Code Generation and Monitoring for Deliberation Components in Autonomous Robots

S. Bernagozzi[1,2], S. Faraci[1], E. Ghiorzi[2,1], K. Pedemonte[2], L. Natale[1], A. Tacchella[2]

***Abstract*— Hand-coded deliberation components are prone to flaws that may not be discovered before deployment and that can be harmful to the robot and its execution environment, including the people within it. To reduce development effort and at the same time increase confidence in robot's safety, we propose to model deliberation components at a conceptual level, to automatically generate code from such models and also to monitor their execution during robot operation. We present two tools, one which compiles models of deliberation components into executable code, and one which generates runtime monitors from the models. We have tested them in simulation, to demonstrate the usefulness of combining together model-based development, code generation, and monitoring.**

I. INTRODUCTION

In spite of substantial technological progress in robotics and AI, the issue of making autonomous decisions in dynamic and unstructured environments while maintaining safe and secure operations is still standing [1]. Moreover, this challenge is faced by development teams where professionals with diverse technical backgrounds and with various levels of experience are involved in the design and implementation of control software. The part that often turns out to be critical is the *deliberation layer*, i.e., the components that are meant to shape the behavior of a robot considering its mission and the current state of the environment. Traditional code-based design, implementation and integration of deliberation components are prone to flaws that can escape simulation and testing and that may not be discovered before deployment where they can cause inefficient or unwanted behaviors. It is to address such issues that researchers have been putting increasing effort in methodologies that can support design and implementation of control architectures that guarantee the expected quality of service as well as stated safety and security requirements [2], [3]. Our research follows on these steps by considering a model-based design (MBD) approach. In particular, we assume that deliberation components are modeled at a conceptual level using well-known abstractions, such as finite-state machines (FSMs) and behavior trees (BTs), and that the implementations of the components interact among them and with other layers of the control architecture through well-defined protocols and a common middleware — ROS 2 in our case [4]. In this context, our main research question is to understand whether we can ease the development of deliberation components and, at the same time, improve the confidence in their correct behavior without introducing unnecessary burden on the developers. We contribute two tools and their experimentation on a case study to show that it is possible to reconcile shorter development time and improved confidence in the implementations: Model2Code (M2C), which automatically compiles conceptual models of deliberation components into executable code compatible with ROS 2 ; and MOnitoring-ON-line (MOON), which generates runtime monitors from the models to improve confidence in the generated code as far as compliance with their specification is concerned. MOON can monitor *properties* and *models*. The former are a set of conditions expressed in formal language that data exchanged through the communication channels in the architecture must satisfy (see Table I for examples of properties). For the latter, MOON verifies that monitored components in the architecture behave according to a given model. We test the combination of M2C and MOON considering a simulated, but realistic, use-case scenario where we demonstrate that (i) M2C generates code that does not trigger unwanted or inefficient behaviors during execution, (ii) MOON can spot different faults injected in the implementation of deliberation components and (iii) model monitors run without excessive overhead even in combination with property monitors. Furthermore, we show that, in some cases, model monitors — that come "for free" by the specification of components — can spot issues that property monitors may not intercept.

Corresponding Author: Stefano Bernagozzi.
stefano.bernagozzi@iit.it
This work was funded by the European Union under the Horizon Europe grant 101070227 (CONVINCE).
We acknowledge the financial support from PNRR MUR Project PE0000013 "Future Artificial Intelligence Research (FAIR)", funded by the European Union – NextGenerationEU, CUP J33C24000430007
[1]*Istituto Italiano di Tecnologia – Genova, Italia*
[2]*Università di Genova – Genova, Italia*

II. PRELIMINARIES

a) CONVINCE project: The tools and methods presented in this work are part of the CONVINCE project (CONtext-aware Verifiable and adaptIve dyNamiC dEliberation) which aims to develop a software toolchain that assists developers in designing and developing fully verified robot deliberation systems [3]. The goal of CONVINCE is to advance the capabilities of robots to perform complex tasks robustly and safely within unstructured environments via autonomous and unsupervised adaptation to the environment and operational context, by developing cognitive deliberation capabilities that ensure safe robot operation over extended periods of time without human intervention. In CONVINCE the reference architecture is made of three layers:

- A **deliberation layer** which orchestrates the various skills in order to obtain the desired behavior, e.g., guide

a group of visitors in a museum and present them some artworks.
- A **skill layer** which implements basic robot capabilities, e.g., understand whether visitors are still engaged or not.
- A **functional layer** which interacts directly with the hardware, e.g., navigation, speech recognition, vision and object detection.

