A Universal Control Construct for Abstract State Machines

Michael Stegmaier, Marcel Dausend, Alexander Raschke(✉), and Matthias Tichy

Institute of Software Engineering and Compiler Construction, Ulm University, 89069 Ulm, Germany
{michael-1.stegmaier,marcel.dausend,alexander.raschke, matthias.tichy}@uni-ulm.de

Abstract. Abstract State Machines can be used to specify arbitrary system behaviour. However, when writing executable specifications one often has to write additional statements which organise how, e.g., in which order, the rules are executed. This reduces the readability and comprehensibility of specifications and can introduce additional defects to them. We propose a new syntax construct for the specification of control flow for the ASM language which improves the compactness and readability of specifications by providing syntactic elements for often manually realised behaviour. This construct enables to parametrise which rules shall be selected for execution and how the selected rules are executed. We illustrate how the control construct can improve the code's readability on some examples. The proposed control construct is also released as a plugin for CoreASM.

Keywords: Abstract State Machines · Control construct · Control flow

1 Introduction

Abstract State Machines (ASMs) (see [5]) allow a formal description of the functional requirements in the analysis and design phase. They are a state-based specification language, as they allow to model a software system or hardware system by states and possible state transitions.

Unlike in finite state machines, states in ASMs don't have names. They are general mathematical structures instead. These mathematical structures are universes (non-empty sets) together with functions operating on the sets. This underlying mathematical approach leads to an improvement of verifiability and reusability [3]. They offer a conceptually simple, yet flexible approach for specifying state transition systems.

In ASMs, control flow is realised through a combination of multiple basic control constructs. The specification of a semantically complex control flow is often hard to realise using the basic control constructs and, hence, results in high nesting depths. High nesting depths increase complexity and therefore deteriorate readability [15].

© Springer International Publishing Switzerland 2016
M. Butler et al. (Eds.): ABZ 2016, LNCS 9675, pp. 37–53, 2016.
DOI: 10.1007/978-3-319-33600-8_2

In this paper, we propose a universal control construct (UCC) that unites different step semantics (**parallel**, **sequence**, **rulebyrule**, **stepwise**) and conditional blocks such as **if**, **while** and **iterate**. Furthermore, it provides the possibility to limit the execution of a block to a given number of repetitions which can be useful for situations like initialisation. Last but not least, it provides a way to select and execute only a subset of rules which can be useful, for example, when choosing a strategy or a heuristic for an algorithm or for the simulation of errors.

In the next section, we review the current support for the specification of control flow in ASM and identify several shortcomings using concrete examples. In Sect. 3, we present the proposed control construct, its syntax and its semantics and compare it to related work in Sect. 4. We conclude the paper in Sect. 5 and give an outlook on future work.

2 Shortcomings of Current ASM Control Constructs

This section shows some shortcomings of current ASM specifications and motivates the introduction of a more powerful universal control construct by means of examples. The meaning of the presented control construct is quite intuitive for the reader and should be understandable without a precise definition of the semantics as given in Sect. 3. For better readability we use (parallel) nesting by indentation.

In complex specifications of real systems, the notion of basic ASMs as a list of guarded updates fired in parallel often does not fit. Usually, after an initialisation phase, several steps have to be performed in sequence. Introducing modes is a common pattern for specifying this behaviour. This class of ASM specifications is named "control state ASMs" [5]. Mode variables have to be defined and each rule is guarded by a mode condition such that only one rule is executed per ASM machine step.

An example for this applied pattern is given in Listing 1. It is the main rule of the specification of the operational semantics of the control construct proposed in this paper. The initial value of the mode is assumed to be INIT.

```
1  rule Main =
2      if mode = INIT then
3          Initialise
4      if mode = SELECT then
5          Selection
6      if mode = PREPARE_EXECUTION then
7          PrepareExecution
8      if mode = EXECUTION then
9          Execution
10     if mode = RESET then
11         Reset
```

Listing 1. Main rule of UCC specification

One problem of the control state ASM pattern is that it is not easy to extract the order of modes from the specification. This is because the subsequent modes are set inside nested guards in separate rules.

In our example, the reader needs the whole specification to gain the insight that (in this particular case) the INIT mode is executed only once and after that, the remaining rules are executed rule by rule in an infinite loop.

