# Evaluating the Practical Impact of Parallelism in `Asmeta`

Andrea Bombarda[1 [0000−0003−4244−9319]], Silvia Bonfanti[1 [0000−0001−9679−4551]], Cesar Cornejo[1 [0000−0003−3716−3607]], Angelo Gargantini[1 [0000−0002−4035−0131]], and Nico Pellegrinelli[1 [0009−0000−4944−6845]]

University of Bergamo, Bergamo, Italy {andrea.bombarda, silvia.bonfanti, cesar.cornejo, angelo.gargantini, nico.pellegrinelli}@unibg.it

**Abstract.** Parallelism is a key semantic feature of Abstract State Machines (ASMs), represented in the `Asmeta` tool set by the `par` construct, which enables synchronous execution of multiple rules. While its theoretical importance is firmly grounded in the ASM formalism, which models synchronous updates through parallel rule execution, an evaluation of its real usefulness in real-world models has received limited attention. This paper presents an experimental study evaluating how and when parallelism is effectively used in `Asmeta`. We analyzed existing specifications to measure the adoption of the `par` construct, and generated sequential variants replacing `par` with `seq` to compare behavior through randomized test suites. Our findings show that `par` is widely adopted and useful: it rather frequently produces results different from those obtained by sequential execution.

**Keywords:** Abstract State Machines · `Asmeta`· Parallelism · Sequential.

## 1 Introduction

Each notation introduces a set of concepts designed to serve as core features for practitioners who adopt it. These concepts are translated in *constructs* that are used when writing artifacts. For instance, object-oriented programming promotes encapsulation and modularity through the constructs of classes and objects. For *Abstract State Machines* (ASMs) [9], a core feature is the synchronous *parallelism*: within a single computation step, all enabled rules are evaluated concurrently over the current state. Their updates are collected into a consistency-checked update set and applied atomically, yielding the next state of the machine [4]. This concept is represented by the `par` construct within the `AsmetaL` notation[1] [1,8]. It allows executing multiple rules in parallel by computing their updates based on the same state, and applying them together.

In practice, with the `par` statement, modelers can express concurrent reactions of independent components and multi-location updates that would otherwise be more complex to be decomposed into sequential steps. While parallelism

[1] The language used in the `Asmeta` tool set.

is a semantic feature of ASMs, `Asmeta` allows users to also model sequential control flow through the `seq` construct: each rule is executed one after the other, so later rules are evaluated over the updates produced by earlier ones. In certain cases, whether the statements are executed sequentially or in parallel has no effect on the update set obtained after all rules have been fired. Thus, one may argue on the real usefulness of the `par` statement in `Asmeta`. More specifically, while one can provide theoretical evidence that the `par` construct is useful, some doubts may remain at a practical level.

A large body of work has established parallelism as a foundational semantic feature of ASMs, justifying compact synchronous updates and parallel rule composition that `Asmeta` implements via the `par` construct [4, 9, 15]. Although intuitively, ASM parallelism aims to capture the essence of synchronous parallel algorithms, [13] claimed that the original theory proposed in [4] was not convincing and proposed a new and simpler one.

In this work, we conduct an empirical study to assess the use of parallelism in `Asmeta` and to analyze how system behavior is affected when sequential execution replaces parallel constructs. A classical approach when it comes to investigating the usefulness of notations in software engineering is to provide users with questionnaires and ask their opinion [3, 14]. However, questionnaires risk to be not objective, especially if the user base is limited and/or respondents contributed to the development of the notation. This is the reason behind our decision to evaluate the usefulness of `par` empirically and objectively.

We evaluated the usefulness of parallelism in `Asmeta` by analyzing and experimenting on 299 specifications. Our results confirm that `par` is used in most of the considered specifications (246 out of 299), and for the specifications we are able to generate the tests for (131 out of 246), in almost half of specifications (63 out of 131) parallelism was useful to express the desired behavior.