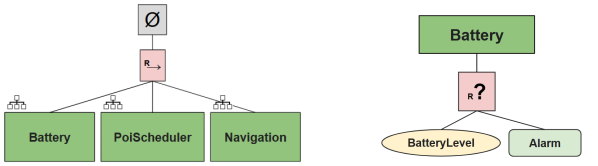
These layers are meant to be connected via a middleware in order to exchange data and coordinate with each other.

b) BTs: Recent surveys [5], [6] demonstrate that BTs [7] are widely used in the robotics community for model-based design of deliberation policies. We consider the following kind of nodes and related graphical syntax for BTs:

- **Root** node represented as "$\emptyset$".
- Control nodes (color-coded in red):
  - **Reactive Sequence** represented as "$^{\mathbf{R}}\rightarrow$".
  - **Reactive Fallback** represented as "$_{\mathbf{R}}?$".
  - **Fallback** represented as "?".
- **Inverter** represented as "$\neq$" (color-coded in blue).
- **Condition Node** represented as an oval (color-coded in yellow).
- **Action Node** represented as a rounded rectangle (color-coded in light green).
- **Sub-Tree** represented as a rectangle (color-coded in green).

Due to their recent adoption, the semantics of BTs is not yet standardized across applications in robotics. While an attempt to propose a formal semantics has been done in [8] and [9], it is easy to find inconsistencies in the implementation of different libraries [10]. In this paper, we consider the semantics implemented by the library `BehaviorTree.cpp`[1] whereon our implementations are based. To understand how BTs work, in Figure 1(a) we present a BT with a *root node* and a single *reactive sequence* child node; this node has three children which correspond to sub-trees. In Figure 1(b) we can see a BT that consists of a *reactive fallback* node with two children, a *condition node* ("BatteryLevel") and an *action node* ("Alarm"). Execution starts when a *tick* signal is sent to the root node, which propagates it down the structure of the tree. We assume that ticks are sent periodically (e.g., once every second) and that the BT returns a result within two consecutive ticks. Inside the BT, a condition node can respond either *success* if the stated condition is true, or *failure* if the stated condition is false. Action nodes may also respond *running*, if the corresponding action is still being accomplished. The execution semantic of the control nodes depends on their nature. A fallback node will tick the first child and, upon the 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 the 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 both nodes, if one of the children returns running, then, at the following iteration, that is the first child to be ticked, while the others keep the result returned at the previous tick. The reactive sequence will act like the sequence, but if the previous result of a child node was running it will send the tick again to all its children.

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



(a) Example of BT with sub-trees (b) Example of a sub-tree

Fig. 1. Example of BTs

c) FSMs: FSMs are commonly used to model policies and to represent either abstract or concrete models of the functional and control stages of a software architecture. FSMs including probabilistic or timed transitions can also represent abstract models of hardware components or the environment. We represent state machines using a subset of the SCXML standard [11], excluding only hierarchical state machines and code scripting. Those features can be an issue: to begin with, the semantics of hierarchical FSMs are somewhat inconsistent across contributions in the literature [12]. Therefore, the implementation is library dependent which would make model monitoring inconsistent. Secondly, code scripting (based on ECMAScript) needs to assign semantics to code fragments, which may become problematic for large pieces of code.
We use the following representation of SCXML:

- **State**: a state of an SCXML machine;
- **Event**: an event received by the FSM, it can be *internal*, i.e. originated from the FSM itself, or *external*, i.e. originated from a source external from the FSM;
- **Transition**: a transition from a state to another, that can be *unconditional* or *event-based*.

