A Model Checkable UML Soccer Player

Valentin Besnard
ERIS, ESEO-TECH
Angers, France
valentin.besnard@eseo.fr

Ciprian Teodorov
Lab-STICC UMR CNRS 6285
ENSTA Bretagne, Brest, France
ciprian.teodorov@ensta-bretagne.fr

Frédéric Jouault
ERIS, ESEO-TECH
Angers, France
frederic.jouault@eseo.fr

Matthias Brun
ERIS, ESEO-TECH
Angers, France
matthias.brun@eseo.fr

Philippe Dhaussy
Lab-STICC UMR CNRS 6285
ENSTA Bretagne, Brest, France
philippe.dhaussy@ensta-bretagne.fr

***Abstract*—This paper presents a UML implementation of the MDETools'19 challenge problem with EMI (our Embedded/Experimental Model Interpreter). EMI is a model interpreter that can be used to execute, simulate, and formally verify UML models on host or embedded targets. The tool's main specificity relies on a single implementation of the language semantics such that consistency is ensured between all development phases: from design to verification and execution activities. Using this approach, we have succeeded in (i) designing a UML model for the challenge problem, (ii) applying formal verification using model-checking on the design model, and (iii) executing this model in order to participate in the challenge.**

***Index Terms*—UML, Model-Driven Engineering, Tool**

I. Introduction

In this paper, we present a solution to the MDETools'19 challenge problem [Wor19] based on EMI, our UML model interpreter. This challenge provides an interesting case study to show the usability of Model-Driven Engineering (MDE) tools and to facilitate comparison and evaluation of such tools. This year, the challenge is a RoboCup-type game where two robots are competing in a soccer game. The goal of the challenge is to design a software application to control one of the robot for scoring the maximum number of goals during a match. The match is officiated by a referee that notifies robots with some information (e.g., end of game, penalty when a soccer player keeps the ball for too much time). A soccer simulator made with Unity gives a graphical view of the soccer game. Two TCP connections (one for the player and the other one for the referee) can be used to connect the designed software application to the soccer simulator and control the robot.

Designing such a software program requires to verify its behavior. With the increasing complexity of systems, software programs are more often exposed to uncertain behaviors, security flaws, as well as design mistakes that could lead to potential critical failures. To verify and validate such software applications, MDE provides capabilities to analyze models of these programs, especially with formal verification techniques (e.g., model-checking), during early design phases.

To this end, our solution (Github repository: https://github.com/ValentinBesnard/mdetools19-emi) for the MDETools'19 challenge problem is a UML [OMG17b] model that can be verified and executed with EMI (our Embedded/Experimental Model Interpreter). EMI is an interpreter of UML models that relies on a single implementation of the language semantics for all activities: simulation, verification, and execution. This tool provides a communication interface such that analysis tools can control the model execution and reuse the operational semantics of our model interpreter for the verification step. As a result, the same combination of model and interpreter is used for model verification and execution on actual systems. This gives more confidence in the fact that what is verified is what is executed in comparison to other techniques that rely on unproved generation steps (e.g., code generation, model transformation). The main specificities of our approach as well as the software architecture of this tool have been presented in [BBJ+18b], [BTJ+19], [BBJ+18a], [BBD+17]. The novelty of this paper is to show usability of this prototype on a case study of the MDE community.

Using this tool, we provide a solution to the MDETools'19 challenge problem using the following process. First, we design a UML model of the system to (i) handle displacements of the robot, and (ii) control suction for aspiring and shooting the ball. Then, we design an abstracted model of the system environment, which has been connected to the system model for verification purposes. The OBP2 model-checker [TLRDD16], [TDLR17] (https://plug-obp.github.io/) has been connected to EMI for applying simulation and model-checking. We have also deployed one observer automaton on the actual system to monitor a safety property at runtime. We finally connected the UML model to the soccer simulator using real TCP connections. The robot has been successfully controlled. In average, it scores 10 goals per match without opponent.

