# Model-based Verification and Monitoring for Safe and Responsive Robots

S. Bernagozzi*†§, S. Faraci*, E. Ghiorzi†*, K. Pedemonte†, A. Ferrando‡, L. Natale*, A. Tacchella†
*Istituto Italiano di Tecnologia – Genova, Italia
†Università di Genova – Genova, Italia
‡Università di Modena e Reggio Emilia – Modena, Italia

***Abstract*—Model-based development (MBD) is gaining widespread acceptance as a design technique in robotics. Not only it eases the work of developers and makes software deployment faster than traditional code-based approaches, but it also enables the application of formal methods without the need of specific training for developers. In particular, (off-line) automated verification is known to improve the quality of software and enable early recognition of hard-to-find bugs whereas (on-line) monitoring helps to close the reality-gap by running either on simulators or actual hardware. In this paper we present a comprehensive approach to MBD, verification, and monitoring that enables developers to design robust control software by featuring both off- and on-line checks in a push-button fashion, i.e., one where all the complexity of formal methods is hidden under the hood of the development tools.**

***Index Terms*—Model-based Development, Verification, Monitoring**

## I. Introduction

Once dismissed as a purely technological matter, methodologies to design and implement control software for robots have been drawing increasing attention from researchers. The need to develop robots that can take autonomous decisions in dynamic and unstructured environments while maintaining safe and secure operations is a formidable challenge that makes traditional code-based design and implementation of software components inadequate for most kinds of applications. Following similar developments in more mature sectors like automotive and aerospace, model-based design (MBD) emerged as a viable solution, to the point that a substantial body of literature is now available on the subject, including methodologies and their successful applications to robotics.

While MBD – coupled with the availability of middleware and off-the-shelf software components – eases the design, implementation, and composition of control architectures, robots are not just complex cyber-physical implements, they are also expected to take autonomous decisions that $(i)$ do not cause damage to the environment or themselves and $(ii)$ do achieve expected results. MBD alone is not enough to guarantee either requirement, and traditional software engineering practices like testing, might be too difficult or expensive to implement, even considering just simulated scenarios. In particular, developers need a way to catch bugs in their designs to avoid expensive fixes at later implementation stages. Also, since not all errors can be caught timely, it is important to ensure that no remaining error impacts adversely on the environment or the robot during operation.

Our proposal is a synergistic combination of three elements: $(i)$ model-based design $(ii)$ off-line automated verification and $(iii)$ on-line monitoring. We claim that all three ingredients are required in order to improve confidence in the control architecture to the point of ensuring that robots remain safe and responsive while maintaining their ability to react to changes in the environment. In particular, model-based design is a fundamental enabler because it shifts focus from code to models which can be assigned a formal semantics to encode their intended behavior. While programs have formal execution semantics as well – the one implemented by the compiler/interpreter – the advantage of models is that their syntax is simpler and their semantics crispier than corresponding codebases. Therefore, automated verification of models is usually simpler than automated verification of programs because lots of implementation details are omitted, but flaws in the logical design and combination of components can still be spotted. The main limit in off-line verification is that the considered models abstract away details. Simulation and real-world operation may behave differently with respect to the models considered during verification, possibly because the robot reacts autonomously to a change in the usual operation scenario. For this reason, monitoring is the third required ingredient, one which ensures that the reality-gap will never affect adversely the operation of the robot. Indeed, the very same requirements that are verified off-line on models, can be also checked on-line in simulation or real operation using monitors generated automatically from the requirements.

To the best of our knowledge, the only framework that targets the same objectives as ours is presented in [1], where the authors describe an approach that starts from a standard, but rigorous, framework to specify and deploy robotic software components which can also automatically synthesize a formal model of these components. However, their approach is reversed with respect to ours in that they execute the resulting

§Corresponding Author: Stefano Bernagozzi.
Email: stefano.bernagozzi@iit.it
This work was funded by the European Union under the Horizon Europe grant 101070227 (CONVINCE). 

formal model in place of a traditional implementation, and show how it provides the opportunity to add verification and monitoring. While several approaches featuring MBD, off-line verification, and monitoring exist and have been successfully applied to robotics (see the extensive survey presented in [1]), most of them do not provide a complete toolchain that spans from design to implementation.

