Encoding Abstract State Machines in PVS

Angelo Gargantini[1] and Elvinia Riccobene[2]

[1] Dipartimento di Elettronica e Informazione - Politecnico di Milano -
`gargantini@elet.polimi.it`
[2] Dipartimento di Matematica e Informatica - Università di Catania -
`riccobene@dmi.unict.it`

Abstract. In this paper we show how the specification and verification system PVS (*Prototype Verification System*) can provide tool support for *Abstract State Machines* (ASMs), especially oriented towards automatic proof checking and mechanized proving of properties. Useful templates are presented which allow encoding of ASM models into PVS without any extra user's skill. We prove the transformation preserves the ASM semantics and provide a framework for an automatic tool, prototypically implemented, which translates ASM specifications in PVS. The ASM specification of the Production Cell given in [BM97] is taken as case study to show how to formalize multi-agent ASMs in PVS and prove properties.

1 Introduction

The *Gurevich's Abstract State Machines* (ASMs) have been successfully used for design and analysis of complex hardware/software systems [Bör99,Bör95]. Through real case studies, ASMs have shown to be a practical method for rigorous system development and to meet the requirements, addressed by Heitmeyer in [Hei98], that formal methods need to have "to be useful to practitioners": a user-friendly notation, useful, easy to understand feedback, integration into standard development process. ASMs use only the standard language and standard methods of programming and mathematics. They are models easy to read, understand and inspectable by the customer, but they are flexible to change, and precise and complete to match the designer's requirements.

However, we believe that the success of a formal method in industrial field also depends on the availability of related automated tools helping the user during the process of specification and subsequent verification and validation. Therefore, machine support for the use of ASMs is extremely useful.

During the last years various investigations have been started in order to verify standard mathematical reasoning about ASMs by interactive or fully automated proof tools. Some encouraging results are already reported in literature (see discussion in [Bör99]). Among theorem provers, PVS (*Prototype Verification System*) has been used in [DGVZ98] to show the correctness of bottom-up rewriting specification for back-end compilers from intermediate language into binary RISC processor code. Dold et al. state that "erroneous rules have been found using PVS and, through failed proof attempts, errors were corrected by inspection of the proof state". Mechanical checking of formal specifications and properties is useful both as a final confirmation of their correctness and for simplifying the process of discovering inconsistencies and proving properties. In fact, experience shows that even the most carefully crafted formal specifications and proofs, when done by hand, can still contain inconsistencies and other errors, and that such errors are discovered only using tools capable to perform some sort of analysis (also simply name checking, type consistency checking or other similar simple analysis). Proofs can also be so long and tedious that machine support can reduce the human effort.

In this paper, we show how PVS can provide tool support for ASMs, especially oriented towards automatic proof checking and mechanized proving of properties. PVS [SOR93], developed by SRI, is a higher order logic specification and verification environment with a Gentzen-like deduction system. PVS provides *both* a highly expressive specification language and automation of proof steps.

Starting from the work of Dold et al. in [Dol98,DGVZ98] we develop suitable PVS theories to encode ASM models in PVS and we prove that our transformation preserves the semantics of ASMs. Our approach differs from that of Dold et al. in the way of encoding the ASM transition system. Our goal is to find a transformation from ASMs to PVS that preserves the layout of the original rule transition system. To achieve this goal we do not take any strong assumption as "only one rule enabled at a time", indeed we allow more rules to be simultaneously executed. Therefore, we do not demand that the user should transform the whole system of rules in one meta rule, transformation which can require skill and ingenuity, especially in case of multi-agent models. Instead, we propose an algorithmic approach of transformation which keeps the set of transition rules as a set of different rules.

We provide useful templates for helping the user during the encoding phase, and useful proof schemes for requirements verification. The suggested encoding scheme is mechanizable and we provide a framework for a tool, that we prototypically implemented, supporting automatic transformation in PVS of ASM models given in the ASM-SL language [Del98].

Our PVS encoding of ASMs has been tested for multi agent models. We report here the PVS specification of the Production Cell given in [BM97] and we discuss our results of the mechanized proofs of its safety and liveness properties. Several proofs are obtained with assumptions, usually regarding the physical environment behavior, that are implicit in the formal description but need to be added in the machine supported approach. Thanks to the PVS features, we formulate such assumptions in a declarative way instead of using the operational way required by the model checking approach used by Winter in [Win97] to show the correctness of the Production Cell specification.

The article is organized as follows. The problem of encoding ASMs in PVS is discussed in Sections 2.1-2.3. Section 2.4 shows how to validate specifications using PVS. Sections 2.5 and 2.6 present proof schemes to support verification of invariant properties and trace properties. Section 3 contains the PVS specification of the Production Cell case study and the results of the verification task. In Section 4 we discuss our work in connection with related results and we conclude outlining future research directions.

2 An encoding of ASMs in PVS

In this section we present our encoding of ASMs in PVS. It consists of a set of PVS theories, hierarchically organized, which define types and functions modeling ASM universes and rules, and of a set of PVS strategies defined in order to simplify the proof conduction. Universe and function representation is based on the work of Dold et al. [Dol98,DGVZ98]. PVS encoding of rules and dynamics of ASMs is significantly changed.

2.1 Abstract State

An ASM *state* models an (abstract) machine state, i.e. the collection of elements the machine "knows", and the functions and predicates it uses to manipulate such elements. Mathematically a *state* is defined as a structure which consists of a collections of domains (sets, also called *universes*, each of which stands for a particular type of elements) with arbitrary functions and relations defined on them. **Universes representation.** Universes have different encoding depending on their being static or dynamic:

- A *static universe* U, i.e. a universe which does not change during computation, is encoded as uninterpreted type `U:TYPE`.
- A *dynamic universe* U, i.e. a universe which may grow during computation, is encoded as a PVS set `U:setof[T]` of elements of type T.
  As elements can not be added to PVS types, only such encoding allows to expand universes. To this purpose, the functions *add* and *union*, (pre-)defined in the PVS theory `sets`, can be used.

