A compact encoding of sequential ASMs in Event-B*

Michael Leuschel[1], Egon Börger[2]

[1] Institut für Informatik, Universität Düsseldorf
Universitätsstr. 1, D-40225 Düsseldorf
leuschel@cs.uni-duesseldorf.de
[2] Dipartimento di Informatica, University of Pisa (Italy)

Abstract. We present a translation of sequential ASMs to Event-B specifications. The translation also addresses the partial update problem, and allows a variable to be updated (consistently) in parallel. On the theoretical side, the translation highlights the intricacies of ASM rule execution in terms of Event-B semantics. On the practical side, we show on a series of examples that the Event-B encoding remains compact and is amenable to proof within Rodin as well as animation and model checking using ProB.
Keywords: ASM, B-Method, Model Checking, Constraint-Solving, Tools.

1 Motivation

ASMs have been used since 1989 to model computational systems of different kinds; various tools have been built to simulate such models [12, 5, 13] and various theorem proving systems have been adopted to prove properties of ASMs, for references see [8, Ch.8.1] and more recent publications in the ABZ Conference Proceedings. Given the close relationship between ASMs and B and Event-B [2] models we want to investigate more closely the relation between ASMs, B and Event-B concepts. In particular we want to clarify whether and how one can translate ASMs to Event-B in a reasonable fashion, without blow-up, such that the translation can be effectively applied and permits the use of the animation, model checking, and constraint solving tool ProB[17] as well as of provers [3, 10, 11] to mechanically support proving properties for ASMs.

2 Background

2.1 ASM background

The syntax and semantics of ASMs is defined in [8, Ch.2.4]. We recapitulate the part considered in this paper.

ASMs are defined as rules which transform structures of a given signature, their 'states'. Without loss of generality one can consider a signature as a family

* Part of this research by the second author was funded by a renewed Forschungspreis grant of the Humboldt Foundation in the summer of 2015.

of function symbols f^n of arity n; predicates (relations) can be dealt with in terms of their characteristic Boolean-valued functions. Thus a state over such a signature is the mathematical structure formed by a set U (called the (super-) universe of the state) together with an interpretation of each function symbol f^n by an n-ary function with arguments and values in U; specific domains can be defined as subsets of the superuniverse U.

There are various constructs which can appear in ASM rules to modify a state by changing the value of some of its functions at some of their arguments, essentially using updates of form $f(s_1, \ldots, s_n) := t$ where s_i, t are terms (functional expressions). In particular the following recursively defined set of ASM rules turned out to be sufficient to describe any sequential algorithm at whatever level of abstraction[1]

skip *// empty action*
$f(s_1, \ldots, s_n) := t$ with terms s_i, t *// assignment rule*
R **par** S *// simultaneous parallel execution*
if ϕ **then** R **else** S with formula ϕ *// case distinction*
let $x = t$ **in** R *// call by value*
choose x **with** ϕ **do** R *// nondeterministic choice*

Notably the **forall** construct for universal quantification and the two clauses for submachines and sequential functional composition in [8, Table 2.2] are missing.

Usually R_1 **par** R_2 **par** ... **par** R_n is written in vertical notation without **par**. Below we also write $S \mid S'$ for S **par** S'.

The parallel composition operator **par** is considered to be characteristic for ASMs. The semantics of $R = R_1$ **par** R_2 **par** ... **par** R_n is to execute all component rules R_i simultaneously, atomically. This means that for one (an atomic) step of R in a given state s, the updates of each R_i are applied together in one step to s. This yields the next state s' (successor state of s) if the computed set of updates is consistent; otherwise the step cannot be computed and the successor state of s is undefined.

Updates are pairs $((f, (a_1, \ldots, a_n)), val)$ of so-called locations $(f, (a_1, \ldots, a_n))$ and a newly to be assigned (not necessarily new) *val*ue for that location; here f is a function symbol (of arity n), $(a_1, \ldots, a_n)$ an argument tuple of elements of some set in the given state s—typically the interpretation in state s of some terms $s_1, \ldots, s_n$ in an assignment rule $f(s_1, \ldots, s_n) := t$—and val the interpretation of t in state s. A set of updates (usually called an 'update set') is **consistent** if it does not contain two pairs $(l, v), (l, v')$ with $v \neq v'$. Thus parallel updates to a same variable (location) are allowed, but they must be consistent. In case the update set is inconsistent (i.e. contains two updates $(l, v), (l, v')$ for the same location l with different values v, v') the next state s' is not defined. Note that 0-ary locations $(f, ())$ are just variables (written f as usual).

For the deterministic form of the sequential ASM Thesis it turned out that the following sequential normal form ASMs suffice:

[1] This fact is known as the sequential ASM Thesis Gurevich proved in [14] from three natural postulates which axiomatize the underlying concept of 'sequential algorithm'.

if ϕ_1 **then** $f_1(s_{1,1}, \ldots, s_{1,n_1}) := t_1$
$\vdots$
if ϕ_k **then** $f_k(s_{k,1}, \ldots, s_{k,n_k}) := t_k$

where the guards ϕ_i are boolean combinations of equations between terms.

