Concurrent Abstract State Machines and ^+CAL Programs

Michael Altenhofen[1] and Egon Börger[2]

[1] SAP Research, Karlsruhe, Germany `Michael.Altenhofen@sap.com`
[2] Università di Pisa, Dipartimento di Informatica, I-56125 Pisa, Italy `boerger@di.unipi.it`

Abstract. We apply the ASM semantics framework to define the **await** construct in the context of concurrent ASMs. We link ^+CAL programs to concurrent control state ASMs with turbo ASM submachines.

1 Introduction

In recent work we made use of the Abstract State Machines (ASM) method [8] to analyze a given cluster protocol implementation. We extracted from the code a high-level model that could be used for the analysis. We also refined the abstract model to an executable CoreASM [11,10] model so that we could run scenarios in the model. The imperative to keep the abstract models succinct and graspable for the human eye led us to work with the **await** construct for multiple agent asynchronous ASMs. In this paper we define this construct for ASMs by a conservative extension of basic ASMs.

1.1 Problem of Blocking ASM Rules

As is well known, **await** *Cond* can be programmed in a non-parallel programming context as **while not** *Cond* **do skip**, where *Cond* describes the wait condition; see the flowchart definition of the control state ASM in Fig. 1, where as usual the circles represent control states (called internal states for Finite State Machines, FSMs) and the rhombs a test.

One has to be careful when using this construct in an asynchronous multi-agent (in the sequel shortly called concurrent) ASM, given that the semantics of each involved single-agent ASM is characterized by the synchronous parallelism of a basic machine step, instead of the usual sequential programming paradigm or interleaving-based action system approaches like the B method [1], where for each step one fireable rule out of possibly multiple applicable rules is chosen for execution. The problem is to appropriately define the scope of the blocking effect of **await**, determining which part of a parallel execution is blocked where **await** occurs as submachine. One can achieve this using control states, which play the role of the internal states of FSMs; see for example Fig. 1 or the following control state ASM, which in case $ctl\_state = wait$ **and not** *Cond* holds produces the empty update set and remains in $ctl\_state = wait$, thus 'blocking'

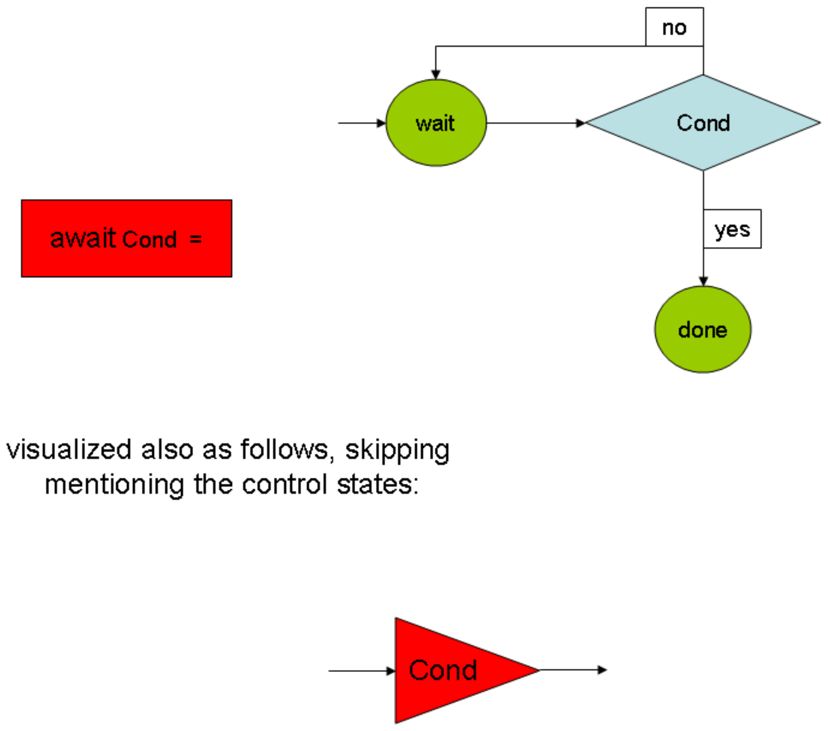

no
await Cond =
wait
Cond
yes
done
visualized also as follows, skipping mentioning the control states:
Cond

Fig. 1. Control State ASM for **await** *Cond*

the execution (under the assumption that in the given ASM no other rule is guarded by $ctl\_state = wait$)[1]:

```
await $Cond$ =
  if $ctl\_state = wait$ then
    if $Cond$ then $ctl\_state := done$
```

However, when the underlying computational framework is not the execution of one rule, but the synchronous parallel execution of multiple transition rules, the explicit use of control states leads quickly to hard to grasp complex combinations of conditions resulting from the guards of different rules where an **await** *Cond* has to be executed. The complexity of the corresponding flowchart diagrams can be reduced up to a certain point using the triangle visualization

[1] 'Blocking' here means that as long as the empty update set is produced, the state of the machine—in particular its control state—does not change.

in Fig. 1. It represents the right half of the traditional rhomb representation for check points in an FSM flowchart—where the right half represents the exit for *yes* (when the checked condition evaluates to true) and the left half the exit for *no* (when the checked condition evaluates to false). Is there a simple definition for the semantics of the **await** construct within the context of synchronous parallel basic ASMs? Otherwise stated, can a definition of the semantics of the **await** *Cond* **do** M machine, namely to wait until *Cond* becomes true and then to proceed with executing M, be smoothly incorporated into the usual semantical definition of ASMs based on constructing update sets?

Such a definition would avoid the need for control states in the high-level definition of **await** (without hindering its implementation by control states). Above all it would preserve a main advantage of the update set construction in defining what is an ASM step, namely to elegantly capture what is intended to be considered as a basic machine step. This is important where one has to work with different degrees of granularity of what constitutes a basic machine step, usually called 'atomic' step to differentiate it from the typical sequence of actions in a standard program with ";" (sequential execution). For example basic ASMs, which consist only of rules **if** *Cond* **then** *Updates*, have been equipped in [7] with a notation for operators to **seq**uentialize or call submachines. It is defined within the ASM semantics framework in such a way that the computation performed by M **seq** N appears to the outside world as one atomic step, producing the overall effect of first executing an atomic M-step and in the thus produced state an atomic N-step; analogously for submachine execution. Machines with these constructs are called turbo ASMs because they offer two levels of analysis, the macro step level and the view of a macro step as a sequence of micro steps (which may contain again some macro steps, etc.).