Remark 1. The concept of type in PVS is very different from the usual meaning of type in programming languages. PVS types abstractly represent entities with common operations. There are only a few built-in PVS types such as numbers, integers and boolean. Other types may be defined either as uninterpreted types, or through the construct `DATATYPE`, or in terms of already defined types by usual type constructors (as lists, records, etc.).

It is possible to encode the superuniverse itself as an uninterpreted type `S` and define every universe `U` as a subset of `S` – as already noted by Dold in [Dol98]–, or as uninterpreted subtype (with the same meaning as subset). However, in this case it is necessary to introduce axioms to guarantee properties on sets, as, for example, disjointness between universes, so not exploiting the strong and powerful type system of PVS. For this reason we suggest to encode universes as uninterpreted types whenever possible, and use the construct `setof` only for dynamic universes.

Functions and states. ASM *basic* functions are classified in *static* functions which remain constant, and *dynamic* functions which may change interpretation during computations. Dynamic functions are furthermore classified in *controlled*, *monitored* and *shared*. Controlled functions occur inside function updates and can change by application of transition rules. Monitored functions can change only due to the environment. Shared functions can change by application of a transition rule but also by some other agents; they formalize the interactions in case of multi-agent computations and combined updates of locations. Functions defined in terms of the basic ones are called *derived*.

We encode functions almost as proposed in [Dol98].

- *Static functions* are encoded as usual PVS functions.
- *Monitored functions* are modeled as PVS functions adding the type `ENV` to the arguments of their domain. `ENV` is an uninterpreted type which represents the external environment, that is everything outside the system's control. A boolean dynamic monitored variable is therefore encoded as a function from `ENV` to `bool`, whereas a generic monitored function $f : S \rightarrow T$ is translated to a function `f:[ENV,S->T]`.
  Our way of interpreting `ENV` is different from that proposed in [Dol98] where `ENV` is the record of all monitored functions. Our view of `ENV` has the advantage of allowing a designer to easily add and remove monitored functions.
- *Controlled functions* are regarded as fields of a record `CTRLSTATE` which represents the controlled part of an ASM state. Therefore, if $f : A \rightarrow B$ and $g : Int \rightarrow Bool$ are controlled functions, the controlled part of the state is defined as `CTRLSTATE: TYPE [# f:[A->B], g:[int->bool] #]`.
- *Shared function* are defined as controlled functions. In a multi-agent ASM, the controlled part of a "global" state is viewed as the collection of all controlled agent's states. Therefore, shared functions, as controlled by at least one agent, are defined as components of the `CTRLSTATE`.
- *Derived function* are encoded as functions defined in terms of their function components in a straightforward way from their ASM specification.

An ASM *state* is compound of its controlled part and the environment. It is defined as the record `STATE: TYPE [# env: ENV, ctrl: CTRLSTATE #]`.

2.2 ASM rules

ASM *(transition) rules* model the actions performed by the machine to manipulate elements of its domains and which will result in a new state. A *transition rule* R_i has the general form:

$$R_i : \textbf{if } cond_i \textbf{ then } update_i$$

$cond_i$, also called the *guard* of the rule, is a statement expressing the condition under which R_i must be applied; $update_i$ consists of finitely many *functions updates*:

$$f(t_1, \ldots, t_n) := t$$

which are executed simultaneously. f is an arbitrary n-ary function and $t_1, ..., t_n$ is the tuple of arguments at which the value of the function is set to t.

How to serialize the parallel rule application. In this section we tackle the problem of finding an algorithm to serialize firing of ASM rules which guarantees the semantics of their parallel application as stated in [Bör99]:

> An ASM M is a finite set of rules [...]. Applying one step of M to a state $\mathcal{A}$ produces as next state another algebra $\mathcal{A}'$, of the same signature, obtained as follows.
> First evaluate in $\mathcal{A}$, using the standard interpretation of classical logic, all the guards of all the rules of M. Then compute in $\mathcal{A}$, for each of the rules of M whose guard evaluates to true, all the arguments and all the value appearing in the updates of this rule. Finally replace, simultaneously for each rule and for all the locations in question, the previous $\mathcal{A}$ function value by the newly computed value.

Let $\{R_1,\ ...,\ R_n\}$ be a set of rules. The naive approach of computing the next state $\mathcal{A}'$ of the current state $\mathcal{A}$ by applying rules R_i, $i = 1 \dots n$, sequentially,[1] does not guarantee the ASM semantics for two reasons: (a) guards $cond_i$ of R_i are not evaluated in the same state $\mathcal{A}$ forall i; (b) terms occurring in the left and right sides of function updates of R_i are not all evaluated in $\mathcal{A}$. The following example may help to convince that this approach is wrong.

Example. Let $\mathcal{A} = (x \rightarrow 0, y \rightarrow 1, z \rightarrow 2)$ be the current state consisting of three integer variables x, y, z, and $\{R_1,\ R_2,\ R_3\}$ be the transition system with rules defined as follows:

$$R_1 : \textbf{if } x = 0 \textbf{ then } y := 5 \qquad R_2 : \textbf{if } y = 5 \textbf{ then } x := 2 \qquad R_3 : z := x + y$$

By sequential rules application, it results:

$$\mathcal{A} : \begin{pmatrix} x \rightarrow 0 \\ y \rightarrow 1 \\ z \rightarrow 2 \end{pmatrix} \overset{R_1}{\rightarrow} \begin{pmatrix} x \rightarrow 0 \\ y \rightarrow \boxed{5} \\ z \rightarrow 2 \end{pmatrix} \overset{R_2}{\rightarrow} \begin{pmatrix} x \rightarrow \boxed{2} \\ y \rightarrow 5 \\ z \rightarrow 2 \end{pmatrix} \overset{R_3}{\rightarrow} \begin{pmatrix} x \rightarrow 2 \\ y \rightarrow 5 \\ z \rightarrow \boxed{7} \end{pmatrix} : \mathcal{A}'$$

whereas by correct computation would be $\mathcal{A}' = (x \rightarrow 0, y \rightarrow 5, z \rightarrow 1)$.
An **algorithm** which overcomes the errors of the naive attempt is given below.