Note that in fact

- **if** ϕ **then** R **else** S is equivalent to

  (**if** ϕ **then** R) **par** (**if not** ϕ **then** S)
- rules of the form **let** $x = t$ **in** R exist only for practical purposes, a hidden form of sequentialization as used for example in initializations. The rule is equivalent to **choose** x **with** $x = t$ **do** R, though **choose** is not needed for the deterministic case. Note that the **let** rule could also be programmed (without using **choose**) by mere **if then else** rules.
- occurrences of **choose** can be moved to the outside, using renaming to avoid clashes between variables. Under the assumption that x is not used in B and S, the construct **if** B **then** (**choose** x **with** ϕ **do** R) **else** S becomes

  choose x **with** ($B \Rightarrow \phi$ **and not** $(B) \Rightarrow x = \mathit{default}$) **do**
  if B **then** R **else** S

  where *default* is some value compatible with the type of x. Indeed, when B is false, the variable x is not used and the role of the assignment $x = \mathit{default}$ is simply to prevent unnecessary non-determinism in the else branch.

In the non-deterministic case of the sequential ASM Thesis one has to consider the bounded choices an algorithm can make—not only the environment—using the **choose** operator. Gurevich's sequential ASM Thesis remains provable but with a slight restatement of one of the three postulates (namely the abstract-state postulate) and the following extension of the normal form to:

choose $i \in \{1, 2, \ldots, k\}$ **do**
  if $i = 1$ **then** R_1
  $\vdots$
  if $i = k$ **then** R_k

Here the transitions rules R_i consist of a finite set of parallel updates. For details see [8, p.306-7].

2.2 Event-B Background

Both the B-method [1] and its successor Event-B [2] are state-based formal methods rooted in set theory. Event-B has a richer refinement notion, with the aim of systems modelling rather than software development. On the other hand, Event-B has a much simpler structure for statements: notably there are no conditionals and no let statements. The static parts of an Event-B model, such as carrier sets, constants, axioms, and theorems are contained in contexts, whereas the dynamic

parts of the model are contained in machines. A machine comprises variables, invariants, and events. An event consists of two main parts: guards and actions. Formally, an event has the following form:

$$\textbf{event } e = \textbf{any } t \textbf{ when } G(x,t) \textbf{ then } S(x,t,x') \textbf{ end}$$

Here, t are the parameters of the event and the guard $G(x,t)$ can be an arbitrarily complex predicate over the state variables x and the parameters t. The statements $S(x,t,x')$, however, are very restricted and consist of parallel assignments of the form

- $v := E(x,t)$ *(deterministic assignment)*
- $v :\in E(x,t)$ *(non-deterministic assignment from a set of values)*
- $v : |P(x,v',t)$ *(non-deterministic assignment using a predicate)*

The statement list can also be empty, which corresponds to **skip**. It is not allowed to assign to the same variable v twice within the same event.

An event is *enabled* if there exists a value for the parameters t which makes the guard $G(x,t)$ true. If no such value exists, the event is *disabled*. Let us present a small example. The following event decrements a variable x by the amount a. In case $x \leq 0$, the event is disabled, as no solution for $a \in 1..x$ exists.

$$\textbf{event } \mathit{decrement} = \textbf{any } a \textbf{ when } a \in 1..x \textbf{ then } x := x - a \textbf{ end}$$

All machines also contain a special event, the *initialisation* which is not allowed to refer to the current state of the variables. The way events are executed in Event-B is somewhat different to ASMs. First, the initialisation event is executed to generate an initial state of the model. In any given state, any enabled event e can be executed atomically, resulting in a new state. All the actions of e are executed in parallel. So, in contrast to ASMs, events are executed in isolation; events cannot be executed in parallel together (but the individual actions of an event are).

Event-B machines contain an *invariant*, which is a predicate over the variables (and constants) of the machine. In order to establish that the invariant is indeed true in all reachable states, proof obligations are generated: one has to prove that the initialisation establishes the invariant and that each event— when enabled— preserves the invariant.

The Event-B language supports a rich set of datatypes, encompassing integers, booleans, user-defined types, sets, relations, and (higher-order) functions. For this paper one notation for functions will be important:

$$\lambda x.P \mid E$$

Given a predicate P and an expression E, this represents the function whose domain is all those values for x which make P true, in which case the function returns the value of E. For example,

$$\lambda x.x > 0 \mid x * x$$

is the squaring function defined for strictly positive integers. In B, functions are just seen as relations which in turn are sets of pairs. The above function could also have been written as

$$\{x \mapsto y \mid x > 0 \wedge y = x * x\}$$

The function $\lambda x.x \in 1..3 \mid x * x$ is a finite function and could thus also have been written as a set extension $\{1 \mapsto 1, 2 \mapsto 4, 3 \mapsto 9\}$. Set operators can also be used for functions, e.g., the predicate $\{2 \mapsto 4\} \subset \lambda x.x > 0 \mid x * x$ is true.

In summary, the differences with ASMs are:

1. richer actions are allowed in ASM rules (conditionals, let, ...)
2. parallel updates to the same variable are allowed in ASMs and not in Event-B (we return to this in Section 4),
3. the way rules are fired (all enabled ones are fired simultaneously in ASMs as opposed to one enabled event in Event-B).