We provide in this paper a similar application of the ASM semantics framework to define the meaning of **await** for concurrent ASMs, in the context of the synchronous parallelism of single-agent ASMs.

1.2 Atomicity in Control State ASMs and ^+CAL

When discussing this issue with Margus Veanes from Microsoft Research at the ABZ2008 conference in London, our attention was drawn to the recently defined language ^+CAL for describing concurrent algorithms. It is proposed in [21] as "an algorithm language that is designed to replace pseudo-code" (op.cit., abstract), the idea being that describing algorithms in the ^+CAL language provides two advantages over traditional pseudo-code whose "obvious problems . . . are that it is imprecise and . . . cannot be executed" (op.cit. p.2): a) "a user can understand the precise meaning of an algorithm by reading its TLA^+ translation", and b) "an algorithm written in ^+CAL . . . can be executed—either exhaustively by model checking or with non-deterministic choices made randomly"(ibid.), using the TLC model checker.

These two features advocated for ^+CAL are not new. ASMs have been used successfully since the beginning of the 1990'ies as an accurate model for pseudo-code, explicitly proposed in this function in [2,3]. A user can *understand the* *precise meaning of ASMs directly*, namely as a natural extension of Finite State Machines (FSMs). Defining rigorously the operational semantics of ASMs uses only standard algorithmic concepts and no translation to a logic language. Furthermore, comprehensive classes of ASMs have been made *executable, using various interpreters*, the first two defined in 1990 at Quintus and at the university of Dortmund (Germany) for experiments with models of Prolog,[2] the more recent ones built at Microsoft Research (AsmL [12]) and as open source project (CoreASM [10]). ASMs have also been *linked to various standard and probabilistic model checkers* [25,9,22,15,16,14,17,23]. Last but not least from ASMs reliable executable code can be compiled, for a published industrial example see [6].

What we find interesting and helpful in ^+CAL is (besides the link it provides to model checking) the succinct programming notation it offers for denoting groups of sequentially executed instructions as atomic steps in a concurrent algorithm, interpreted as basic steps of the underlying algorithm. As mentioned above, such a malleable atomicity concept allowing sequential subcomputations and procedure calls has already been provided through the **seq**, **iterate** and submachine concepts for turbo ASMs [7], which turn a sequence or iteration of submachine steps into one atomic step for the main machine. However the label-notation of ^+CAL, which is a variation of the control state notation for FSMs, will be more familiar to algorithm designers and programmers than the **seq** notation (in AsmL [12] the name **step** is used instead of **seq**) and is more concise. We will illustrate that one can exploit the ^+CAL notation in particular as a convenient textual pendant to the FSM flowchart diagram description technique for control state ASMs which contain turbo ASM submachines. As side effect of linking corresponding features in ^+CAL and in concurrent ASMs one obtains an ASM interpretation for ^+CAL programs, which supports directly the intuitive understanding of ^+CAL constructs and is independent of the translation of ^+CAL to the logic language TLA^+.[3]

In Section 2 we extend the standard semantics of ASMs to concurrent ASMs with the **await** construct. For the standard semantics of ASMs we refer the reader to [8]. In Section 2 we link the corresponding constructs in ^+CAL and in the class of concurrent constrol state ASMs with **await**.

2 Concurrent ASMs with the await Construct

A basic ASM consists of a signature, a set of initial states, a set of rule declarations and a main rule. A rule is essentially a parallel composition of so-called transition rules **if** *Cond* **then** *Updates*, where *Updates* is a finite set of assignment statements $f(e_1, \ldots, e_n) := e$ with expressions e_i, e. In each step of the machine all its transition rules that are applicable in the given state are exe-

[2] see the historical account in [4]

[3] For the sake of completeness we remark that there is a simple scheme for translating basic ASMs to TLA^+ formulae which describe the intended machine semantics. Via such a translation one can model check ASMs using the TLC model checker of TLA^+.

cuted simultaneously (synchronous parallelism); a rule is applicable in state S if its guard *Cond* evaluates in S to true.

In more detail, the result of an M-step in state S can be defined in two parts. First one collects the set U of all updates which will be performed by any of the rules **if** *Cond* **then** *Updates* that are applicable in the given state S; by update we understand a pair (l, v) of a location l and its to be assigned value v, read: the value the expression *exp* of an assignment $f(e_1, \ldots, e_n) := e$ evaluates to in S. The reader may think of a location as an array variable (f, val), where *val* consists of a sequence of parameter values to which the expressions e_i in the left side $f(e_1, \ldots, e_n)$ of the assignment evaluate in S. Then, if U is consistent, the next (so-called internal) state $S + U$ resulting from the M-step in state S is defined as the state that satisfies the following two properties:

- for every location (f, val) that is not element of the update set U, its value, written in standard mathematical notation as $f(val)$, coincides with its value in S (no change outside U),
- each location (f, val) with update $((f, val), v) \in U$ gets as value $f(val) = v$ (which may be its previous value in S, but usually will be a new value).

In case U is inconsistent no next state is defined for S, so that the M-computation terminates abruptly in an error state because M can make no step in S. For use below we also mention that in case the next internal state $S + U$ is defined, the next step of M takes place in the state $S + U + E$ resulting from $S + U$ by the environmental updates of (some of) the monitored or shared locations as described by an update set E.