Let $\{R_1\ ,...,\ R_n\}$ be a set of transition rules, R_i of the form **if** $cond_i$ **then** $update_i$ with $update_i$ a sequence of function updates $f(t_1, t_2, \dots, t_n) := t$, and $\mathcal{A}$ be the current state.
The algorithm can be stated as follows:

1. For all rules $R_1\ ,...,\ R_n$, mark R_i if enabled at the state $\mathcal{A}$.
2. For all i such that marked R_i, evaluate in $\mathcal{A}$ all terms $t_1, t_2, \dots, t_n, t$ occurring in function updates $f(t_1, t_2, \dots, t_n) := t$ of $update_i$.
3. For all i such that marked R_i, sequentially perform the function updates in $update_i$ using the values computed at step 2.

[1] I.e., first evaluate the guard of R_1 and, if it is true, perform the updates defined in R_1 to obtain $\mathcal{A}_1$; then apply R_2 to $\mathcal{A}_1$ and obtain $\mathcal{A}_2$; and so on till computing the final state $\mathcal{A}_n$ to be taken as $\mathcal{A}'$.

The proposed algorithm initially values all the guards $cond_i$ in $\mathcal{A}$ and only afterwards, sequentially, performs all the function updates. Applying the algorithm to the above example we get the following correct transformation:

$$\mathcal{A} : \begin{pmatrix} x \to 0 \\ y \to 1 \\ z \to 2 \end{pmatrix} \overset{R_1}{\to} \begin{pmatrix} x \to 0 \\ y \to \boxed{5} \\ z \to 2 \end{pmatrix} \overset{R_2}{\to} \begin{pmatrix} x \to 0 \\ y \to 5 \\ z \to 2 \end{pmatrix} \overset{R_3}{\to} \begin{pmatrix} x \to 0 \\ y \to 5 \\ z \to \boxed{1} \end{pmatrix} : \mathcal{A}'$$

Rule R_2 does not affect the state $\mathcal{A}$ because not enabled, and R_3 correctly computes the new value of z using the values of x and y in $\mathcal{A}$.

Inconsistent updates. If a location is updated to different values by simultaneously finiring rules, then we speak of "inconsistent updates". Although the location value should remain unchanged due to the non-execution of inconsistent updates [Gur95], according to our algorithm, the location would take the last value assigned to it. For this reason, at the moment, we are able to consider only consistent ASM models. It would be possible to discover inconsistent updates through simulation of suitable test cases. However, simulation can never guarantee absence of inconsistency. Applying a formal verification technique similar to that used for proving invariant properties in Section 2.4, we might discover inconsistent updates. Checking inconsistency of ASM specifications is a problem still under investigation. Note that the problem of firing inconsistent updates is not taken in consideration in [Dol98,DGVZ98].

How to deal with extend. The proposed algorithm does not give the expected result when dealing with combination of rules which simultaneously add elements to a same set by the construct **extend**. This problem arose trying to encode in PVS the ASM model of crypto-protocol given in [BR98]. In this model, different agents can send at the same time messages into the TRAFFIC (a set of messages) representing the network channel common to all agents.

To be convinced why **extend** must be dealt with particular attention, consider the following example. Let $\mathcal{A}$ be a state where a set Δ has value $\emptyset$. The following rules are enabled in $\mathcal{A}$:

$$R_1\text{: } \textbf{if } \Delta = \emptyset \textbf{ then extend } \Delta \textbf{ with } a$$
$$R_2\text{: } \textbf{if } \Delta = \emptyset \textbf{ then extend } \Delta \textbf{ with } b$$

Encoding "**extend** Δ **with** u" as a function update of the form $\Delta := \Delta \cup \{u\}$, by our algorithm, we get the following transformation:

$$\mathcal{A} : (\Delta \to \{\}) \overset{R_1}{\to} (\Delta \to \{a\}) \overset{R_2}{\to} (\Delta \to \{b\}) : \mathcal{A}'$$

However, as both rules are applicable, according to the correct semantic of **extend**, we expect the result $\Delta = \{a, b\}$ in the final state $\mathcal{A}'$.

The algorithm needs to be adjusted to deal with extend. A correct solution should (a) evaluate expressions in the current state; (b) fire updates not involving

extend against the current state; (c) if a rule R_i contains an update **extend** U **with** u, add u to the set U interpreted not in the current state, but in the state obtained after firing all rules preceding R_i (otherwise we could loose updates of U by other possible **extend**s of previous rules).
The **revisited version of the algorithm** follows:

1. For all rules R_1 ,..., R_n, mark R_i if enabled at the state $\mathcal{A}$.
2. For all i such that marked R_i, evaluate in $\mathcal{A}$ all terms occurring in function updates of $update_i$ with exception of those representing universes as arguments of **extend**.
3. Let, unless renaming, $\{R_1, ..., R_m\}$ be the set of all marked rules. Assume $s_0 = \mathcal{A}$. Sequentially, compute s_i from s_{i-1}, $i = 1 \dots m$, performing function updates of $update_i$ of rule R_i in s_{i-1}: use values computed at step 2 for all terms except those representing universes within **extend** which must be evaluated in s_{i-1}. The final state s_m be $\mathcal{A}'$.

