain in *F*, i.e. latches `v11` and `v12`. Now we have to edit two files that were made by this procedure to remove the excess primary outputs of *S* and reflect that *S* is not called `planet.mv`.

Now we look at `planet.mv` (which corresponds to *S*) and remove the outputs corresponding to the first four latches, `v7,v8,v9,v10`:

```
.inputs v0 v1 v2 v3 v4 v5 v6
.outputs v13.6 v13.7 v13.8 v13.9 v13.10 v13.11 v13.12 \
v13.13 v13.14 v13.15 v13.16 v13.17 v13.18 v13.19 v13.20 \
v13.21 v13.22 v13.23 v13.24 v7 v8 v9 v10 v11 v12
```

becomes

```
.inputs v0 v1 v2 v3 v4 v5 v6
.outputs v13.6 v13.7 v13.8 v13.9 v13.10 v13.11 v13.12 \
v13.13 v13.14 v13.15 v13.16 v13.17 v13.18 v13.19 v13.20 \
v13.21 v13.22 v13.23 v13.24 v11 v12
```

Finally, we edit `planets.script` and replace the second input file with `planet.mv` instead of `planet.blif`, since this is the new specification *S*:

```
solve_fsm_equ       planetf.blif       planet.blif       \
v0,v1,v2,v3,v4,v5,v6,v11,v12 v7,v8,v9,v10 planetxs.aut
```

We now look at the inputs and outputs of *S, F*, and *X*

`planet.mv`:
Primary inputs: `v0 v1 v2 v3 v4 v5 v6`
Primary outputs: `{v11} {v12} {v13.10} {v13.11} {v13.12} {v13.13} {v13.14} {v13.15} {v13.16} {v13.17} {v13.18} {v13.19} {v13.20} {v13.21} {v13.22} {v13.23} {v13.24} {v13.6} {v13.7} {v13.8} {v13.9}`

`planetf.blif`:
Primary inputs: `v10 v7 v8 v9 v0 v1 v2 v3 v4 v5 v6`
Primary outputs: `{v11} {v12} {v13.10} {v13.11} {v13.12} {v13.13} {v13.14} {v13.15} {v13.16} {v13.17} {v13.18} {v13.19} {v13.20} {v13.21} {v13.22} {v13.23} {v13.24} {v13.6} {v13.7} {v13.8} {v13.9}`

`planetxs.aut`:
Primary inputs: `v0,v1,v2,v3,v4,v5,v6,v11,v12,v7,v8,v9,v10`
Primary outputs: `cc`

Note that *F* has extra inputs that come from *X*, `v7,v8,v9,v10` (the *v* variables) and *X* has extra inputs that come from the latches of *F*, `v11,v12`, (the *u* varuables).

We are now set to take advantage of the simulation relation. We execute

```
source planetS.script
dcmin planetxs.aut planetxs-dcmin.aut
```

We need to use `dcmin` to minimize the result `planetxs.aut`:

```
dcmin planetxs.aut planetxs-dcmin.aut
```

The reason this will work well is that inputs to the solution automaton `planetxs.aut` must agree with the product state in the variables `v11,v12`. For example if the product state had 0 1 in these positions, then any input with `v11` = 1 or `v12` = 0 would be an input that would never occur, and its transition would be directed to the accepting don't care state. Hence, it can be used by `dcmin` in minimizing the result. Looking at the relative sizes we see

```
print_stats_aut planetxs.aut
"csf": incomplete (48 st), deterministic, non-progressive
(48 st), and non-Moore (48 st).
13 inputs (13 FSM inputs) 49 states (49 accepting) 120
trans
Inputs = { v0,v1,v2,v3,v4,v5,v6,v11,v12,v7,v8,v9,v10 }
```

and

```
print_stats_aut planetxs-dcmin.aut
"csf": complete, deterministic, progressive, and Moore.
13 inputs (13 FSM inputs) 13 states (12 accepting)  61
trans
Inputs = { v0,v1,v2,v3,v4,v5,v6,v11,v12,v7,v8,v9,v10 }
```

We note that it is possible to do the following procedure (without editing any files):

```
read_blif_mv planet.blif
latch_expose
latch_split 0-3
source planetS.script
dcmin planetxs.aut planet-dcmin.aut
```

but this would force the outputs of *X* to be aligned with the states of *S* and would overly constrain the solution. If this is tried on the example, the final solution `planet-dcmin.aut` would have 3 more states than with the edited version. Also, in checking the particular solution `planeta.blif`, we note that it has 16 states all of which are deterministic, and can't be state minimized, whereas `planetxs-dcmin` has 12 states all of which are non-deterministic (probably ND only in the outputs).

Appendix

Commands in BALM

The following list contains a one line summary of all the commands available in BALM.

Automata manipulation commands:

- `complement`: complement an automaton (a non-deterministic automaton will be automatically determinized first)
- `complete`: complete an automaton by adding a don't-care state
- `contain`: check language containment of two automata
- `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 an automaton
- `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

Automata viewing commands:

- `plot_aut`: visualize an automaton using DOT and GSVIEW
- `print_lang_size`: compute the number of I/O strings accepted by the maximum prefix-closed sub-automaton of an automaton
- `print_nd_states`: print information about non-deterministic states of an automaton
- `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 $F \square X \subseteq S$

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 non-deterministic 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

Bibliography

[BLIFMV] `BLIF-MV` manual.
`http://www-cad.eecs.berkeley.edu/~vis/usrDoc.html`

[FSA] FSA6.2XX: Finite State Automata Utilities.
`http://odur.let.rug.nl/~vannoord/Fsa`

[Grail] The Grail+ Project.
`http://www.csd.uwo.ca/research/grail`

[Graphviz] Graphviz - Graph Visualization Software.
`http://www.graphviz.org/`

[MiB03] A. Mishchenko and R. K. Brayton. A theory of nondeterministic networks. In *Proc. Int'l Conf. on Computer-Aided Design*, 2003.

[MBJ+05] A. Mishchenko et al. Efficient solution of language equations using partitioned representations. In *Proc. Design Automation and Test in Europe*, March 2005.

[Sip97] M. Sipser. *Introduction to the Theory of Computation*. PWS Publishing Co.,1997.

[YVB+01] N. Yevtushenko, T. Villa, R. K. Brayton, A. Petrenko, and A. Sangiovanni-Vincentelli. Solution of parallel language equations for logic synthesis. In *Proc. Int'l Conf. on Computer-Aided Design*, pages 103--110, 2001.

[YVB+03] N. Yevtushenko, T. Villa, R. K. Brayton, A. Petrenko, and A. Sangiovanni-Vincentelli. Compositionally progressive solutions of synchronous language equations. In *Proc. Int'l Workshop on Logic and Synthesis*, pages 148--155, 2003.

[YVB+05] N. Yevtushenko, T. Villa, R. K. Brayton, A. Petrenko, and A. Sangiovanni-Vincentelli. Sequential synthesis by language equation solving. Submitted to *IEEE Transaction on Computer-Aided Design, 2005*.