The remainder of this paper is structured as follows. Section II makes a brief presentation of EMI. Section III presents the design of our UML model before we explain how it has been analyzed in Section IV. Then, we execute this model with the actual soccer simulator in Section V. In Section VI, we discuss limitations and strengths of EMI, and in Section VII we review some related work. Finally, we conclude this paper in Section VIII.

II. Brief Presentation of EMI

EMI (https://plug-obp.github.io/bare-metal-uml/) is a tool dedicated to the verification and execution of UML models. This section makes a reminder of its specificities presented in [BBJ+18b], [BTJ+19].

The main characteristic of this model interpreter is to rely on a single implementation of the language semantics for all activities: simulation, verification, and execution. This unique definition of the language semantics is encoded into our model interpreter as its operational semantics. A communication interface is also provided to connect analysis tools to this model interpreter. This interface enables to control model execution through the interpreter and to reuse the same implementation of the language semantics. As a result, the same couple (model + operational semantics) is used for model verification and actual execution.

In comparison to classical approaches, this technique avoids semantic gaps caused by code generation or model transformations from the design model. Our technique also avoids to build, prove, and maintain an equivalence relation between the executable code and the model used for formal verification. In our approach, the same model and the same definition of the language semantics are used for verification and execution. This offers two main benefits: (1) it ensures that what is verified is what is executed, and (2) it facilitates the understanding of analysis results that are directly expressed in terms of design concepts.

Using EMI, we can apply multiple verification and validation (V&V) activities directly on the design model. The OBP2 model-checker can be used to perform trace-based simulation and LTL model-checking [BBJ+18b]. It is also possible to encode safety properties into UML observer automata and to deploy these observer automata on the actual target to make runtime monitoring [BTJ+19]. The same observer automata can also be used in model-checking during early verification phases. If model-checking checks the design model exhaustively, it requires an abstracted model of the system environment to perform the verification. Therefore, monitoring offers a good complementarity to model-checking in case of bad environment abstractions or state-space explosion.

This model interpreter is particularly relevant for executing UML models of embedded systems. For this purpose, it can be deployed on bare-metal (i.e., without OS) on embedded targets (e.g., STM32 discovery) or on host computers running a Linux Operating System (OS). Only the latter platform has been used in the context of this case study. The interested reader should refer to [BBJ+18b] for more details about model deployment.

III. Design of the UML Model

To address the MDETools challenge, the first step has been to design a UML model of the system. This section describes the main design principles used for this purpose.

A. Modular UML Model

For executing and verifying this software application, we made a modular UML model such that different environment models can be connected to the system at different times. This concept is shown on the composite structure diagram of our model in Figure 1. The *Main* composite structure is composed of the *system* and of an *environment*. Both components are linked together through ports and communicate by sending signals. This modular UML model is divided in several files: one that defines signals and common interfaces, one for the *system*, one for each *environment*, and one for the *Main* composite structure that instantiate both the *system* and an *environment*. For this model, we design two environment models: an abstracted environment model and a concrete environment model. The abstracted environment model is an abstraction of the actual environment to perform trace-based simulation and formal verification. The concrete environment implements the two TCP connections to the soccer simulator: one for the player and the other one for the referee.

This UML model conforms to the UML subset supported by EMI, which can be represented by class, composite structure, and state machines diagrams. In this subset, states machines transitions can have a trigger (i.e., a received event that triggers the transition), a guard (i.e., a predicate), and an effect. All

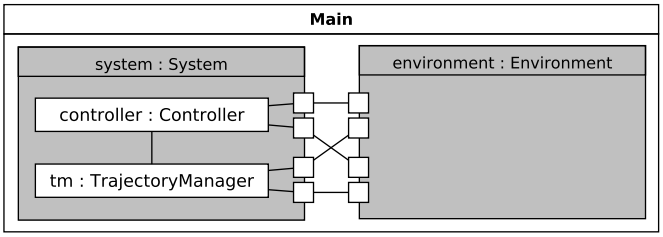

Figure 1: General composite structure diagram of the UML model.