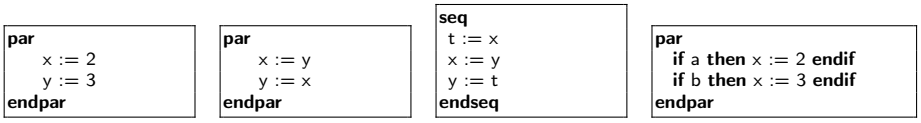
The remainder of the paper is structured as follows. Section 2 introduces ASMs and their parallelism, with the `Asmeta` notation. Section 3 details the empirical experimental methodology we adopted in our evaluation and presents the obtained results. In Section 4, we discuss potential threats to the validity of our findings and experiments. Finally, Section 5 concludes the paper.

## 2 `Asmeta` and parallelism

This section examines the semantics and practical relevance of parallel execution in system modeling based on the ASM formalism[2].

The `par` statement is used in `Asmeta` to express parallel behavior by executing rules in parallel. Semantically, all rules are evaluated simultaneously over the current state, and all location updates are collected into a single *update set* that is applied atomically to produce the next state. As a consequence, the `par` construct can be interpreted as indicating that the contained statements can be executed in any order. Some examples illustrating the use of `par` in `Asmeta` are shown in Fig. 1. In Fig. 1a, two functions $x$ and $y$ are simultaneously updated, and the resulting update set is $\{x = 2, y = 3\}$. Since all terms are

[2] For theoretical foundations of ASMs, see [9]; for further details on `Asmeta`, see [1, 8].




(a) Simple par (b) Swap (c) Swap by seq (d) Inconsistent updates

Fig. 1: Examples of parallel and sequential blocks

evaluated with respect to the current state, the `par` construct can easily express, for instance, operations such as value swapping without auxiliary variables. The code in Fig. 1b correctly swaps $x$ and $y$ in a single parallel step. Achieving the same effect with a sequential construct (`seq`) would require the more classical implementation shown in Fig. 1c.

As illustrated by the swap example, the use of `par` enables a more compact and declarative specification, eliminating the need for temporary variables. More generally, it facilitates the transition from one state to the next in spirit closer to the stepwise semantics of finite state machines rather than the imperative paradigm of general-purpose programming languages.

However, parallel updates may give rise to *inconsistent update* situations. As shown in Fig. 1d, when both conditions `a` and `b` are satisfied, the model attempts to assign two different values to `x` within the same step. Such inconsistencies can occur only under `par`, and their detection is beneficial, as it alerts the modeler to unintended interactions among rules. In this sense, `par` serves as a safeguard that exposes conflicting updates, thereby contributing to the correctness and clarity of the specified behavior.

## 3 Evaluation of the practical impact

In this section, we describe the empirical evaluation methodology we have adopted to assess the usefulness of parallelism and `par` construct in `Asmeta`. In particular, we defined two research questions (RQs) to guide our evaluation:

**RQ1** Is parallelism in `Asmeta` used?
**RQ2** Is parallelism in `Asmeta` useful?

**RQ1: Is parallelism in `Asmeta` used?** To evaluate if the `par` construct is used in `Asmeta`, we select all specifications from the original `Asmeta` GitHub repository at `https://github.com/asmeta/asmeta/tree/master/asm_examples`, excluding duplicates, those marked as "old," or written solely for testing purposes. Then, we count how many of the selected `Asmeta` specifications contain `par` in the main module. Specifications, scripts, and results are available online [6].

*Findings:* We found that `par` is used in 246 specifications over 299. It is therefore apparent that parallelism is a construct frequently used by modelers. However, there are cases in which parallelism is used even though it may not be strictly required. For instance, in Fig. 1a, the same behavior could have been achieved executing rules sequentially. We investigate this aspect in the following RQ.

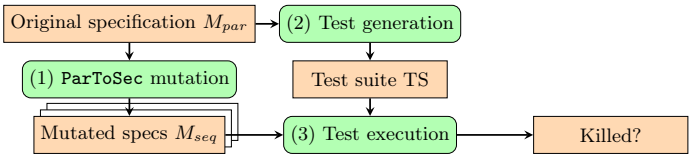



Fig. 2: Experimental process

**RQ2: Is parallelism in `Asmeta` useful?** To objectively evaluate the usefulness of `par`, we have devised the methodology depicted in Fig. 2. Given an `Asmeta` specification, $M_{par}$, containing some `par` blocks, we automatically generate a set of `Asmeta` specifications $M_{seq}$ by replacing each `par` individually, one after the other[3], with a `seq` block (Step 1 in Fig. 2). For this, we use the *Parallel to Sequential* mutation operator defined in [5]. In this way, $M_{par}$ and each $M_{seq}$ differ only in the parallelism among rules, which is substituted by a more classical sequential activation. This mutation completely changes the semantics of the `Asmeta` specification, but it may or may not change its behavior. More precisely, if `par` plays a meaningful role in $M_{par}$, then $M_{seq}$ exhibits different behavior. Conversely, if $M_{seq}$ behaves identically to $M_{par}$, `par` is not useful in $M_{par}$.

