Journal of Universal Computer Science, vol. 3, no. 5 (1997), 689-701
submitted: 20/12/96, accepted: 20/5/97, appeared: 28/5/97 © Springer Pub. Co.

Model Checking for Abstract State Machines

Kirsten Winter
GMD FIRST
Rudower Chaussee 5, D-12489 Berlin, Germany
email: kirsten@first.gmd.de

Abstract: In this paper, we discuss the use of a model checker in combination with the specification method of Abstract State Machines (ASMs). A schema is introduced for transforming ASM models into the language of a model checker. We prove that the transformation preserves the semantics of ASMs and provide a theoretical framework for a transformation tool. Experience with model-checking the ASM model of the Production Cell demonstrates that this approach offers effective support for verifying ASM specifications.

Key Words: Formal Methods, Abstract State Machines, Model Checking, Tool Support, Requirement Specification, Verification

1 Introduction

Abstract State Machines (ASMs) [Gurevich 95] denote a formal method that allows us to write succinct and understandable specifications for a wide spectrum of applications. The process of specification and verification is guided by the notion of stepwise refinement. The developer can choose any level of abstraction to focus on specific aspects of the system. With ASMs, we have a formal method for practical use, as the list of contributions in [Börger 95] shows.

When using formal methods for developing safety-critical systems, one of the major goals is to verify that the safety requirements are satisfied. Interactive *Theorem provers* are adequate for showing the properties of infinite state systems, but their drawback is the often immense human effort needed for verification. *Model checkers* are based on a fully automated state exploration technique, but their application is restricted to systems with a not-too-large finite state space.

In the model-checking approach, *system models* are specified as automata or transition systems. The model checker verifies whether the given properties are satisfied by the system model. These *requirements* can be formalized as a temporal logic or as an automaton like the system model. Model checking can be done either by completely searching the state space of the system model and testing whether the logical formula is satisfied in every state, or by testing bisimulation of the two given automata.

One proposal to overcome the problem of state explosion is to provide a correctness-preserving reduction to an abstract model with a reduced state space [Clarke et al. 92], [Dingel, Filkorn 96], [Bharadwaj, Heitmeyer 97]. Using ASMs, we can start from the opposite direction by exploiting the possibility of modeling at arbitrary levels of abstraction: we start with an abstract ground model (in the sense of [Börger 95a]) to obtain an overview of the problem, and then refine – ideally in a provably correct way – the model in several steps in order to get more concrete versions. In this process, which characterizes the ASM method, we look for an adequate level of abstraction, where the model operates in a state space that can be handled by the model checker, and in which we are able to express the requirements that are to be checked.

We explain this novel use of model-checking ASMs using the example of the Production Cell. We have chosen the SMV model checker that offers the language facilities needed to simulate ASM models. We transform the ASM model of the Production Cell [Börger, Mearelli 97] into the language of SMV. (We chose the refined model of the Production Cell which also operates in a finite state space and allows us to formalize all safety and liveness properties.) The required safety and liveness properties [Lewerentz, Lindner 95a], [Börger, Mearelli 97] are formalized in a temporal logic. The resulting system model can be checked against the formalized requirements automatically.

In Section 2, we explain our choice of the SMV model checker. Section 3 describes the transformation of the ASM rules and the structure of the system model. In Section 4, we describe the (language for the) requirement specification and give the results of model-checking the requirements of the Production Cell. We conclude with a discussion of some related research and a look at possible future work.

2 The Choice of SMV for ASMs

We decided to use SMV [McMillan 93], because like ASMs it is based on transition systems and on states that are denoted by the values of the state variables. Internally SMV treat transitions symbolically as binary decision diagrams (which provide very efficient algorithms), but this representation is hidden at the level of the description language. The SMV checks a temporal logic formula against the system specification and outputs a counterexample if the system fails to meet this requirement.

An SMV system model is built as follows. After declaring the state variables, the conditions for the initial state are given. This is followed by the definition of rules for assignments and conditions for transition behavior. Then the requirement specification (in our case, a set of safety and liveness properties) is given in terms of the computation-tree logic CTL. For the sake of structuring, one can divide the system into modules. As semantics for executing the modules we can choose *interleaving* concurrency or *simultaneous* execution. Within one module, all enabled assignments and transitions are executed simultaneously. The communication between different modules has to be modeled using *global variables*.