3 Translating Conditional and Parallel Statements

Both Event-B and ASM allow parallel updates but differ in a quite fundamental way. Indeed, as mentioned in Section 2.2, Event-B only allows parallel updates of disjoint variables, such as $x := 1||y := 2$. However, the following is not allowed in Event-B (or classical B):

$$x := 1||x := y$$

ASMs, however, do allow this parallel assignment but then impose a consistency condition on all parallel updates.[2] For this example, this implies that the update is allowed in case $y = 1$ and considered inconsistent otherwise. If the update set to be applied to a state s is inconsistent, the next state s' is not defined. When combined with conditional statements — which Event-B does not support— the translation becomes even more intricate. Let us take an ASM machine with a variable x initialised to 0 and the rule depicted in Fig. 1.

$$\textbf{if } x > 10 \textbf{ then } x := x - 1 \textbf{ |if } x < 5 \textbf{ then } x := x + 1$$

Fig. 1. Simple ASM rule

[2] This is related to the circumstance that for reasons of generality arbitrary terms s, not only variables, are permitted on the left side of an assignment statement $s := t$. Therefore a machine may contain assignment statements $t_1 := t$ and $t_2 := t'$ with syntactically different $t_i = f(t_{i,1}, \ldots, t_{i,n})$ for which however in some state S their evaluation may yield the same arguments $eval_S(t_{1,j}) = eval_S(t_{2,j})$ for all $1 \leq j \leq n$ resulting in two updates $(l, eval_S(t))$ and $(l, eval_S(t'))$ to the same location $l = (f, (eval_S(t_{1,1}), \ldots, eval_S(t_{1,n}))$ so that the consistency condition $eval_S(t) = eval_S(t')$ is required.

Suppose we are interested in proving that this rule preserves the invariant $x \in \mathbb{N}$. How can we translate this to Event-B, without using conditional actions and parallel assignments to the same variable x?

3.1 A simple translation using case distinctions

The first solution that comes to mind is to encode every possible path through the rule as one event in Event-B. In this case there are in principle four possible paths: each of the two if-conditions can be either true or false. A translation of this ASM machine using four events is presented in Fig. 2.

```
machine ASM_4
variables x
invariants @inv x∈ℕ
events
  event INITIALISATION begin @ini x≔0
  end
  event asmif_tt when @gtt x>10 ∧ x<5 theorem @thm x-1=x+1 then
     @act  x ≔ x−1
   end
  event asmif_tf when @gtf x>10 ∧ ¬(x<5) then
     @dec  x ≔ x−1
   end
  event asmif_ft when  @gft ¬(x>10) ∧ x<5  then
     @inc  x ≔ x+1
   end
  event asmif_ff when @gff ¬(x>10) ∧ ¬(x<5) end
end
```

Fig. 2. Non-linear translation of **if** $x > 10$ **then** $x := x - 1$ |**if** $x < 5$ **then** $x := x + 1$ from Fig. 1

On the positive side, this machine can be be animated and model checked with ProB and proven fully automatically in Rodin with the standard autotactics. In other words, we have established that our ASM rule preserves the invariant $x \in \mathbb{N}$. Note that, for the case that both assignments are triggered (`asmif_tt`), we have added the guard theorem $x + 1 = x - 1$ to encode that these two assignments yield the same result. In Event-B this gives rise to the following proof obligation (where the guards and invariants are in the hypotheses):

$$x \in \mathbb{N} \land x > 10 \land x < 5 \models x + 1 = x - 1$$

As the hypotheses are unsatisfiable, the theorem can be proven. We have thus also proven that no conflict between the parallel assignments to x can occur. If we change the test $x < 5$ to, e.g., $x < 15$ in Fig. 1 and 2 the corresponding guard theorem proof obligation $x \in \mathbb{N} \land x > 10 \land x < 15 \models x + 1 = x - 1$ can no longer be discharged.

On the negative side, however, this translation can lead to a combinatorial blow up in the number of events. The many events will share many common predicates (i.e., $x > 10$ or $x < 5$ above) which will be re-evaluated by tools such as a model checker or even the provers. In this paper, we try to find a translation into Event-B which does *not* lead to such an explosion of the events, but which remains linear in the size of the original ASM.

3.2 Translation using update functions

One idea of our solution is to encode the updates to variables into composable *update functions*. Basically, an update function u for the variable x will be used to construct an Event-B assignment of the form

$$x := u(x)$$

This solution can later be extended (in Section 4) to deal with the important issue of partial updates, but it also solves the problem of conditional total updates. Let us return to our ASM rule from Fig. 1, where we now add the implicit else branches:

$$\textbf{if } x > 10 \textbf{ then } x := x - 1 \textbf{ else skip} \mid \textbf{if } x < 5 \textbf{ then } x := x + 1 \textbf{ else skip}$$

The idea is that every branch First, the update function for the (implicit) else branches is the identity function $\mathrm{id}_{\mathbb{Z}}$ for the type $\mathbb{Z}$ of x, defined by:

$$\mathrm{id}_T = \lambda v.v \in T \mid v$$

Second, the update function for the assignment $x := x - 1$ is the function $cst_{\mathbb{Z}}(x-1)$ defined by (where x' does not occur in C):[3]

$$cst_T(C) = \lambda v.v \in T \mid C$$

Similarly, the update function for the assignment $x := x + 1$ is the function $cst_{\mathbb{Z}}(x+1)$. While id_T copies the old value v of a variable, $cst_T(C)$ simply ignores it and overwrites it with C.