Our proposed construct aims at overcoming this weakness as shown in Listing 2. UCC allows for defining that rules are executed only once (**at most** 1 **times**) and that the other rules are executed rule by rule per machine step (**stepwise**). In this example, the order of modes is non-linear which is also a common case. Depending on whether a new selection should be made, the mode following RESET either is SELECT or PREPARE_EXECUTION. This non-linearity cannot directly be specified using **stepwise**, therefore the guard in line 6 is needed. If this guard evaluates to **false** the conditional rule is treated as a **skip**. Thus, the UCC forces the user to make the condition under which rules are executed more explicit and more visible. Obviously, this circumstance is not always an advantage. In specifications realising very complex automata it might be better to not linearise the sequence of the modes.

```
1  rule Main =
2    perform always stepwise
3      perform at most 1 times
4        Initialise
5      end
6      if shouldSelect then
7        Selection
8      PrepareExecution
9      Execution
10     Reset
```

Listing 2. Improved MAIN rule using UCC

Another example for better readability of specifications by hiding technical (yet necessary for execution) details is given in Listing 3. For testing and demonstration purposes, the specified system behaves normally or it simulates a subset of three different errors. The current behaviour shall be chosen non-deterministically for each step. Listing 3 specifies an environment for a safety-critical system that should be able to cope with different kinds of errors. An arbitrary subset of the error simulating rules is chosen and executed in parallel. If there was no error the environment should behave normally.

```
1  choose errorsToSimulate ⊆ {SimulateSensorError,
       SimulateTemperatureError, SimulateCommunicationError} do
2    if |errorsToSimulate| = 0 then
3      NormalBehaviour
4    else
5      forall r ∈ errorsToSimulate do
6        r
```

Listing 3. A subset of all errors can occur simultaneously

Using the UCC, the complex rule of Listing 3 can be condensed into the succinct rule of Listing 4.

```
1 perform always single variable selection
2     NormalBehaviour
3     perform any nonempty variable selection
4         SimulateSensorError
5         SimulateTemperatureError
6         SimulateCommunicationError
```

Listing 4. Equivalent specification as in Listing 3 using UCC

The following example (Listing 5) shows excerpts of a specification of the A* algorithm [14] using ASMs. The A*-algorithm is a heuristic method to determine the shortest path between two nodes in a directed graph with only positive edge weights. To illustrate the algorithm, the sliding puzzle has been chosen, which is also known as 15-puzzle (see [12]). In this puzzle, there are fifteen numbered tiles and one free place. The goal of the game is to repeatedly move tiles into the free place until the desired state is reached. In this specification, multiple heuristics have been realised.

Listing 5 shows the initialisation rule of this specification. Besides the already mentioned typical initialisation mode, the heuristic to use for the algorithm is chosen in this rule. Since all functions in ASMs are globally accessible there is no simple way to ensure that the heuristic will not change throughout a complete run of the algorithm (see Listing 5, line 9).

```
1  rule InitialiseAstar =
2      if not initialised then
3          seq
4              MakePuzzle
5              FindEmpty(InitialState)
6              root ← CreateNode(InitialState, undef, undef)
7              OpenListEnqueue(root)
8          endseq
9          choose h ∈ HEURISTIC do heuristic := h
10         initialised := true
```

Listing 5. The rule InitialiseAstar of *Specification of A**

Using UCC, the rule InitialiseAstar becomes significantly shorter and now only consists of the actual initialisation of the algorithm. Using **perform at most 1 times in sequence** the contained ruleblock will be executed in sequence and at most once. This way we make sure that the algorithm is initialised only once. We can omit the function `initialised` now because UCC takes care of making sure that the initialisation is never re-executed.

```
1 rule InitialiseAstar =
2   perform at most 1 times in sequence
3     MakePuzzle
4     FindEmpty(InitialState)
5     root ← CreateNode(InitialState, undef, undef)
6     OpenListEnqueue(root)
```

Listing 6. The rule InitialiseAstar of *Specification of A** using the proposed construct

Furthermore, the heuristic does not have to be decided during the initialisation anymore (Listing 6). Instead, the heuristic can be chosen permanently at the point where it is needed (see Listing 7).

```
1 derived GetHeuristicalValue(state) = return value in
2     perform single fixed selection
3         value := CalcGoalHeuristic(state)
4         value := CalcMisplacedTiles(state)
5         value := CalcManhattan(state)
```

Listing 7. The rule GetHeuristicalValue of *Specification of A** using the proposed construct

By using the **single fixed selection**, we ensure that a single rule (a single heuristic in this case) is selected and will always be selected (fixed) for the complete run of the specification. This use of the UCC has the following advantages:

- A permanent random decision does not need to take place during the initialisation anymore.
- The function storing the random permanent decision can be omitted. The UCC can remember its random decision and it ensures that its decision cannot be changed from outside.
- The decision is made at the point where it is needed. The reader won't need to search the specification in order to find out how the decision is being made.

