AsmetaF: a flattener for the ASMETA framework

Paolo Arcaini*
National Institute of Informatics
Japan
arcaini@nii.ac.jp

Riccardo Melioli
Elvinia Riccobene
Dipartimento di Informatica, Università degli Studi di Milano
Italy
riccardo.melioli@studenti.unimi.it
elvinia.riccobene@unimi.it

Abstract State Machines (ASMs) have shown to be a suitable high-level specification method for complex, even industrial, systems; the ASMETA framework, supporting several validation and verification activities on ASM models, is an example of a formal integrated development environment. Although ASMs allow modeling complex systems in a rather concise way –and this is advantageous for specification purposes–, such concise notation is in general a problem for verification activities as model checking and theorem proving that rely on tools accepting simpler notations.

In this paper, we propose a *flattener* tool integrated in the ASMETA framework that transforms a general ASM model in a *flattened* model constituted only of update, parallel, and conditional rules; such model is easier to map to notations of verification tools. Experiments show the effect of applying the tool to some representative case studies of the ASMETA repository.

1 Introduction

Abstract State Machines [11, 10] (ASMs) is a formal specification method based on model refinement, which has been used in several application domains and case studies [4, 1, 7]. To overcome the lack of tool support and foster the use of ASMs for rigorous software development, in 2008 we started the ASMETA (ASM mETAmodeling) project[1] with the goal of developing a set of tools supporting different activities of the ASM-based system development process [4], and operating in an integrated way to reuse model information. Today, ASMETA [6] exists as a framework for specification, validation (by simulation, scenario construction, model-based testing) and verification (static analysis, model checking, symbolic verification, refinement correctness, runtime verification), as well as automatic code generation. Exploiting the Model-Driven methodology (which is at the base of the whole framework development), some of these tools have been developed from scratch, while many others have been obtained by mapping ASMs (usually by exploiting *model2model* or *model2text* transformations) into the native formalisms of already existing tools (e.g., model checkers and SMT solvers) in order to exploit their functionalities. Our whole project is, indeed, based on the idea that the ASMETA tool-set should be a formal *integrated* development environment for ASMs. However, the integration of tools into the ASMETA framework has caused some difficulties that justify the work we present here and that we motive in the following.

Motivation ASMs use a plain mathematical notation to model a system configuration (i.e., a *state*) in terms of a mathematical algebra, and use a set of powerful rule constructors to specify system behavior (i.e., state *transitions*). ASMs provide, therefore, a powerful language that permits to describe complex systems in a rather concise way. Although it is an advantage when modeling, this notational conciseness can be a problem for tools integration: target languages have their own syntax and semantics, and translating an ASM to a target model by maintaining the intended computational model is not a trivial work; moreover, ASM specifications must often be translated to less expressive languages, and implementing

*The author is supported by ERATO HASUO Metamathematics for Systems Design Project (No. JPMJER1603), JST.

[1] http://asmeta.sourceforge.net/

Submitted to:
F-IDE 2018

these transformation tools is rather complicated, as the semantics of the complex rule constructors of the ASM language must be taken into account and guaranteed.

In the past, different mappings have been developed to model checkers as SPIN [14] and NuSMV [2], to SMT solvers [5, 3], and to C++ code [9]. All these target notations, although can in principle represent the same class of systems as ASMs, have syntaxes that are very different from the ASM notation, with less expressive constructs; therefore, the integration of these tools into ASMETA usually supports only specific classes of ASMs. Some constructs of the ASM formalism are indeed difficult to translate in the target notation, and, although possible, we did not implement such translations because too much complex (e.g., the mappings to model checkers NuSMV and SPIN do not support variable arguments in functions). On the other hand, we observe that tools integrated into ASMETA usually perform similar pre-processing of supported ASM constructs (e.g., translation of unbounded parallelism of the forall rule is usually implemented by an unfolding), and that this pre-processing could be extracted from the integrated tools and made separately available for all the integrations.