The paper is structured as follows. In Section II we introduce the basic elements we build upon. In Section III we describe a scenario to test our methodology that we introduce in Section IV. The results of our tests are described in Section V, and in Section VI we conclude the paper with some remarks and an agenda for future research.

## II. Background

### A. Behavior Trees

Behavior Trees (BTs) [2] are gaining acceptance in the robotics community for model-based design of deliberation policies [3]. Since their adoption is relatively recent, the semantics of BTs is not yet standardized. Therefore, while their graphical syntax is widely agreed upon, it is not infrequent to find inconsistent semantics across different publications and code libraries [4]. In the following, we give a brief informal introduction to the syntax and semantics of BTs considering the one proposed in [5] as a first attempt to overcome some of the aforementioned inconsistencies. In Figure 1 we can see a BT with a *root node* "$\emptyset$" and a single *reactive sequence* "$\xrightarrow{\text{R}}$" child node; this node has three children which correspond to subtrees. The subtree in Figure 2 consists of a *reactive fallback* "R?" node with two children, a *condition node* ("BatteryLevel") and an *action node* ("Alarm"). The execution of the main BT starts by sending a *tick* signal to the root node and by propagating it down the structure of the tree. A condition node responds either *success* or *failure* depending on whether the stated condition is true or false, whereas an action node may also respond *running* if the action is still being accomplished. The response of reactive fallback and reactive sequence nodes depends on their children's: a fallback node will tick the first child and, upon success of the child it will return success, whereas upon failure of the child subsequent nodes will be ticked; a sequence node will tick the first child and, upon success of the child, will tick the subsequent nodes. Therefore, a fallback node returns failure only if all children fail, while a sequence node returns success only if all children succeed. In the case of reactive sequence and fallback, running is returned when the ticked child returns running. While this introduction is by no means exhaustive, it is enough to make sense of most of our case study; further details will be given in the following sections when necessary.

### B. Finite State Machines

Finite State Machines (FSMs) are a well-established paradigm in model-based design to represent policies at the deliberation stage and to describe either abstract or concrete models of elements and the functional and control stages of a software architecture. Abstract models of the environment can also be provided in terms of FSMs, possibly including probabilistic or timed transitions. In our paper, we consider state machines described with a subset of the SCXML standard [6], where we disallow hierarchical state machines and code scripting. Indeed, such features would make formal treatment challenging: the semantics of hierarchical FSMs are somewhat inconsistent across contributions in the literature [7], and code scripting (based on ECMAScript) would require assigning semantics to code fragments. Aside from these provisions, we consider the standard syntax of SCXML and the semantics described (informally) in [6], where each machine has states and transitions among them, and transitions are either unconditional or based on events which are either internal, i.e., coming from the machine itself, or external, i.e., coming from some other machine. Both internal and external events are queued and processed according to their order of occurrence. Handling of external events enables several machines to run in an asynchronously parallel fashion.

### C. Middleware

We assume that the implementation of the control architecture whose models are given in terms of BTs and FSMs is based on ROS2 [8], a communication middleware widely used in the robotics community. It is mainly composed of three types of primary interfaces: topics, services, and actions. Topics are based on the publisher-subscriber pattern, so they can have multiple publishers and subscribers for the same topic. In addition to that, each topic has a message type which specifies the data that will be published or read on that topic. On the other side, both services and actions are based on the request-response pattern, but services are meant for computation guaranteeing a quick turnaround time, while actions are for general, potentially time-consuming computations. While the client of the service is blocked until the server provides a response, actions support asynchronous interaction through two services — one for making the request and one for requesting the result — and a topic which, after the request has been made, sends feedback to the client with the status of the computation.