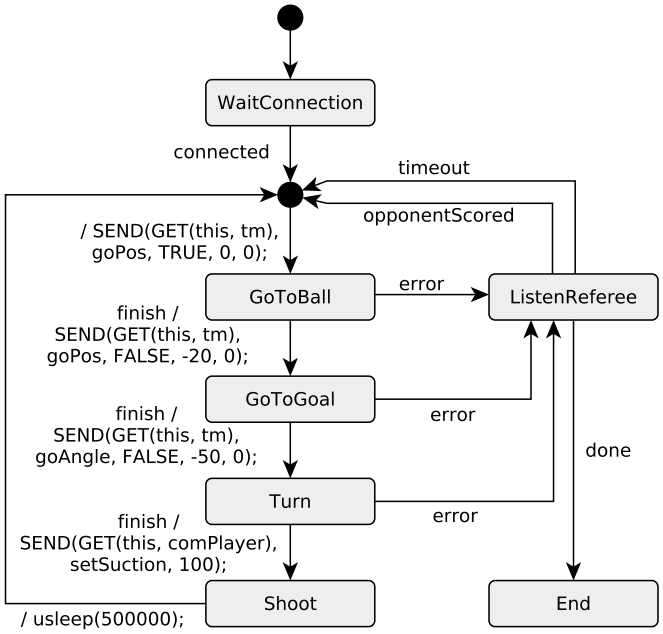

Figure 2: State machine of the Controller.

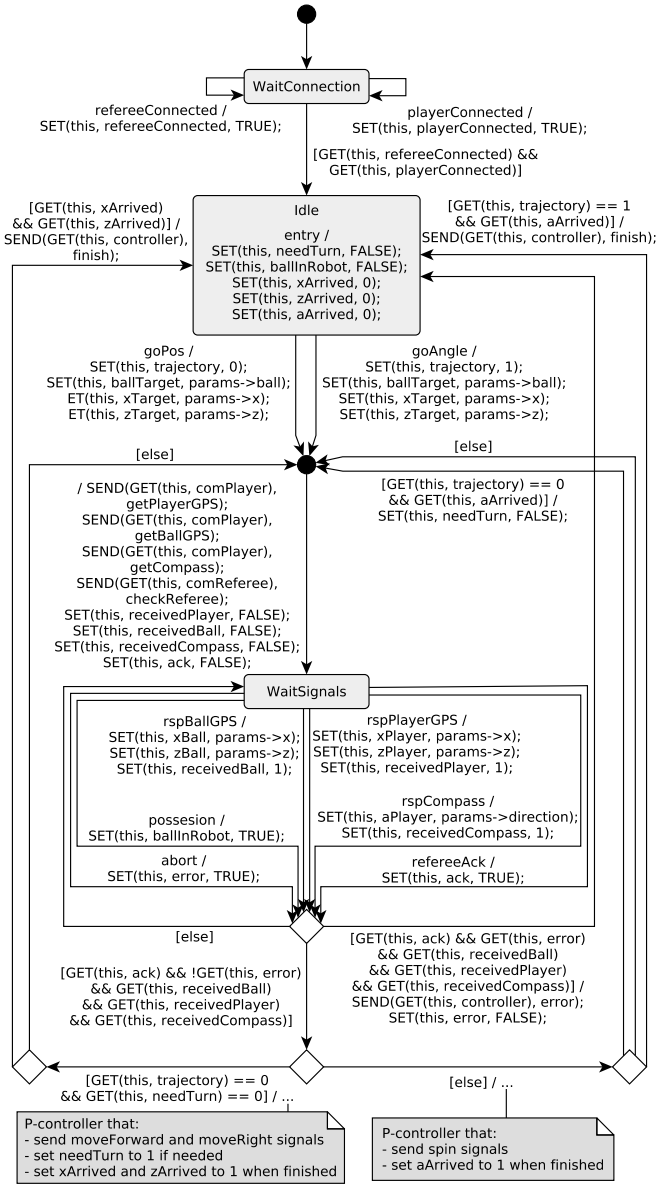

Figure 3: State machine of the TrajectoryManager.

transitions without triggers rely on completion events for being fired. To define the fine-grained behavior, we use an action language to describe transitions guards and effects. In our tool, the action language is based on the C programming language with additional C macros to access UML objects. These expressions are valid C statements that can be executed as they stand by our model interpreter. A more detailed presentation of our action language is given in [BBJ+18b]. Hence, this UML subset enables to describe both the structural and the behavioral parts of UML components.