One drawback of the approach, however, is that the structure of the ASM transition rules has to be transformed to match the SMV language and may be changed significantly. Moreover, the report facilities of SMV are limited. It is not possible to inspect the possible computation paths if no failure is found. An additional simulation tool for analyzing behavior would be helpful.

3 The System Model

In this section, we introduce schemas for effective coding from ASM rules into the SMV language that preserve the semantics. This yields a basis for a compilation algorithm supporting the transformation task.

It is well known that the execution time of the model-checking algorithm of the SMV depends significantly on the order of the variables. At the end of this section, we describe how to find an adequate *variable ordering*.

3.1 The Transformation from an ASM Model to a SMV Model

For our case study, we use the Production Cell [Lewerentz, Lindner 95a] modeled with ASMs [Börger, Mearelli 97]. We restrict our investigation to simple update rules, sequential and conditional rules [Gurevich 95], because no other constructs are needed for this example. Further work has to be done to obtain transformation schemas for other rule constructs of ASMs.

3.2 Transformation Schema for an Update Instruction

Following [Gurevich 95], the semantics of an *update instruction* $R: \; f(\bar{t}) := t_0$ is that the current value of t_0 is mapped to the function f at the current value of $\bar{t}$, i.e. it is given formally by firing its update $\alpha = (l, y)$ at the current state of the system, where $l = (f, \mathit{Val}_S(\bar{t}))$, $y = \mathit{Val}_S(t_0)$ and Val_S is the evaluation function with respect to the current state S. At all other argument values, the function itself and all other functions remain unchanged. We confine our attention to updates that map a location into a finite range.

The SMV language deals with variables over a finite domain, which we use to denote the locations l of the updates. Updating of a variable can be done by the next operator $\mathbf{next}(\cdot)$, which describes how the value of the variable will be changed in the next state. If no transition behavior is specified, a variable behaves nondeterministically. To ensure that a variable will not be changed, we have to state this explicitly. Otherwise the variable can take any value from its range (i.e. for each value there exists a computation path in the internal computation tree).

The next operator can be used in SMV within an assignment or to describe the transition behavior invariantly. Here, we choose the following schema:

$$\mathbf{f}(\bar{\mathbf{t}}) := \mathbf{t_0}$$

$$\Downarrow$$

$$\mathbf{ASSIGN}\ \mathbf{next}(l) := y$$

with the location $l = (f, \mathit{Val}_S(\bar{t}))$ and the value $y = \mathit{Val}_S(t_0)$

The semantics for SMV programs is given as a function that maps a program fragment p onto a formula $[|p|]$ as its denotation. Since an expression is a formula, its semantics is denoted by itself: $[|e|] = e$. It is a peculiarity of SMV that expressions are implicitly treated as nondeterministic. They evaluate to sets of possible values. A constant is mapped to a singleton.

The semantics of the assignment of the next operator is given in [McMillan 93] as follows:

$$[|\mathbf{ASSIGN}\ \mathbf{next}(l) := y\ |]_R = (\ l' \in (\mathbf{running} \rightarrow ([|y|], l))\ )$$

where l' is the value of l in the next state. $\rightarrow$ stands for the *if-then-else* operator, which has the intuitive meaning that, if the value of the left side is true, then the expression evaluates to the first element on the right side, or else to the second element. We can read the semantics of assignments as follows: if the module to which the statement belongs is running, l' evaluates to an element of the set of possible values of the expression y, otherwise it remains as it is.

In the semantics of ASMs, it is intended that every module be active, i.e. running is always true, and (for sequential ASMs without parallelism) y is a constant. Therefore, the schema for transforming ASM updates into SMV assignments is semantically correct.

3.3 Transformation Schema for Sequences and Conditional Rules

The semantics of a guarded transition rule in ASMs is given by the update, which should be fired when the rule is executed. The semantics of firing the rule R : **if** $\mathtt{g_0}$ **then** $\mathtt{R_0}$ **elseif** $\mathtt{g_1}$ **then** $\mathtt{R_1}$ ... **elseif** $\mathtt{g_k}$ **then** $\mathtt{R_k}$ **endif** is that whenever g_i holds in a state S, and for all $j < i$, g_j does not hold, the update to fire is given by the rule R_i. In [Gurevich 95], the semantics of R is given in terms of update sets: $Update(R, S) = Update(R_i, S)$ if g_i is the first guard that holds, and $Update(R, S) = \emptyset$ if none of the guards is true in S (i.e. nothing should be done).