### D. Statistical Model Checking

To perform off-line verification we utilize SCAN[1] (Statisti-Cal ANalyzer) a statistical model checker designed to verify models of asynchronous processes exchanging data over FIFO buffers — a *channel system* in the parlance of [9], where each buffer is a *channel* and data items are *messages*. In SCAN, the underlying model features also timed transitions and invariants [9]. SCAN accepts SCXML-specified FSMs running asynchronously (section II-B) which are internally converted to a channel system, where event queues are modeled by channels, and SCXML events are modeled by messages.

The staple feature of SCAN is to sample concrete execution of channel systems to find the probability that a property is satisfied, within specific confidence bounds. In practice,

[1]https://github.com/convince-project/scan

SCAN runs sufficiently many Monte Carlo simulations of the model to prove a statistical result characterized by precision and confidence parameters. To determine the number of tests required, we adopt the Adaptive criterion from [10] which updates the halting threshold dynamically, based on the test results as they are obtained. Other than providing statistically sound success rate results, SCAN can save the execution traces it produces in terms of events being sent and received by SCXML machines, thus allowing developers to directly relate the execution traces to the specified model and properties.

In particular, SCAN deals with temporal properties expressed in Metric Temporal Logic with past-tense connectives (past-MTL) [11] with super-dense time semantics [12] and state-event support [13], which is meant to align it with the monitoring system (section II-E) so that the same properties verified on the models can be monitored at runtime on the corresponding implementations. We consider standard past-MTL properties written with the following syntax:

$$\varphi := \top \mid p \mid \neg\alpha \mid \alpha \wedge \beta \mid \alpha\, \mathbf{S}[a:b]\, \beta$$

where $p$ is a member of a finite set of *predicates*, $\alpha$ and $\beta$ are formulas. Past-MTL is interpreted over timed-traces where each element of the trace has a timestamp. Let $w$ be a timed trace, and $t$ be a timestamp of $w$, then we write $w, t \models \varphi$ to mean that $w$ satisfies $\varphi$ at time $t$ according to the following rules:

- $w, t \models \top$ for every $t \in w$
- $w, t \models p$ if predicate $p$ is true in $w$ at time $t$
- $w, t \models \neg\alpha$ when $w, t \not\models \alpha$
- $w, t \models \alpha \wedge \beta$ when $w, t \models \alpha$ and $w, t \models \beta$
- $w, t \models \alpha\, \mathbf{S}[a:b]\, \beta$ when there exists $t' \in [t-b, t-a]$ such that $w, t' \models \beta$ and for all $t''$ such that $t' < t'' < t$ it is also the case that $w, t'' \models \alpha$ for each $t''$.

Intuitively, the temporal operator *since* ($\mathbf{S}$) states that $\beta$ should be satisfied at some time $t'$ preceding $t$ in the interval $[a, b]$ and $\alpha$ is satisfied ever since. Other propositional connectives are dealt with the usual equivalences, e.g., $\alpha \vee \beta \equiv \neg(\neg\alpha \wedge \neg\beta$ and $\alpha \implies \beta \equiv \neg\alpha \vee \beta$, and additional temporal operators *past eventually* ( $\mathbf{P}$ ) and *historically* ( $\mathbf{H}$ ) can be defined as

$$\mathbf{P}[a:b]\, \alpha := \top\, \mathbf{S}[a:b]\, \alpha \qquad \mathbf{H}[a:b]\, \alpha := \neg\, \mathbf{P}[a:b]\, \neg\alpha$$

Corresponding unbounded versions of temporal connectives $\mathbf{S}$, $\mathbf{P}$ and $\mathbf{H}$ can also be given. For instance, $t, w \models \mathbf{H}\, \alpha$ if $\alpha$ is always satisfied before $t$. In SCAN, predicates are restricted to the messages sent along the channels corresponding to messages that can be monitored in the real system. We choose not to verify the internal state of the processes, as models might be an abstraction over the actual components in the system, and thus their actual state neither corresponds to some state in the actual component, nor it can be monitored.