Remark 2: *Reserve* and multisets. According to the ASM semantics, elements added to extended universes are "new" and imported from *Reserve*. To guarantee that, Dold et al. introduce a predicate `new` to control an element does not already belong to the set to be extended, and a function `sort_update` which extends a set by new elements. However, PVS does not allow sets to contain different occurrences of the same element, and even Dold's formalization does not avoid such a problem. When an ASM specification requires the universe to contain multiple occurrences of a same element, we suggest to encode a universe as a multiset instead of a set. Keeping in mind that in PVS a set of elements of type `T` is a function from `T` to `bool`, as natural extension, a multiset can be encoded as a function from `T` to natural numbers: the value of the function represents the number of occurrences of its argument in the multiset. Functions and predicates on sets, like `add`, `member`, `emptyset`, etc., must be redefined. The PVS theory to model multisets follows:

```
multisets [T: TYPE]: THEORY
  BEGIN
   multiset: TYPE = [T-> nat]
   x, y: VAR T
   a: VAR multiset
% an element x is member of a only if the occurrences
% of x in a are greater than 0
   member(x, a): bool = a(x) > 0
   empty?(a): bool = (FORALL x: NOT member(x, a))
   emptyset: multiset = lambda x: 0
   nonempty?(a): bool = NOT empty?(a)
% the function “add” returns the same multiset modified
% increasing by one the number of occurrences of the added
% element
   add(x, a): (nonempty?) = lambda y:
            if x = y then a(y)+ 1 else a(y) endif
  END multisets
```

Remark 3. The rule encoding proposed in [Dol98,DGVZ98] is based on the (strong) assumption that rules "concern distinct (disjoint) cases". As a consequence, one may infer that (a) at every step only one rule is enabled, and therefore (b) all rules "may be combined into a single state transition function, defined by a case construct or nested conditionals". Such a single transition function is called "one-step interpreter" by Dold et al. This assumption of disjoint cases is restrictive, since in many applications more then one rule might fire at a time. In case of possible simultaneous firing of many rules, a unique transformation function might still model the ASM dynamics as well, however we prefer keeping the set of transition rules as set of distinct rules, without forcing the user to define an equivalent transformation function from the current state to the next one. The advantages are two: first stylistic, because we preserve the ASMs structure, and second practice, because writing one equivalent global transformation function requires skill and ingenuity. Furthermore, due to the assumption that only one rule can be applied in one step, the one-step interpreter is absolutely not suitable in case of multi-agent ASMs.

Remark 4. Another possible approach for rule encoding would be to define the value of every variable and function (location) in the next step by means of axioms. For example, the update $f(x) := t$ would be translated into an axiom of the form `update : AXIOM f(next(s))(x) = t` , where `next(s)` stands for the next state. In this way it would not be necessary to introduce the state as a record and we could consider the state as simple uninterpreted type. This approach is similar to that taken by [Win97] to translate ASMs in SMV. However, this translation does not preserve the original form of the rules and problems rise when a rule consists of more than one function update, or when more rules update the same function (location). In both cases, a complex transformation is needed as described in detail in [Win97].

2.3 Implementing ASM rules in PVS

In this section we show how the proposed algorithm has been implemented in PVS. Keep in mind that in our encoding a *state* consists of a controlled part (`CTRLSTATE`) modifiable by the rules, and the environment (`ENV`).

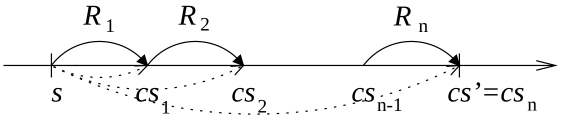
Rule encoding. We consider a rule as a function `Ri(current,intCtrl)`. The first argument `current: STATE` represents the current state s (controlled part and environment), and it is used to evaluate guards (step 1 of the algorithm) and compute terms (step 2 of the algorithm). The second argument `intCtrl: CTRLSTATE` represents an intermediate controlled state which is the result of applying the previous $i-1$ rules, and is used to compute those updates extending sets (step 3 of the algorithm). `Ri(current,intCtrl)` yields a new controlled (intermediate) state.

In PVS a rule is defined as `Ri:[STATE,CTRLSTATE -> CTRLSTATE]`.

Computation of the next state by composition of rules. To compute the next state of a current one by application of a set of rules, we define the function `next` as explained below.

Let $R_1, R_2, \ldots, R_n$ be a set of rules, and s the current state (its controlled part be cs). The controlled part cs' of the next state s' is inductively defined as follows:

1. $cs_0 = cs$
2. for $i = 1, \ldots, n : cs_i = R_i(s, cs_{i-1})$
3. $cs' = cs_n$



This algorithm defines only how to compute the controlled part of s' since the rules do not change the monitored part (i.e. the environment).

The set of all ASM rules is encoded as a list `rules` of functions from state and controlled state to a controlled state:

```
rules: list[[STATE,CTRLSTATE->CTRLSTATE]]
```

The application of the list `rules` of rules using as current state `s0` and as intermediate controlled state `cs_i`, is given by the following recursive definition[2]:

```
apply(s0,cs_i,rules) : recursive CTRLSTATE =
 if null?(rules)    then cs_i
 else apply(s0,car(rules)(s0,cs_i),cdr(rules))
endif measure length(rules)
```

The controlled part of the next state of `s` is defined applying the list `rules` of all ASM rules, and taking `s` as initial current state and its controlled part as initial intermediate controlled state:

```
nextCtrlState(s:STATE): CTRLSTATE = apply(s,s‘ctrl,rules)
```

The monitored part (the environment) of the next state of `s` is defined by the following function returning a random value for `ENV`:

```
nextEnv(s:STATE) : ENV
```

The next state of `s` is the composition of the two next (controlled and monitored) parts:

```
next(s:STATE): STATE =
(#env:= nextEnv(s), ctrl:= nextCtrlState(s) #)
```

[2] `car` and `cdr` are built-in PVS functions yielding head and tail of a list, respectively.

Templates for rules encoding. We report here the two templates for rules encoding. They distinguish between rules with and without **extend**ing updates. Let $R_i \equiv$ **if** $cond_i$ **then** $update_i$ be a transition rule.

1 If $update_i$ is a sequence of function updates of the form $f(t_1, \ldots, t_n) := t$, then R_i is translated in PVS as follows:

```
Ri(current, intCtrl) : CTRLSTATE =
 IF cond_i(current) THEN intCtrl
          WITH [f:=f(intCtrl) WITH [(t1,...,tn):=t]]
 ELSE intCtrl    ENDIF
```

If the guard `cond_i` is true in the `current` state then the updated intermediate controlled state `intCtrl` is returned; otherwise the state `intCtrl` is returned unchanged (all terms `ti` and `t` are computed in the `current` state).