If the inner rules R_i are update rules for the same location, it suffices to expand the transformation schema for update instructions to a transformation schema that is guarded. In an SMV model, guarded updates can be specified with a **case** expression within an assignment. A case expression returns the value of the first expression on the right-hand side, of which the corresponding condition on the left-hand side is true. The last branch should always be guarded with *true* (i.e. 1) to indicate the default case. This is used to describe explicitly that the variable will remain as it is if the transition rule will not fire, which is the desired semantics within ASMs. This is exactly the semantics of executing a guarded transition rule in ASM, if only one location is the subject of all updates R_i. We use the schema:

if $\mathtt{g_0}$ **then** $\mathtt{R_0}$ **elseif** $\mathtt{g_1}$ **then** $\mathtt{R_1}$ ... **elseif** $\mathtt{g_k}$ **then** $\mathtt{R_k}$ **endif** with $\forall i \;\; 0 \leq i \leq k$: $R_i == f(\bar{t}) := t_i$	$\Rightarrow$	**ASSIGN next** (l) := **case** $g_0 : y_0$; $g_1 : y_1$; $\vdots$ $g_k : y_k$; $1 : l$; **esac** ;

where location $l = (f, Val_S(\bar{t}))$and values $y_i = Val_S(t_i)$ for all i.

To enable this schema to be used for conditional transition rules in general, we have to modify the transition rules in two steps. First, we have to flatten the structure of the ASM rule; and second, we have to reorder these simple derived rules and assemble the rules which update the same location.

1. Every if-then-elseif construction containing nested rules has to be broken down into a flat sequence of simple if-then-else rules with only update instructions inside. We simplify the if-then-else rules to if-then structures and duplicate the rules (or the guards of the rules) in order to get one rule for each update.
   To be more precise, we are obliged to take the following steps to transform a nested conditional transition rule into the required flat structure:

$$\begin{array}{c} \textbf{if } \mathtt{g_0} \textbf{ then } \mathtt{R_i} \textbf{ else } \mathtt{R_j} \\ \text{with } R_i = R_{i_1}, \ldots, R_{i_n} \text{ and } R_j = R_{j_1}, \ldots, R_{j_m} \\ \text{and } \forall p \;:\; 1 \leq p \leq n \;:\; R_{i_p} == f_q(\bar{t_p}) := t_{0_p} \\ \text{and } \forall q \;:\; 1 \leq q \leq m \;:\; R_{j_q} == f_q(\bar{t_q}) := t_{0_q} \end{array}$$

$$\Downarrow$$

$$\textbf{if } \mathtt{g_0} \textbf{ then } \mathtt{R_i} \;,\; \textbf{if } \neg\mathtt{g_0} \textbf{ then } \mathtt{R_j}$$

$$\Downarrow$$

$$\begin{array}{c} \textbf{if } \mathtt{g_0} \textbf{ then } \mathtt{R_{i_1}} \;,\; \ldots \;, \textbf{if } \mathtt{g_0} \textbf{ then } \mathtt{R_{i_n}}, \\ \textbf{if } \neg\mathtt{g_0} \textbf{ then } \mathtt{R_{j_1}} \;,\; \ldots \;,\; \textbf{if } \neg\mathtt{g_0} \textbf{ then } \mathtt{R_{j_m}} \end{array}$$

   The semantics of a sequence of transition rules $R = R_1, \ldots, R_n$ is given by the conjunction of the respective update sets $\mathit{Update}(R_1, S) \cup \ldots \cup \mathit{Update}(R_n, S)$. In our case, we have the update set $\mathit{Update}(R_{i_1}, S) \cup \ldots \cup \mathit{Update}(R_{i_n}, S) \cup \mathit{Update}(R_{j_1}, S) \cup \ldots \cup \mathit{Update}(R_{j_m}, S)$. Each of the subsets is a singleton and guarded by g_0 or $\neg g_0$, respectively. Whenever g_0 is true, the update set is given by $\mathit{Update}(R_{i_1}, S) \cup \ldots \cup \mathit{Update}(R_{i_n}, S)$. Whenever g_0 is false, the update set is given by $\mathit{Update}(R_{j_1}, S) \cup \ldots \cup \mathit{Update}(R_{j_n}, S)$. We can see that the semantics is preserved by the transformation.
2. In the next step, we search for all update rules that map the same location to various values, i.e. for all rules R_i with update set $\mathit{Update}(R_i, S) = \{(l, \cdot)\}$, where l is the location of the update. This set of updates at location l can be transformed into the SMV language in the following way:

$$\begin{array}{ccl} & & \textbf{ASSIGN next } (l) := \\ & & \qquad\qquad \textbf{case} \\ \textbf{if } \mathtt{g_0} \textbf{ then } \mathtt{R_0} \;, & & \qquad\qquad\quad g_0 : y_0 \;; \\ \vdots & \Rightarrow & \qquad\qquad\quad \vdots \\ \textbf{if } \mathtt{g_h} \textbf{ then } \mathtt{R_h} & & \qquad\qquad\quad g_h : y_h \;; \\ & & \qquad\qquad\quad\; 1 : l \;\;\; ; \\ & & \qquad\qquad \textbf{esac} \;; \end{array}$$

   where for all $R_i,\ 0 \leq i \leq h,$ the update location is some $l = \mathit{Val}_S(f(\bar{t}))$, and the value is some $y_i = t_i$, which may differ for different indices i.

Similar to the simple next operation, the semantics of an assignment containing a case block is given by

$$\left[\left|\begin{array}{l} \textbf{ASSIGN next } (l) := \\ \quad \textbf{case} \\ \qquad g_0 : y_0 \ ; \\ \qquad \quad \vdots \\ \qquad g_h : y_h \ ; \\ \qquad \ 1 : l \quad ; \\ \quad \textbf{esac} \ ; \end{array}\right|\right]_R = \begin{array}{r} g_0 \rightarrow (\ (l' \in (\textbf{running} \rightarrow ([|y_0|], l))), \\ g_1 \rightarrow (\ (l' \in (\textbf{running} \rightarrow ([|y_1|], l))), \\ g_2 \rightarrow \ldots \\ \ldots, g_h \rightarrow (\ (l' \in (\textbf{running} \rightarrow \\ ([|y_h|], l))), \\ (l' = l))\ldots) \end{array}$$

If g_i is the first condition that evaluates to true, then the value of l' is a member of the set of possible values of $[|y_i|]$ whenever the module is running. With deterministic evaluation of values y_i and permanently running modules, we have l' evaluating to y_i whenever g_i is true.

With the proposed transformation steps we get a set of assignments with the structure shown above. Under the appropriate conditions, each of them assigns a new value to one location only. We can see that this set has exactly the same semantics as the initially given ASM rule.

3.4 Transformation Schema for ASM Modules

As mentioned above, the SMV language allows modules to be used as an encapsulated collection of declarations that may depend on specific parameters. They have to be instantiated within the calling module (i.e. the parent). The variable **running** is an implicit parameter of an instance of a module, which is inherited by the **running** variable of its parent. Hence, a module is always running when the parent module is. We get the semantics of *true concurrency* in the sense of *simultaneous execution*. On the other hand, if we instantiate a module with the keyword **process**, then the semantics is that of interleaving concurrency: no two modules with the same parent run at the same time.

In ASMs, modules and all the transition rules inside a module are running concurrently. We transform ASM modules into SMV modules that are instantiated without the keyword **process**, which yields the right semantics of simultaneous execution.

We also borrow some ideas from the schema of Havelund and Shankar [Havelund, Shankar 95]. To be able to simulate the communication with *global variables*, we need a module *state* consisting of the declaration of all variables that are used in common. Since, in the case of the Production Cell, the naming of the functions is unique (with the exception of *bottomPos* and *topPos* that we rename for the press module), we chose to declare *all* variables in the module *state* – not only the common variables. This means that only one parameter is given to all modules: the module *state*. (If we decide to decompose more complex systems in order to check the modules separately, we certainly have to distinguish between global and local variables of the ASM model.) We preserve the module structure as given in the ASM model. For the Production Cell, we get six modules that describe the transitional behavior of the respective agents in the ASM model. Finally, a module *main* is used to instantiate all modules. We get the following schema for the overall SMV system model:

```
MODULE state
VAR  ...  (declaration of all variables)
INIT  ...  (initialization of all variables)

MODULE FeedBelt (s)
ASSIGN  ...; (assignments of FeedBelt)
            ⋮
MODULE DepositBelt (s)
ASSIGN  ...; (assignments of DepositBelt)

MODULE main
VAR
     s   : state;
     FB  : FeedBelt (s);
           ...
     DB  : DepositBelt (s);
SPEC  ... (CTL formula as system specification)
```

With this transformation schema, the module structure of the ASM program is preserved. All modules are running whenever the module *main* or the system is running.