### *E. Monitoring*

ROSMonitoring [14] is a tool to monitor ROS interfaces for a system. The tool has three components: instrumentation, oracle, and monitor. The instrumentation is used to create a monitor to be subsequently placed as a man-in-the-middle between two components of the system; the oracle is used to check whether the monitored parts satisfy certain properties given in a formal language; finally, the monitor is used to take the messages from the monitored topic or service and to forward them to the oracle. The oracle we consider is based on Reelay [11] to implement past-MTL property checking. In our case, monitoring is different between topics and services: for topics, the monitor leverages the publish-subscribe architecture of ROS: it only listens to the topic for new messages and logs them leaving unchanged the communication between the components, for services it acts as a real man-in-the-middle, so it receives the message from the first component and it forwards it to the second component.

## III. CASE STUDY

We consider a simulation example representing a simplified humanoid robot museum tour guide task. The simplified task requires the robot to navigate to two designated points of interest (POIs), while checking the percentage of its battery and the presence of the group of visitors it is guiding.

The behavior tree in figure 1 represents the simulated task. It comprises three subtrees: battery management, POI scheduling, and navigation.

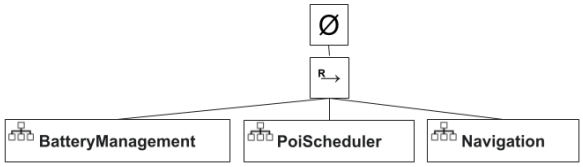


Fig. 1. Case study Behavior Tree: each of the three leaf nodes is a sub-tree. Here we can see the logic, at first we check the battery and if it is low we manage it, then we set the current PoI with the PoI scheduler and finally we navigate to the current PoI.

The Battery Management subtree checks if the robot battery level (in percentage) is below a given threshold. If so, it starts an alarm. This is done using a ReactiveFallback control node, the "BatteryLevel" condition node, and the "Alarm" action node, as shown in Figure 2.

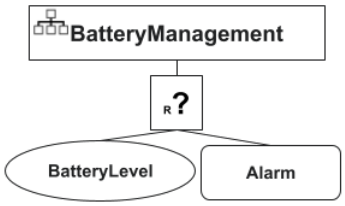


Fig. 2. Battery Management sub-tree: it checks the battery level and if it is below 30% it ticks the alarm skill.

The POI scheduling subtree, shown in Figure 3, handles the setting of two POIs. The subtree is composed of two identical branches, one for each POI. Each branch is composed of a "IsPoiDone" condition leaf which checks whether the POI has been executed; if false the POI is set as the current POI

with the "SetPoi" leaf, otherwise, it executes the branch of the second POI. When both PoIs have been visited, the "Reset" leaf is ticked, both PoIs are marked as not done and the tour starts from the beginning.

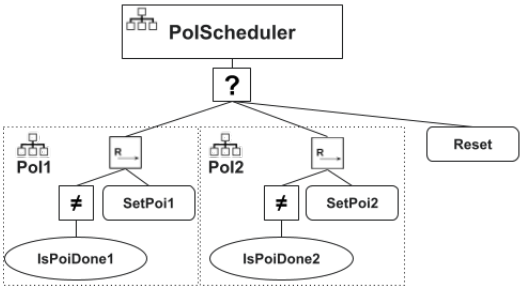


Fig. 3. Poi Scheduler sub-tree: it checks on the blackboard if a PoI is already done, and if so it goes to the next one. If all the PoIs are done it reset everything to start again.

The first part of the Navigation tree (Figure 4) is the visitors following sub-tree which comprises the "VisitorsFollowing" node that checks if the group of visitors is present and performs appropriate actions. For simplification, here the fallback is a simple "Wait" action. In the second part of the navigation tree the "GoToPoiAction" activates the navigation component to bring the robot to the current POI, while "SetCurrentPoiDone" sets that POI as done when it is reached.