Events are managed according to SCXML specification, i.e., all of them are queued and processed according to their occurrence. In addition, we have enriched our language with middleware-specific elements to retrieve implementation-specific information. Specifically, in the case of ROS 2 , we have implemented "services" and "topics" to specify if an event is generated or generates a message over them. Those elements are fundamental for the code generation and monitoring, explained in Sections III and IV.

d) Middleware: The use of a middleware for the communications among components enables us to distribute them on different machines with low effort, allowing the control architecture to scale as needed. In principle, our approach can be implemented on top of any middleware, but in this paper we assume that BT and FSMs models are based on ROS 2 [4], a communication middleware widely used in the robotics community. ROS 2 is mainly composed of three primary interfaces:

- **Topics**: are based on the publisher-subscriber pattern, their communication is many to many meaning that we can have multiple publishers and multiple subscribers for each topic. They are asynchronous and typed.
- **Services**: are based on the client-server pattern. They are synchronous and typed. Their communication is many to one, meaning that multiple clients can connect to one server.
- **Actions**: are a fusion between services and topics, and they are made for long computations. An action is composed by two services, one for the start of the computation and one for getting the result. In addition to that, there is one topic that, once the computation is started, gives to the client a feedback on the status of the computation. This allows the client to continue its work and retrieve the result upon action completion.

III. M2C

The skill layer (Section II-a) is implemented using FSMs based on the SCXML language (Section II-c). In these models, events trigger state transitions, enabling the machine to change its behavior and interact with the external world. Although SCXML state machines can be compiled and executed, they cannot directly interface with the functional layer, as these elements are not defined in SCXML. This introduces a gap between the skill and the functional layer, requiring an interface that translates SCXML events into actionable commands that interact with the system according to the chosen framework. M2C is a tool designed to automate the generation of this interface system for each state machine. The tool allows users to focus on the high-level design of the state machines while automating the generation of the underlying interface, whose manual development would be time-consuming and error-prone. M2C employs a template-based approach, taking as input the SCXML models and generating as output the corresponding ROS 2 C++ code that translates SCXML events into ROS 2 interfaces as stated in Section II-d. The input files include the SCXML model and the XML file describing the system architecture specifying the definition of the interfaces. Starting from a set of C++ template files, it produces the following output: C++ header and source of the executable skill, a main file containing the event handling code and the corresponding ROS 2 translation, the `CMakeLists.txt` configuration file and the XML ROS 2 package file. The output can be built directly with *CMake* without further editing. The code of the tool can be found on github[2].

IV. MOON

MOON is a runtime monitor developed for CONVINCE, implemented on top of ROSmonitoring [13], to which we have added a model verification and an automatic monitor generation.

MOON accepts the same model specification as M2C and provides monitor generation for properties and models. As of now, monitor generation is only available for services and topics, with the aim to extend it also to actions in the future. For inspecting messages travelling across channels and verify properties, MOON relies on asynchronous communication facilities provided by the publish-subscribe mechanisms available in ROS 2 . For monitoring models, on the other hand, it introduces specific man-in-the-middle components that intercepts messages exchanged by ROS 2 services. This introduces a certain degree of overhead, which can be reduced using introspection facilities available in recent distributions. Implementing model monitoring using these facilities is in progress. Once the messages are inspected, they are forwarded to the oracle. The oracle supports monitoring through property specification and through model specification. In the former case the oracle checks that the monitored communication channels satisfy a property as described in [14], and in this case we can both monitor the call of the service and the response of the service, depending on what is specified in the property. For model monitoring, the oracle considers the FSM of a skill created at design-time and ensures that the execution of the skill implementation — generated by M2C from the very same FSM — is compliant to the model. The oracle is currently implemented using the Apache Commons SCXML library[3], in order to support SCXML models. Our model specification is compiled into a fully compliant SCXML state machine, called the Oracle State Machine, which is the state machine that used at runtime to evaluate if the skill is compliant to its FSM specification. The oracle processes inputs and outputs of the monitored skill and simulates its behavior by forwarding the monitored communication to the Oracle State Machine, and compares produced outputs with monitored ones, alerting in case any discrepancy is found. In general, monitors need to be configured manually by providing information on the relevant parts of the system that need to be monitored; however, since all relevant information is included in our model specification, MOON automatically generates the monitor configuration files. The code of the tool can be found on github[4].