By way of an example, Figure 1 shows a fragment of the SMV model of the Production Cell. (In the SMV code, we shorten the variable names and remove the macro definitions as given in [Börger, Mearelli 97].)

```
MODULE DepositBelt (s)
ASSIGN
    next(s.Critical):=
      case
      (s.DBMot=run & !s.Critical) & s.PieceInLightBarr : 1;
      (s.DBMot=run & s.Critical) & !s.PieceInLightBarr : 0;
       1 : s.Critical;
      esac;
    next(s.DBMot):=
      case
       (s.DBMot=run & s.Critical) & !s.PieceInLightBarr : idle;
       (s.DBMot=idle) & !s.PieceAtBeltEnd : run;
       1 : s.DBMot;
      esac;
    ...
```

Figure 1: The Module Deposit Belt in the SMV Production Cell

3.5 The Variable Ordering

As stated above, the SMV description of the system is treated as a Boolean formula. Internally, a graph representation as binary decision diagrams (BDDs) is used, which is *reduced* and *ordered* as well (we get ROBDDs). To construct this graph, the tool sequentially looks at each variable of the formula. The order in which these variables are evaluated determines the size of the ROBDD.

In ROBDDs, transition systems are represented as a conjunction of several transition conditions (or assignments). In order to make the graph structure as small as possible, we assemble the variables as they occur in the conjunction such that single conjunctive terms can be evaluated completely in their respective sequence. The evaluation thus can be aborted as soon as one conjunctive term evaluates to false.

As a consequence, we have to assemble the variables in the order they occur in the assignments. Using this simple heuristic, we made good progress in the case study. This procedure could be automated for all ASMs.

4 Specification of the System Requirements

In our experiments with the ASM model of the Production Cell, we first consider the *safety properties* of the system as listed in [Börger, Mearelli 97] (Section 4.2) and then the *liveness property* (Section 4.3). Furthermore, we show how model checking can be used to support manual proving by checking particular lemmas like the *Agent Progress Lemma* as introduced for the liveness proof in [Börger, Mearelli 97] (Section 4.4).

We formulate the system requirements in the temporal logic CTL, which is based on atomic formulas that express information about states using state variables and Boolean connectors. In addition, we need only the following path quantifiers:

- $\mathsf{AG}\varphi$ (φ holds in *every* reachable *state* of every path (i.e. always)),
- $\mathsf{AX}\varphi$ (in all reachable *next states* of every path, the formula φ holds) and
- $\mathsf{AF}\varphi$ (in every path *eventually* φ holds).

4.1 The Safety Properties of the Production Cell

In the ground model, the safety properties are based on several assumptions concerning the system. In the concrete model, these assumptions are used to refine the behavior of the abstract model. We therefore choose the refined ASM model as the system description we want to check. Continuous intervals as for the angle values were treated as a finite set of discrete values, because "only a finite number of the real values are relevant" [Börger, Mearelli 97], (Section 4.3).

The safety properties of the Production Cell are given as properties of its components, but the complexity of the system allows us to check the system as a whole. We specify each property as a single formula and form the conjunction of this set of formulas in order to get the requirement specification.

By way of an example, we discuss here the safety property of the press. The example shows how dependencies between the behavior of several modules can be checked easily. We have the following two safety requirements for the press:

1. *The press is not moved downward if it is in the bottom position; it is not moved upward if it is in the top position*
   This is easily described using the state variables that model the motion of

the press motor and the oracle functions for the position of the press: if the bottom position is reached, the press motor should not move downward; similarly, if the top position is reached, the press motor should not move upward.
We have to bear in mind that the condition of reaching the top or bottom position is also a condition for switching the motor to idle. Thus we profit by the next step quantification of CTL stipulating that whenever the boundary is reached in *all possible next states*, the motor should not move the press in this direction.[1] We get:

$$\begin{array}{ll} \mathsf{AG}\ \ (bottomPositionPress & -> \ \ \mathsf{AX}\ \ (PressMot \neq down)) \\ \wedge\ \mathsf{AG}\ \ (topPositionPress & -> \ \ \mathsf{AX}\ \ (PressMot \neq up)) \end{array}$$