This necessity was again confirmed by our recent work on devising a new mapping to the probabilistic model checker PRISM[2] that will be used for ASM-based analysis of cyber-physical systems in the context of the ERATO MMSD project [15]: we realized that instead of trying to directly map any ASM in PRISM (that provides an extremely limited language), it would have been better to go through a *simpler*, but still equivalent, ASM that uses a limited set of ASM constructs. Such simpler ASM would have been as the result of the pre-processing phase of other integrated tools.

Contribution To simplify the porting of ASMs towards other modeling languages, to reuse tools for model validation and verification, and to foster tool integration into the ASMETA framework, we here propose a *flattener* (`AsmetaF`) that, given an ASM model M, produces an equivalent model M_f that only contains update, conditional, and parallel rules; we consider the M_f model to be in a *normal form*. The idea is that translating the normal form to the target languages of verification frameworks (e.g., NuSMV, SMT-LIB) or code is much easier than considering ASMs containing any possible construct. We are currently using `AsmetaF` in the development of the mapping of ASM to PRISM. Moreover, we have integrated the tool with the AsmetaSMV tool; this has allowed us to support a wider set of specifications, namely those having variable function arguments. In the future, we will integrate it in verification tools of the ASMETA framework and in future integrated tools requiring flattening.

Paper structure Sect. 2 presents some background on the ASM method. Sect. 3 introduces the flattener transformations, and describes how we validated the approach. Sect. 4 describes some preliminary experiments, Sect. 5 reviews some related work, and Sect. 6 concludes the paper.

2 Abstract State Machines

Abstract State Machines (ASMs) [11, 10] are transition systems based on the concept of *state* representing the instantaneous system configuration, and *transition rules* describing the change of state.

ASM *states* are multi-sorted first-order structures, i.e., domains of objects with functions and predicates defined on them. An ASM state S is represented by a set of couples (*location*, *value*). ASM *locations*, namely pairs (*function-name*, *list-of-parameter-values*), represent the abstract ASM concept of basic object containers (memory units). Location *updates* represent the basic units of state change and they are given as assignments, each of the form $loc := v$, where loc is a location and v its new value.

ASM *transition rules* express how function interpretations are modified from one state to the next one, and therefore describe the system configuration changes. The basic form of a transition rule is

[2] `http://www.prismmodelchecker.org/`

the `conditional rule`: "**if** *Condition* **then** *Updates*", where *Updates* is a set of function updates (or `update rules`) of the form $f(t_1,\ldots,t_n) := t$ which are simultaneously executed when *Condition* is true; f is an n-ary function and $t_1,\ldots,t_n,t$ are terms. Due to their parallel execution, we require updates to be consistent, i.e., no pair of updates can simultaneously update the same location to different values.

Besides update and conditional, there is a finite set of *rule constructors* to model submachine calls (`macro (call)` rule), simultaneous parallel actions (`par` rule), non-determinism (`choose` rule), unrestricted synchronous parallelism (`forall` rule), abbreviation on terms or rules (`let` rule). There are also derived rule constructors, as the `case` rule that is defined as alternative disjointed guarded rules.[3]

Functions remaining unchanged during the computation are *static*. Those updated by agent actions are *dynamic*, and distinguished in *monitored* (read by the machine and modified by the environment) and *controlled* (read and written by the machine).

A *computation* of an ASM is a finite or infinite sequence $S_0, S_1, \ldots, S_n, \ldots$ of machine states, where S_0 is an initial state and each S_{n+1} is obtained from S_n by simultaneously firing all the transition rules which are enabled in S_n. The (unique) *main rule* is a transition rule and represents the starting point of the computation. An ASM can have more than one *initial state*. It is possible to specify state *invariants*.

A *multi-agent ASM* models concurrent and distributed computations. It is defined as a set of pairs $M = \{(a, ASM(a))\}$ where a is an element of a predefined set *Agent*, and $ASM(a)$ is a machine specifying its behavior. A predefined function *program* on *Agent* associates an agent with its ASM, and a special function *self* : *Agent*, interpreted by each agent a as itself, allows for self-reference in transition rules.

ASMETA [6] is a tool-set for ASMs, which provides basic functionalities for specification and model analysis techniques (validation, verification, testing, model review, requirements analysis, runtime monitoring, etc.). AsmetaL is the textual notation to encode ASM models into ASMETA.