In the next section, the syntax and semantics of the UCC is defined in details. It supports the control constructs already available in (Turbo-)ASM as well as additional description possibilities as shown in this section. This unification also reduces the nesting depth and, thus, improves readability [15].

3 A Universal Control Construct for ASM

The goal of the proposed control construct is to provide a succinct high-level description scheme, formulated in intuitive terms, one can use to specify complex control structures and reduce nesting depth. In this section, its syntax and semantics are defined.

3.1 Syntax

For our control construct, we propose the syntax defined by the following grammar shown in Listing 8. The nonterminals `Term`, `ConstantTerm` and `Rule` are defined as expected [7].

```
Selection = 'all' | SubSelection;
SubSelection = SubSelectionSize ('variable' | 'fixed') 'selection';
SubSelectionSize = ('any' ['nonempty']) | 'single';
Enabled = ('always' | EnabledAtMost | EnabledUntil) [EnabledReset];
EnabledAtMost = 'at most' ConstantTerm 'times';
```

```
EnabledUntil = 'until' ('no updates' | 'no change');
EnabledReset = 'resetting' 'on' ConstantTerm;
StepSemantics = 'in' ('parallel' | 'sequence') | 'rulebyrule' | 'stepwise';
Condition = (('if' | 'while') Term) | 'iterate';
RuleBlock = Rule {Rule};
Semantics = ?Any sequence of Selection, StepSemantics, Enabled?;
PerformRule = 'perform' [Semantics] [Condition] RuleBlock ['end'];
```

Listing 8. The resulting grammar for UCC

The syntax and the corresponding semantics are split up into four groups which cover different orthogonal aspects of the control construct: (1) *Selection* supports selecting a subset of rules from the rule block, (2) *Enabled* determines whether the construct should be enabled or not, (3) *StepSemantics* determines the step semantics to be used for the execution of the selected rules, and (4) *Condition* provides a condition that has to be **true** before the selected rules can be executed. The first group of keywords is the *Selection* group:

- **all** - The whole ruleblock is selected for execution.
- **any** - A random subset of rules from the ruleblock is selected for execution. It can be attributed by the keyword **nonempty** to avoid selecting an empty subset.
- **single** - A single random rule from the ruleblock is selected for execution.

An **any** or **single** selection can either be **variable**, i.e., a selection is made for every evaluation of the construct, or **fixed**, i.e., the selection is permanently made in the first evaluation of the construct and reused for consecutive evaluations. The second group of keywords is the *Enabled* group:

- **always** - The construct is always enabled.
- **at most** n **times** - The construct is enabled at most n times.
- **until no change** - The construct is only enabled until there's no update resulting from the evaluation of the construct or all the resulting updates are trivial [10], that is, no update (if any) resulting from the evaluation of the construct does change the value of any function.
- **until no updates** - The construct is enabled until there are no updates resulting from the evaluation of the construct.

If the construct is not enabled anymore, it will not do anything. The construct can only be re-enabled if a reset condition is specified in the **resetting on** part. The third group of keywords is the *StepSemantics* group:

- **in parallel** - The rules are executed in parallel.
- **in sequence** - The rules are executed in sequence.
- **rulebyrule** - The rules are executed rule by rule. That is, in the first evaluation of this construct the first rule is executed, in the second evaluation the second rule is executed and so on. After the last rule of the block the first rule is executed again and so on.

– **stepwise** - Similar to **rulebyrule** the rules are executed rule by rule. The difference is that with **stepwise**, the same rule is executed for every evaluation during the same machine step. In the next machine step, the next rule is executed and so on. This difference is explained in more detail on the next page.

The fourth group of keywords is the *Condition* group:

– **if** - The selected rules are only executed if the specified condition evaluates to **true**.
– **while** - The selected rules are executed in a loop as long as the specified condition evaluates to **true**. This corresponds to the semantics of the turbo rule **while**.
– **iterate** - The selected rules are executed in a loop as long as they produce updates. This corresponds to the semantics of the turbo rule **iterate**.

In general, the construct must not be confused with a loop. Most notably, the *Enabled* part does not specify a looping condition. For example, **at most** n **times** does not mean that the construct loops n times. Instead it means that the construct must not be evaluated more than n times at all. After the nth repetition, the construct is disabled and cannot be executed anymore. That is, it will behave like a **skip**. The construct is re-enabled if the condition provided for **resetting on** evaluates to **true**. But it is possible to make the construct loop by either using **while** or **iterate**. While there is a significant similarity in the descriptions of **iterate** and **until no updates**, there is a major difference. The keyword **iterate** causes the construct to loop while the keyword **until no updates** does not.