2. *The press only closes when there is no robot arm inside it.*
The press is closing/closed when the motor is running *up*. In this case, neither arm 1 nor arm 2 should ever be in the press. In terms of the ASM press model, arm i ($i = 0, 1$) is in the press if the angle has the value *ArmiToPress* and the arm is completely extended. This leads to the following formula:

$$\begin{array}{l} \mathsf{AG}\ \ (PressMot = up \\ \quad -> \ \ \neg(Arm1Ext = Arm1IntoPress\ \wedge\ Angle = Arm1ToPress) \\ \qquad \wedge\ \neg(Arm2Ext = Arm2IntoPress\ \wedge\ Angle = Arm2ToPress)) \end{array}$$

We have similar simple formulas for the safety requirements of the other agents. The behavior of the oracle functions (that determine the behavior of the modules) has to be specified to reflect the *Cell Assumption* that requires a reasonable system environment.

The model checker concluded that the conjunction of all safety formulas is satisfied by the SMV model of the Production Cell. The output shows the resources used:

```
-- specification AG (s.FBM = on & s.deliv & !s.pfl → !s.... is true
resources used:
user time: 3.8 s, system time: 1.03333 s
BDD nodes allocated: 43529
Bytes allocated: 18087936
BDD nodes representing transition relation: 25718 + 1
```

4.2 The Liveness Property of the Production Cell

We formalize the liveness property, reflecting the whole process as a sequence of different steps, and check that always the next action of this sequence (in the sense of the Blank Progress Lemma in [Börger, Mearelli 97], see below) will eventually be executable.

We split the cyclic process into subprocesses at the points where "blanks can progress" owing to the actions of the cell components. For the liveness property, it suffices to show that every blank is forwarded by every component of the cell

[1] In the real-life system, we have to require that the boundary be set with a sufficient margin in order to be able to stop the motor at a safe distance to prevent a collision.

in the regular sequence: it is transported by the feed belt to the elevating rotary table, then moved by the robot to the press, and from there to the deposit belt, where the traveling crane puts it back on the feed belt. (This is what is stated in the Blank Progress Lemma [Börger, Mearelli 97].)

The presence and location of "blanks" in the system is formalized in terms of the ASM model notions, e.g. a blank is transported from the beginning of the feed belt to the end when the feed belt changes from *NormalRun* to *CriticalRun*. (By the Cell Assumption, it is guaranteed that within these phases a blank will be passed through.) We define CTL formulas like

$$\mathsf{AG}\ \ (action\_executable\ \ -> \ \ \mathsf{AF}\ \ (next\_action\_executable))$$

which are satisfied if it is always (in every path in every state) the case that once *action* is executable, then eventually *next_action* will be executable. Following the Blank Progress Lemma, we consider all pairs of blank-forwarding actions and their successor blank-forwarding actions. The liveness property is expressed by the conjunction of all these "one-step-liveness" formulas.

Moreover, to ensure that *action_executable* will be reached at all, we have to verify that *eventually the action will be executable*: (AF *action_executable*) has to be checked for every first action in the set of action pairs.

Integrating the Agent Progress Lemma used in [Börger, Mearelli 97] in the proof of the Blank Progress Lemma, we obtain the following sequence of actions characterizing the control-critical moments in the life of a blank in the Production Cell: $FeedBelt$ is in $NormalRun$, $FeedBelt$ is in $CriticalRun$, $Table$ is *stopped_in_load_position*, $Table$ is *stopped_in_unload_position*, *Robot* has *unloaded_the_table*, *Press* is $OpenForLoading$, *Robot* has *loaded_the_press*, *Press* is $ClosedForForging$, *Press* is $OpenForUnloading$, *Robot* has *unloaded_the_press*, *Robot* has *loaded_the_DepositBelt*, *DepositBelt* is in $NormalRun$, *DepositBelt* is in *CriticalRun*, $Crane$ has *unloaded_the_DepositBelt*, $Crane$ has $loaded\_the\_FeedBelt$.