3 Flattener

In order to improve tools integration in ASMETA and to overcome some shortcomings due to the high level and concise mathematical notation of the ASMs w.r.t. less expressive (in terms of conciseness) formalisms of the integrated tools, we developed a *flattener*. Given an ASM M written in *general form*—i.e., containing any kind of rule and any level of nesting—, the flattener produces an ASM M_f in *normal form* (if all the flattener transformation rules are applied). An ASM is in normal form if, in the main rule, it only contains a parallel rule composed of a set of update rules and conditional rules (without else branch); each conditional rule must contain either an update rule or a parallel of update rules.

The flattener applies a series of transformations shown in Table 1 and described in the following.

`MCR`**: Macro Call rule Remover** A *macro rule* is a named rule r with some formal parameters $v_1, \ldots, v_n$, and a rule body R defined in terms of the parameters. A *macro call rule* is an invocation of rule r with actual parameters $t_1, \ldots, t_n$. The flattener transformation `MCR` replaces each occurrence of a call rule r with the macro rule body R; occurrences of formal parameters in the rule are replaced by the actual parameters used in the macro call rule. In multi-agent ASMs, given a specific subset *AgentKind* of *Agent*, a macro rule *rAgentKind* specifies the program of all agents in *AgentKind*, and, by the keyword

[3] All above mentioned rule constructors are characteristics of the so called *basic ASMs*, which dispose of potentially unrestricted non-determinism and parallelism (appearing in the form of the *choose* and *forall* rules defined above) and to distinguish a version with flat specifications from structured versions (by using the *macro call* rule). Besides basic ASMs, there are advanced classes of ASMs having mechanisms for domain extention (`extend` rule), action sequentialization (`seq` rule), and invocation of sub-machines reporting values. In the current work, we do not take in consideration such advanced classes of ASMs that will be addressed as future work (see Sect. 6).

	Original ASM	Flattened ASM
MCR	**rule** r(v_1 **in** D_1, ...,v_n **in** D_n) = R[v_1,...,v_n] ... r[t_1,...,t_n] *//macro call rule*	*//Macro rule r is removed* R[$v_1 \mapsto t_1$, ..., $v_n \mapsto t_n$]
MCR	**rule** rAgentKind = R[self] ... program(a) *//a is an AgentKind agent* ... **agent** AgentKind: rAgentKind[]	*//Macro rule rAgentKind is removed* R[self $\mapsto$ a] *//Program declaration for AgentKind is removed*

FR	**forall** v_1 **in** D_1, ..., v_n **in** D_n
	with guard[v_1, ..., v_n] **do**
	R[v_1, ..., v_n]
		$(d_1^1,\ldots,d_n^1)$, ..., $(d_1^m, \ldots, d_n^m) \in D_1\times\ldots\times D_n$ with $m = \prod_{j=1}^{n} \lvert D_j \rvert$
		par
		if guard[$v_1 \mapsto d_1^1$, ..., $v_n \mapsto d_n^1$] **then**
		R[$v_1 \mapsto d_1^1$, ..., $v_n \mapsto d_n^1$]
		endif
		...
		endpar

ChR	**choose** v **in** D **with** guard[v] **do** R[v] [**ifnone** R_{none}]	**function** f_{choose} = chooseone({v **in** D | guard[v] : v}) **if** isDef(f_{choose}) **then** R[v $\mapsto$ f_{choose}] [**else** R_{none}] **endif**
AR	f(t_1, ..., t_n)	**let** ($v_1 = t_1$, ..., $v_n = t_n$) **in** f(v_1, ..., v_n) **endlet**
LR	**let** (v_1=t_1, ..., v_n=t_n) **in** R[v_1, ..., v_n] **endlet**	D_1, ..., D_n are the domains of $t_1,\ldots,t_n$ and $(d_1^1,\ldots,d_n^1)$, ..., $(d_1^m, \ldots, d_n^m) \in D_1\times\ldots\times D_n$ with $m = \prod_{j=1}^{n} \lvert D_j \rvert$ **par** **if** $t_1 = d_1^1$ and ... and $t_n = d_n^1$ **then** R[$v_1 \mapsto d_1^1$, ..., $v_n \mapsto d_n^1$] **endif** ... **endpar**
CaR	**switch**(t) **case** t_1: R_1 ... **case** t_n: R_n [**otherwise** R_o] **endswitch**	**par** **if** t = t_1 **then** R_1 **endif** ... **if** t = t_n **then** R_n **endif** [**if** t != t_1 and ... and t != t_n **then** R_o **endif**] **endpar**