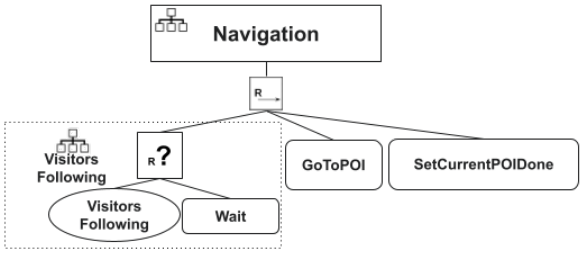


Fig. 4. Navigation sub-tree: at first it checks if visitors are following the robot, if not it waits for them. Then it goes to the current PoI and, once reached, it marks the current PoI as done.

We consider five requirements that the task described above should satisfy, namely:

1) If the battery level is above or equal to 30% threshold then the alarm should not be active.
2) If the battery level drops below the 30% threshold then the alarm should activate.
3) The battery level should be published every $t$ seconds.
4) If POI_1 is not completed it should be selected.
5) The state of the visitors should be published every $t$ seconds.

## IV. Methodology

As shown in Figure 5, we assume that the design structure comprises four levels: task plan, plugins, skills, and components. Each level is formally modeled in XML and has a C++ executable counterpart. In our case, the *task plan* is a policy

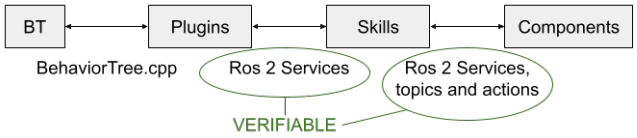


Fig. 5. System interfaces: here we can see the whole architecture with the interfaces between the various pieces and where the monitoring can happen.

represented in XML by a behavior tree. Its leaf nodes are handled by two different kinds of *plugins*, one for actions the other for conditions. Plugins are simple ROS nodes that directly interact with the *skills*, by forwarding the tick to them. The skills, which have the prefix of the name equal to the leaf nodes, provide the desired functionality by orchestrating the *components* that interact with the environment. For plugins and components, the models are abstract SCXML formalizations of the actual executable code and, therefore, are given as input only to the model-checking tool; while the code of task plan and skills is directly generated from the model. The generation of the code for task plan and skills gives us the opportunity to monitor them at run-time, allowing us to discover bugs due to reality-gaps in the models. While in principle it is possible to extend our approach to also monitor components, we are focused on verification and monitoring of task plan and skills.

An overview of our methodology is shown in Figure 6. Since the model comprises also some custom tags to ease development over the ROS2 middleware[2], we need to convert it to plain SCXML to give it as input to SCAN. This step is achieved by AS2FM software[3], which outputs the whole system as a set of SCXML files that are compliant with the standard. For the execution of the model in simulation instead, the behavior tree can be directly executed using BehaviorTree.cpp[4]. The plugins, and the components are executable, while for the skills we need a conversion to an executable code. This step is done via the model2code software [5], which takes as input a state machine and generates an executable which executes that state machine and interacts via ROS2, as in Figure 5.

For each requirement we proceed in three steps: at first we choose the correct pattern following the patterns presented in [11], then we write those properties in natural language in a manner that can be directly translated into the patterns and at the end we write the resulting PAST-MTL formula based on the selected pattern. All the intermediate result of this process can be seen in Table I.

