Boosting LLM-Based Software Generation by Aligning Code with Requirements

Tom Yaacov
Ben-Gurion University of the Negev
tomya@post.bgu.ac.il

Achiya Elyasaf
Ben-Gurion University of the Negev
achiya@bgu.ac.il

Gera Weiss
Ben-Gurion University of the Negev
geraw@cs.bgu.ac.il

***Abstract*—Emerging LLM-based code generation tools enable programmers to specify desired functionality and automatically generate code. However, these tools fall short in comparison to human ability when it comes to creating complete system models from requirements. This is because humans typically formulate a software design before implementing a system. In this paper, we propose to use the behavioral programming (BP) model-based paradigm as a general design approach that allows for the direct translation of requirements of any reactive systems into code. We demonstrate that each requirement can be automatically transformed into a dedicated code module without the need for a global view of the system. The key lies in BP's capability to enable modules to implement both scenarios and anti-scenarios separately. This means that each module can independently define behaviors that may happen, must happen, and must not happen. Subsequently, an application-agnostic execution engine interprets and interweaves these modules at runtime to generate cohesive system behavior consistent with system requirements. The fact that each requirement is translated into a small module also facilitates the verification of its implementation, thereby helping to reduce errors in LLM code generation. We present an initial evaluation of our approach and demonstrate how the characteristics of BP aid in generating aligned and correct implementations.**

***Index Terms*—behavioral programming, large language models, requirement engineering**

I. Introduction

Large Language Models (LLMs), such as OpenAI's GPT [1] or Google's BERT [2], have sparked a transformative revolution in automatic text generation, and have demonstrated remarkable capabilities in understanding and producing human-like text across various domains, including natural language processing [3], content generation [4], and software engineering [5]. This breakthrough technology has enabled the automation of routine programming tasks and introduced new possibilities for enhancing developer productivity and software quality.

While LLMs are proving useful in bridging requirements and their implementation, there is still a gap between the current state-of-the-art and the vision of code-less programming [6]. In this paper, we identify two main factors that can be improved with a new approach that we propose:

1) Current practices typically involve programmers creating software designs and then tasking LLMs with implementing parts of the design. We argue that it may be feasible to utilize a unified design approach so that LLMs can directly handle multifaceted requirements.
2) LLMs often introduce errors that are challenging for programmers and other stakeholders to identify. We state that the implementation can be formally verified using techniques such as model-checking by isolating and keeping the code that implements each requirement separate and concise.

To illustrate the challenge of using LLMs for architecting systems directly from requirements, we turn to a well-known use case: an event-driven level-crossing system [7]–[9]. This example describes a system functioning as a controller for a gate at a railway crossing—an intersection between a railway line and a road at ground level. The railway line has a sensor that signals the controller whenever the train approaches, enters, and leaves the crossing zone. Based on these signals, the system manages the raising and lowering of barriers to ensure the safety of passing trains.

The following requirements are taken from Elyasaf et al. [9]:

1) When a train passes, the sensor system activates the exact event order: approaching, entering, and leaving.
2) The barriers are lowered when a train approaches and then raised.
3) A train may not enter while barriers are raised.
4) The barriers may not be raised while a train passes, i.e., it approached but did not leave.

To emphasize how LLM implements a system out of its requirements, we prompted OpenAI's latest LLM model (gpt-4-turbo-preview) [10] model and asked it to implement the level-crossing system that adheres to the requirements above. The task involved requesting the model to produce a Python function to simulate the specified system. The function was expected to generate a potentially varying trace of system events among all feasible traces. Apart from these instructions, the model could implement the system as it saw fit, i.e., add classes and methods if needed. The entire prompt we used is provided in the appendix [11], and the code used to run it is available at https://github.com/bThink-BGU/Papers-2024-MoDRE-BP-LLM. The generated implementation is given in Listing 1.