For conditionals, we construct update functions by inserting conditions into the update functions of each branch. This scheme is defined formally below. First, we need the following two auxiliary definitions (where v is a variable not occurring in u):

- $cond(P, u) = \lambda v.P \mid u(v)$
- $if(P, u_1, u_2) = cond(P, u_1) \cup cond(\neg P, u_2)$

We can now formally define the ternary relation $S \rightsquigarrow_x u$, denoting that the ASM statement S results in the update function u for the variable x. We assume that, like above, a conditional statement without else branch is first translated into a conditional statement whose else branch is **skip**.

[3] An overwrite particle in the terminology of [15].

We also have the set of location (entry) names *LocEntry*, which are the names of functions/variables which are updated in the ASM machine under consideration. From the point of view of the Event-B translation, these will be the variable names of the B machine. Below, $type(E)$ refers to the type of an expression.

$$\frac{}{\mathbf{skip} \rightsquigarrow_y \mathrm{id}_{type(y)}} \; y \in LocEntry$$

$$\frac{}{x := E \rightsquigarrow_x cst_{type(x)}(E)}$$

$$\frac{}{x := E \rightsquigarrow_y \mathrm{id}_{type(y)}} \; y \in LocEntry \setminus \{x\}$$

$$\frac{S \rightsquigarrow_x u \wedge S' \rightsquigarrow_x u'}{\mathbf{if}\ B\ \mathbf{then}\ S\ \mathbf{else}\ S' \rightsquigarrow_x if(B, u, u')}$$

$$\frac{S_1 \rightsquigarrow_x u_1 \wedge S_2 \rightsquigarrow_x u_2}{S_1 \mid S_2 \rightsquigarrow_x u_1 \circ u_2}$$

Rule 1 above stipulates that **skip** does not modify any location, i.e., we always obtain the identity function as update function. Rule 2 says that an assignment to x does indeed modify location x, while Rule 3 stipulates that it does not influence locations $y \neq x$, i.e., we obtain the identity update function $\mathrm{id}_{type(y)}$ for y. In the case of the parallel composition in the fifth rule, how do we know that the update functions u_1 and u_2 are compatible ? Also, the last rule has two obvious solutions: $u_1 \circ u_2$ and $u_2 \circ u_1$. Which one should be chosen? Basically, we will add proof obligations to ensure that $u_1 \circ u_2 = u_2 \circ u_1$, which in turn guarantees that the update functions are compatible. Indeed, commutativity is one way of defining compatible updates (see the functional applicative algebras in [15]). As such we will add a guard theorem $u_1 \circ u_2 = u_2 \circ u_1$ to the translated events; this theorem will result in a proof obligation but does not influence the event execution as such.[4] Actually, for technical reasons we will add the following weaker guard theorem, which only requires commutativity for the actual values of x encountered:

$$u_1(u_2(x)) = u_2(u_1(x))$$

This theorem can be dealt with more easily by the Rodin provers and by tools such as ProB. The precise computation of these guard theorems, also encompassing partial updates, will be formalised later in Section 4.

Note that the above rules distinguish between **skip** and $x := x$: the former has the update function $\mathrm{id}_{\mathbb{Z}}$ while the latter has the update function $cst_{\mathbb{Z}}(x) = \lambda v.v \in \mathbb{Z} \mid x$. This is very important. Indeed, $x := x \mid x := x - 1$ is inconsistent while **skip**$\mid x := x - 1$ is valid and equivalent to $x := x - 1$. Indeed, for the latter we have commutativity of the update functions:

$$\mathrm{id}_{\mathbb{Z}} \circ cst_{\mathbb{Z}}(x-1) = cst_{\mathbb{Z}}(x-1) \circ \mathrm{id}_{\mathbb{Z}} = cst_{\mathbb{Z}}(x-1)$$

[4] See http://handbook.event-b.org/current/html/theorems.html.

but not for the fomer:

$$cst_{\mathbb{Z}}(x) \circ cst_{\mathbb{Z}}(x-1) = cst_{\mathbb{Z}}(x) \neq cst_{\mathbb{Z}}(x-1) \circ cst_{\mathbb{Z}}(x)$$

We can now use the following inference rules to construct the action parts of an Event-B event for an ASM rule R as follows

$$\frac{R \rightsquigarrow_x u}{R \rightsquigarrow_{act} x := u(x)} \; x \in LocEntry \wedge u \neq \mathrm{id}_{type(x)}$$

To construct the Event-B action we simply take all solutions A for $R \rightsquigarrow_{act} A$ and put them into parallel and then add the commutativity theorems.

Let us return to our ASM rule R from Fig. 1. We have $R \rightsquigarrow_x u_1 \circ u_2$ with

- $u_1 = if(x > 10, cst(x-1), id)$ and
- $u_2 = if(x < 5, cst(x+1), id)$.