## V. Experimental Evaluation

In order to ensure the correctness and reliability of the system we want to answer two main questions: i) are results

[2]The tags allow to specify communication over ROS2 interfaces, but these ought to be expanded into corresponding models of such protocols in terms of plain SCXML machines

[3]https://github.com/convince-project/AS2FM

[4]https://www.behaviortree.dev/

[5]https://github.com/convince-project/model2code

| Property No. | Pattern | Text | PAST-MTL formulae |
|---|---|---|---|
| 1 | ABSENCE BEFORE | It is always the case that the "alarm" does not occur before the "battery_level $< 30\%$" occurs | $\mathbf{H}\,(battery\_level \geq 30\% \implies \mathbf{H}\,(\neg alarm))$ |
| 2 | RESPONSE GLOBALLY | It is always the case that "alarm" responds to "battery level $< 30\%$" within $t_1$ | $\mathbf{H}\,((alarm \implies \mathbf{P}\,(battery\_level < 30\%)) \wedge \neg(\neg alarm\ \mathbf{S}[t_1 :]\ battery\_level < 30\%))$ |
| 3 | RECURRENCE GLOBALLY | It is always the case that "battery_published" at least every $t_2$ time units | $\mathbf{H}\,((\mathbf{P}[t_2 :] True) \implies (\mathbf{P}[: t_2]\, battery\_pub))$ |
| 4 | RESPONSE GLOBALLY | It is always the case that "POI_1_selected" responds to not "POI_1_completed" | $\mathbf{H}\,((POI1\_sel \implies \mathbf{P}\ \neg POI1\_comp) \wedge \neg(\neg POI1\_sel\ \mathbf{S}[t_3 :]\ \neg POI1\_comp))$ |
| 5 | RECURRENCE GLOBALLY | It is always the case that "visitors_published" at least for every $t_5$ seconds | $\mathbf{H}((\mathbf{P}[t_4 :] True) \implies (\mathbf{P}[: t_4]\, visitors\_pub))$ |

TABLE I

PROPERTIES WITH THE RELATED PATTERN TAKEN FROM [11], THE TEXT EXPRESSED IN FORMAL LANGUAGE AND THE CORRESPONDENT PAST-MTL FORMULA

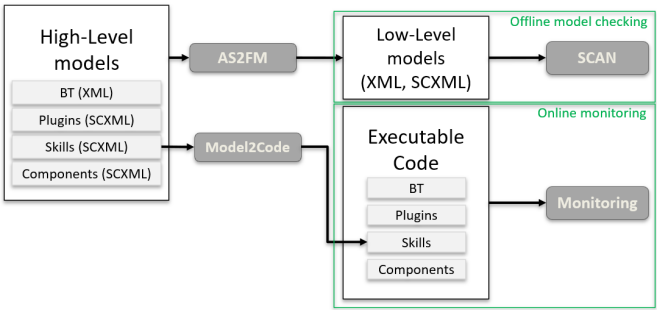


Fig. 6. Methodology overview: on the left we can see the input of the whole pipeline, while on the right the outputs and their usage.

of SCAN reliable and can the monitoring find flaws due to the reality gap? ii) can the use of both frameworks enhance the reliability of the system? To answer these questions we have tested separately SCAN and ROSMonitoring using a simulation, which can be found in the Github repository[6]. For the SCAN program, we have a model of the whole system that can be simulated, while for ROSMonitoring we have a simulation with all components (Navigator, Alarm, Battery and Visitors Following) that can be executed stand-alone.

### A. Off-line verification

The models of the system translated into plain SCXML using AS2FM and the properties written in XML have been given as input to the SCAN model checker. The selected SCAN parameters are the default ones: $0.95$ for confidence, $0.01$ for precision, and the maximum length of execution trace is set to $500000$ steps. SCAN takes a maximum of two minutes to check each property, and the resulting precision decreases with the number of runs until it reaches $\pm 0.01$.

Table II shows the success rate for each property considering different temporal bounds (multiples of $100ms$). The first property has a success rate equal to one and does not have a time bound. The bound of property 2 has been found manually by trials: in this case the bound depends on the interaction among ticks and scheduling of different processes and it is not relevant per se, but it shows that a bound exists. Time bounds for properties 3 and 5 are correct because they corresponds to the rate at which the components publish the corresponding values (i.e., 1Hz), threfore a period of 10 time units, in both cases. The properties are always true given a bound greater or equal to 10, and false otherwise. The bound of property 4 is two time units since the system needs to be initialized and, at the beginning, the POI 1 is not yet selected. The bound corresponds to the time required by the Behavior Tree to tick the first leaf nodes and retrieve their values from the components before setting the POI 1 as selected.