We found that the model's implementation violates all specified requirements. Further, a review of the code reveals that it constructs a sequence of events that is composed of three parts: a random sequence (`pre_evts`), a fixed sequence (`mandatory_sequence`), and another random sequence

```python
def railway_crossing_events():
  evts = ["Approaching", "Entering", "Leaving",
          "Lower", "Raise"]
  sequence = []
  mandatory_sequence = ["Approaching", "Lower", "Entering",
                        "Leaving", "Raise"]
  sequence.extend(mandatory_sequence)
  pre_evts = random.sample(evts,
                          k=random.randint(0, len(evts)))
  sequence = pre_evts + sequence
  post_evts = random.sample(evts,
                           k=random.randint(0, len(evts)))
  sequence += post_evts
  if sequence[-1] != "Raise":
    sequence.append("Raise")
  return sequence
```

Listing 1: The level-crossing example, implemented by GPT 4 as a general Python implementation.

(`post_evts`). Certainly, it does not enforce the desired behavior in any meaningful way, and it is unclear how the different parts of the code relate to the requirements. This lack of alignment makes it difficult to fix the code to accurately reflect the intended behavior and hinders maintenance. Additionally, we note that the inclusion of the fixed sequence by the system results in over-specification, as this sequence is not necessarily required (for instance, if a second train approaches before the barriers are raised). We prompted the model several times, offering to use classes and methods. Despite multiple attempts, our efforts were unsuccessful.

In light of this small experiment, we identified the following research questions:

R1. Can LLMs be applied to translate multifaceted requirements into implementations without requiring a software design phase?

R2. Can developers use verification techniques effectively to identify and handle the errors produced by LLMs?

LLM coding benchmarks, such as HumanEval [12], MultiPL-E [13], and MBPP [14], primarily focus on generating code snippets dedicated to a specific task, e.g., software interview questions, or mathematical computations. We could not find LLM benchmarks for generating complete systems from requirements with cross-cutting aspects. Thus, we believe that raising these questions and striving towards higher-level benchmarks is important for the requirements-engineering community. Nevertheless, the focus of this paper is to emphasize the importance of aligning requirements to code for this task. Specifically, we illustrate how a modeling approach where each requirement is implemented in a separate dedicated code segment can help address the above two questions.

Encouraged by a recent position paper by Harel et al. [15], we suggest the behavioral programming (BP) model-based paradigm [16]. This approach allows the decomposition of the model generation process into smaller, more manageable parts. This modular approach not only harmonizes well with the capabilities of LLMs but also offers a structured methodology for composing and orchestrating these code fragments into larger, coherent software systems. Additionally, one of BP's primary design goals is the direct alignment with software requirements, as articulated in natural language by humans [17], [18]. This alignment can potentially reduce complexity and simplify the translation of requirements into an indirect implementation, offering an additional advantage:

R3. Can a programming model where each requirement is implemented in a separate module help LLMs translate multifaceted requirements into code?

In the following sections, we will introduce BP and illustrate how its properties offer an advantage in the context of automatic model generation. It is worth noting that while this paper focuses on BP, we do not claim that it is the only formalism that has these properties; rather, we claim that these properties are important in addressing this challenge. Other approaches can be used to address these challenges, provided they have similar attributes.

II. Behavioral Programming and BPpy

Behavioral Programming (BP) [16] is a model-based paradigm that enables users to specify the behavior of reactive systems directly, in alignment with their perception of the system requirements. In BP, users write scenarios, known as *b-threads*, representing behaviors the system should include or avoid. Each scenario is standalone, focusing on a specific aspect of the system behavior, typically a single requirement. At runtime, an application-agnostic execution engine interprets and seamlessly interweaves these scenarios to produce cohesive system behavior consistent with the specified requirements. This execution mechanism is based on a synchronization protocol introduced by Harel et al. [17]. The protocol involves each b-thread submitting a statement before selecting an event produced by the b-program. When a b-thread is ready to submit a statement, it synchronizes with its peers and specifies the events it requests, waits for (without requesting), or blocks. After the statement submission, the b-thread is paused. Once all b-threads have submitted their statements, the b-program reaches a *synchronization point*, where an event arbiter selects a single event that was requested and not blocked, and resumes all b-threads that requested or waited for that event. The remaining b-threads stay paused, and their statements are considered in the subsequent synchronization point. This process is repeated throughout the program's execution.