The reader who is interested in technical details can find a precise (also a formal) definition of this concept in the AsmBook [8, Sect.2.4]. In particular, there is a recursive definition which assigns to each of the basic ASM constructs[4] P its update set U such that $yield(P, S, I, U)$ holds (read: executing transition rule P in state S with the interpretation I of the free variables of P yields the update set U), for each state S and each variable interpretation I. We extend this definition here by defining $yield($**await** $Cond, S, I, U)$. The guiding principle of the definition we are going to explain is the following:

- (the agent which executes) an ASM M becomes blocked when at least one of its rules, which is called for execution and will be applied simultaneously with all other applicable rules in the given state S, is an **await** *Cond* whose *Cond* evaluates in S to false,
- (the agent which executes) M is unblocked when (by actions of other executing agents which constitute the concurrent environment where M is running) a state S' is reached where for each **await** *Cond* rule of M that is called for execution in S', *Cond* evaluates to true.

In the sequential programming or interleaving-based action systems context there is at each moment at most one applicable rule to consider and thus at most one **await** *Cond* called for execution. In the ASM computation model, where at each moment each agent fires simultaneously all the rules that are applicable, it

[4] **skip**, **par**, **if then else**, **let**, **choose**, **forall**, **seq** and machine call

seems natural to block the computation at the level of agents, so that possibly multiple **await** *Cond* machines have to be considered simultaneously. Variations of the definition below are possible. We leave it to further experimentation to evaluate which definition fits best practical needs, if we do not want to abandon the advantage of the synchronous parallelism of basic ASMs (whose benefit is to force the designer to avoid sequentiality wherever possible).

The technical idea is to add to the machine signature a location *phase* with values *running* or *wait*,[5] which can be used to prevent the application of the internal state change function $S+U$ in case *Cond* of an **await** *Cond* called for execution does not evaluate to true in S. In this case we define **await** *Cond* to yield the *phase* update $(phase, wait)$ and use the presence of this update in U to block the state change function $S+U$ from being applied. This leads to the following definition:

```
$yield(\textbf{await}\ Cond, S, I, U) =$
    $\emptyset$                  **if** $Cond$ is true in $S$ // proceed
    $\{(phase, wait)\}$ **else** //change $phase$ to $wait$
```

We now adapt the definition of the next internal state function $S+U$ to the case that U may contain a *phase* update. The intuitive understanding of an **await** *Cond* statement is that the executing agent starts to wait, continuously testing *Cond* without performing any state change, until *Cond* becomes true (through actions of some other agents in the environment). In other words, a machine M continues to compute its update set, but upon encountering an **await** *Cond* with false *Cond*ition it does not trigger a state change. We therefore define as follows (assuming $yields(M, S, I, U)$):

- If $phase = running$ in S and U contains no update $(phase, wait)$, it means that no **await** *Cond* statement is called to be executed in state S. In this case the definition of $S+U$ is taken unchanged from basic ASMs as described above and $phase = running$ remains unchanged.
- If $phase = running$ in S and the update $(phase, wait)$ is an element of U, then some **await** *Cond* statement has been called to be executed in state S and its wait *Cond*ition is false. In this case we set $S+U = S+\{(phase, wait)\}$. This means that the execution of any **await** *Cond* statement in P whose *Cond* is false in the given state S blocks the (agent who is executing the) machine P as part of which such a statement is executed. Whatever other updates—except for the *phase* location—the machine P may compute in U to change the current state, they will not be realized (yet) and the internal state remains unchanged (except for the *phase* update).
- If in state S $phase = wait$ holds and the update $(phase, wait)$ is element of U, we set the next internal state $S+U$ as undefined (blocking effect without internal state change). This definition reflects that all the **await**

[5] *phase* could be used to take also other values of interest in the concurrency context, besides *running* and *wait* for example *ready*, *suspended*, *resumed*, etc., but here we restrict our attention to the two values *running* and *wait*.

Cond statements that are called for execution in a state S have to succeed simultaneously, i.e. to find their *Cond* to be true, to let the execution of P proceed (see the next case). In the special case of a sequential program without parallelism, in each moment at most one **await** *Cond* statement is called for execution so that in this case our definition for ASMs corresponds to the usual programming interpretation of **await** *Cond*.

- If $phase = wait$ holds in S and U contains no *phase* update $(phase, wait)$, it means that each **await** *Cond* statement that may be called to be executed in state S has its *cond* evaluated to true. In this case the next internal state is defined as $S + U + \{(phase, running)\}$, i.e. the internal state $S + U$ as defined for basic ASMs with additionally *phase* updated to *running*. Otherwise stated when all the waiting conditions are satisfied, the machine continues to run updating its state via the computed set U of updates.

The first and fourth case of this definition imply the conservativity of the resulting semantics with respect to the semantics of basic ASMs: if in a state with $phase = running$ no **await** *Cond* statement is called, then the machine behaves as a basic ASM; if in a state with $phase = wait$ only **await** *Cond* statements with true waiting *Cond*ition are called, then the machine switches to $phase = running$ and behaves as a basic ASM.

One can now define **await** *Cond M* as parallel composition of **await** *Cond* and M. Only when *Cond* evaluates to true will **await** *Cond* yield no *phase* update $(phase, wait)$ so that the updates produced by M are taken into account for the state change obtained by one M-step.

```
await $Cond$ $M$ =
  await $Cond$
  $M$
```

Remark. The reason why we let a machine M in a state S with $phase = wait$ recompute its update set is twofold. Assume that **await** *Cond* is one of the rules in the **then** branch M_1 of $M =$ **if** *guard* **then** M_1 **else** M_2, but not in the **else** branch. Assume in state S *guard* is true, $phase = running$ and *Cond* is false, so that executing **await** *Cond* as part of executing M_1 triggers the blocking effect. Now assume that, due to updates made by the environment of M, in the next state *guard* changes to false. Then **await** *Cond* is not called for execution any more, so that the blocking effect has vanished. Symmetrically an **await** *Cond* that has not been called for execution in S may be called for execution in the next state S', due to a change of a *guard* governing **await** *Cond*; if in S' *Cond*ition is false, a new reason for blocking M appears that was not present in state S. Clearly such effects cannot happen in a sequential execution model because there, a program counter which points to an **await** *Cond* statement will point there until the statement proceeds because *Cond* became true.

The above definition represents one possibility to incorporate waiting into the ASM framework. We are aware of the fact that its combination with the definition of turbo ASMs may produce undesired effects, due to the different scoping disciplines of the two constructs.[6] Other definitions of **await** *Cond* M with different scoping effect and combined with non-atomic sequentialization concepts should be tried out.