The difference between **rulebyrule** and **stepwise** occurs when using them in conjunction with loops. Loops allow a construct to be evaluated multiple times during the same machine step. With **rulebyrule**, one rule after another is executed within the loop. With **stepwise**, always the same rule is executed during the loop. Only if the UCC is reached again in another machine step, the next rule is executed.

```
1 forall i in {1, 2} do
2    perform rulebyrule r1, r2, r3
3    perform stepwise r1, r2, r3
```

Listing 9. Example for stepwise vs. rulebyrule

Figure 1 illustrates the difference between **stepwise** and **rulebyrule** using the example in Listing 9. It shows a time line on which each machine step is indicated by a vertical bar. Each row shows the rules that are to be executed by the perform rule in the respective iteration of the loop. The rule that is executed in the current iteration is marked. The rules to be executed with **rulebyrule** are above the time line and the rules to be executed with **stepwise** are beneath the arrow.

In our syntax, we allow every sequence of the groups of keywords (see Listing 8). This way we improve the learning curve of our construct by

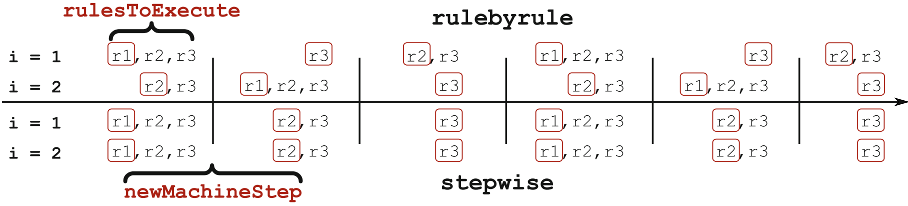

Fig. 1. Illustration of stepwise vs. rulebyrule

reducing the cognitive load of the specification author. He already has to remember all the keywords he wants to use, so at least he does not have to remember the sequence as well. This is not a threat to readability because the groups of keywords are semantically independent from each other.

An obvious criticism of the UCC would be the introduction of many different keywords. In terms of a programming language, this is a clear disadvantage. For one thing, all those keywords are reserved and cannot be used as identifiers anymore, for another, the author of a specification has to remember the keywords to effectively make use of the control construct. So initially, writing specifications with the UCC can be even more difficult. But at the same time, the control construct approaches a natural language even closer by using many different keywords. Getting closer to a natural language usually improves readability for non-experts. Since ASM specifications are often used for communication with customers, certain emphasis should be placed on readability.

3.2 Semantics

In the following specification of the operational semantics, we use the following auxiliaries to work with lists:

- *head* returns the head of the list, that is, the first element of the list.
- *tail* returns the tail of the list, that is, a list of all elements following the first element.

In general, there are different ways to describe the semantics of ASM constructs. One possibility is to define rules of inference for the update set for a construct [5]. Another possibility is the operational approach used in the design specification of CoreASM (see [6]). In this formalisation, the interpretation of each expression is described using ASMs producing a complex value containing the calculated update set and the result of the evaluation. In this approach, all partial update sets are collected and at a machine step applied to the current state.

In this paper, we also describe the semantics using ASMs. Instead of defining the partial interpretation of each part of each construct, we describe the interpreter of the UCC as a whole. Although it is not easily possible to integrate our definition into an execution engine like CoreASM, the translation process is

straightforward. The advantage of the presented approach is the linearity of the description. We split the semantics into four phases that are described in detail in the following sections:

1. SELECTION A subset of rules is selected from the ruleblock.
2. PREPAREEXECUTION If the construct is still enabled, the selected rules are copied for execution.
3. EXECUTION The selected rules are executed and the copy of the selection is adapted according to the specified step semantics.
4. RESET The repetition state is reset if the given reset condition evaluates to **true**.