V. EXPERIMENTAL EVALUATION

A. Use case description

Consider a scenario where a humanoid robot guides visitors on a laboratory tour. To simplify the simulation, speech-based interaction is not considered. The robot navigates the laboratory to reach two predefined points of interest (POIs), while actively ensuring that visitors are following, stopping when they fall behind. It also monitors its battery level, triggering an alarm when it drops below a certain threshold. According to the architecture defined in Section II-a we have three layers. The functional layer is not implemented by us and we only know about its interfaces, the skill layer is implemented with FSMs as defined in Section II-c, and the deliberation layer is implemented with a BT, as defined

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

[3]`https://github.com/apache/commons-scxml`

[4]`https://github.com/convince-project/MOON`

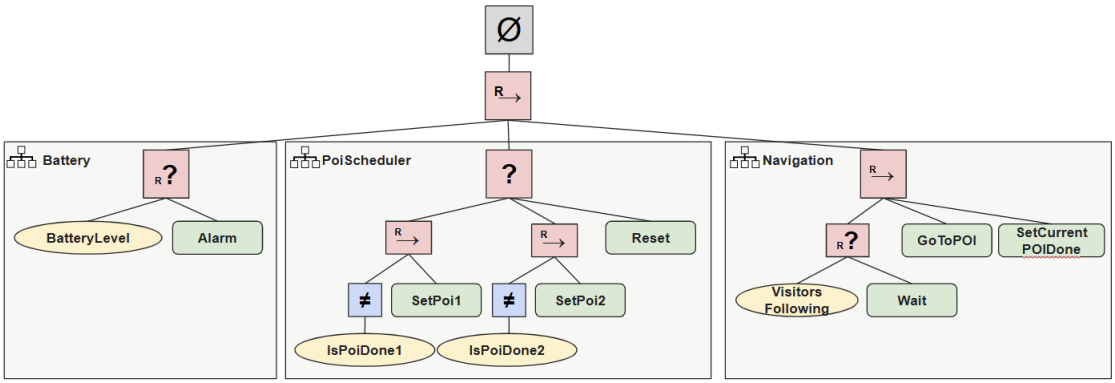


Fig. 2. BT of the case study: expanded version

in Section II-b. All the communications happens through the ROS 2 middleware. The robot's policy is encapsulated in a BT, whose compact representation in sub-trees is shown in Figure 1(a), while the extended version is shown in Figure 2. The Battery sub-tree is composed of a reactive fallback control node that ticks the "BatteryLevel" leaf, which returns failure when the battery of the robot is below the threshold, in which case the Alarm leaf node is ticked and an alarm is triggered. The "PoiScheduler" sub-tree handles the setting of the current PoI through a fallback node whose first and second branches refer to the first and the second PoI, respectively. The "IsPoiDone" leaf nodes checks if a specific PoI is reached, while the "SetPoi" nodes set a specific PoI as the current one. The "Reset" leaf node is ticked when the tour is over to start a new tour and reset each PoI. The "Navigation" sub-tree checks for the presence of the visitors and waits for them if not present. Otherwise, the "GoToPOI" action leaf node is ticked, returns "Running" and the robots move to the current destination. When the PoI is reached, the "SetCurrentPOIDone" is ticked and that PoI is marked.

B. Monitoring

We consider the skills "BatteryLevel", "Alarm", "IsPoiDone1", "SetPoi1" for model monitoring, since they have a different internal logic and rely on different ROS 2 interfaces, i.e., the "BatteryLevel" skill relies only on topics from the environment while other skills have one or more services to communicate with. Additionally, some properties related to system-wide requirements are available. Each property, as shown in Table I, has an assigned pattern which is translated to a temporal logic formula from which a property monitor can be generated — see [14] for details. We run both model monitors and property monitors to check for configurations where model monitors can complement property monitors while providing an even greater level of automation. In particular, while generating property monitors requires users to define specific requirements, model monitors are generated from the FSMs designed by developers, without additional effort placed on them. To evaluate the ability of the system to detect anomalous circumstances, bugs are injected into the selected skills, either on the SCXML State Machine or in the generated code. We consider both methods because they give us more guarantees when assessing the robustness of the overall approach. For instance, changing an SCXML event we can simulate a wrong translation of that event by M2C. We introduced three different categories of faults:

1) **Unresponsiveness**: the skill does not respond to the tick.
2) **Incorrect response**: the skill responds to the tick but not always correctly.
3) **Incorrect parameter names**: the skill has a spelling error, for example in the event, service, or topic name.

The code used for the experiments can be found on github[5].