3 Linking Concurrent ASMs and ^{+}CAL Programs

The reader has seen in the previous section that in the ASM framework a basic step of a machine M is simply a step of M, which computes in the given state with given variable interpretation the set U of updates such that $\mathit{yield}(M, S, I, U)$ and applies it to produce the next state $S + U$. Once an update set U has been used to build $S + U$, there remains no trace in $S + U$ about which subsets of *Updates*, via which M-rule **if** *Cond* **then** *Updates* that is executable in the given state, have contributed to form this state. Thus each step of an ASM is considered as atomic and the grain of atomicity is determined by the choice made for the level of abstraction (the guards and the abstract updates) at which the given algorithm is described by M. Typically precise links between corresponding ASMs at different levels of abstraction are established by the ASM refinement concept defined in [5].

The ASM literature is full of examples which exploit the atomicity of the single steps of an ASM and their hierarchical refinements. A simple example mentioned in the introduction is the class of turbo ASMs, which is defined from basic ASMs by allowing also the sequential composition $M = M_1$ **seq** M_2 or iteration $M =$ **iterate** M_1 of machines and the (possibly recursive) call $M(a_1, \ldots, a_n)$ of submachines for given argument values a_i. In these cases the result of executing in state S a sequential or iterative step or a submachine call is defined by computing the comprehensive update set, produced by the execution of all applicable rules of M, and applying it to define the next state. The definition provides a big-step semantics of turbo ASMs, which has been characterized in [13] by a tree-like relation between a turbo ASM macro step and the micro steps it hides (see [8, 4.1]).

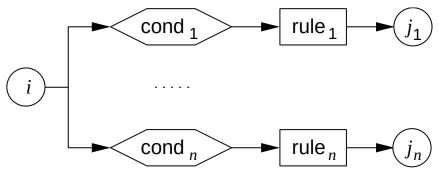
Another way to describe the partitioning of an ASM-computation into atomic steps has been introduced in [3] by generalizing Finite State Machines (FSMs) to control state ASMs. In a control state ASM M every rule has the following form:

```
Fsm($i, cond, rule, j$) =
  if $ctl\_state = i$ then
    if $cond$ then
      $rule$
      $ctl\_state := j$
```

Such rules are visualized in Fig. 2, which uses the classical graphical FSM notation and provides for it a well-defined textual pendant for control state

[6] Consider for example M **seq await** *Cond* or **await** *Cond* **seq** M, where M contains no **await**, applied in a state where *Cond* evaluates to false. The update $(\mathit{phase}, \mathit{wait})$ of **await** *Cond* will be overwritten by executing M even if *Cond* remains false.

ASMs with a rigorously defined semantics. We denote by $dgm(i, cond, rule, j)$ the diagram representing $\textsc{Fsm}(i, cond, rule, j)$. We skip *cond* when it is identical to *true*. In control state ASMs each single step is controlled by the unique current control state value and a guard. Every M-step leads from the uniquely determined current *ctl_state* value, say i, to its next value j_k (out of $\{j_1, \dots, j_n\}$, depending on which one of the guards $cond_k$ is true in the given state[7])—the way FSMs change their internal states depending on the input they read. Otherwise stated the control state pair (i, j_k) specifies the desired grain of atomicity, namely any $rule_k$ constituting a single machine step. This is essentially what is used in ^+CAL to indicate atomicity, except for the notational difference that the control states are written as labels and the fact that ^+CAL is based upon the sequential programming paradigm (see the details below).



```
if $ctl\_state = i$ then
  if $cond_1$ then
    $rule_1$
    $ctl\_state := j_1$
      ...
  if $cond_n$ then
    $rule_n$
    $ctl\_state := j_n$
```

Fig. 2. Flowchart for control state ASMs

However, each $rule_k$ may be a complex ASM, for example another control-state ASM whose execution may consists of multiple, possibly sequential or iterated substeps.[8] If their execution has to be considered as one step of the main machine M where $rule_k$ appears, namely the step determined by going from control state i to control state j_k, a notation is needed to distinguish the control states within $rule_k$ from those which define the boundary of the computation segment that is considered as atomic.

There are various ways to make such a distinction. The M **seq** N operator can be interpreted as composing control state ASMs with unique *start* and *end* control state, namely by identifying $end_M = start_N$. Suppressing the visualization of this intermediate control state, as indicated in Fig. 3, provides a way to render also graphically that the entire machine execution leading from control state $start = start_M$ to control state $end = end_N$ is considered as atomic.

[7] If two guards $cond_k, cond_l$ have a non empty intersection, in case $ctl\_state_k \neq ctl\_state_l$ an inconsistent update set U is produced so that by definition $S + U$ is undefined. If instead a non-deterministic interpretation of FSM rules is intended, this non-determinism can be expressed using the ASM **choose** construct.

[8] Replacing $\textsc{Fsm}(i, rule, j)$ by $\textsc{Fsm}(i, M, j)$ with a new control state ASM M is a frequent ASM refinement step, called procedural in [5].

This corresponds to the distinction made in the ^+CAL language: it is based upon sequential control (denoted by the semicolon) the way we are used from programming languages, but specific sequences of such sequential steps can be aggregated using labels, namely by defining as (atomic) *step* each "control path that starts at a label, ends at a label, and passes through no other labels" (op.cit., p.19). ^+CAL programs appear to be equivalent to control state ASMs with turbo submachines; the labels play the role of the control states and the turbo submachines the role of sequentially executed nonatomic ^+CAL code between two labels. This is explained by further details in Sect. 3.1 and illustrated in Sect. 3.2 by writing the major example from [21] as a turbo control state ASM.