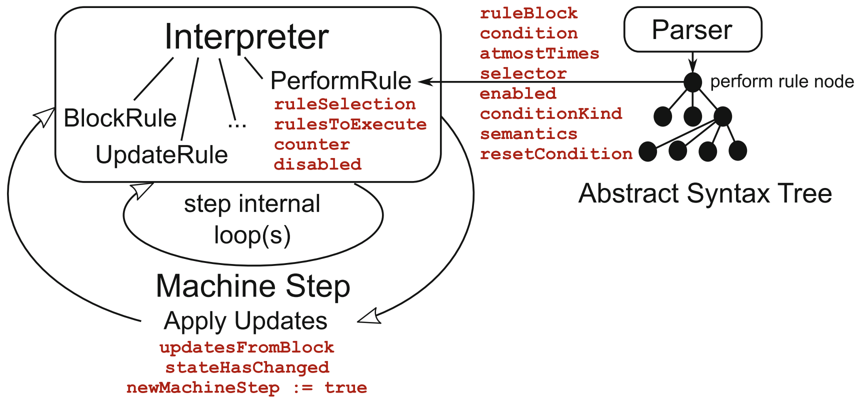

Fig. 2. Run Loop

For the description we assume that an occurrence of the UCC is already parsed and available in a proper model (see Fig. 2) that provides access to the following parts of the construct:

- The contained block of rules (accessed via the function ruleBlock).
- The condition of the *Condition* group (accessed via the function condition).
- The value of the term (n) provided to **at most** n **times** (accessed via the function atmostTimes).
- The selector ∈ {**ALL**, **SINGLE**, **ANY**, **ANY_NON_EMPTY**} (accessed via the function selector).
- The keyword from the *Enabled* group ∈ {**ATMOST_N_TIMES**, **UNTIL_NO_CHANGE**, **UNTIL_NO_UPDATES**, **ALWAYS**} (accessed via the function enabled).
- The keyword from the *Condition* group ∈ {**IF**, **WHILE**, **ITERATE**} (accessed via the function conditionKind).
- The step semantics ∈ {**IN_PARALLEL**, **IN_SEQUENCE**, **RULEBYRULE**, **STEPWISE**} (accessed via the function semantics).

- The reset condition of the *Enabled* group (accessed via the function resetCondition).

Furthermore, the following functions are used to maintain the state of the construct:

- ruleSelection holds the current selection and results from the SELECTION phase.
- rulesToExecute holds the rules that are to be executed in the EXECUTION phase. It results from the PREPAREEXECUTION phase. It is initialised as [].
- counter keeps track of the number of executions when **at most** n **times** is used. Its value is only relevant for the PREPAREEXECUTION phase. In that phase it gets increased for each execution. It is initialised as 0 and can only be reset to 0 in the RESET phase (if resetCondition evaluates to **true**).
- disabled is a flag that keeps track of whether the construct has been disabled. Its value is only relevant for the PREPAREEXECUTION phase. In that phase it gets set to **true** depending on the condition associated with the keyword specified for enabled. It is initialised as **false** and can only be reset to **false** in the RESET phase (if resetCondition evaluates to **true**).
- updatesFromBlock is a flag indicating whether the last RUN of the rules has produced updates. Its value is relevant for the PREPAREEXECUTION phase and the EXECUTION phase. Its value implicitly results from running the rules. It is initialised as **true**.
- stateHasChanged is a flag indicating whether the last RUN of the rules has changed the state, i.e., the updates resulting from running the rules change the value of a function. Its value is only relevant for the PREPAREEXECUTION phase. Its value implicitly results from running the rules. It is initialised as **true**.
- newMachineStep is a flag indicating whether the last interpretation of the construct was in another machine step than the current. Its value is only relevant for the PREPAREEXECUTION phase. Its value is set to **true** by the environment as soon as a new machine step is started. The construct sets its value to **false** in the PREPAREEXECUTION phase in order to remember that the construct has already been interpreted in the current machine step. This distinction is required for the difference between the semantics of **stepwise** and **rulebyrule**.

3.3 SELECTION Phase

The SELECTION phase is specified by the rule SELECTION (see Listing 10). It selects the rules to execute from the ruleBlock. With the keyword **all**, the selection corresponds to the whole ruleBlock (see Listing 10, line 4). With the keyword **single**, only an arbitrary single rule is selected (see Listing 10, line 6). With the keyword **any**, an arbitrary subset of rules is selected (see Listing 10, line 8). This selection can be empty. If the keyword **any** is attributed by the keyword **nonempty** an arbitrary subset of rules that is not empty is selected from the ruleBlock (see Listing 10, line 10).

```
1  rule Selection =
2    case selector of
3      ALL:
4        ruleSelection := ruleBlock
5      SINGLE:
6        choose r ∈ ruleBlock do ruleSelection := [r]
7      ANY:
8        choose s ⊆ ruleBlock do ruleSelection := s
9      ANY_NON_EMPTY:
10       choose s ⊆ ruleBlock with s ≠ ∅ do ruleSelection := s
11   endcase
12   mode := PREPARE_EXECUTION
```