2 If $update_i$ has the form **extend** *Delta* **with** *alpha*, then R_i is translated in PVS as follows:

```
Ri(current, intCtrl) : CTRLSTATE =
IF cond_i(current) THEN intCtrl
               WITH [Delta := add(alpha,Delta(intCtrl))]
ELSE intCtrl   ENDIF
```

If the guard `cond_i` is true, the element *alpha* is added to *Delta* evaluated in `intCtrl`; otherwise `intCtrl` is returned unchanged.

Example 1. The following examples may help to better understand rules encoding. Let the ASM rules be

R_1 : **if** $x = 0$ **then** $y := 5$ R_2 : **if** $y = 5$ **then** $x := 2$ $R_3 : z := x + y$

The controlled state is defined as a record of three variables:

```
CTRLSTATE: TYPE = [# x: int,  y: int,  z: int #]
```

The three rules are defined as follows:

```
R1(current, intCtrl) : CTRLSTATE =
 IF x(current) = 0    THEN intCtrl  WITH [y := 5]
 ELSE intCtrl  ENDIF
R2(current, intCtrl) : CTRLSTATE =
 IF y(current) = 5    THEN intCtrl  WITH [x := 2]
 ELSE intCtrl  ENDIF
R3(current, intCtrl) : CTRLSTATE =
 intCtrl  WITH [z := x(current) + y(current)]
```

Example 2 (with sets).

R_1: **if** $\Delta = \emptyset$ **then extend** Δ **with** a
R_2: **if** $\Delta = \emptyset$ **then extend** Δ **with** b

Assuming that elements of the set Δ belong to a certain type `elementType`, and a and b are two constants of that type:

```
elementType : TYPE
a,b : elementType
```

the controlled state is defined as a record containing only the set Δ:

```
CTRLSTATE: TYPE = [# Delta: SETOF[elementType]#]
```

The two rules become:

```
R1(current, intCtrl) : CTRLSTATE =
 IF empty?(Delta(current))  THEN intCtrl
           WITH [Delta := add(a,Delta(intCtrl))]
 ELSE intCtrl  ENDIF
R2(current, intCtrl) : CTRLSTATE =
 IF empty?(Delta(current))  THEN intCtrl
           WITH [Delta := add(b,Delta(intCtrl))]
 ELSE intCtrl  ENDIF
```

Non determinism. In ASM non-determinism can be expressed using the constructor **choose** which allows to fire a rule $R(x)$ choosing randomly an x satisfying given conditions:

choose x **in** U **s.t.** *g(x)*
R(x)

In PVS **choose** can be encoded by a monitored function from the environment to the subset of U consisting of elements satisfying the condition $g(x)$.

```
chooseX : [ENV-> {x:U|g(x)}]
```

The subcase **choose** x **in** U is encoded by `chooseX : [ENV->U]`. The non-determinism is captured by leaving undefined (without no specified mathematical low) the function `chooseX`. Therefore, for a given environment `e`, the value of `chooseX(e)` in `{x:U|g(x)}` is not determined.

2.4 Validating specifications through simple proofs

After having specified system universes, functions and rules, the designer should check whether the specification is correct or not, i.e. if it meets users' needs and if it satisfies all the desired properties (requirements).

The first step in this direction is to check some possible behaviors of the system as specified and compare them with the desired behaviors. This approach is followed by ASM simulators (like ASM Workbench and ASM-Gofer). A similar approach might be followed in our encoding, probing the specification by means of "formal challenges". With this term we mean putative theorems, i.e. properties that should be true if the specification is correct. The designer should start from formally specifying and proving very simple statements and then gradually prove more complex properties. Only at the end he/she should try to prove the complete requirements.

In our Example 1 a very simple property is: "if in state s x is 0, y is 1 and z is 2, then in the next state x should be equal to 0, y to 5 and z to 1". It can be encoded as the following lemma:

```
prop1: lemma
s`ctrl= (#x:=0,y:=1,z:=2#) => next(s) = (#x:=0,y:=5,z:=1#)
```

We have proved the lemma `prop1` simply using the PVS decision procedures of rewriting and symbolic execution. More complex and complete properties might contain quantification on system quantities and, therefore, to be proven they might need the use of more advanced PVS strategies.

The main goal of these short proofs is to gain a deeper understanding of the system, to become confident in the correctness of the proposed description and model, and discover possible errors or faults as early as possible, since it is widely acknowledged that the cost of correcting specification errors is order of magnitudes higher in the later stages of the life cycle of the system (like during testing or even during its normal functioning). At the end the designer should be able to formally state and prove the actual requirements. The verification of system requirements is the subject of the following sections.

2.5 Using induction to prove invariants

In mathematics induction is widely used to prove properties that hold for every natural number. In formal models that describe system behavior as a sequence of states (as the ASM approach does), the same scheme can be used to prove system *invariants*, i.e. properties holding in every state. In this case induction is based on the following theorem:

Theorem 1. *Let S_0 be the set of all initial states and $P(s)$ a property of the state s. If*

(i.) $P(s_0)$ holds $\forall s_0 \in S_0$;

(ii.) $P(s) \to P(s')$, $\forall s, s'$ states such that $s' = next(s)$

then P is an "invariant".

In our encoding we have defined and proved theorem 1 as follows:

```
induction: THEOREM
        (forall(s:STATE): P(s)=> P(next(s)) and
        (forall(s:(init)): P(s)) implies INV(P)
```

where `(init)` denotes the set of initial states and `P` the property to prove as invariant (`INV(P)` means that `P` is true in the initial state and in every reachable state).

This theorem, along with an "ad hoc" strategy that we have defined in the file `pvs-strategies`, provides an induction scheme that can be used to prove invariants in ASM models.

2.6 Trace properties

A *trace* is an (infinite) sequence $s_0, s_1, \ldots, s_n, \ldots$ of states – any state s_i should be considered as compound of the controlled part and the environment – satisfying the property that: *a*) s_0 is a valid initial state, and *b*) for every pair of subsequent states s_i, s_{i+1} it holds that s_{i+1} is the next state of s_i.