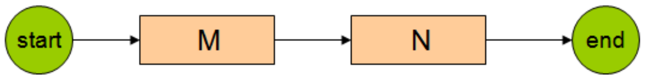


Fig. 3. Sequential composition of control state ASMs

3.1 Control State ASM Interpretation of ^+CAL Programs

^+CAL is proposed as an algorithm language to describe multiprocess algorithms. The chosen concurrency model is interleaving:

> A multiprocess algorithm is executed by repeatedly choosing an arbitrary process and executing one step of that process, if that step's execution is possible. [20, p.26]

This can be formalized verbatim by an ASM-scheme MultiProcess for multi-agent ASMs, parameterized by a given set *Proc* of constituting processes. The execution behavior of each single process is defined by the semantics of basic (sometimes misleadingly also called sequential) ASMs. The ASM **choose** construct expresses choosing, for executing one step, an arbitrary process out of the set $CanExec(Proc)$ of those $P \in Proc$ which can execute their next step:[9]

```
MultiProcess($Proc$) =
  choose $P \in CanExec(Proc)$
    $P$
```

[9] There are various ways to deal with the constraint "if that step's execution is possible". Using the ASM **choose** operator has the effect that in case $CanExec(P)$ is empty, nothing happens (more precisely: an empty update set is produced whose application does not change the current state). If one wants this case to be interpreted as explicitly blocking the scheduler, it suffices to add the **await** $CanExec(P) \neq \emptyset$ machine. The predicate $CanExec(P)$ is defined inductively.

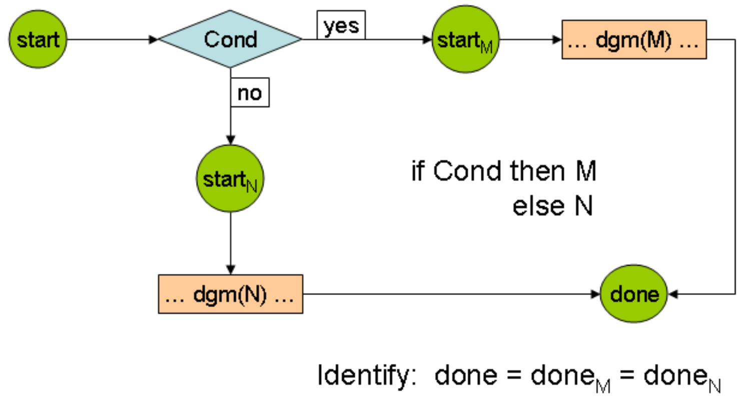
Therefore for the rest of this section we focus on describing the behavior of single ^+CAL programs by ASMs, so that MultiProcess becomes an interpreter scheme for ^+CAL programs. The program behavior is determined by the execution of the statements that form the program body (called algorithm body in ^+CAL), so that we can concentrate our attention on the operational description of ^+CAL statements and disregard here the declarations as belonging to the signature definition.

We apply the FSM flowchart notation to associate to each ^+CAL program body P a diagram $dgm(P)$ representing a control state ASM $asm(P)$ which defines the behavior of P (so that no translation of P to TLA^+ is needed to define the semantics of P). Each label in P is interpreted as what we call a *concurrent control state*. The other control states in $dgm(P)$ are called *sequential control states* because they serve to describe the sequentiality of micro-steps (denoted in P by the semicolon), the constituents of sequences which are considered as an atomic step. They are the control states that are hidden by applying the **seq**, **while** and submachine call operators. Since the construction of $dgm(P)$ uses only standard techniques we limit ourselves here to show the graphical representation for each type of ^+CAL statements. Out of these components and adding rules for the evaluation of expressions one can build a ^+CAL interpreter ASM, using the technique developed in [24] to construct an ASM interpreter for Java and JVM programs. We leave out the print statement and the assert statement; the latter is of interest only when model checking a ^+CAL program.

As basic statements a ^+CAL program can contain assignment statements or the empty statement **skip**. Program composition is done via the structured programming constructs sequencing (denoted by the semicolon), **if then else**, **while** together with **await** (named **when**) statements (a concurrent pendant of **if** statements), two forms of choice statements (nondeterminism), statements to call or return from subprograms. Since statements can be labeled, also *Goto l* statements are included in the language, which clearly correspond to simple updates of *ctl_state* resp. arrows in the graphical representation.

For the structured programming constructs the associated diagram $dgm(stm)$ defining the normal control flow consists of the traditional flowchart representation of FSMs, as illustrated in Fig. 3 and Fig. 4. One could drop writing "yes" and "no" on the two exits if the layout convention is adopted that the "yes" exit is on the upper or hight half of the rhomb and the "no" exit on the lower or left half, as is usually the case. In Fig. 4 we explicitly indicate for each diagram its control state for begin (called *start*) and end (called *done*), where each $dgm(stm)$ has its own begin and end control state, so that *start* and *done* are considered as implicitly indexed per *stm* to guarantee a unique name. Most of these control states will be sequential control states in the diagram of the entire program that is composed from the subdiagrams of the single statements. These sequential control states can therefore be replaced by the turbo ASM operator **seq** as done in Fig. 3, which makes the atomicity of the sequence explicit.

await statements are written **when** *Cond*; in ^+CAL. The semantical definition of $asm(\textbf{await}\ Cond)$ has been given within the ASM framework in Sect. 2,



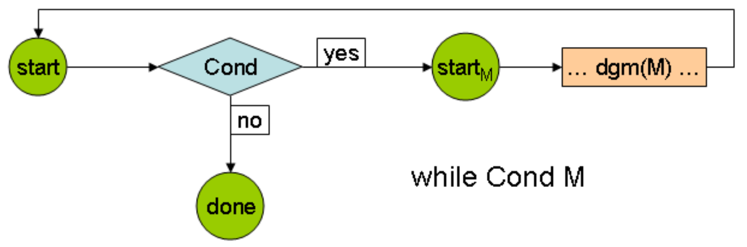


Fig. 4. Control State ASM for Structured Programming Concepts

extending the sequential programming definition in Fig. 1. For the visualization we define $dgm(\textbf{when}\ Cond;)$ by the triangle shown in that figure.