We can compute these update functions and their composition as follows:

- $cst(x-1) = \lambda v.v \in \mathbb{Z} \mid x-1$
- $u_1 = (\lambda v.x > 10 \mid x-1) \cup (\lambda v.\neg(x > 10) \mid v)$
- $u_2 = (\lambda v.x < 5 \mid x+1) \cup (\lambda v.\neg(x < 5) \mid v)$
- $u_1 \circ u_2 = (\lambda v.x > 10 \mid x-1) \cup (\lambda v.\neg(x > 10) \wedge x < 5 \mid x+1) \cup$
  $(\lambda v.\neg(x > 10) \wedge \neg(x < 5) \mid v)$
- $u_2 \circ u_1 = (\lambda v.x < 5 \mid x+1) \cup (\lambda v.\neg(x < 5) \wedge x > 10 \mid x-1) \cup$
  $(\lambda v.\neg(x > 5) \wedge \neg(x > 10) \mid v) =$
  $u_1 \circ u_2$ because $x > 10 \wedge x < 5$ is unsatisfiable.

```
machine ASM1
variables x
invariants @inv x∈ℕ
events
  event INITIALISATION begin @ini x≔0
  end
  event asmifs // if x>10 then x:=x-1 | if x<5 then x:=x+1
   any u1 u2 where
    @g1 u1 = (λv·v∈ℤ ∧ x>10 ∣ x-1)  ∪  (λv·v∈ℤ ∧ ¬(x>10)∣ v)
    @g2 u2 = (λv·v∈ℤ ∧ x<5 ∣ x+1)  ∪  (λv·v∈ℤ ∧ ¬(x<5)∣ v)
    theorem @comm u1(u2(x)) = u2(u1(x))
    then
     @a  x ≔ u1(u2(x))
   end
end
```

Fig. 3. Event-B Translation of an ASM Rule with parallel conditional update on the same variable

Figure 3 shows the complete Event-B translation of the above example with the commutativity theorem. We use the syntax of the text editor Camille [6]. The

labels such as `@g1`, `@g2` and `@a` only play a role for user feedback during proof; they have no attached semantics. For readability we have extracted the two composed update functions into event parameters. The machine has three proof obligations related to the translated event `asmifs`, all of which can be automatically discharged using the SMT prover plugin [10, 11]. However, the standard autotactics of Rodin no longer discharge all of them. Indeed, proving commutativity typically requires case distinctions, which the SMT prover plugin is good at but not so much the Atelier-B provers ML and PP. ProB can animate and model check the model; it has 6 distinct states ($x \in 0..5$).

4 Partial Update Problem

In Event-B a function—just like a relation— is seen like a set of tuples. An array is just a special case of a function, where the domain is a contiguous set of indexes. We will adopt the same view for our ASM machines. Below we also use $\lambda v.E$ as a shortcut for $\lambda v.v \in \mathit{type}(E) \mid E$.

The parallel update problem becomes more interesting when partial updates are concerned. An assignment like $f(2) := 3$ is a partial update in the sense that the function f is only changed at the position 2. A possible value for f would be $\{1 \mapsto 0, 2 \mapsto 0\}$ and the effect of the above assignment would be to change f into $\{1 \mapsto 0, 2 \mapsto 3\}$. Furthermore, in Event-B the above assignment is seen as syntactic sugar for $f := f \lhd\!\!\!+ \{2 \mapsto 3\}$, where the override operator $\lhd\!\!\!+$ can be defined as follows:

$$r \lhd\!\!\!+ s = \{x \mapsto y \mid x \mapsto y \in s \vee (x \mapsto y \in r \wedge x \notin \mathit{dom}(s))\}$$

As such Event-B does not allow the parallel updates:

$$f(1) := 2 \mid f(2) := 3$$

as indeed they get translated to the following parallel total updates of f:

$$f := f \lhd\!\!\!+ \{1 \mapsto 2\} \mid f := f \lhd\!\!\!+ \{2 \mapsto 3\}$$

First, parallel updates of the same variable are not allowed in Event-B. Second, even if they were, these two total updates are not consistent, as $f \lhd\!\!\!+ \{1 \mapsto 2\} = \{1 \mapsto 2, 2 \mapsto 0\} \neq \{1 \mapsto 0, 2 \mapsto 3\}$ for our value of $f = \{1 \mapsto 0, 2 \mapsto 0\}$ above. In ASMs, however, these parallel partial updates are allowed (and are not seen as equivalent to a total update) and result in $f = \{1 \mapsto 2, 2 \mapsto 3\}$.

The situation is somewhat similar to the conditional updates in Section 3, and our solution is the same: we represent partial updates also as update functions which can be composed and which have to be checked for commutativity. So, in the example above we would generate two update functions:

1. $u_1 = \lambda v.v \lhd\!\!\!+ \{1 \mapsto 2\}$
2. $u_2 = \lambda v.v \lhd\!\!\!+ \{2 \mapsto 3\}$

We have commutativity $u_1 \circ u_2 = u_2 \circ u_1 = \lambda v.v \lhd\!\!+ \{1 \mapsto 2, 2 \mapsto 3\}$ and we can achieve the combined parallel effect of the assignments using the single assignment

$$f := u_1(u_2(f))$$

Let us consider again the assignment $x := x+1$ from Section 3. There we have used the update function $u_c = cst_{\mathbb{Z}}(x+1)$. However, an alternate update function would have been $u_{cum} = \lambda v.v \in \mathbb{Z} \mid v+1$. Note that u_c is idempotent and does not commute with $u'_c = cst_{\mathbb{Z}}(x-1)$. u_{cum} on the other hand is not idempotent

$$u_{cum} \circ u_{cum} = \lambda v.v \in \mathbb{Z} \mid v+2$$

but does commute with $u'_{cum} = \lambda v.v \in \mathbb{Z} \mid v-1$:

$$u_{cum} \circ u'_{cum} = u'_{cum} \circ u_{cum} = \mathrm{id}_{\mathbb{Z}}$$

So, the update function u_{cum} corresponds to another, *cumulative* interpretation of $x := x+1$; when executed in parallel with itself it increments x by 2. u_{cum} is an example of what are called particles in [15].