In our encoding a trace is formalized as a sequences of states satisfying the property of being a *trace* stated above:

```
trace: TYPE = {x : sequence[STATE] | member(first(x),init)
                 and forall n: nth(x,n+1) = next(nth(x,n))}
```

Trace properties are properties on traces, i.e. properties which are expressed and proved in terms of traces. The two most common types of trace properties are:

- properties holding in every state of every trace (*always*), or
- properties holding at least in one state in every trace (*eventually*).

We can express in PVS that a property *stateProp* holds in every state of the trace t by the following predicate:

```
always(t,stateProp): bool = FORALL n: stateProp(nth(t,n))
```

We have proved the equivalence between this approach based on traces and that based on invariants by proving the following lemma

Lemma 1. *"stateProp" is an invariant iff it always holds in every trace of the system.*

In PVS:

```
equivalence: LEMMA
        INV(stateProp) <=> forall t: always(t,stateProp)
```

We express in PVS that a property *reachProp* holds in a state of the trace t by the following predicate:

```
eventually(t,reachProp): bool = EXISTS n: reachProp(nth(t,n))
```

The *always* property is normally used to express "safety" requirements ("nothing bad will never occur"): properties that must be true in every state. The *eventually* property normally expresses "liveness" properties ("something good will eventually happen") modeling requirements that must be eventually true.

3 A Case Study: Production Cell

In this section we explain this novel use of PVS as tool support for ASMs using as case study the Production Cell model given in [BM97]. The main purpose of this section is not to show that the Production Cell specification of Börger and Mearelli satisfies safety and liveness properties. That has been already proved in [Win97] by means of the model checker SMV. We simply like to show, through a concrete example, how to apply our method of encoding ASM specifications in PVS, and how to mechanize proofs.

3.1 A brief introduction of the Production Cell Case Study

The production cell control problem was posed in [LL95] as case study derived from "an actual industrial installation in a metal-processing plant in Karlsruhe" to obtain a "realistic, comparative survey" for testing "the usefulness of formal methods for critical software systems and to prove their applicability to real-world examples" [LL95]. Börger and Mearelli propose a solution of the production cell control problem in [BM97], and show how to integrate the use of ASMs into a complete software development life cycle.

> *. . . the production cell is composed of two conveyor belts, a positioning table, a two-armed robot, a press, and a traveling crane. Metal plates inserted in the cell via the feed belt are moved to the press. There, they are forged and then brought out of the cell via the other belt and the crane. [LL95]*

The system is specified "as a distributed ASM with six modules, one for the *agents*" – the Feed Belt, the Robot, the Press, the Deposit Belt, the Traveling Crane, the Elevating Rotary Table – "composing the production cell, and working together concurrently where each of the component ASMs follows its own clock. Each of the agents represents a sequential process which can execute its rules as soon as they become enabled. The sequential control of each agent is formalized using a function currPhase: Agent → Phase which yields at each moment the *current phase of the agent*"[BM97]. Please refer to [BM97] for further details.

3.2 The PVS specification

For convenience, we report below the signature and the module of the Ground-CELL Program for the Feed Belt (FB).
Monitored function: PieceInFeedBeltLightBarrier
Shared function: TableLoaded (between FB and the elavating rotary table ERT)
Derived functions:
  TableInLoadPosition ≡ currPhase(ERT) = StoppedInLoadPosition
  TableReadyForLoading ≡ TableInLoadPosition and not TableLoaded
Controlled functions: FeedBeltFree,
  currPhase(FB) ∈ {NormalRun,Stopped,CriticalRun}.

Module:

```
FB _NORMAL.
    if currPhase = NormalRun and PieceInFeedBeltLightBarrier
    then  FeedBeltFree:= True
          if TableReadyForLoading then currPhase:= CriticalRun
                                  else currPhase:= Stopped
FB_STOPPED.
    if currPhase = Stopped and TableReadyForLoading
    then currPhase:= CriticalRun

FB_CRITICAL.
    if currPhase = CriticalRun and PieceInFeedBeltLightBarrier
    then  currPhase:= NormalRun
          TableLoaded:= True
```

Initialization: currPhase = NormalRun, FeedBeltFree = True,
PieceInFeedBeltLightBarrier = False

We now report the PVS encoding of the Feed Belt specification. To describe the Feed Belt we define a type containing all the possible values of the feed belt phase:

```
FBPhase : TYPE = {NormalRun,Stopped,CriticalRun}
```

The controlled function FeedBeltPhase is then included as component of the record CTRLSTATE which represents the controlled part (controlled and shared functions) of the (global) state:

```
CTRLSTATE : TYPE =
 [#   FeedBeltPhase : FBPhase,
      FeedBeltFree  : bool, % controlled by the FB
      TableLoaded   : bool, % controlled by the FB and ERT
      ...#]
```

The dots are replaced by the controlled part of the other five agents, that we skip for the sake of conciseness. Monitored variable is defined as function from the environment to its domain as:

```
PieceInFeedBeltLightBarrier : [ENV->bool]
```

Starting from definitions of monitored and controlled functions, derived functions are defined as:

```
TableInLoadPosition(s:CTRLSTATE) : bool =
     ERTPhase(s) = StoppedInLoadPosition
TableReadyForLoading(s:CTRLSTATE) : bool =
     TableInLoadPosition(s) and not TableLoaded(s)
```

The initial state is modeled by a predicate over the states (we report only the part concerning FB):

```
init(s:STATE) : bool =
FeedBeltPhase(s)= NormalRun and FeedBeltFree(s)
and not PieceInFeedBeltLightBarrier(s) ...
```

For the rules we report only the example of the FB_NORMAL rule:

```
FB_NORMAL(current,intCtrl): CTRLSTATE =
if FeedBeltPhase(current) = NormalRun and
   PieceInFeedBeltLightBarrier(current)
then intCtrl with
    [FeedBeltFree := true,
    FeedBeltPhase := if TableReadyForLoading(current)
         then CriticalRun
         else Stopped
         endif]
else intCtrl
endif
```

3.3 The Safety Properties

Using PVS we have proved all the safety properties (for the Feed Belt, the Robot, the Press, the Deposit Belt, and the Traveling Crane) of the Production Cell as given in [BM97]. For some of these properties, the encoding in PVS and the proof are straightforward. Others require some user effort and skill. In order to discuss the degree of interaction necessary for proving in PVS the requirements of the Production Cell case study, we present some selected examples of proved properties having different degree of complexity.