NR	**if** $guard_1$ **then**
	if $guard_2$ **then** R_t
	else R_e **endif**
	endif
		par
		if $guard_1$ and $guard_2$ **then** R_t **endif**
		if $guard_1$ and not($guard_2$) **then** R_e **endif**
		endpar

Table 1: Flattener transformations

program, it is possible to invoke the program of an agent *a* in *AgentKind*; in rule *rAgentKind*, the keyword *self* is used to reference the current agent executing the rule. MCR flattens also program invocations; an invocation *program*(a) is replaced with the rule *R* (body of the agent rule), where each occurrence of *self*

is replaced with a. At the end, all the macro rules declared in the ASM model are removed.

`FR`**: Forall rule Remover** In a *forall rule*, the rule R is executed in parallel with all the values of variables $v_1, \ldots, v_n$ satisfying the *guard*. The flattener transformation `FR`, for each combination $\overline{d} = (d_1, \ldots, d_n)$ of values of the domains $D_1, \ldots, D_n$, builds a conditional rule (without else branch)[4] whose guard is that of the forall rule, instantiated over values $\overline{d}$ (i.e., variables $v_1, \ldots, v_n$ are replaced by values $d_1, \ldots, d_n$); in a similar way, the rule in the *then* branch is the rule R of the forall body instantiated over $\overline{d}$.

`ChR`**: Choose rule Remover** In a *choose rule*, the rule R is executed with a value of v, nondeterministically chosen, that satisfies *guard*. If such value does not exist, the choose rule does nothing. The flattener transformation `ChR` embeds the non-deterministic choice in a derived function f_{choose} that randomly selects one of the values of the choose domain; the rule is replaced by a conditional rule that checks whether f_{choose} is defined (i.e., it is possible to select a value from the domain) and, if so, calls R instantiated over f_{choose}. In a choose rule, it is also possible to specify a rule R_{none} that must be executed when no choice can be performed; in the flattened version, this rule is reported in the *else* branch.

`AR`**: Arguments Remover** Function locations are identified at runtime by interpreting the terms used as function arguments. Such feature is usually particularly difficult to handle in target notations; NuSMV, for example, allows to specify arrays (that could be used to model functions), but does not allow to dynamically accessing them. The flattener transformation `AR` removes terms used as function arguments and replaces them by suitable let rules (that can then be flattened by the flattener transformation `LR`).

`LR`**: Let rule Remover** A *let rule* associates logical variables $v_1, \ldots, v_n$ to terms $t_1, \ldots, t_n$; the rule body R is defined in terms of the variables. The flattener transformation `LR` removes the rule by considering all the possible values assumed by the terms; for each combination $\overline{d} = (d_1, \ldots, d_n)$ of values of the terms domains, a conditional rule is created: the guard checks whether the terms assume the values in $\overline{d}$, and the *then* rule is the rule body R of the let rule, instantiated over $\overline{d}$.

`CaR`**: Case rule Remover** In a case rule, a term t is compared with terms $t_1, \ldots, t_n$, each one associated with a rule R_i to be executed if t evaluates as t_i. An optional *otherwise* branch can specify a rule R_o to execute when t does not match to any of the t_i. The flattener transformation `CaR` introduces a parallel of conditional rules, each checking whether t is equal to t_i and then executing the corresponding rule R_i in the then branch. An additional conditional rule is added if the *otherwise* branch is present.

`NR`**: Nesting Remover** A nested conditional rule is replaced by parallel conditional rules, by unfolding the rules and aptly combining their guards.