Figure 4 contains a few more example update functions. For queues we assume the representation of a sequence of length n as a total function with domain $1..n$. Within the update function for pop, $(\{1\} \lhd v)$ removes the first element from the sequence v without adjusting the indices of the other elements; that is achieved by using relational composition with the successor function succ. Our translation to Event-B can in principle cope with any of these interpretations, cumulative or not. In the remainder of the paper we will only focus on total assignment and function update, though.

Description	Syntax	Update function u
Regular total assignment	$x := y$	$\lambda v.y$
Cumulative integer addition	$x+ = \Delta$	$\lambda v.v \in \mathbb{Z} \mid v + \Delta$
Function update	$x(s) := t$	$\lambda v.v \lhd\!\!+ \{s \mapsto t\}$
Function update level 2	$x(s_1)(s_2) := t$	$\lambda v.v \lhd\!\!+ \{s_1 \mapsto (v(s_1) \lhd\!\!+ \{s_2 \mapsto t\})\}$
Cumulative set addition	$x\cup = y$	$\lambda v.v \cup y$
Cumulative set removal	$x\setminus = y$	$\lambda v.v \setminus y$
Queue push	$push(x, e)$	$\lambda v.v \cup \{(card(v)+1) \mapsto e\}$
Queue pop	$pop(x)$	$\lambda v.(\mathrm{succ}\,;(\{1\} \lhd v))$

Fig. 4. Example update functions ($\lambda v.E$ is a shortcut for $\lambda.v \in type(E) \mid E$)

The core idea is that two partial updates are consistent if they commute and the practical question is how can we check this, in particular when multiple update functions are combined. We could try out all permutations, but that quickly blows up. Note that the composition of multiple update functions is always associative, as function composition $\circ$ is associative. We return to this issue later.

5 Translation Scheme

We now finish the translation scheme started in Section 3.2, adding rules for partial updates and describing how the guard theorems are generated. Basically, we have already described the rule $ASM \rightsquigarrow_{act} x := u(x)$ and below we describe the relation $ASM \rightsquigarrow_{thm} G$. Let $\{A_1, \ldots, A_k\} = \{A \mid ASM \rightsquigarrow_{act} A\}$ and let $\{G_1, \ldots, G_m\} = \{G \mid ASM \rightsquigarrow_{thm} G\}$. Then we generate the Event-B event:

$$\textbf{event } ASM = \textbf{ when theorem } G_1 \ldots \textbf{theorem } G_m \textbf{ then } A_1 | \ldots | A_k \textbf{ end}$$

Observe that we have no parameters and no proper guard, just guard theorems. The guard theorems just give rise to proof obligations which ensure that the update functions commute and we have consistent updates. We could (and actually do so in the implementation) lift the individual update functions u in the A_j's to be parameters and reuse them within the G_i's. We already did so in Fig. 3.

In addition to the basic update functions id, $cst(C)$, and $if(P, u_1, u_2)$ we now need one more construct defined as follows:

- $upd(s,t) = \lambda v.v \Leftarrow \{s \mapsto t\}$

We extend the inference rules for $\rightsquigarrow$ from Section 3.2 by the two following ones:

$$\frac{}{x(E) := F \rightsquigarrow_x upd(E, F)}$$

$$\frac{}{x(E) := F \rightsquigarrow_y \mathrm{id}_{type(y)}} \; y \in LocEntry \setminus \{x\}$$

The guard theorems are generated using this inference rule for $\rightsquigarrow_{thm}$:

$$\frac{ASM \rightsquigarrow_x u_1 \circ \ldots u_k}{ASM \rightsquigarrow_{thm} perm_x((u_1 \circ \ldots u_k))}$$

We define $perm_x(u)$ to be true if u is a basic update function not of the form $u_1 \circ u_2$. Otherwise we have:

$$perm_x(u_1 \circ u_2) =_{def} (u_1(u_2(x)) = u_2(u_1(x))) \wedge perm_x(u_1) \wedge perm_x(u_2)$$

Take the following ASM rule $f(x) := 1$ **|if** $y \neq x$ **then** $f(y) := -1$ where we wish to establish that the invariant $f(x) \geq 0$ is preserved by the rule. The result of our translation can be found in Fig. 5.