Listing 10. Selection rule

3.4 PrepareExecution phase

The PrepareExecution phase is specified by the rule PrepareExecution (see Listing 11). It determines the rules to execute. With the keyword **always**, this always is the whole selection (see Listing 11, line 19). With the keyword **at most**, the rules to execute are the selection repeated n times with n being a constant natural number (see Listing 11, lines 5 – 7). With the keyword **until no updates**, the rules to execute only are the selection if there are updates resulting from the previous step (see Listing 11, line 14 – 17). Otherwise a flag is set to disable the construct (see Listing 11, lines 14, 15). With the keyword **until no change**, the rules to execute are the selection only if the state has changed in the previous step, i.e., there are updates from the previous step that actually change the value of a function in the state (see Listing 11, lines 9 – 12). Otherwise a flag is set to disable the construct (see Listing 11, lines 9, 10).

```
1  rule PrepareExecution =
2     if rulesToExecute = [] then
3        case enabled of
4           ATMOST_N_TIMES:
5              if counter < atmostTimes then
6                 rulesToExecute := ruleSelection
7                 counter := counter + 1
8           UNTIL_NO_CHANGE:
9              if not stateHasChanged then
10                disabled := true
11             else if not disabled then
12                rulesToExecute := ruleSelection
13          UNTIL_NO_UPDATES:
14             if not updatesFromBlock
15                disabled := true
16             else if not disabled then
17                rulesToExecute := ruleSelection
18          ALWAYS:
19             rulesToExecute := ruleSelection
```

```
20        endcase
21     else if semantics = STEPWISE and newMachineStep then
22        rulesToExecute := tail(rulesToExecute)
23     newMachineStep := false
24     mode := EXECUTION
```

Listing 11. PrepareExecution rule

3.5 Execution Phase

The Execution phase is specified by the rule Execution (see Listing 12). With the keyword **if**, the rules to execute are executed if the specified condition evaluates to **true** (see Listing 12, line 4). With the keyword **while**, the rules to execute are executed as long as the specified condition evaluates to **true** (see Listing 12, line 5). With the keyword **iterate**, the rules to execute are executed as long as they produce at least one update (see Listing 12, line 6). At the same time the rules to execute are adjusted according to the specified step semantics. That is, in case of **rulebyrule** only the very first rule of the current rule selection is consumed (see Listing 12, lines 8, 9).

```
1  rule Execution =
2     if rulesToExecute ≠ [] then
3        case conditionKind of
4           IF: if condition then Run
5           WHILE: while condition do Run
6           ITERATE: while updatesFromBlock do Run
7        endcase
8        if semantics = RULEBYRULE then
9           rulesToExecute := tail(rulesToExecute)
10       else if semantics ≠ STEPWISE
11          rulesToExecute := []
12    mode := RESET
13
14 rule Run =
15    case semantics of
16       IN_PARALLEL:
17          forall r ∈ rulesToExecute do
18             r
19       IN_SEQUENCE:
20          foreach r ∈ rulesToExecute do
21             r
22       RULEBYRULE, STEPWISE:
23          if rulesToExecute ≠ [] then
24             let r = head(rulesToExecute) in
25                r
26    endcase
```

Listing 12. Execution rule and Run rule

The rule Run actually executes the rules. With the keyword **in parallel**, the rules are executed in parallel (see Listing 12, lines 17 – 19). With the keyword **in sequence**, the rules are executed in sequence (see Listing 12, lines 21, 22). The **foreach** rule is the sequential counterpart to the **forall** rule, i.e., each iteration of the loop is computed in sequence instead of in parallel. With the keyword **rulebyrule**, or the keyword **stepwise** the rules are executed one by one (see Listing 12, lines 24 – 26), i.e., with each evaluation only exactly one rule is executed.

3.6 Reset Phase

The Reset phase is specified by the rule Reset (see Listing 13). It resets the state of the *Enabled* part if the provided condition evaluates to **true** (see Listing 13, lines 2 – 4). If the keyword **variable** is used the next phase will be the Selection phase (see Listing 13, lines 5, 6). If the keyword **fixed** is used the next phase will be the PrepareExecution phase (see Listing 13, line 8). So in case of a **fixed** selection the selection stays untouched thus will be permanent.

```
1 rule Reset =
2   if resetCondition then
3      disabled := false
4      counter := 0
5   if selection = VARIABLE_SELECTION and rulesToExecute = []
     then
6      mode := SELECT
7   else
8      mode := PREPARE_EXECUTION
```

Listing 13. Reset rule

4 Related Work

In the following, we compare the presented UCC with the control structures found in other formal specification languages.

TurboASM [4] is an extension of basic ASMs that introduces control constructs with sequential step semantics. We compare our construct with this common extension in order to show that the UCC covers the possibilities of Turbo-ASMs completely.