To provide an illustration of these concepts, we begin with a short example of a *b-program* (a set of b-threads) implemented in BPpy [19], an implementation of BP in Python. This example is adapted from one of the sample b-programs presented by Harel et al. [16], specifying a system responsible for controlling the mixing of hot and cold water from two respective taps. Listing 2 contains two b-threads, named `add_hot` and `add_cold`, each requesting the event of pouring a small amount of hot and cold water, respectively, three times. B-threads in BPpy are implemented as Python generators—functions capable of pausing themselves and passing data back to their caller at any point using the `yield` idiom. They can then be resumed when re-invoked using the `send` method. Statements submitted by the b-threads are structured as instances of the `sync` class, containing events or event sets labeled by the `request`, `block`, or `waitFor` arguments. BPpy's execution mechanism initiates by independently invoking each b-thread generator and awaiting its statement yield. Once all the statements are collected, an event is selected, and the program resumes its execution based on the protocol mentioned above.

```
@bp.thread
def add_hot():
    for i in range(3):
        yield sync(request=BEvent("HOT"))
@bp.thread
def add_cold():
    for i in range(3):
        yield sync(request=BEvent("COLD"))
```

Listing 2: The `add_hot` and `add_cold` b-threads. Each request to pour an amount of hot and cold water three times.

In contrast to many other paradigms, BP grants developers the flexibility not to be bound by a single predefined behavior for the implemented system. Instead, the system has the freedom to select any behavior aligned with all the defined b-threads. For example, a b-program consisting of the two b-threads in Listing 2 does not enforce a specific order for pouring cold and hot water. Consequently, its execution can produce all sequences containing exactly three occurrences of the `COLD` event and three occurrences of the `HOT` event. Examples of such sequences include `HOT,HOT,HOT,COLD,COLD,COLD` or `COLD,HOT,HOT,COLD,HOT,COLD`.

To illustrate further, consider that after running the initial version of the system, a safety concern arises, prompting the introduction of a new requirement, stating that having two consecutive `HOT` events is undesirable. While we can modify the `add_hot` b-thread by incorporating new conditions and statements, the BP paradigm advocates for preserving the alignment between existing b-threads and their respective requirements and adding a new b-thread. This approach fosters an incremental and modular development style, allowing developers to add or remove behaviors independently without impacting other b-threads. Thus, we add the `control` b-thread in Listing 3, which iteratively waits for the `HOT` event and then blocks it while waiting for any subsequent event using the `All` event set.

```
@bp.thread
def control():
    while True:
        yield sync(waitFor=BEvent("HOT"))
        yield sync(waitFor=All(), block=BEvent("HOT"))
```

Listing 3: The `control` b-thread, which prevents the `HOT` event from occurring consecutively.

The code provided in Listing 4 demonstrates how b-programs are instantiated and executed in BPpy. A `BProgram` class instance is defined with a list of b-threads, an event selection strategy (arbiter), and an optional listener. The `SimpleEventSelectionStrategy` class randomly selects an event from the set of enabled events, i.e., events that are requested and not blocked, uniformly. Alternative execution mechanisms can also be employed to resolve this non-determinism. This flexibility allows for the integration of various event selection strategies tailored to specific optimization needs, such as resource considerations or predefined objectives. Further, BPpy supports program listeners—entities that receive notifications at the initiation and completion of the b-program or upon event selection during its execution. This feature facilitates the integration of a b-program within a host application.

```
ess = bp.SimpleEventSelectionStrategy()
program_listener = bp.PrintBProgramRunnerListener()

b_program = BProgram(bthreads=[add_hot(), add_cold(),
                               control_temp()],
                     event_selection_strategy=ess,
                     listener=program_listener)
b_program.run()
```

Listing 4: Instantiating and executing a b-program in BPpy.

While the examples and experiments presented in this paper use BPpy, we note that BP has been implemented in other languages, including C++ [20], Java [17], and JavaScript [21]. Some of the ideas presented here can carry over to these frameworks as well. Additionally, BP has been applied in various applications, such as a tool suite for model-driven testing [22], a fully functional nano-satellite [23], and a reactive IoT building [24], highlighting its versatility and potential.

III. A Demo of How BP Can Boost Code Generation

To illustrate how BP can boost model generation from requirements, we turn back to the level-crossing example. We prompted OpenAI's GPT-4 model once again, instructing it to implement the level-crossing system according to the requirements outlined in Section I. In this attempt, the prompting asked it to create a b-program representing the system. The entire prompt we used is provided in the appendix [11], and the code used to run this prompting is available at https://github.com/bThink-BGU/Papers-2024-MoDRE-BP-LLM. The output b-program is given in Listing 5.