VI. EXPERIMENTAL RESULTS

A. Bug discovering

For the tests we use a single fault model, i.e., in each configuration a single bug is injected, ensuring that this is the only difference from the nominal behavior of the system. The monitored properties are shown in Table I, while a summary of the tests is shown in Table II.

a) Unresponsiveness (T1-2): In these cases the skill does not respond to the tick, so the BT is stuck waiting for the response. For T1 we have modified the SCXML of the "BatteryLevel" skill in order to achieve this behavior. Once executed, the model monitor detects right away a difference between the outputs of the skill and its specification, since the latter has an additional event, i.e., the tick response. MOON keeps comparing the outputs and asserts that there is an anomaly after a timeout. The property monitor in this case uses mainly P1 and P2. P1 is always verified since the "Alarm" skill is never ticked, therefore MOON does not detect the anomaly; in case of P2, an anomaly is detected, but only after the battery level drops below 30% following its initial value of 100%. In test T2 we have modified the "SetPoi1" skill in the same way as "BatteryLevel" in T1. Also in this case, the model monitor discovers the anomaly in the same way as in T1, while the property monitor does not detect it because the monitor sets "POI_1_selected" to true once it receives the tick response, which never happens.

[5]https://github.com/convince-project/CGM-IROS2025

ID	Pattern	Property Description
P1	ABSENCE BEFORE	It is always the case that the "alarm" does not occur before the "battery_level $< 30\%$" occurs
P2	RESPONSE GLOBALLY	It is always the case that "alarm" responds to "battery level $< 30\%$" within t
P3	RESPONSE GLOBALLY	It is always the case that "POI_1_selected" responds to not "POI_1_completed"within t

TABLE I
MONITORED PROPERTIES

b) Incorrect response (T3-4): In these cases the skill responds to the tick but the response is aleatory due to errors in the state machines. Specifically, in T3 we inject the fault in the SCXML of the "BatteryLevel" skill to randomly return success or failure. Similarly, in T4 in we inject the fault in the SCXML of the "Alarm" skill, which randomly returns success or failure but it also never forwards the alarm to the functional layer. In both tests, the model monitor detects the anomaly at the first occurrence. The property monitor P1 detects the fault in T3 at the same time as the model monitor, since the "BatteryLevel" skill returns "Failure" and the BT ticks the alarm when the battery charge is still high. The monitor P2, instead, does not become false because it never happens that the battery drops below 30%. In T4, the monitors P1 and P2 never become false because they only check the tick of the skill and not its response.

c) Incorrect parameter name (T5-6): In these cases the skill has a misspelling in the name of a parameter. Particularly, in T5 the "BatteryLevel" skill is subscribed to a misspelled ROS 2 topic, so it does not read the level of the battery, keeping its internal level variable at 100%. Instead, in T6 the "IsPoiDone1" skill has 2 interfaces communicating with the functional layer (due to the man-in-the-middle of MOON) one monitored and the other not monitored by the model monitor. In this case, the skill calls the interface that is not monitored, so the model monitor does not receive any information about the call to the functional layer. In T5, the model monitor detects the anomaly when the battery level drops below 30%. Since the model monitor cannot see that the internal state of the variable and the response to the tick is always successful, the model monitor does not detect any anomaly. Instead, when the battery level drops below 30%, the skill returns always a successful answer while the one in the monitor returns failure. The property monitor detects the anomaly at the same time as the model monitor with the property P2, while P1 never detects anything since alarm is never ticked. Finally, in T6 the model monitor detects the anomaly immediately, while the property monitor does not detect any anomaly because the program seemed to be working correctly from its standpoint.

B. Overhead added

To test how much overhead our monitor added, we have retrieved the tick time of each skill with each configuration:

- **conf 1** no monitors,
- **conf 2** only property monitors active,
- **conf 3** only model monitors active,
- **conf 4** all monitors active.

As we can see in Table III, the tick time of each skills with both monitors increases as expected. The main difference between property monitors and model monitors is that property monitors use asynchronous communication from ROS 2 instead of a synchronous man-in-the middle. The overhead added by the model monitors is on average 200-250 millisecond per skill. This is because the execution of the tick blocks while waiting for the execution of the monitors, which executes synchronously with the service call. These results demonstrates that monitoring properties and models together is feasible. The added overhead is not negligible but it is acceptable in this context, as we focus on monitoring components at the deliberation level, whose response time is larger than the added overhead. In addition, we expect that most overhead due to model monitoring can be reduced by replacing the man-in-the-middle approach with asynchronous channel inspection based on ROS 2 introspection mechanisms.