Basic statements *stm* are represented by the traditional FSM graph of Fig. 2, in Sect. 3 denoted by $dgm(start, asm(stm), done)$. The statement **skip**; "does nothing" and thus has the behavior of the homonymous ASM $asm(\textbf{skip}\ ;) =$ **skip**, which in every state yields an empty update set. An assignment statement in ^+CAL is a finite sequence $stm = lhs_1 := exp_1 \mid\mid \ldots \mid\mid lhs_n := exp_n$ of assignments, executed by "first evaluating the right-hand sides of all its assignments, and then performing those assignments from left to right". This behavior is that of the following ASM, where the expression evaluation is performed in parallel for all expressions in the current state, whereafter the assignment of the computed values is done in sequence.[10]

[10] It seems that this treatment of assignment statements is related to the semantics of nested EXCEPTs in TLA^+ and thus permits a simple compilation.

$$\begin{array}{l} asm(lhs_1 := exp_1 \mid\mid \ldots \mid\mid lhs_n := exp_n\ ;) = \\ \quad \textbf{forall}\ 1 \leq i \leq n\ \textbf{let}\ x_i = exp_i \\ \qquad lhs_1 := x_1\ \textbf{seq} \ldots \textbf{seq}\ lhs_n := x_n \end{array}$$

The behavior of statements *Goto l ;* is to "end the execution of the current step and causes control to go to the statement labeled *l*" [20, p.25], wherefor such statements are required to be followed (in the place where they occur in the program) by a labeled statement. Thus the behavior is defined by $asm(Goto\ l\ ;) = (ctl\_state := l)$ and $dgm(Goto\ l\ ;)$ as an arrow leading to control state *l* from the position of the *Goto l ;*, which is a control state in case the statement is labeled.

The two constructs expressing a non determinstic choice can be defined as shown in Fig. 5.

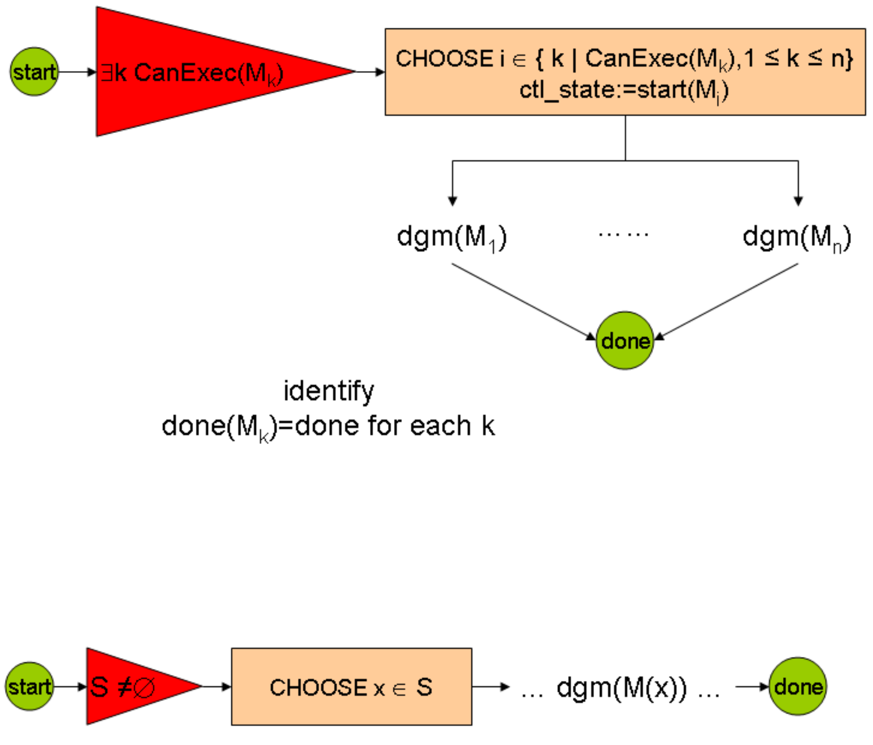


Fig. 5. Control State ASMs for Choice Statements

The first type of non deterministic statement **either** M_1 **or** M_2 **or** ... **or** M_n; chooses an executable statement among finitely many statements M_i. It is defined to be executable if and only if one of M_i is executable. This means that the execution of the statement has a blocking character, namely to wait as long as none of the substatements is executable. Similarly the second type of non deterministic statement, written **with** $id \in S$ **do** M;, is meant to choose an element in a set S if there is one and to execute M for it. The statement is considered as not executable (blocking) if the set to choose from is empty.

Since the ASM **choose** construct is defined as non blocking, but yields an empty update set in case no choice is possible, we make use of the **await** construct to let **choose** have an effect only when there is something to choose from. We write *CanExec* for the executability predicate.

```
$asm(\textbf{either } M_1 \textbf{ or } M_2 \textbf{ or} \ldots \textbf{or } M_n;) =$
  $\textbf{await forsome } 1 \leq j \leq n \ CanExec(M_j)$
  $\textbf{choose } i \in \{j \mid CanExec(M_j) \textbf{ and } 1 \leq j \leq n\}$
    $asm(M_i)$

$asm(\textbf{with } id \in S \textbf{ do } M;) =$
  $\textbf{await } S \neq \emptyset$
  $\textbf{choose } id \in S$
    $asm(M(id))$
```

The remaining ^+CAL statements deal with procedure call and return in a standard stack machine like manner. Define *frame* as quadruple consisting of a control state, the values of the procedure's arguments respectively of its local variables and the procedure name. We denote the frame stack by a location *stack* and the current frame by a quadruple of four locations *ctl_state*, *args* (which is a sequence, of any finite length, of variables standing for the parameters), *locals* (which is a sequence, of any finite length, of local variables) and *proc* (which denotes the currently executed procedure). For a call statement $P(expr_1, \ldots, expr_n)$;, "executing this call assigns the current values of the expressions $expr_i$ to the corresponding parameters $param_i$, initializes the procedure's local variables, and puts control at the beginning of the procedure body", which "must begin with a labeled statement" [20, p.27]. As preparation for the return statement one has also to record the current frame on the frame *stack*. We denote the sequence of local variables of P by $locVars(P)$ and their initial values by *initVal*. For the sake of brevity, for sequences *locs* of locations and *vals* of values we write $locs := vals$ for the simultaneous componentwise assignment of the values to the corresponding locations, to be precise for the machine $asm(locs_1 := vals_1 \mid\mid \ldots \mid\mid locs_n := vals_n \ ;)$ defined above where $locs = (locs_1, \ldots, locs_n)$ and $vals = (vals_1, \ldots, vals_n)$. Let $start_P$ denote the label of the first statement of P.

```
$asm(P(exp_1, \ldots, exp_n)) =$
  $\textsc{PushFrame}(P, (exp_1, \ldots, exp_n)$
$\textbf{where}$
```

```
$\textsc{PushFrame}(P, exps) =$
    $stack := stack.[ctl\_state, args, locals, proc]$ // push current frame
    $proc := P$
    $args := exps$ // pass the call parameters
    $locals := initVal(locVars(P))$ // initialize the local variables
    $ctl\_state := start_P$ // start execution of the procedure body
```