> In a specification $S$, we say that `par` is *useful* iff there is at least one `par` block in $S$ that cannot be replaced by a `seq` block, with identical body, without changing the behavior of $S$.

Now, the problem of evaluating the usefulness of `par` is reduced to checking the behavioral equivalence between a specification ($M_{par}$) and a set of specifications ($M_{seq}$), and this can be proved or disproved in several ways. In this paper, we propose using *random test generation* and *mutation* analysis, which are effective for disproving, but not proving, behavioral equivalence.

First, a test suite *TS* is generated from $M_{par}$ by using a random test generator [1] (Step 2 in Fig. 2). *TS* contains a set of tests in the `Avalla` language, which are composed by multiple `step`, `set` and `check` commands [11]. Being generated from $M_{par}$, *TS* will consist only of test cases that successfully execute that specification. Finally, *TS* is executed over all $M_{seq}$ specifications (Step 3 in Fig. 2). If all the tests in *TS* pass, then we can say that the mutated specification shows an identical behavior w.r.t. the original one, and changing from `par` to `seq` did not cause any behavioral modification - at least for the behavior the tests were able to cover. Instead, if any test in *TS* fails, we say that the mutation is killed and we were able to find a case in which changing from `par` to `seq` would alter the behavior of the specification.

*Findings:* Using the experimental methodology outlined above and considering the same set of specifications as in RQ1, we obtained the following results. Out of the 246 specifications, the full process completed successfully for 131. Mutants could be generated for all specifications, but the process could not be completed

[3] Alternatively, we could have replaced all the `par`s in one shot, but this would have made impossible to assess the individual contribution of each `par`.

```
par
  if state = ON then state := OFF endif
  if state = OFF then state := ON endif
endpar
```

Fig. 3: Control State ASM

```
par
  x := y
  y := 6
endpar
```

Fig. 4: `seq` order matters

for some due to constructs not supported by either the validator or our test generator. Among the 131 successfully processed specifications, at least one mutant was killed for 63 of them (48.1%), indicating that parallelism (i.e., the use of `par`) matters in nearly half of the cases according to our definition of usefulness.

Upon analyzing the results, we were surprised by the large number of specifications (68) for which no mutant was killed. Although our findings come with certain limitations, which we discuss in the next section, we identified two peculiar scenarios in which `par` can (or cannot) be replaced by `seq`.

*Control State ASMs* are ASMs in which one of the rules has a series of conditional rules checking the current state and setting a new state in case of meaningful events [10]. For these machines, the `par` is useful because omitting it would cause the ASM to skip some states during execution. For instance, in the ASM shown in Fig. 3, a sequential execution would result in the specification remaining permanently in the `ON` state. Overall, we observed that `par` is useful when it contains rules updating locations that are subsequently used by other rules within the same execution step.

*Order matters.* In some cases, the inability (or ability) to kill mutants is instead caused by the specific ordering of rules within the `par`. For example, in the specification shown in Fig. 4, the `par` construct appears not useful solely because of the order in which the two assignments were written. If one had written the two updates in the opposite way, the `par` would have been useful.

## 4 Threats to validity

In this section, we describe the main threats to validity according to the scheme proposed in [19] and elaborate on mitigation strategies.