Simplifier Applying the previous flattener transformations could produce some terms only containing constants; such terms can be evaluated statically at parsing time. Therefore, in order to avoid unnecessary rules in the flattened models, after the application of a flattener transformation, we apply two *simplifiers*:

1. `TS` visits all the terms and, if possible, evaluates them or simplifies them; for example, a function term $and(a, true)$ is simplified to a, $3 < 4$ is simplified to *true*, $2 + 1$ is simplified to 3, and so on; `TS` can simplify logical, mathematical, and relational terms;
2. `RS` visits all the rules and, if possible, removes or simplifies them; for example a conditional rule with guard equal to *true* is replaced with its *then* rule.

Application order of flattener transformations All the flattener transformations are applied in the order in which they have been presented. The order guarantees that no construct that should be flattened

[4]Note that in AsmetaL the domains of a forall must be finite, so the number of generated conditional rules will be finite.

is not. Indeed, a transformation could introduce some constructs that are further flattened by another one; namely, LR must be executed after AR because AR introduces let rules that are then flattened by LR; in a similar way, NR must be applied after all the other transformations because it must remove the nesting they introduce. However, the chosen order is not the only possible; indeed, although there are couples of transformations that must be executed in a given order, the order of other couples could be exchanged. We will discuss about the *best* order in the experiments (see **RQ1** in Sect. 4).

Tool implementation The flattener has been implemented in the tool AsmetaF. The tool has been designed in a modularized way such that the user can decide which flattener transformations to apply; in some cases, it may be not necessary to flatten all the ASM constructs, as some of them could be natively supported by the target language. For example, a programming language as C supports nesting, and so it is not necessary to remove it. The tool is meant to be used as pre-processing step of other tools. However, we provide a standalone version for demonstration purposes.[5]

3.1 Validation of the approach

The proposed flattener transformations preserve the ASM semantics; however, it could be that their implementation in AsmetaF is not correct. In order to guarantee the correctness of AsmetaF, we should prove semantic equivalence between original and flattened models, but this is in general difficult to achieve. Therefore, we perform two kinds of validation, *syntactic* and *semantic*.

In *syntactic validation*, we simply check whether the produced flattened ASM is syntactically correct, i.e., it can be parsed correctly by the ASMETA parser.

In the *semantic validation*, by means of model checking and scenario-based validation, we try to check that the semantics of the model is preserved. We use the AsmetaSMV tool to check that the temporal properties specified in the original model are equally evaluated in the flatten model. The tool AsmetaV, instead, allows to write *scenarios* (similar to test cases) that drive the model simulation and check that the ASM state (values of controlled locations) is as expected; we run the scenarios written for the original model also on the flattened model and we expect that it passes them.

In the future, we plan to devise a more systematic way to perform validation. For example, we could automatically produce scenarios achieving rule coverage of the original and target models: the target model should pass scenarios generated for the original one (to check that the flattener preserves the behavior), and the original model should pass scenarios generated for the target one (to check that the flattener does not introduce additional behaviors).

4 Experiments

We applied all the transformations to 13 representative models of the ASMETA repository[6] as a Landing Gear System [4], a hemodialysis device [1], a device for measuring amblyopia, and a termination detection algorithm by Dijkstra (from Dagstuhl Seminar 13372[7]). Note that some of the case study models were obtained by refinement and we took the last refined model. Table 2 reports, for all the models, the number of their rules. The table also reports, for each kind of rule, the average number among the

[5] A jar file of the tool can be downloaded from https://goo.gl/vShfbJ

[6] All the original and the flattened models, together with the scenarios used for validation, are available at http://fmse.di.unimi.it/sw/FIDE2018AsmetaF.zip