For this example the proofs no longer go through automatically in Rodin: we have to do manual case distinctions on $x = y$. In future, one should probably generate a library of "update function proof rules"; we return to this issue below. We can animate and model check the system using ProB without problem though.

```
event asm // f(x) := 1 || if y/= x then f(y) := -1
 any u1 u2 where
  @g1 u1 =   (λg·g∈ℤ↔ℤ| g⩤{x↦1})
  @g2 u2 =   (λg·g∈ℤ↔ℤ ∧ y≠x| g⩤{y↦-1}) ∪
             (λg·g∈ℤ↔ℤ ∧ y=x| g)
  theorem @commutativity u1(u2(f)) = u2(u1(f))
  then
   @a  f ≔ u1(u2(f))
 end
```

Fig. 5. Event-B Translation of an ASM Rule with partial udpates and conditionals

Some Optimisations In order to make the translation more amenable to proof and animation, we suggest a few optimisation rules (some of which we have implemented in our prototype). Let us first define this notation:

$$u_1 \diamond u_2 =_{def} (u_1 \circ u_2) = (u_2 \circ u_1)$$

Note that $u_1 \diamond u_2$ is symmetric. The following selection of results hold for commutativity of our update functions and can ease proving our guard theorems:

$$\mathrm{id}_T \diamond u \Leftrightarrow \top$$
$$cst_T(C_1) \diamond cst_T(C_2) \Leftrightarrow (C_1 = C_2)$$
$$cst_T(C) \diamond upd(s,t) \Leftrightarrow (C(s) = t)$$
$$upd(s,t) \diamond upd(s',t') \Leftrightarrow (s \neq s' \vee t = t')$$

It would make sense to add these as theorems to our translation, to ease automatic proving.

Initialisation The Event-B initialisation allows no parameters. In case parameters are used in an ASM initialisation, we either need to translate these into a second initialisation event (along with a boolean variable *isInitialised*). Alternatively, one can use Event-B constants to represent the parameters.

Well-Definedness The ASM statement `f(x) := undef` is translated to domain subtraction in Event-B $f := \{x\} \lhd f$. Apart from that we suppose well-definedness in Event-B style.

Translating Choose and Forall The proof of the Parallel ASM Thesis in [7] yields the following normal form for parallel ASMs R (i.e. sequential ASMs with the additional **forall** x **with** ϕ **do** M construct):

forall $x \in U_R$
  if $Cond_1(x)$ **then** $upd_1(x)$
  ⋮
  if $Cond_r(x)$ **then** $upd_r(x)$

for a multiset term U_R, single assignments $upd_i(x)$ of the form $f(t_1, \ldots, t_n) := t$ and some r depending on the given ASM program R. Essentially the term U_R represents the multiset U of updates to be applied by R—the empty set in case R computes an inconsistent multiset of updates so that if $U \neq \varnothing$ the assignments $upd_i(x)$ are consistent.

Suppose now we have a parallel ASM rule **forall** i **with** $\Phi(i)$ **do** $S(i)$ in normal form (i.e. where **forall** appears only as the outer constructor of the ASM).

If the set $\{i \mid \Phi(i)\}$ is static, the machine can be translated to a sequential machine as follows. One can transform $\Phi(i)$ into $i \in \{K_1, \ldots, K_n\} \wedge \Phi'(i)$, where the expressions $K_1, \ldots, K_n$ are known statically. Then we can generate the following sequential ASM which is equivalent to the given ASM:

if $\Phi'(K_1)$ **then** $S(K_1)$
if $\Phi'(K_2) \wedge K_2 \notin \{K_1\}$ **then** $S(K_2)$
$\vdots$
if $\Phi'(K_n) \wedge K_n \notin \{K_1, ..., K_{n-1}\}$ **then** $S(K_n)$

to which one can apply the translation rules described above.

6 Prototype, Discussions and Future Work

Prototype Implementation and Experiments

A prototype translator has been implemented within ProB, using Prolog to implement our inference rules. It uses the classical B syntax to express ASMs. To avoid generating errors for parallel assignments to the same variable, a preference within ProB has been added to turn off a variety of static checks. The translator generates the update functions and theorems as described above. We have then manually copied and pasted the result into Rodin.

However, while writing the paper we have streamlined and improved the notations and also the transformation process. Initially we experimented with various other approaches, notably collecting the updates in sets. All of these were more cumbersome than the solution presented here and actually less amenable to automated proof.

The prototype implementation should now be rewritten using the simpler concepts and should also work on "real" ASM input files rather than use classical B syntax. We plan to do this in future work. Still, we were able to experiment with some non-trivial ASM rules and animate, model check, and prove them.

Discussion and Future Work

Translating Monitored Variables ASMs foresee certain variables (locations) to be monitored and implicitly modified by the environment (between two internal ASM steps, see [8, Def.2.4.22]). We can translate this behaviour by adding a "turn" variable and the respective guards to alternate between executing the ASM rules proper and the environment which modifies the monitored variables.

Translating Derived Functions ASMs also foresee derived functions such as:

```
derived isMaster(m) = (index(m)=0)
```

There are various ways these could be translated. One solution would be to add new definition using the Event-B Theory Plugin [9]. Alternatively, one could add a new function in a context which takes all variables as arguments. For this example that would be: $isMaster = \lambda m.m \in Agents \mid bool(index(m)) = 0$.

Translating Submachine calls These use "call by reference" in ASM. In our experiments we have thus expanded such rule calls like macros (possibly with renaming to avoid clashes and variable capture).