*Internal validity* is a concern arising when the design of a study may compromise the accuracy of the results. To mitigate this risk, we have carefully checked the code executed in our experiments to see if other factors could have caused the outcome, such as errors in the tools or in the experimental code we wrote. However, potential threats to the internal validity still exist. First, to perform mutation analysis in RQ2, we have forced the behavior of the analyzed specifications to be deterministic even when specifications perform non-deterministic choices, to avoid flaky tests [7]. Second, in case of *inconsistent updates*, random test generation raises an exception, and it does not produce any scenario; therefore, the evaluation does not take into account the capability of the `par` construct to detect *inconsistent updates*. Initially, in the evaluation process, we had to set the number of tests and steps in each test to generate test cases. To avoid explicitly specifying the number of tests and steps, we introduced a 10

minutes timeout and iteratively increased the number of test cases with more steps, up to 20 iterations or until the first 5 iterations failed to generate any valid scenario. While this approach reduces the risk of selecting incorrect configuration parameters, it is important to acknowledge that using more tests (or steps) than those we encountered during our experiments within the timeout period could have yielded different results. By increasing timeout and max number of iterations, we may kill more mutants and increase the number of specifications for which `par` is actually useful. Likewise, when using random test cases, we cannot guarantee full coverage of the `Asmeta` specification [5]. As a result, some mutants may survive not because they behave identically to the original specification, but because the generated tests never exercise the mutated part.

*Construct validity* concerns arise when the link between theory and observation is weakened. A potential threat in our study lies in assuming that our chosen measures appropriately capture the usefulness of parallelism in `Asmeta`. We selected them based on established practice: usage frequency is widely used as a proxy for practical relevance [12,18], and mutation has been used in several contexts, such as assessing test suite effectiveness [16], source code optimization [17], and anomaly detection [2]. We acknowledge that the results for RQ2 are highly dependent on the definition of *usefulness* adopted in this study.

*External validity* is concerned with whether we can generalize the results outside the scope of the presented study. Killing mutants is a sufficient condition for proving usefulness of `par` but it is not a necessary one: it is not suitable for proving that `par` is not useful. We may explore formal proofs for that in future works, for instance using a model checker or an SMT solver. In our analysis, we considered all specifications available in the `Asmeta` GitHub repository. Although this constitutes an extensive dataset, most specifications were developed for research purposes. Nevertheless, the considered set also includes some real-world case studies, partially mitigating the external validity threat. In this paper, we applied our empirical evaluation to parallelism in `Asmeta`. However, we believe that the same methodology can be applied to other constructs or formal notations where a mutation excluding the analyzed construct exists.

## 5 Conclusion

In this paper, we empirically assessed the usefulness of parallelism and of the `par` construct in `Asmeta`. We found that most `Asmeta` specifications (82%) employ `par` (RQ1) and for many of them (52%) it is useful (RQ2). Thus, we can consider the `par` advantageous, at least in the presence of certain modeling styles that represent common pattern in ASMs. However, we discovered that in many other cases it could be replaced by `seq` without changing the behavior. Although parallelism is a fundamental feature of ASMs, to the best of our knowledge, this is the first work to empirically investigate whether `Asmeta` users employ `par` out of an actual need for parallelism or simply because it is the most commonly suggested composition strategy for ASMs. Also, some users may be more familiar with `seq`, while others may benefit from `par`. Further investigation of the pros and cons of parallel constructs may be needed, including required effort and ease of expressing complex behavior, e.g., via user questionnaires.

**Acknowledgments.** The work of Andrea Bombarda is supported by the project ANTHEM (AdvaNced Technologies for Human-centrEd Medicine) - PNC0000003 – CUP: B53C22006700001.