[6]https://github.com/convince-project/MBVM-SIMPAR2025

| Property ID | Temporal Bound | Success Rate |
|---|---|---|
| 1 | - | 1.00 |
| 2 | $t_1 \geq 62$ | 1.00 |
| | $t_1 < 62$ | 0.00 |
| 3 | $t_2 \geq 10$ | 1.00 |
| | $t_2 < 10$ | 0.00 |
| 4 | $t_3 \geq 2$ | 1.00 |
| | $t_3 < 2$ | 0.00 |
| 5 | $t_4 \geq 10$ | 1.00 |
| | $t_4 < 10$ | 0.00 |

TABLE II

MODEL CHECKING RESULTS (TEMPORAL BOUNDS ARE IN MULTIPLES OF $100ms$)

### B. Monitoring

We tested all the properties in Section III for various time bounds, in order to catch the ones which lead the properties to a failure when running on actual code. Results can be found in table III. The first property has no time bounds and it is verified correctly by our monitor. Time bounds for properties 3 and 5 are correct because they correspond to the rate at which the components "Battery" and "Visitors Following" publish their messages through topics (i.e., 1 Hz): this explains also the time bound which reflects the publishing rate of the topic. For the last two properties their value is independent of the publication frequency but it is due to the response of the system. We have found the value for property 2 by trials corresponding to 2.5 seconds. By looking at the traces we determined that this value corresponds, as expected, to the average time required by the alarm to become active after the drop of the battery. This is equivalent to approximatly two ticks of the behavior tree. For property number 4 instead, the bound value is related to the time that the behavior tree needs to reach the next action. Also in this case the bound corresponds to the time required

by the Behavior Tree to tick the first leaf nodes and retrieve their values from the components before setting the POI 1 as selected.

| Property ID | Temporal Bound [s] | Success Rate |
|---|---|---|
| 1 | - | 1.00 |
| 2 | $t_1 \geq 2.5$ | 1.00 |
| | $t_1 < 2.5$ | 0.00 |
| 3 | $t_2 \geq 1$ | 1.00 |
| | $t_2 < 1$ | 0.00 |
| 4 | $t_3 \geq 0.2$ | 1.00 |
| | $t_3 < 0.2$ | 0.00 |
| 5 | $t_4 \geq 1$ | 1.00 |
| | $t_4 < 1$ | 0.00 |

TABLE III
MONITORING RESULTS

## VI. CONCLUSIONS AND FUTURE WORK

Considering the experiments presented in the previous section, we can conclude that the overall methodology can deal successfully with simple but non-trivial systems. In particular, leveraging statistical model checking with SCAN maintains the promise of scaling well: the internal channel system corresponding to our case study contains more than 50 processes and 400 channels which would make it prohibitively large for state-of-the-art classical model checkers. Furthermore, SCAN can spot bugs at the conceptual level that relate to timing constraints: all properties tried turn out to be satisfied as long as the choice of time bounds is consistent with the structure of the model and the interaction among the components. However, monitoring is required to fill the reality-gap and ensure that bounds can be stated so that such constraints will be fulfilled also during execution. Indeed, monitoring takes into account time bounds in wall-clock seconds, whereas SCAN considers abstract "ticks" which may realize differently depending on the speed at which the overall system runs to. In the future, we plan to extend our methodology to consider more complex case studies enabling the monitoring of the components and wrap the toolchain behind a single easy-to-use graphical user interface. We are also planning to extend the experimental analysis to include other state-of-the-art systems, like SMC-STORM [15], that can be integrated for verification and monitoring within our framework.