The **Feed Belt Safety Property:** *the feed belt does not put metal blanks on the table if the latter is already loaded or not stopped in loading position,* has been quickly encoded considering that the feed belt puts metal blanks only when it is in the CriticalRun phase[3]:

```
FeedBeltSafety: theorem
FeedBeltPhase(s) = CriticalRun
  => ElevatingRotaryTablePhase(s) = StoppedInLoadPosition
     and not TableLoaded(s)
```

The proof of this property, reported below, is immediate (as also its hand proof in [BM97]):

```
("" (ASM-INDUCT)
      (("1" (COMPUTE-NEXT) (GRIND))
       ("2" (TYPEPRED "is!1") (GRIND))))
```

(ASM-INDUCT) applies the induction theorem presented in Section 2.5. By induction the proof is split in two parts: the induction step (proved by the branch "1") and the initial state (branch "2"). (COMPUTE-NEXT) is a strategy defined in our encoding and expands the definitions of the next state and the rules. (GRIND) is a PVS command that rewrites remaining definitions, splits the cases and applies the decision procedures of PVS. (TYPEPRED "is!1") recalls the type definition of the initial state "is!1". The last (GRIND) expands the definition of initial state and applies the PVS decision procedures.

This proof case shows that the user effort to write properties and obtain relative proofs might be very low. According to our experience, all simplest properties whose proof do not involve assumptions about the environment, require a minimal user interaction and their proofs can be performed using induction, case splitting, decision procedures, and rewriting rules. However, many other properties have to be proved taking in consideration the interaction of the system with the environment. In these cases, proving properties might require greater user effort and skill. As example, consider the first part of the **Press Safety Property 1:** *the press is not moved downward if it is in its bottom position.* Its PVS encoding is straightforward:

[3] In this lemma and in the following ones, s is to be considered as universally quantified over the set of all reachable states from an initial state.

```
PressSafety1a: theorem
     PressBottomPosition(s) =>
             not PressMotorDown(Press(next(s)))
```

However, to prove this property, we have to introduce an assumption about the monitored function `BottomPosition` asserting that *if the press is closed for forging, then it is not already in the bottom position*:

```
notBottom: axiom
PressPhase(s)= ClosedForForging => not BottomPosition(s)
```

This is an obvious implication considering how the system works, but we have to explicitly state that by means of an axiom. For other properties we introduce similar assumptions by means of axioms, and recall these axioms during proofs. These assumptions often concern the correct behavior of the sensors, and sometimes are missing in the original description because implicitly assumed. This again shows that automatic support may help to uncover errors forcing the designer to precisely introduce every assumption. Note that these assumptions are introduced by means of logical statements, similar to those given in [BM97], while a model checker would require us to express them in an operational way. Upon introducing the `notBottom` assumption, the proof of the Press Safety Property 1 is obtained applying induction, expanding the definitions and applying the PVS decision procedures. Another example we like to report here is the **Press Safety Property 2**: *The press does only close when no robot arm is positioned inside it.* In order to encode this property in a concise form, we introduce two (derived) boolean functions: *PressIsClosing* and *ArmInPress,* defined as

```
PressIsClosing(s): bool = PressMot(PressPhase(s)) = up
```

i.e. the press is closing only when its motor is going up, and

```
ArmInPress(s): bool =
     Arm1Ext(s) > 0 and Angle(s) = Arm1ToPress or
     Arm2Ext(s) > 0 and Angle(s) = Arm2ToPress
```

Using these definitions the property becomes:

```
PressSafety2: theorem
    PressIsClosing(s) => not ArmInPress(s)
```

To prove this property we model the angle of the robot arm by the monitored variable `Angle` encoded as function on the environment:

```
Angle :[ENV-> real]
```

Then we formalize all the assumptions about the movement of the robot: how the robot rotates, how the angle changes, and how the sensors communicate to the robot when to stop. We prove the Press Safety Property 2 using induction, case analysis (considering all the possible robot phases in the current and in the next state), recalling the assumptions about the robot movement, and applying the automatic decision procedures of PVS.

3.4 The Liveness Property

We have proved the liveness property as well, i.e. that the system never goes in deadlock. We have stated it as "*every component in the system will eventually change its state*". This statement is weaker than the property proved (by hand) in [BM97]. Börger and Mearelli establish also a performance valuation about the number of pieces that the cell is able to process cyclically. The proof of this performance property is under investigation. Our property is similar to the "progress agent property" stated in [BM97].

In order to prove liveness, we model and specify some assumptions about the environment (those called "Cell Assumption" in [BM97]). For example we assume that every object on the feed belt keeps moving until eventually it arrives at its destination and the monitored function `PieceInFeedBeltLightBarrier` becomes true:

```
FBAssumption: axiom
   FeedBeltPhase(s) = NormalRun =>
   exists(ns:(followingOrNow(s))):PieceInFeedBeltLightBarrier(ns)
```

`followingOrNow(s)` is the set of all states obtained repeatedly applying the function `next` to the state `s`, and of the state `s` itself.

We make similar assumptions for every state of every component of the cell, thus we assume that every component keeps moving till the monitored function of interest (which selected on the basis of the state and the component) eventually changes its value. Starting from these assumptions we prove, for example, the liveness of the FeedBelt:

```
FeedBeltProgress: lemma
   exists (ns: (following(s))):
       not FeedBeltPhase(ns) = FeedBeltPhase(s)
```

Using the same approach we are able to prove similar properties for every agent.