The Vienna Development Method (VDM) [9] is a well established formal specification language which was originally developed by IBM. The specification language VDM-SL hides the theoretical background from less-experienced users. The state is defined by data structures built on abstract data types. VDM does not support the selection of a random subset of rules or based on priorities. The loop constructs do not limit the execution of rules for the complete run, but only for the current construct execution.

Henshin [1] is a formal language based on the graph transformation formalism [13]. Henshin uses graph transformation rules for the specification of state

changes and provides different control constructs to specify which rules to execute and in which order. While Henshin supports all of the control construct features shown in Table 1, graph transformations are a different formalism compared to abstract state machines.

The Very High Speed Integrated Circuit Hardware Description Language (VHDL) [2] is a hardware description language for electronic design automation to describe systems such as integrated circuits. It's a high level specification language that can also be used as a general purpose parallel programming language.

Table 1 shows whether these languages explicitly support seven different control construct features. The first five control construct features cover how rules are executed, e.g., whether it can be specified to execute rules in parallel or in sequence. The last two aspects cover how rules are selected for execution, e.g., whether it is possible to specify priorities to select the rules to execute.

Note that all these specification languages support to manually realise the different control construct features, e.g., one could realise a random selection of rules by manually calling a random method to decide whether the rule should be executed for each rule. Hence, the table shows whether the UCC is *explicitly* realised by a syntax element. For example, Henshin specifically provides a so-called *IndependentUnit* to non-deterministically select rules for execution.

Table 1. Control constructs in formal specification languages

Supported Feature	UCC	TurboASM	VDM	Henshin	VHDL
Parallel execution	yes	yes	yes	yes	yes
Sequential execution	yes	yes	yes	yes	yes
Limit execution count	yes	no	no	yes	no
Sequential Loop (while)	yes	yes	yes	yes	yes
Conditional execution	yes	yes	yes	yes	yes
Random selection of rules	yes	yes	no	yes	no
Priority-based selection of rules	no	no	no	yes	no

The table shows that there is a common subset of features, i. e., parallel execution, sequential execution, conditional execution and sequential loops. But limiting the execution count or selecting a subset of rules to execute or even selecting a random subset of rules is only supported by some languages.

Control constructs can also be found in every imperative programming language. Simple loop constructs like `for` and `while` are covered in UCC by **`while`** and **`at most`** `n` **`times`**. Continuation with the next iteration (`continue`) and early exit of a loop (`break`) are not directly supported in our construct but could be simulated by appropriate guards. The programming

languages Perl[1] and Ruby[2] support additional loop constructs. While `redo` restarts the current iteration, `retry` (in Ruby only) resets the entire loop. Restarting the current iteration is not possible in UCC, but the entire loop can be restarted by **`resetting on`**.

Random selection of rules has already been addressed by Gurevich in the context of bounded-choice nondeterminism [10]. He introduces the construct **`choose among`** to support non-deterministic choice algorithms like non-deterministic Turing machines. Applications of this construct are found in [11] where the actions of a thread are modeled as non-deterministic bounded choice between different rules (cf. Listing 14).

```
1 rule EXECUTE PROGRAM : choose among
2     WM-EE transfer
3     Create var
4     Create thread
```

Listing 14.)]Example usage of **`choose among`** (from [11])

A similar construct is used by Börger in [5, p. 294] to describe computations in Cold [8]. He chooses one or multiple rules from a given set of rules to realize non-deterministic rule execution (cf. Listing 15).

```
1 COLDUSE(Proc) = choose n ∈ ℕ, choose p_1, ..., p_n ∈ Proc
2     p_1 seq ... seq p_n
```

Listing 15.).]Random selection of rules that are executed in sequence (from [5]).

These kinds of non-deterministic selection from a set of rules are provided by our construct, too. In contrast to Gurevich and Brger, we allow a selection of rules that is permanent for the current run of a machine. For example, this extension can be used to specify heuristics that are modeled by different rules as in Listing 7.

In [5, p. 39] a rule CycleThru is introduced, that cycles through a sequence of rules and executes them one by one. The "stepwise" execution of UCC can be simulated with that rule but resetting on a condition is not easily possible. A conditional update of the current position in parallel may result in an inconsistent update because this location is also set (and most likely to a different value) during the execution of CycleThru.