We checked our specification of liveness against the SMV model of the concrete system of the Production Cell. We found that the system satisfies the property if (and only if) the initial condition guarantees that there are at least two blanks in the system. (This has to do with the order of the robot actions; see the discussion in [Börger, Mearelli 97].) Since we do not model an operator that puts new blanks on the feed belt, we have to change the initial condition in order to get more than one blank within the system (see the Insertion Priority Assumption and the proof of the Strong Performance Property in [Börger, Mearelli 97]). We have determined that initially $FeedBeltMot$ is $idle$, $FeedBeltFree$ is false, $PieceInFeedBeltLightBarrier$ is true, Robot is $WaitingInUnloadTablePos$, $TableLoaded$ is true, $PressLoaded$ is true, $DepBeltMot$ is idle, $DepositBeltIsReadyForLoading$ is false, $PieceAtDepBeltEnd$ is true, and $CraneMagnet$ is *on*. This initial condition models the situation that seven blanks are in the Production Cell. If we add more blanks (e.g. by setting $Arm1Mag$ to *on*), the model checker finds a deadlock.

With respect to the changes listed above and a complete model of the system environment, we get the following output:

```
-- specification AG (s.FBM = on & !s.deliv → AF (s.FBM =... is true
resources used:
user time: 10.1333 s, system time: 0.95 s
BDD nodes allocated: 56832
Bytes allocated: 18350080
BDD nodes representing transition relation: 25718 + 1
```

We conclude that the system model satisfies the liveness property as claimed in the problem definition.

4.3 The Agent Progress Lemma

Börger and Mearelli [Börger, Mearelli 97] introduce the **Agent Progress Lemma** in order to prove the liveness of the system. Proof of this lemma requires inspection of the rules. We suggest supporting the user with the model checker in order to check the interactive behavior of the overall system.

The Agent Progress Lemma is split into six parts. Each part describes the progress of one of the agents with respect to the values of the interface functions. We formalize each of the parts in one CTL formula. The conjunct of these formulas gives the formalization of the progress requirement of the agents.

Model-checking the formulas of progress verifies that the SMV model satisfies the progress requirements if we choose the same initial condition as described in the last section. We conclude that the Agent Progress Lemma holds for the given system model.

5 Related Work and Conclusion

Initially, model checkers were developed to verify circuits specified as Boolean functions. More recently, the area of applications has been extended to consider more generally the verification of embedded or concurrent systems. A combination of model checkers and more sophisticated specification languages (as compared with Boolean functions) is needed. In this context, several authors have discussed the transformation from an operational specification language to the language of a chosen model checker. [Day 1993] uses the Voss model checker to apply model checking to Statecharts. [Grahlmann, Best 93] present a system environment for developing programs with Petri nets that includes a model checker. Bharadwaj and Heitmeyer report on their experiences in combining SCR with the model checker SPIN in [Bharadwaj, Heitmeyer 97].

In the context of the ASM methodology, our work is related to the "Abstract State Machine/Virtual Architecture" (ASM/VA, formerly known as EAM) by Del Castillo, Durdanović and Glässer [Del Castillo et al. 96]. With their ASM-based specification and design environment, they also address the scope of tool support. Among other things, they supply the developers with a simulation tool for ASMs. Compared with the ASM/VA, the SMV model checker provides very limited simulation facilities. The user will see only one computation path if a counterexample is reported in the case of failure. On the other hand, it is not possible to use the ASM/VA for automated verification of system requirements.

In this paper, we evaluated the model-checking approach using the given ASM model of the Production Cell [Börger, Mearelli 97]. It was to be expected that additional effort would be needed for formalization. The informal natural-language descriptions of the safety and liveness properties had to be transformed into temporal-logic formulas. In particular, the behavior of the environment had to be defined, because otherwise every possible behavior would be examined by the model checker, yielding counterexamples that are not relevant assuming a well-behaved environment.

To verify safety and liveness properties of the ASM Production Cell, we first transformed the ASM model into a SMV model by applying the transformation steps described above. We proved (in Section 3) that the transformation steps preserve the semantics of ASMs. We specified the behavior of the environmental variables (i.e. the oracle functions). We modeled the safety and liveness requirements as CTL formulas and checked them against the derived SMV model. The results show that safety and liveness are satisfied by the SMV model. From the correctness proofs of the transformation steps, we conclude that the ASM model is correct with respect to the requirements under consideration.