VII. CONCLUSIONS AND RELATED WORK

Our experimental results provide convincing evidence that automating the generation of skill implementations and monitoring the resulting code can effectively increase confidence in the correct behavior of deliberation components without increasing the burden of developers. Our tests also demonstrate that model monitors can successfully complement property monitors and do not add additional burden on the developers since they do not require them to elicit specific properties. To the best of our knowledge, the only framework that targets the same objectives as ours is presented in [2], [15], where the authors describe a rigorous approach 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 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 [2]), most of them do not provide code generation from conceptual models (as done by M2C) together with property and model monitoring (as provided by MOON), both on top of a widely used and supported middleware such as ROS 2 .

REFERENCES

[1] K. Rajan and A. Saffiotti, "Towards a science of integrated ai and robotics," *Artificial Intelligence*, vol. 247, pp. 1–9, 2017, Special Issue on AI and Robotics, ISSN: 0004-3702.

Test ID	Type of fault	Skill	Model Monitor	Property monitor
T1	Unresponsiveness	BatteryLevel	Detects after timeout	P1 - No detections P2 - Detection when battery_level $< 30\%$ P3 - Not relevant
T2	Unresponsiveness	SetPoi1	Detects after timeout	P1, P2 - Not relevant P3 - No detections
T3	Incorrect response	BatteryLevel	Detects at first occurrence	P1 - Detects at first occurrence P2 - No detections P3 - Not relevant
T4	Incorrect response	Alarm	Detects at first occurrence	P1, P2 - No detections P3 - Not relevant
T5	Incorrect parameter	BatteryLevel	Detects at first occurrence	P1 - No detections P2 - Detection when battery_level $< 30\%$ P3 - Not relevant
T6	Incorrect parameter	IsPoiDone1	Detects at first occurrence	P1, P2 - not relevant P3 - No detections

TABLE II

EXPERIMENTAL RESULTS

Skill Name	Average Tick Time (μs)			
	conf 1	conf 2	conf 3	conf 4
BatteryLevel	3,877	4,103	56,943	65,670
IsPoiDone1	6,906	7,030	314,621	327,816
SetPoi1	6,094	7,663	308,686	333,764
Alarm	4,699	8,151	284,138	298,117
ROOT	29,904	37,036	810,121	864,467

TABLE III

AVERAGE TICK TIME FOR EACH SKILL AND THE ROOT OF THE BT

[2] S. Dal Zilio, P.-E. Hladik, F. Ingrand, and A. Mallet, "A formal toolchain for offline and run-time verification of robotic systems," *Robotics and Autonomous Systems*, vol. 159, p. 104 301, 2023, ISSN: 0921-8890. DOI: https://doi.org/10.1016/j.robot.2022.104301.

[3] C. Street, Y. Warsame, M. Mansouri, M. Klauck, C. Henkel, M. Lampacrescia, M. Palmas, R. Lange, E. Ghiorzi, A. Tacchella, *et al.*, "Towards a verifiable toolchain for robotics," in *Proceedings of the AAAI Symposium Series*, vol. 4, 2024, pp. 398–403.

[4] 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.

[5] 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.

[6] R. Ghzouli, T. Berger, E. B. Johnsen, S. Dragule, and A. Wasowski, "Behavior trees in action: A study of robotics applications," in *Proceedings of the 13th ACM SIGPLAN international conference on software language engineering*, 2020, pp. 196–209.

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

[8] E. Ghiorzi and A. Tacchella, *Execution semantics of behavior trees in robotic applications*, 2024. arXiv: 2408.00090 [cs.RO].

[9] F. Ingrand, "A formal implementation of behavior trees to act in robotics," *arXiv preprint arXiv:2502.11904*, 2025.

[10] 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.

[11] 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/.

[12] 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.

[13] 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.

[14] S. Bernagozzi, S. Faraci, E. Ghiorzi, K. Pedemonte, A. Ferrando, L. Natale, and A. Tacchella, "Model-based verification and monitoring for safe and responsive robots," in *2025 IEEE International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR)*, 2025, pp. 1–6. DOI: 10.1109/SIMPAR62925.2025.10979077.

[15] F. Ingrand, "Recent trends in formal validation and verification of autonomous robots software," in *2019 Third IEEE International Conference on Robotic Computing (IRC)*, 2019, pp. 321–328. DOI: 10.1109/IRC.2019.00059.