## References

1. Arcaini, P., Bombarda, A., Bonfanti, S., Gargantini, A., Riccobene, E., Scandurra, P.: The ASMETA Approach to Safety Assurance of Software Systems, pp. 215–238. Springer International Publishing, Cham (2021). https://doi.org/10.1007/978-3-030-76020-5_13
2. Arcaini, P., Gargantini, A., Riccobene, E., Vavassori, P.: A novel use of equivalent mutants for static anomaly detection in software artifacts. Information and Software Technology **81**, 52–64 (2017). https://doi.org/10.1016/j.infsof.2016.01.019
3. ter Beek, M.H., Ferrari, A.: Empirical Formal Methods: Guidelines for Performing Empirical Studies on Formal Methods. Software **1**(4), 381–416 (2022). https://doi.org/10.3390/software1040017
4. Blass, A., Gurevich, Y.: Abstract state machines capture parallel algorithms: Correction and extension. ACM Transactions on Computational Logic **9**(3) (2008). https://doi.org/10.1145/1352582.1352587
5. Bombarda, A., Bonfanti, S., Cornejo, C., Gargantini, A., Pellegrinelli, N.: Evaluating coverage and fault detection capability of scenarios for the validation of asmeta specifications. In: Deshmukh, J., Havelund, K., Pinto, A. (eds.) NASA Formal Methods. Springer Nature Switzerland, Cham (2026)
6. Bombarda, A., Bonfanti, S., Cornejo, C., Gargantini, A., Pellegrinelli, N.: Replication Package for "Evaluating the Practical Impact of Parallelism in Asmeta" (2026). https://doi.org/10.5281/zenodo.18550923
7. Bombarda, A., Bonfanti, S., Gargantini, A., Pellegrinelli, N.: Eliminating Flakiness: Deterministic Control for Validating Nondeterministic Asmeta Specifications. In: Dutle, A., Humphrey, L., Titolo, L. (eds.) NASA Formal Methods. pp. 100–115. Springer Nature Switzerland, Cham (2025). https://doi.org/10.1007/978-3-031-93706-4_7
8. Bombarda, A., Bonfanti, S., Gargantini, A., Riccobene, E., Scandurra, P.: ASMETA Tool Set for Rigorous System Design. In: Formal Methods. pp. 492—-517. Springer Nature Switzerland, Cham (2024). https://doi.org/10.1007/978-3-031-71177-0_28
9. Börger, E., Stärk, R.: Abstract State Machines: A Method for High-Level System Design and Analysis. Springer Verlag (2003). https://doi.org/10.1007/978-3-642-18216-7
10. Borger, E., Stark, R.F.: Abstract State Machines. Springer, Berlin, Germany (Apr 2013)
11. Carioni, A., Gargantini, A., Riccobene, E., Scandurra, P.: A Scenario-Based Validation Language for ASMs. In: Börger, E., Butler, M., Bowen, J.P., Boca, P. (eds.) Abstract State Machines, B and Z. pp. 71–84. Springer, Berlin, Heidelberg (2008). https://doi.org/10.1007/978-3-540-87603-8_7
12. Costa, D., Andrzejak, A., Seboek, J., Lo, D.: Empirical Study of Usage and Performance of Java Collections. In: Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering. p. 389–400. Association for Computing Machinery, New York, NY, USA (2017). https://doi.org/10.1145/3030207.3030221

13. Ferrarotti, F., Schewe, K.D., Tec, L., Wang, Q.: A new thesis concerning synchronised parallel computing – simplified parallel ASM thesis. Theoretical Computer Science **649**, 25–53 (2016). https://doi.org/10.1016/j.tcs.2016.08.013
14. Gross, A., Jurkiewicz, J., Doerr, J., Nawrocki, J.: Investigating the usefulness of notations in the context of requirements engineering. In: 2012 Second IEEE International Workshop on Empirical Requirements Engineering (EmpiRE). pp. 9–16 (2012). https://doi.org/10.1109/EmpiRE.2012.6347684
15. Gurevich, Y.: Abstract State Machines: An Overview of the Project. In: Seipel, D., Turull-Torres, J.M. (eds.) Foundations of Information and Knowledge Systems. pp. 6–13. Springer, Berlin, Heidelberg (2004)
16. Jia, Y., Harman, M.: An Analysis and Survey of the Development of Mutation Testing. IEEE Transactions on Software Engineering **37**(5), 649–678 (2011). https://doi.org/10.1109/TSE.2010.62
17. López, J., Kushik, N., Yevtushenko, N.: Source code optimization using equivalent mutants. Information and Software Technology **103**, 138–141 (2018). https://doi.org/10.1016/j.infsof.2018.06.013
18. Salmani Nodoushan, M.A.: Measurement theory in language testing: Past traditions and current trends. i-manager's Journal on Educational Psychology **3**(2), 1–12 (2009). https://doi.org/10.26634/jpsy.3.2.1023
19. Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B., Wesslén, A.: Experimentation in Software Engineering. Springer Berlin Heidelberg (2024). https://doi.org/10.1007/978-3-662-69306-3