We can observe that, in contrast to the general implementation in Listing 1, the b-program implementation complies with Requirement 1, Requirement 2, and Requirement 4, with Requirement 3 being amendable with a minor adjustment (waiting for `Lower` instead of `Leaving`). Further, the alignment of each b-thread with a requirement makes the generated model easier to update and maintain.

IV. An Initial Evaluation of the Approach

To reinforce our claims, we expanded the level-crossing example and conducted an initial experiment with 20 system specifications involving a total of 149 requirements. Among these specifications, 10 were presented without context, simply describing systems using generic events (A, B, C, etc.) and

```
@bp.thread
def req_1():
    while True:
        yield sync(waitFor=BEvent("Approaching"))
        yield sync(waitFor=BEvent("Entering"))
        yield sync(waitFor=BEvent("Leaving"))
@bp.thread
def req_2():
    while True:
        yield sync(waitFor=BEvent("Approaching"))
        yield sync(request=BEvent("Lower"))
        yield sync(waitFor=BEvent("Leaving"))
        yield sync(request=BEvent("Raise"))
@bp.thread
def req_3():
    while True:
        yield sync(waitFor=BEvent("Approaching"))
        yield sync(block=BEvent("Entering"),
                   waitFor=BEvent("Leaving"))
@bp.thread
def req_4():
    while True:
        yield sync(waitFor=BEvent("Approaching"))
        yield sync(block=BEvent("Raise"),
                   waitFor=BEvent("Leaving"))
```

Listing 5: The level-crossing example, as implemented by GPT 4 as a b-program. The b-program contains a small bug that can be fixed with a minor adjustment.

including requirements regarding their ordering (e.g., "A must be followed by B"). The other 10 specifications depicted a system within the context of a frame story, as in the level-crossing example. All GPT 3.5 system implementations and the code we used for this evaluation are publicly available at https://github.com/bThink-BGU/Papers-2024-MoDRE-BP-LLM.

To evaluate the alignment of the implemented systems with their respective requirements, a sample of 100 system-generated runs was taken from each implementation and tested for validity against each requirement. Subsequently, we computed the number of requirements in which the b-program implementation exhibited better alignment based on the generated runs and vice versa. The results are available in Table I. Out of the 149 requirements, the b-program showed better alignment in 52 cases, while the non-BP implementation performed better in 37. The rest of the 60 requirements were evenly aligned in both approaches. We calculated that the probability of a random Bernoulli variable producing such an advantage for BP or less is 95.5%, indicating that the advantage of BP over plain Python in this evaluation is statistically significant. We believe that the root cause for this better alignment lies in BP's modularity and unified design, which allowed the model to implement the systems directly.

V. Verification and Validation Support

LLMs often introduce errors that prove challenging for programmers and other stakeholders to detect, underscoring the growing importance of validating the generated model for correctness. We state that the implementation can be formally verified using techniques such as model-checking, achieved by isolating and keeping the code that implements each requirement separate and concise. This enables new development cycle approaches where, for instance, a new

Specification	#Requirements	General	BP
r1	8	1	3
r2	7	2	1
r3	8	2	4
r4	8	5	1
r5	5	2	3
r6	8	2	4
r7	8	2	2
r8	5	0	5
r9	9	3	5
r10	6	1	3
rs1	4	1	3
rs2	4	1	2
rs3	8	3	0
rs4	10	4	4
rs5	8	1	1
rs6	9	1	3
rs7	9	3	0
rs8	9	0	3
rs9	8	3	1
rs10	8	0	4
Total	**149**	**37**	**52**

Table I: The initial experiment results, with 20 system specifications. Among these specifications, 10 are presented without any context (r1-r10), and 10 depict a system within the context of a frame story (rs1-rs10). Columns **General/BP** show the number of requirements out of **#Requirements** where the general/BP implementation exhibited better alignment. The remaining requirements were evenly aligned in both implementations.