## REFERENCES


[1] S. Dal-Zilio, P. Hladik, F. Ingrand, and A. Mallet, "A formal toolchain for offline and run-time verification of robotic systems," *Robotics Auton. Syst.*, vol. 159, p. 104 301, 2023.

[2] M. Colledanchise and P. Ögren, *Behavior trees in robotics and AI: An introduction*. CRC Press, 2018.

[3] M. Iovino, E. Scukins, J. Styrud, P. Ögren, and C. Smith, "A survey of behavior trees in robotics and ai," *Robotics and Autonomous Systems*, vol. 154, p. 104 096, 2022.

[4] M. Colledanchise and L. Natale, "On the implementation of behavior trees in robotics," *IEEE Robotics and Automation Letters*, vol. 6, no. 3, pp. 5929–5936, 2021.

[5] E. Ghiorzi and A. Tacchella, *Execution semantics of behavior trees in robotic applications*, 2024. arXiv: 2408.00090 [cs.RO]. [Online]. Available: https://arxiv.org/abs/2408.00090.

[6] J. Barnett, R. Akolkar, R. Auburn, M. Bodell, D. C. Burnett, J. Carter, S. McGlashan, T. Lager, M. Helbing, R. Hosn, T. Raman, K. Reifenrath, N. Rosenthal, and J. Roxendal, *State chart xml (scxml): State machine notation for control abstraction*, last accessed 2024-10-18, 2015. [Online]. Available: https://www.w3.org/TR/scxml/.

[7] É. André, S. Liu, Y. Liu, C. Choppy, J. Sun, and J. S. Dong, "Formalizing UML state machines for automated verification - A survey," *ACM Comput. Surv.*, vol. 55, no. 13s, 277:1–277:47, 2023.

[8] S. Macenski, T. Foote, B. Gerkey, C. Lalancette, and W. Woodall, "Robot operating system 2: Design, architecture, and uses in the wild," *Science Robotics*, vol. 7, no. 66, eabm6074, 2022. DOI: 10.1126/scirobotics.abm6074. [Online]. Available: https://www.science.org/doi/abs/10.1126/scirobotics.abm6074.

[9] C. Baier and J.-P. Katoen, *Principles of model checking*. MIT press, 2008.

[10] C. E. Budde, P. R. D'Argenio, A. Hartmanns, and S. Sedwards, "An efficient statistical model checker for nondeterminism and rare events," *International Journal on Software Tools for Technology Transfer*, vol. 22, no. 6, pp. 759–780, 2020.

[11] D. Ulus, "Online monitoring of metric temporal logic using sequential networks," *arXiv preprint arXiv:1901.00175*, 2019.

[12] H. S. Sarjoughian and S. Sundaramoorthi, "Super-dense time trajectories for devs simulation models," in *Proceedings of the Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium*, ser. DEVS '15, Alexandria, Virginia: Society for Computer Simulation International, 2015, pp. 249–256, ISBN: 9781510801059.

[13] S. Chaki, E. M. Clarke, J. Ouaknine, N. Sharygina, and N. Sinha, "State/event-based software model checking," in *Integrated Formal Methods*, E. A. Boiten, J. Derrick, and G. Smith, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 128–147, ISBN: 978-3-540-24756-2.

[14] A. Ferrando, R. C. Cardoso, M. Fisher, D. Ancona, L. Franceschini, and V. Mascardi, "Rosmonitoring: A runtime verification framework for ros," in *Towards Autonomous Robotic Systems*, A. Mohammad, X. Dong, and M. Russo, Eds., Cham: Springer International Publishing, 2020, pp. 387–399, ISBN: 978-3-030-63486-5.

[15] M. Lampacrescia, M. Klauck, and M. Palmas, "Towards verifying robotic systems using statistical model checking in storm," in *Bridging the Gap Between AI and Reality*, B. Steffen, Ed., Cham: Springer Nature Switzerland, 2025, pp. 446–467, ISBN: 978-3-031-75434-0.