[7] https://www.dagstuhl.de/de/programm/kalender/semhp/?semnr=13372

Model	Rule								
	Update	Parallel	Conditional	Forall	Choose	Case	Let	MacroCall	All
CoffeeVendingMachine	2	1	3	0	1	0	0	2	9
DijkstraTermination	9	6	8	1	3	0	0	9	36
ferrymanSimulator_raff1	5	1	3	0	0	0	0	2	11
GameOfLife	2	0	3	1	0	0	0	1	7
GilbreathCardTrick	15	5	7	2	3	1	0	9	42
HemodialysisRef3	146	78	228	1	0	0	0	192	645
LandingGearSystem_3L	38	15	9	0	0	5	0	4	71
OneWayTrafficLight	5	9	8	0	0	0	0	16	38
PetriNet	1	0	0	1	1	0	0	1	4
philosophers1	6	3	4	0	1	0	0	3	17
Roulette	4	2	3	0	1	0	0	4	14
SluiceGateMotorCtl	9	7	8	0	0	0	0	4	28
StereoacuityRaff5	20	6	11	0	0	0	0	15	52
AVG	20.15	10.23	22.69	0.46	0.77	0.46	0	20.15	74.92

Table 2: Benchmarks size

Model	Flattener transformation							Simplifier		
	MCR	FR	ChR	AR	LR	CaR	NR	TS	RS	Time (sec)
CoffeeVendingMachine	2	0	1	1	1	0	2	0	0	0.01
DijkstraTermination	9	1	18	12	42	0	5	0	0	0.11
ferrymanSimulator_raff1	2	0	0	3	9	0	2	1	1	0.08
GameOfLife	1	1	0	0	0	0	2	0	0	0.02
GilbreathCardTrick	9	4	3	50	50	1	4	12	0	0.37
HemodialysisRef3	192	1	0	0	0	0	8	0	0	0.94
LandingGearSystem_3L	4	0	0	0	0	6	4	0	0	0.03
OneWayTrafficLight	16	0	0	12	20	0	1	96	16	0.04
PetriNet	1	1	1	4	4	0	1	0	0	0.01
philosophers1	3	0	1	10	5414	0	2	20102	1802	59.93
Roulette	4	0	1	1	1	0	3	37	37	1.36
SluiceGateMotorCtl	4	0	0	4	4	0	1	2	2	0.01
StereoacuityRaff5	15	0	0	0	0	0	6	0	1	0.06
AVG	20.15	0.62	1.92	7.46	426.54	0.54	3.15	2892.86	265.57	4.84

Table 3: Applied flattener transformations and execution time

models. We observe that the update, the conditional, and the macro call rules are the most used ones.

RQ1: *Which are the most applied flattener transformations?*
We are here interested in finding which are the transformations that are applied more often. Table 3 reports how many times each transformation is applied to each model. Since MCR is used at the beginning, it is applied exactly the same number of times as the number of macro call rules (see Table 2); note that, although MCR could be applied at any stage during the flattening process, it makes sense to use it at the beginning since it is applied so many times (it is the second most used transformation). Applying it after some other transformations (e.g., FR) would probably increase even more the number of times it is used.

The most used transformation is LR; although the original models do not contain any let rule, these are introduced by AR. Note that in some models (e.g., philosophers1) the number of applications of LR is

Model	Rule				
	Update	Parallel	Conditional	All	Δ %
CoffeeVendingMachine	6	2	5	13	44%
DijkstraTermination	276	13	252	541	1403%
ferrymanSimulator_raff1	145	1	145	291	2545%
GameOfLife	32	1	32	65	829%
GilbreathCardTrick	900	2	898	1800	4186%
HemodialysisRef3	214	54	154	422	-35%
LandingGearSystem_3L	46	19	18	83	17%
OneWayTrafficLight	72	9	56	137	261%
PetriNet	8	1	8	17	325%
philosophers1	118800	1	118800	237601	1397553%
Roulette	5404	2703	2702	10809	77107%
SluiceGateMotorCtl	20	9	8	37	32%
StereoacuityRaff5	30	7	16	53	2%
AVG	9688.69	217.08	9468.77	19374.54	25759%

Table 4: Size of the flattened models

much higher than that of AR, because the let rules are nested: during the flattening, the inner let rule is visited as many times as the number of conditional rules created by outer let rule.

The value reported for NR is the difference between the maximum nestings of the starting model and of the flattened one (i.e., how many nesting levels have been removed). We observe that, on average, 3.15 levels of nesting are removed, meaning that the developers tend to write quite nested models.