prompt is constructed based on the verification results and fed back to the LLM. In this section, we introduce BPpy's model-checking support, which draws upon extensive research on the analysis of b-programs [25]–[27]. BPpy facilitates model checking through both explicit and symbolic modes. In the explicit mode, the program is validated through assertions in b-threads using Depth First Search. For instance, Listing 6 demonstrates how Requirement 3 can be verified in the explicit mode by adding a b-thread, which raises an assertion error if a train enters while barriers are raised.

```
@bp.thread
def check():
    while True:
        e = yield sync(waitFor=[BEvent("Entering"),
                                BEvent("Lower")])
        assert e == BEvent("Lower")
        yield sync(waitFor=BEvent("Raise"))

def bp_gen():
    ess = bp.SimpleEventSelectionStrategy()
    return BProgram(bthreads=[req_1(), req_2(), req_3(),
                              req_4(), check()],
                    event_selection_strategy=ess)

ver = DFSBProgramVerifier(bp_gen, max_trace_length=1000)
ok, counter_example = ver.verify()
```

Listing 6: Verifying Requirement 3 in the level-crossing b-program using explicit model-checking.

A symbolic model checker mode is available for general LTL support and larger state space. This mode explores only the state spaces of individual b-threads, bypassing the composite product space. Subsequently, this product space is analyzed using PyNuSMV [28] (a Python binding for NuSMV [29]), avoiding the explicit enumeration of all states. Detailed insights into the automatic translation of b-programs to SMV models are available in [19]. The verifier operates in two modes: Binary Decision Diagrams (BDD) and SAT-based Bounded Model Checking (BMC). The specification to be verified is written in NuSMV LTL specification format. For example, Listing 7 demonstrates how Requirement 4 can be verified symbolically using BDDs.

```
evt_list=[BEvent("Approaching"),BEvent("Entering"),
         BEvent("Leaving"),BEvent("Lower"),BEvent("Raise")]
s="G((event=Approaching)-> (event!=Raise U event=Leaving))"

ver = SymbolicBProgramVerifier(init_bprogram, evt_list)
result, ce = ver.verify(spec=s,
                        type="BDD",
                        find_counterexample=True)
```

Listing 7: Verifying Requirement 4 in the level-crossing b-program using symbolic model-checking.

VI. Conclusion

In this paper, we proposed the behavioral programming (BP) model-based approach to enhance the use of LLMs automatic model generation. We illustrate how LLMs can be applied to translate multifaceted requirements into implementations without requiring a software design phase by employing BP's unified design approach. Further, we suggest that a comprehensive environment, with supporting verification and validation methodology, is imperative for developers to effectively handle errors produced by LLMs. Moving forward, we plan to conduct more extensive and rigorous experiments. These experiments will involve additional model-driven approaches and an analysis of their formal verification methodologies within the context of models generated by LLMs.

References

[1] A. Radford, K. Narasimhan, T. Salimans, and I. Sutskever, "Improving Language Understanding by Generative Pre-Training," *OpenAI*, 2018.

[2] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," May 2019, arXiv:1810.04805 [cs].

[3] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[4] J. Wei, Y. Tay, R. Bommasani, C. Raffel, B. Zoph, S. Borgeaud, D. Yogatama, M. Bosma, D. Zhou, D. Metzler *et al.*, "Emergent abilities of large language models," *arXiv preprint arXiv:2206.07682*, 2022.

[5] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[6] J. Liu, C. S. Xia, Y. Wang, and L. Zhang, "Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation," Oct. 2023, arXiv:2305.01210 [cs].

[7] N. Leveson and J. Stolzy, "Safety Analysis Using Petri Nets," *IEEE Transactions on Software Engineering*, vol. SE-13, no. 3, pp. 386–397, 1987.

[8] A. Mazzeo, N. Mazzocca, S. Russo, and V. Vittorini, "A Systematic Approach to the Petri Net Based Specification of Concurrent Systems," in *Safety-Critical Real-Time Systems*. Boston, MA: Springer US, 1997, pp. 3–20.

[9] A. Elyasaf, T. Yaacov, and G. Weiss, "What Petri Nets Oblige us to Say: Comparing Approaches for Behavior Composition," *IEEE Transactions on Software Engineering*, vol. 49, no. 04, pp. 2303–2317, Apr. 2023, place: Los Alamitos, CA, USA Publisher: IEEE Computer Society.