5 Conclusion and Future Work

The goal of our approach is to improve the specification of *Abstract State Machines*. The proposed construct has been validated in several ways. On one hand, by defining transformation rules which transform any expression using the proposed control construct into a semantically equivalent block of standard ASM,

[1] https://www.perl.org.
[2] https://www.ruby-lang.org.

on the other hand, by implementing it as a CoreASM Plugin[3]. Additionally, a test suite has been developed which can be executed using the provided implementation. While this test suite primarily validates the implementation itself, it also demonstrates that the proposed semantics are actually applicable. Furthermore, the implementation allows to run any specification that uses UCC.

Using some examples, we have shown that in several situations UCC helps to simplify definitions. In future it should be determined and explored what other use cases for this control construct can be found and how existing specifications can be simplified by using it.

Furthermore, specifications using the UCC should be presented to non-software engineers to measure the level of understanding. These experimental studies must be conducted in order to obtain reliable results.

Acknowledgements. First ideas to introduce a UCC originate from a CoreASM workshop in Ulm some time ago. We thank Vincenzo Gervasi, Roozbeh Farahbod, and Simone Zenzaro for inspiring discussions and valuable comments on a preliminary version of this paper. We also thank the anonymous reviewers for their extensive and valuable reviews that helped us improve the paper significantly.

References

1. Arendt, T., Biermann, E., Jurack, S., Krause, C., Taentzer, G.: Henshin: advanced concepts and tools for in-place EMF model transformations. In: Petriu, D.C., Rouquette, N., Haugen, Ø. (eds.) MODELS 2010, Part I. LNCS, vol. 6394, pp. 121–135. Springer, Heidelberg (2010)
2. Ashenden, P.J.: The Designer's Guide to VHDL, 3rd edn. Morgan Kaufmann Publishers Inc., San Francisco (2008)
3. Börger, E.: The origins and the development of the ASM method for high level system design and analysis. J. UCS **8**(1), 2–74 (2002)
4. Börger, E., Bolognesi, T.: Remarks on Turbo ASMs for functional equations and recursion schemes. In: Börger, E., Gargantini, A., Riccobene, E. (eds.) ASM 2003. LNCS, vol. 2589, pp. 218–228. Springer, Heidelberg (2003)
5. Börger, E., Stärk, R.F.: Abstract State Machines. A Method for High-Level System Design and Analysis. Springer, Heidelberg (2003)
6. Farahbod, R.: CoreASM: an extensible modeling framework & Tool environment for high-level design and analysis of distributed systems. Ph.D. thesis, Simon Fraser University.(2009)
7. Farahbod, R., Gervasi, V., Glässer, U.: CoreASM: an extensible ASM execution engine. In: Proceedings of the 12th International Workshop on Abstract State Machines, ASM 2005. pp. 153–166 (2005)
8. Feijs, L.M.G., Jonkers, H.B.M.: Formal Specification and Design. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge (1992)
9. Fitzgerald, J.S., Larsen, P.G., Verhoef, M.: Vienna Development Method. Wiley Encyclopedia of Computer Science and Engineering (2008)

[3] http://github.com/coreasm/coreasm.plugins/tree/master/org.coreasm.plugins.universalcontrol.

10. Gurevich, Y.: Sequential abstract-state machines capture sequential algorithms. ACM Trans. Comput. Logic (TOCL) **1**(1), 77–111 (2000)
11. Gurevich, Y., Schulte, W., Wallace, C.: Investigating Java concurrency using abstract state machines. In: Gurevich, Y., Kutter, P.W., Odersky, M., Thiele, L. (eds.) ASM 2000. LNCS, vol. 1912, pp. 151–176. Springer, Heidelberg (2000)
12. Johnson, W.W., Story, W.E.: Notes on the "15" puzzle. Am. J. Math. **2**(4), 397–404 (1879)
13. Rozenberg, G. (ed.): Handbook of Graph Grammars and Computing by Graph Transformation : Foundations, vol. 1. World Scientific Pub Co, Singapore (1997)
14. Russell, S., Norvig, P.: Artificial Intelligence: A Modern Approach, 3rd edn. Prentice Hall Press, Upper Saddle River (2009)
15. Schroeder, A.: Integrated program measurement and documentation tools. In: Proceedings of the 7th International Conference on Software Engineering, ICSE 1984, pp. 304–313. IEEE Press (1984)

http://www.springer.com/978-3-319-33599-5

Abstract State Machines, Alloy, B, TLA, VDM, and Z
5th International Conference, ABZ 2016, Linz, Austria, May
23-27, 2016, Proceedings
Butler, M.; Schewe, K.-D.; Mashkoor, A.; Biro, M. (Eds.)
2016, XXI, 426 p. 143 illus., Softcover
ISBN: 978-3-319-33599-5