A return statement consists in the inverse machine PopFrame. If *ctl* is the point of a call statement in the given program, let $next(ctl)$ denote the point immediately following the call statement.

```
$asm(return) =$
  let $stack = stack'.[ctl, prevArgs, prevLocs, callingProc]$ in
    $ctl\_state := next(ctl)$ // go to next stm after the call stm
    $args := prevArgs$ // reassign previous values to args
    $locals := prevLocs$ // reassign previous values to locals
    $proc := callingProc$
    $stack := stack'$ // pop frame stack
```

3.2 Fast Mutex Example

Fig. 6 illustrates the diagram notation explained in Sect. 3.1 for ^{+}CAL programs.

The ^{+}CAL program for the fast mutual exclusion algorithm from [19] is given in [21]. Fig. 6 does not show the declaration part, which is given in the signature definition. We write Ncs and Cs for the submachines defining the non critical resp. critical section, which in ^{+}CAL are denoted by an atomic **skip** instruction describing—via the underlying stuttering mechanism of *TLA*—a nonatomic program. Given the structural simplicity of this program, which says nothing about the combinatorial complexity of the runs the program produces, there is only one sequential subprogram. It corresponds to two simultaneous updates, so that the sequentialization can be avoided and really no turbo ASM is needed because there are no control states which do not correspond to ^{+}CAL labels. This case is a frequent one when modeling systems at an abstract level, as the experience with ASMs shows. In general, the synchronous parallelism of ASMs drives the model designer to avoid sequentialization as much as possible and to think instead about orthogonal components which constitute atomic steps.

Comparing the two representations the reader will notice that even the layouts can be made to be in strict correspondence, so that each of the labeled lines in the textual description corresponds to a line starting a new control state subdiagram. This is in line with the following well-known fact we quote from [18, p.72]:

> The visual structure of **go to** statements is like that of flowcharts, except reduced to *one* dimension in our source languages.

We can confirm from our own work the experience reported in [21] that the notation works well for programs one can write on a couple of pages, making

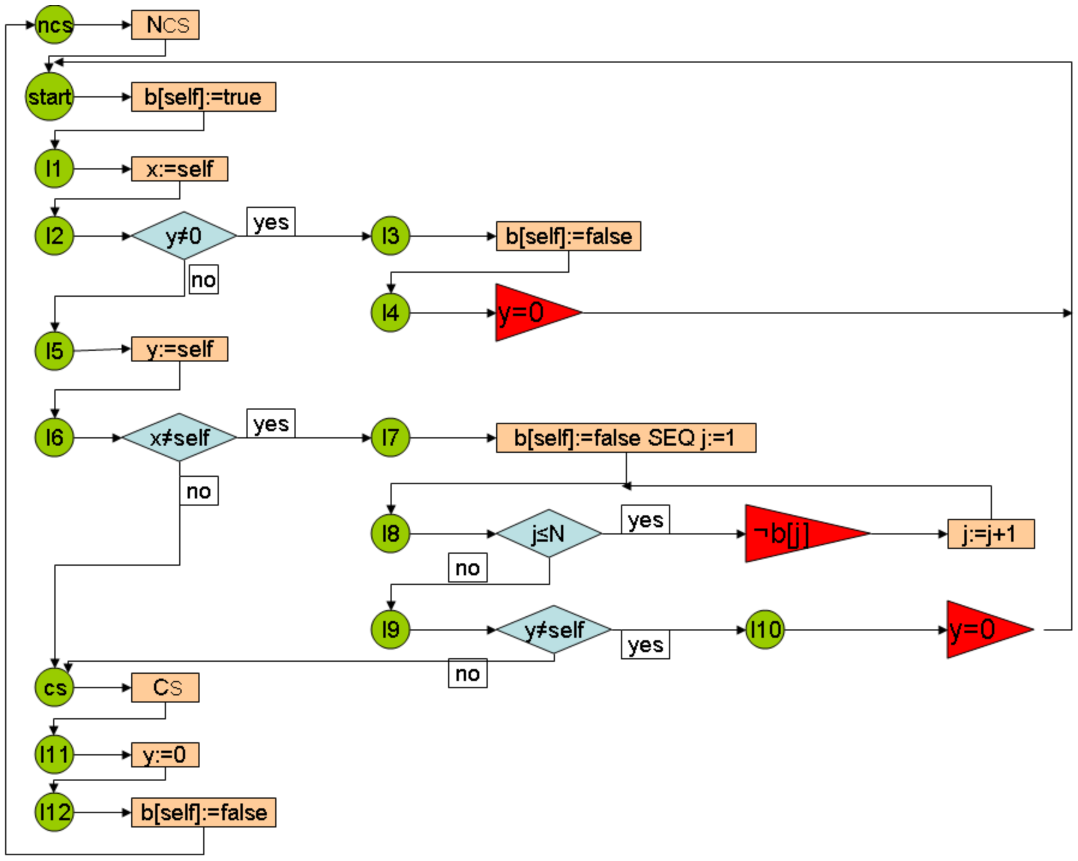


Fig. 6. Control state ASM for the Fast Mutual Exclusion ^+CAL Program

judicious use of procedures where possible to cut down the size of each single program one has to analyze. Such programs seem to be the main target for ^+CAL code and model checkable TLA^+ translations for concurrent algorithms with combinatorially involved behaviour. For larger programs flowcharts present some small advantage over the representation of programs as strings, however, as Knuth continues op.cit.:

> . . . we rapidly loose our ability to understand larger and larger flowcharts; some intermediate levels of abstraction are necessary.