[10] OpenAI. (2023) gpt-4-turbo-preview. [Online]. Available: https://openai.com/blog/new-models-and-developer-products-announced-at-devday

[11] T. Yaacov, A. Elyasaf, and G. Weiss, "Boosting llm-based software generation by aligning code with requirements — appendix," https://github.com/bThink-BGU/Papers-2024-MoDRE-BP-LLM/blob/main/appendix.pdf, 2024, [Accessed 07-05-2024].

[12] OpenAI. (2021) Humaneval. [Online]. Available: https://github.com/openai/human-eval

[13] Northeastern University Programming Research Lab. (2024) Multipl-e. [Online]. Available: https://github.com/nuprl/MultiPL-E

[14] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le, and C. Sutton, "Program Synthesis with Large Language Models," Aug. 2021, arXiv:2108.07732 [cs].

[15] D. Harel, G. Katz, A. Marron, and S. Szekely, "On Augmenting Scenario-Based Modeling with Generative AI," in *Proc. 12th Int. Conf. on Model-Driven Engineering and Software Development (MODELSWARD)*, 2024.

[16] D. Harel, A. Marron, and G. Weiss, "Behavioral programming," *Communications of the ACM*, vol. 55, no. 7, pp. 90–100, 2012.

[17] ——, "Programming Coordinated Behavior in Java," in *ECOOP 2010 – Object-Oriented Programming*, T. D'Hondt, Ed. Berlin, Heidelberg: Springer, 2010, pp. 250–274.

[18] A. Elyasaf, "Context-Oriented Behavioral Programming," *Information and Software Technology*, vol. 133, p. 106504, May 2021, publisher: Elsevier BV.

[19] T. Yaacov, "BPpy: Behavioral programming in Python," *SoftwareX*, vol. 24, p. 101556, Dec. 2023.

[20] D. Harel and G. Katz, "Scaling-Up Behavioral Programming: Steps from Basic Principles to Application Architectures," in *Proceedings of the 4th International Workshop on Programming based on Actors Agents & Decentralized Control*, ser. AGERE! '14. New York, NY, USA: Association for Computing Machinery, Oct. 2014, pp. 95–108.

[21] M. Bar-Sinai, G. Weiss, and R. Shmuel, "BPjs: an extensible, open infrastructure for behavioral programming research," in *Proceedings of the 21st ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*. New York, NY, USA: Association for Computing Machinery, 2018, pp. 59–60.

[22] M. Bar-Sinai, A. Elyasaf, G. Weiss, and Y. Weiss, "Provengo: A Tool Suite for Scenario Driven Model-Based Testing," Aug. 2023, arXiv:2308.15938 [cs].

[23] M. Bar-Sinai, A. Elyasaf, A. Sadon, and G. Weiss, "A scenario based on-board software and testing environment for satellites," in *The 59th Israel Annual Conference on Aerospace Sciences (IACAS), 2019*, 2019.

[24] A. Elyasaf, A. Marron, A. Sturm, and G. Weiss, "A Context-Based Behavioral Language for IoT." in *MODELS Workshops*, 2018, pp. 485–494.

[25] D. Harel, R. Lampert, A. Marron, and G. Weiss, "Model-checking behavioral programs," in *Proceedings of the ninth ACM international conference on Embedded software*. New York, NY, USA: Association for Computing Machinery, 2011, pp. 279–288.

[26] D. Harel, A. Kantor, G. Katz, A. Marron, L. Mizrahi, and G. Weiss, "On composing and proving the correctness of reactive behavior," in *2013 Proceedings of the International Conference on Embedded Software (EMSOFT)*. IEEE, 2013, pp. 1–10.

[27] D. Harel, G. Katz, A. Marron, and G. Weiss, "The effect of concurrent programming idioms on verification: A position paper," in *2015 3rd International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*, Feb. 2015, pp. 363–369.

[28] S. Busard and C. Pecheur, "PyNuSMV: NuSMV as a Python Library," in *NASA Formal Methods*, ser. Lecture Notes in Computer Science, G. Brat, N. Rungta, and A. Venet, Eds. Berlin, Heidelberg: Springer, 2013, pp. 453–458.

[29] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri, "NUSMV: a new symbolic model checker," *International Journal on Software Tools for Technology Transfer*, vol. 2, no. 4, pp. 410–425, Mar. 2000.