In our concept of transforming an ASM model into the description language of the model checker SMV, we break down the structure of the transition rules. Only simple updates or guarded rules that update one location can be expressed by the language of the SMV. Some of the transformation steps can be carried out automatically, as the given schemas show. For a complete model, however, the transformation has to be guided by the user in order to handle the remaining formalizations. In future work, we have to develop schemas for the remaining rule constructors that were not used in the ASM model of the Production Cell. A compilation algorithm should be implemented for automatic transformation; it could be enhanced with interaction facilities for user guidance.

To cover all proof obligations, the combined use of a model checker and a theorem prover would be interesting. Dingel and Filkorn [Dingel, Filkorn 96] suggest a procedure of this sort. Also, a more powerful simulation facility would help to analyze system behavior if it fails to meet the requirements. To provide adequate tool support, a combination of the model checker SMV and the ASM/VA [Del Castillo et al. 96] might be fruitful.

Acknowledgments My special thanks are due to E. Börger and L. Mearelli for encouraging my work on their ASM model of the Production Cell. I also wish to thank S. Herrmann and J. Burkhardt for discussing with me the problems that had to be solved, as well as T. Santen, A. Sodan and C. Sühl for their helpful comments.

References

[Bharadwaj, Heitmeyer 97] Bharadwaj, R., Heitmeyer, C.: "Verifying SCR Requirements Specification Using State Exploration"; Proc. First ACM SIGPLAN Workshop on Automatic Analysis of Software, Jan., (1997).

[Börger 95] Börger, E.: "Annotated Bibliography on Evolving Algebras"; in Börger, E. (Eds.): "Specification and Validation Method"; Oxford University Press, (1995), 37-51.

[Börger 95a] Börger, E.: "Why Use Evolving Algebras for Hardware and Software Engineering?", Bartosek, M., Staudek, J., Wiedermann, J. (Eds.), SOFSEM'95; LNCS 1012, (1995), 236-271.

[Börger, Mearelli 97] Börger, E., Mearelli, L.: "Integrating ASMs into the Software Development Life Cycle"; contribution in this volume.

[Burch et al. 92] Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwan, L.J.: "Symbolic model checking 10^{20} states and beyond"; Information and Computation, 98, 2 (1992), 142-170.

[Bryant 86] Bryant, R.E.: "Graph-Based Algorithms for Boolean Function Manipulation"; IEEE Transactions On Computers, C-35, 8, (1986).

[Clarke et al. 92] Clarke, E., Grumberg, O., Long, D.: "Model-Checking and Abstraction"; Proc. 19th ACM Symposium on Principles of Programming Languages, ACM Press, (1992), 343–354.

[Chang et al. 92] Chang, E., Manna, Z., Pnueli, A.: "The Safety-Progress Classification"; Dep. of Comp. Science, Stanford Univ., STAN-CS-92-1408, (1992).

[Day 1993] Day, N.: "A Model Checker for Statecharts (Linking CASE Tools with Formal Methods)"; Tech.Report 93-35, Dep. of Computer Science, Univ. of British Columbia, Vancouver, B.C., Canada, (1993).

[Del Castillo et al. 96] Del Castillo, G., Durdanović, I., Glässer, U.: "An Evolving Algebra Abstract Machine"; Proc. CSL'95, LNCS 1092, (1996), 191-214.

[Dingel, Filkorn 96] Dingel, J., Filkorn, T.: "Model checking for infinite state systems using data abstraction, assumption-commitment style reasoning and theorem proving"; Proc. CAV '95, LNCS 939, (1996), 54-69.

[Grahlmann, Best 93] Grahlmann, B., Best, E.: "PEP – More than a Petri Net Tool"; Procs. of TACAS'96, Springer LNCS 1055, (1996).

[Gurevich 95] Gurevich, Y.: "Evolving Algebras 1993: Lipari Guide"; E. Börger (Eds.): "Specification and Validation Methods"; Oxford University Press, (1995).

[Havelund, Shankar 95] Havelund, K., Shankar, N.: "Experiments in Theorem Proving and Model Checking for Protocol Verification"; SRI International Menlo Park, Report, USA (1995).

[Lewerentz, Lindner 95a] C. Lewerentz, T. Lindner: "Formal Development Of Reactive Systems. Case Study "Production Cell""; Springer LNCS 891 (1995), 9-21.

[Long 93] Long, D.,E.: "Model Checking, Abstraction and Compositional Verification"; CMU Report, USA (1993).

[McMillan 93] McMillan, K.: "Symbolic Model Checking"; Kluwer Academic Publishers, Boston (1993).