4 Related Work and Conclusions

Several attempts of applying both theorem provers and model checkers to ASM models have been performed. In [SA97] the KIV (Karlsruhe Interactive Verifier) system has been used to mechanically verify the proof of correctness of the Prolog to WAM transformation. PVS has been used in [DGVZ98,Dol98] to perform mechanical verification of the correctness of back-end rewrite system (BURS) specifications. A model checker approach is reported in [Win97] where correctness of the Production Cell specification of Börger and Mearelli has been proved through SMV. Recently, an interface from the ASM Workbench to the SMV model checking tool, based on an ASM-to-SMV transformation, has been presented in [DW99].

A direct comparison of our approach can be done with the work of Dold et al. using PVS and with the Winter's work using SMV.

Along our presentation, especially in remarks 1,2 and 3, we have discussed differences between Dold et al.'s approach and ours. We provide a more natural translation of ASMs in PVS keeping the set of transition rules as a set of different rules, instead of forcing the user to define an equivalent transformation function in terms of one meta rule. We also provide the user with useful templates to guide his/her formalization of ASMs in PVS. These templates have also allowed us to provide a framework for a tool to automatically translate ASM specifications in PVS. This tool is under development and we plan to integrate it into the ASM-Workbench system. In addition we present proof schemes to encode and prove invariants and properties on traces.

Although the approach based on the model checker SMV allows properties verification in a completely automatic manner (unless the well known state explosion problem), the advantages of using our approach regard both the specification and the verification phase. PVS can easily manage specifications with infinite sets and/or infinite agents, has a powerful language (to represent functions, sets, lists and so on), has a strong type system, and can use the usual logical constructs (like universal and existential quantifications). Proof can be performed almost automatically in the simplest cases (as shown in the Production Cell case study). For more complex properties, in any case, our encoding can be used to check proofs done by hand or to support the user during the proof in an interactive way. In connection with the results presented in [Win97], we like to remark that the model checking approach can deal only with a finite set of agents and each agent having a finite number of possible states. This is the case of the Production Cell, under the assumption that continuous intervals (for example, the robot angle values) can be treated as finite sets of discrete values. This assumption was indeed used by Winter in [Win97], while it is not necessary in our approach, since we are able to deal with infinite sets (for example, we treat the robot angle as a real number). The correctness proof (with its results) of the Production Cell specification as it is shown in [Win97] has to be related to the added formalization of the environmental behavior. It is a mayor benefit of our approach that the assumptions regarding the interaction of the system and the environment can be formalized in a logical than in an operational way (i.e. in terms of transition rules) as required in [Win97].

Concluding, we like to stress our confidence that the proposed PVS encoding also works well for multi-agent ASMs which are very complex to treat. It is not so hard to imagine how difficult can be performing mechanized proof verification of properties regarding interleaving computations of agents. The case study we present here is an example of a multi-agent system, but with a limited number of agents. However, the method has been successfully applied to analyze properties of crypto-protocols, where an unlimited number of agents run simultaneously. Security and authentication properties of the ASM specification presented in [BR98] have been proved in PVS using the technique of invariants. We have voluntarily left the presentation of these results out because they would require a specific treatment.

Acknowledgments. We kindly like to thank Egon Börger for his useful advice. We also thank anonymous referees for their helpful suggestions.

References

[BM97] E. Börger and L. Mearelli. Integrating ASMs into the Software Development Life Cycle. *Journal of Universal Computer Science*, 3(5):603–665, 1997.

[Bör95] E. Börger. Why Use Evolving Algebras for Hardware and Software Engineering? In M. Bartosek, J. Staudek, and J. Wiederman, editors, *Proceedings of SOFSEM'95, 22nd Seminar on Current Trends in Theory and Practice of Informatics*, volume 1012 of *LNCS*, pages 236–271. Springer, 1995.

[Bör99] E. Börger. High level system design and analysis using abstract state machines. In D. Hutter, W. Stephan, P. Traverso, and M. Ullmann, editors, *Current Trends in Applied Formal Methods (FM-Trends 98)*, number 1641 in LNCS, pages 1–43. Springer-Verlag, 1999.

[BR98] G. Bella and E. Riccobene. A Realistic Environment for Crypto-Protocol Analyses by ASMs. In *Proceedings of the 28th Annual Conference of the German Society of Computer Science.* Technical Report, Magdeburg University, 1998.

[Del98] G. Del Castillo. The ASM Workbench: an Open and Extensible Tool Environment for Abstract State Machines. In *Proceedings of the 28th Annual Conference of the German Society of Computer Science.* Technical Report, Magdeburg University, 1998.

[DGVZ98] A. Dold, T. Gaul, V. Vialard, and W. Zimmerman. ASM-Based Mechanized Verification of Compilter Back-Ends. In *Proceedings of the 28th Annual Conference of the German Society of Computer Science.* Technical Report, Magdeburg University, 1998.

[Dol98] Axel Dold. A formal representation of abstract state machines using pvs. Technical Report Verifix Report Ulm/6.2, Universitat Ulm, July 1998.

[DW99] G. Del Castillo and K. Winter. Model Checking Support for the ASM High-Level Language. Technical Report TR-RI-99-209, Universität-GH Paderborn, June 1999.

[Gur95] Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.

[Hei98] C. Heitmeyer. On the Need for Parctical Formal Methods. In *Proceedings of FTRTFT'98, 5th Intern. Symposium Real-Time Fault-Tolerant Systems*, volume 1486 of *LICS*, pages 18–26. Springer, 1998.

[LL95] C. Lewerentz and T. Linder, editors. *Formal Development of Reactive Systems. A Case Study "Production Cell".* Number 891 in LNCS. Springer, 1995.

[SA97] G. Schellhorn and W. Ahrendt. Reasoning about Abstract State Machines: The WAM Case Study. *Journal of Universal Computer Science*, 3(4):377–413, 1997.

[SOR93] N. Shankar, S. Owre, and J. Rushby. The PVS proof checker: A reference manual. Technical report, Computer Science Lab., SRI Intl., Menlo Park, CA, 1993.

[Win97] K. Winter. Model Checking for Abstract State Machines. *Journal of Universal Computer Science*, 3(5):689–701, 1997.