The needed abstractions can be provided in control state ASMs by using separately defined complex submachines, which in the flowcharts appear as simple rectangles to be executed when passing from one to the next control state. This follows an advice formulated by Knuth op.cit. as one of the conclusions of his discussion of structured programming with **go to** statements:

> . . . we should give meaningful names for the larger constructs in our program that correspond to meaningul levels of abstraction, and we should define those levels of abstraction in one place, and merely use their names (instead of including the detailed code) when they are used to build larger concepts.

Acknowledgement We thank L. Lamport and S.Merz for the discussion of a preliminary version of this paper. We also thank M. Butler and A. Prinz for some helpful comment and S. Tahar for pointing out a relevant reference.

To appear in: Proc. WADT'08, Springer LNCS, 2009, eds. A. Corradini and U. Montanari.

References

1. J.-R. Abrial. *The B-Book*. Cambridge University Press, Cambridge, 1996.
2. E. Börger. Why use Evolving Algebras for hardware and software engineering? In M. Bartosek, J. Staudek, and J. Wiederman, editors, *Proc. SOFSEM'95, 22nd Seminar on Current Trends in Theory and Practice of Informatics*, volume 1012 of *Lecture Notes in Computer Science*, pages 236–271. Springer-Verlag, 1995.
3. 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)*, volume 1641 of *Lecture Notes in Computer Science*, pages 1–43. Springer-Verlag, 1999.
4. E. Börger. The origins and the development of the ASM method for high-level system design and analysis. *J. Universal Computer Science*, 8(1):2–74, 2002.
5. E. Börger. The ASM refinement method. *Formal Aspects of Computing*, 15:237–257, 2003.
6. E. Börger, P. Päppinghaus, and J. Schmid. Report on a practical application of ASMs in software design. In Y. Gurevich, P. Kutter, M. Odersky, and L. Thiele, editors, *Abstract State Machines: Theory and Applications*, volume 1912 of *Lecture Notes in Computer Science*, pages 361–366. Springer-Verlag, 2000.
7. E. Börger and J. Schmid. Composition and submachine concepts for sequential ASMs. In P. Clote and H. Schwichtenberg, editors, *Computer Science Logic (Proceedings of CSL 2000)*, volume 1862 of *Lecture Notes in Computer Science*, pages 41–60. Springer-Verlag, 2000.
8. E. Börger and R. F. Stärk. *Abstract State Machines. A Method for High-Level System Design and Analysis*. Springer, 2003.
9. G. Del Castillo and K. Winter. Model checking support for the ASM high-level language. In S. Graf and M. Schwartzbach, editors, *Proc. 6th Int. Conf. TACAS 2000*, volume 1785 of *Lecture Notes in Computer Science*, pages 331–346. Springer-Verlag, 2000.
10. R. Farahbod et al. *The CoreASM Project*. http://www.coreasm.org.
11. R. Farahbod, V. Gervasi, and U. Glässer. CoreASM: An Extensible ASM Execution Engine. *Fundamenta Informaticae XXI*, 2006.
12. Foundations of Software Engineering Group, Microsoft Research. AsmL. Web pages at http://research.microsoft.com/foundations/AsmL/, 2001.

13. N. G. Fruja and R. F. Stärk. The hidden computation steps of turbo Abstract State Machines. In E. Börger, A. Gargantini, and E. Riccobene, editors, *Abstract State Machines 2003–Advances in Theory and Applications*, volume 2589 of *Lecture Notes in Computer Science*, pages 244–262. Springer-Verlag, 2003.
14. A. Gargantini, E. Riccobene, and S. Rinzivillo. Using Spin to generate tests from ASM specifications. In E. Börger, A. Gargantini, and E. Riccobene, editors, *Abstract State Machines 2003–Advances in Theory and Applications*, volume 2589 of *Lecture Notes in Computer Science*, pages 263–277. Springer-Verlag, 2003.
15. A. Gawanmeh, S. Tahar, and K. Winter. Interfacing ASMs with the MDG tool. In E. Börger, A. Gargantini, and E. Riccobene, editors, *Abstract State Machines 2003–Advances in Theory and Applications*, volume 2589 of *Lecture Notes in Computer Science*, pages 278–292. Springer-Verlag, 2003.
16. A. Gawanmeh, S. Tahar, and K. Winter. Formal verification of asms using mdgs. *Journal of Systems Architecture*, 54(1-2):15–34, January-February 2008.
17. U. Glässer, S. Rastkar, and M. Vajihollahi. Computational Modeling and Experimental Validation of Aviation Security Procedures. In S. Mehrotra, D. D. Zeng, H. Chen, B. M. Thuraisingham, and F.-Y. Wang, editors, *Intelligence and Security Informatics, IEEE International Conference on Intelligence and Security Informatics, ISI 2006, San Diego, CA, USA, May 23-24, 2006, Proceedings*, volume 3975 of *LNCS*, pages 420–431. Springer, 2006.
18. D. Knuth. Structured programming with goto statements. *Computing Surveys*, 6, December 1974.
19. L. Lamport. A fast mutual exclusion algorithm. *ACM Transactions of Computer Systems*, 5(1):1–11, 1987.
20. L. Lamport. A +CAL user's manual.P-syntax version. URL http://research.microsoft.com/users/lamport/tla, June 29 2007.
21. L. Lamport. The +CAL algorithm language. URL http://research.microsoft.com/users/lamport/tla/pluscal.html, February 14 2008.
22. C. N. Plonka. Model checking for the design with Abstract State Machines. Diplom thesis, CS Department of University of Ulm, Germany, January 2000.
23. A. Slissenko and P. Vasilyev. Simulation of timed Abstract State Machines with predicate logic model-checking. *J. Universal Computer Science*, 14(12):1984–2007, 2008.
24. R. F. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine: Definition, Verification, Validation.* Springer-Verlag, 2001. .
25. K. Winter. Model checking for Abstract State Machines. *J. Universal Computer Science*, 3(5):689–701, 1997.