RQ2: *Which is the effect of flattening?*
We are now interested in evaluating the effect of applying the flattener to the models. Table 4 reports the size of the flattened models in terms of number of update, parallel, and conditional rules. The table also reports the total number of rules and their percentage change w.r.t. the original model (Δ). We can observe that usually the flattened model has many more rules. The model that has the greatest increment in the number of rules is philosophers1; indeed, the model has several dynamic function arguments that, when flattened by AR and LR, produce several rules (see Table 3).

However, there are also some models for which the number of rules is similar or also decreases; these models are already quite *flatten*: for example, the original model of HemodyalisisRef3 already contains almost only update, parallel, and conditional rules (see Table 2), and the application of the flattener has the effect of reducing the conditional rules by merging some of them through NR (see Table 3).

We can interpret Δ as an index of the *conciseness* of the model: the higher Δ is, the more concise the original model is. Indeed, a very concise model (as philosophers1) specifies, by using few powerful rules, a complex behavior; when flattened, this results in a big number of rules.

RQ3: *Does the simplification have any effect?*
We here check whether the simplification of terms and rules (embedded in all the flattener transformations) has some effect. Table 3 reports, for each model, the number of simplifications performed by the two simplifiers. We observe that, for more than half of the models, the simplifications are actually applied. For example, in the flattening of OneWayTrafficLight and philosophers1, several terms are simplified; this is due to the fact that both models contain several guards of conditional rules that depend

on functions with dynamic arguments. When these arguments are flattened by AR and LR, some resulting guards can be simplified by TS either to *true* or to *false*; as a consequence of the simplifications of guards, some conditional rules can be simplified by RS, either by removing them (if the guard is *false*) or by replacing them with the *then* branch (if the guard is *true*).

RQ4: *Which is the computational effort required by the flattener?*

To answer this question, we performed 100 executions of the flattener over all the models on a macOS machine, 3.1 GHz Intel Core i5, and 16GB. Table 3 reports, for each model, the average time (in seconds) taken by the flattener, and the average time among models. We observe that for almost all models the execution time is less than 1.5 secs. We can notice that the model for which it takes longer (59.93 secs for philosophers1) is a very concise model that has been flattened a lot (see Tables 3 and 4).

5 Related work

Flattening a model in order to simplify it is a rather common activity. The authors in [13] present flattening transformations for state machines equipped with hierarchy and parallelism, in order to transform models into executable code or inputs for model-based testing and verification techniques.

Model checkers use flatteners to simplify the notation of the models, as for example the NuSMV flattener [12], that produces a synchronous flat model from a modular description of a model.

Model flattening has been exploited also in [8] to reduce model to code, in particular to generate efficient C code from B formal models in the domain of smart card applications.

In the context of the ASMs, Winter [16] proposed some ad-hoc transformations for mapping ASMs to SMV models; differently, our flattener aims at producing a normal form of the model more widely usable for transformation to several other tools. A different ASM model refactoring approach appears in [17] where a number of refactoring patterns are presented to restructure ASM models with the goal of improving their intelligibility, maintainability, abstraction, and conciseness. In a way, applying such patterns produces an effect which is opposite w.r.t. our flattener transformations: the latter ones may increase the model size, and may compromise model intelligibility; on the other side, they provide a normal form of the model in term of a very limited number of rule constructors, and the flattened model is not to be intended for readability and comprehension, but to facilitate tools integration.

6 Conclusions

In this paper, we propose a *flattener* tool, AsmetaF, integrated in the ASMETA framework that transforms an ASM in a *flattened* model constituted only of update, parallel and conditional rules. The goal of the flattener is to support a pre-processing of ASM transformations towards tools having less expressive notational constructs, as, for example, those of verification tools. We claim that ASM flattening can improve the strengths of ASMETA as formal integrated development environment for ASMs.

In ASMs, functions are total by assigning the undef value to undefined locations. In the translators developed in the past, handling the undef has been challenging, and only some partial solutions (i.e., only for some domains) have been proposed. As future work, we plan to devise a special transformation able to produce an ASM in which the undef can not occur and it is handled explicitly in the model.