Bounded Exploration Postulate Sequential ASMs satisfy Gurevich's bounded exploration postulate so that there can only be finitely many changes to locations in one step. In B (as in parallel ASMs) you can do infinitely many changes, e.g., the assignments $f := \mathbb{N} \times \{1\}$ or $f := \{d \mapsto r \mid d \in \mathbb{N} \wedge r = d * x\}$ set the value of f in infinitely many locations, as do the non-sequential parallel ASMs **forall** $n \in \mathbb{N}\, f(n) := 1$ resp. **forall** $n \in \mathbb{N}\, f(n) := n * x$.

Validation To prove our translation correct we would need an expression of ASM semantics in B; but this is what we are trying to develop in the first place. However, once we have a tool in place we could cross check the results of various tools, such as [12] or [4].

By combining all updates for a variable, our translation avoids re-evaluating the same predicate multiple times (unlike the naive translation in Fig. 2). However, the same predicate can still appear in different update functions for different variables. We propose to solve this issue by using ProB's common-sub-expression elimination.[5]

To what extent our technique can really scale to bigger ASM machines is still open. As far as proving is concerned, the commutativity theorems usually require a series of case distinctions. As such, provers such as the SMT prover plugin [10, 11] or the ProB Disprover [16] are probably essential for our translated models. Theorems about applicative algebras in [15] may be of help. Simulation and model checking can also be done, but it still remains open what the performance will be compared to running CoreASM[12] or using AsmetaSMV [4].

Outside of tooling support for ASMs, we hope that our work has also clarified the intricacies of ASM rules to Event-B researchers, and also shown the ASM researchers the power of B's predicates and expressions.

In summary, we have developed a translation from sequential ASMs to Event-B machines, which prevents a blow-up of the number of events. The translated machines can be validated using the Event-B tools. We hope that the efforts help in bringing the ASM and B communities closer to together.

Acknowledgement We would like to thank the reviewers of ABZ'16 for very useful feedback. The second author thanks Laurent Voisin for discussing with

[5] A recent feature of ProB; it needs to be explicitly enabled via the `CSE` preference; it does not work across multiple events and thus cannot be applied to Fig. 2.

him during the Dagstuhl seminar *Integration of Tools for Rigorous Software Construction and Analysis* (September 8-13, 2013) the problem of an Asm2EventB translation.[6]

References

1. J.-R. Abrial. *The B-Book*. Cambridge University Press, 1996.
2. J.-R. Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 2010.
3. J.-R. Abrial, M. Butler, and S. Hallerstede. An open extensible tool environment for Event-B. In Z. Liu and J. He, editors, *Proceedings ICFEM'06*, LNCS 4260, pages 588–605. Springer-Verlag, 2006.
4. P. Arcaini, A. Gargantini, and E. Riccobene. Asmetasmv: A way to link high-level ASM models to low-level NuSMV specifications. In *Proceedings ABZ'10*, pages 61–74, 2010.
5. The Abstract State Machine Metamodel website. http://asmeta.sourceforge.net, 2006.
6. J. Bendisposto, F. Fritz, M. Jastram, M. Leuschel, and I. Weigelt. Developing Camille, a text editor for Rodin. *Software: Practice and Experience*, 41(2):189–198, February 2011.
7. A. Blass and Y. Gurevich. Abstract State Machines capture parallel algorithms: Correction and extension. *ACM Trans. Computational Logic*, 8(3):19:1–19:32, 2008.
8. E. Börger and R. F. Stärk. *Abstract State Machines. A Method for High-Level System Design and Analysis*. Springer, 2003.
9. M. J. Butler and I. Maamria. Practical theory extension in event-b. In Z. Liu, J. Woodcock, and H. Zhu, editors, *Theories of Programming and Formal Methods - Essays Dedicated to Jifeng He on the Occasion of His 70th Birthday*, volume 8051 of *Lecture Notes in Computer Science*, pages 67–81. Springer, 2013.
10. D. Déharbe. Automatic Verification for a Class of Proof Obligations with SMT-Solvers. In *Proceedings ASM 2010*, pages 217–230, 2010.
11. D. Deharbe, P. Fontaine, Y. Guyot, and L. Voisin. SMT solvers for Rodin. In *Proceedings ABZ'2012*, LNCS 7316, pages 194–207. Springer, 2012.
12. R. Farahbod et al. *The CoreASM Project*. http://www.coreasm.org and https://github.com/coreasm/.
13. Foundations of Software Engineering Group, Microsoft Research. AsmL. http://research.microsoft.com/foundations/AsmL/, 2001.
14. Y. Gurevich. Sequential Abstract State Machines capture sequential algorithms. *ACM Trans. Computational Logic*, 1(1):77–111, July 2000.
15. Y. Gurevich and N. Tillmann. Partial updates. *Theoretical Computer Science*, 336(2-3):311–342, 2005.
16. S. Krings, J. Bendisposto, and M. Leuschel. From Failure to Proof: The ProB Disprover for B and Event-B. In *Proceedings SEFM'2015*, LNCS 9276. Springer, 2015.
17. M. Leuschel and M. J. Butler. ProB: an automated analysis toolset for the B method. *STTT*, 10(2):185–203, 2008.

[6] See http://drops.dagstuhl.de/opus/volltexte/2014/4358/.