As future step, we also plan to include in our flattener suitable transformations to handle extend and seq rules. This would allow reducing in normal form also advanced ASM classes that would become accessible to verification tools not yet able to support some classes of models.

References

[1] Paolo Arcaini, Silvia Bonfanti, Angelo Gargantini, Atif Mashkoor & Elvinia Riccobene (2018): *Integrating formal methods into medical software development: The ASM approach*. Science of Computer Programming 158, pp. 148 – 167.

[2] Paolo Arcaini, Angelo Gargantini & Elvinia Riccobene (2010): *AsmetaSMV: a way to link high-level ASM models to low-level NuSMV specifications*. In: Proceedings of the 2nd International Conference on Abstract State Machines, Alloy, B and Z (ABZ 2010), LNCS 5977, Springer, pp. 61–74.

[3] Paolo Arcaini, Angelo Gargantini & Elvinia Riccobene (2016): *SMT-based automatic proof of ASM model refinement*. In: Software Engineering and Formal Methods: 14th International Conference, SEFM 2016, Proceedings, Lecture Notes in Computer Science, Springer International Publishing, Cham, pp. 253–269.

[4] Paolo Arcaini, Angelo Gargantini & Elvinia Riccobene (2017): *Rigorous development process of a safety-critical system: from ASM models to Java code*. International Journal on Software Tools for Technology Transfer 19(2), pp. 247–269.

[5] Paolo Arcaini, Angelo Gargantini & Elvinia Riccobene (2018): *SMT for state-based formal methods: the ASM case study*. In: Automated Formal Methods, Kalpa Publications in Computing 5, EasyChair, pp. 1–18.

[6] Paolo Arcaini, Angelo Gargantini, Elvinia Riccobene & Patrizia Scandurra (2011): *A model-driven process for engineering a toolset for a formal method*. Software: Practice and Experience 41(2), pp. 155–166.

[7] Paolo Arcaini, Roxana-Maria Holom & Elvinia Riccobene (2016): *ASM-based formal design of an adaptivity component for a Cloud system*. Formal Aspects of Computing 28(4), pp. 567–595.

[8] Didier Bert, Sylvain Boulmé, Marie-Laure Potet, Antoine Requet & Laurent Voisin (2003): *Adaptable Translator of B Specifications to Embedded C Programs*. In: FME 2003: Formal Methods, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 94–113.

[9] Silvia Bonfanti, Marco Carissoni, Angelo Gargantini & Atif Mashkoor (2017): *Asm2C++: A Tool for Code Generation from Abstract State Machines to Arduino*. In: NASA Formal Methods, Springer International Publishing, Cham, pp. 295–301.

[10] Egon Börger & Alexander Raschke (2018): *Modeling Companion for Software Practitioners*. Springer, Berlin, Heidelberg.

[11] Egon Börger & Robert Stärk (2003): *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer Verlag.

[12] Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani & Armando Tacchella (2002): *NuSMV 2: An OpenSource Tool for Symbolic Model Checking*. In: Computer Aided Verification, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 359–364.

[13] X. Devroey, M. Cordy, P. Y. Schobbens, A. Legay & P. Heymans (2015): *State machine flattening, a mapping study and tools assessment*. In: 2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW), pp. 1–8.

[14] Angelo Gargantini, Elvinia Riccobene & Salvatore Rinzivillo (2003): *Using Spin to Generate Tests from ASM Specifications*. In: Abstract State Machines 2003, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 263–277.

[15] Ichiro Hasuo (2017): *Metamathematics for Systems Design*. New Generation Computing 35(3), pp. 271–305.

[16] Kirsten Winter (1997): *Model Checking for Abstract State Machines*. Journal of Universal Computer Science (J.UCS) 3(5), pp. 689–701.

[17] Hamed Yaghoubi Shahir, Roozbeh Farahbod & Uwe Glässer (2012): *Refactoring Abstract State Machine Models*. In: Abstract State Machines, Alloy, B, VDM, and Z: Third International Conference, ABZ 2012, Pisa, Italy, June 18-21, 2012. Proceedings, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 345–348.