B. System Design

The *system* component is composed of two objects: a *controller* that describes the high-level strategy of the soccer player, and a trajectory manager (*tm*) that can control the robot according to different kinds of trajectories.

The *controller* behavior is depicted on the state machine in Figure 2. Once the connection is established with the soccer simulator, the *controller* tries to perform the following list of actions in loop during all the match duration:

1) Move towards the ball and aspire it when the robot is close to it.
2) Move to the shooting position (`x=-20, z=0`) located in front of the opponent's goal
3) Turn towards the opponent's goal
4) Shoot the ball
5) Restart to 1).

For all trajectories, the *controller* goes to the next step each time the trajectory manager sends the *finish* signal. In case of failure, the trajectory manager sends an *error* signal to indicate that some messages sent by the referee must be taken into account. In case of timeout (i.e., the player keeps the ball during too much time) or if the opponent has scored, the *controller* restarts the sequence by moving to the ball position. If the match is finished, the *done* signal is consumed and the state machine goes to the *End* state.

The state machine of the trajectory manager (*tm*) is illustrated in Figure 3. Multiple kinds of trajectories are provided:

- Turn towards the ball
- Turn towards the direction given by a point at (`x`, `z`) coordinates
- Go to a position specified with (`x`, `z`) coordinates
- Go to the ball position and aspire it.

When the trajectory asked by the *controller* is finished, a *finish* signal is sent to it.

The first three trajectories finish when the target direction or position has been reached. The last trajectory, used to take the ball, is a bit more complex. This trajectory is a combination of both trajectories 1 and 3 to move simultaneously the robot to the ball position and to orient it towards the ball. When the robot is close to the ball, the suction is activated to aspire the ball while the robot continues its movement. The trajectory is considered complete when the ball has been taken (i.e., the *possession* signal has been received).

To control the robot, the state machine of the trajectory manager begins by sending requests to the environment to get new values of sensors (e.g., GPS and compass). A signal *checkReferee* is also sent to the referee to know if some interesting events (e.g., *timeout*, *done*) have occurred. When all information have been received, the trajectory manager checks which trajectory has to be made for computing new commands to apply on motors. This step is repeated until the trajectory finishes or an *abort* signal is received. In the latter case, the trajectory is interrupted and an *error* signal is sent to the controller.

To control movements of the robot, the trajectory manager used P-controllers. The control algorithm is divided in two parts: one for displacements of the robot using `x` and `z` coordinates, and the other one for the orientation of the robot.

For displacements, we used one P-controller for moving the robot forward or backward (with the *moveForward* signal), and another one for moving the robot left or right (with the *moveRight* signal). For the orientation, only one P-controller is used to make rotate the robot (with the *spin* signal). For all these controllers, we use only a proportional component (P) because the robot behavior is almost perfect and it also simplifies the verification task.

IV. Analysis Activities

To verify the behavior of this UML model, we design an abstracted environment that can be connected to the system. Using this setup, we have applied multiple analysis activities to increase the confidence in our model.

A. Design of an Abstracted Environment Model

To analyze our UML design model, we need an abstraction of the environment to close the system for the verification step. For this purpose, we have designed an abstracted environment composed of a *player* and a *referee*. The *Main* composite structure with this environment model is represented in Figure 4. The environment model used for model-checking usually needs to make some abstractions to avoid state-space explosion. These abstractions can be used to refine the model and reduce the number of possible scenarios. For this abstracted environment model, our goal is to focus verification efforts on control flows. As a result, we will not verify the correctness of P-controllers (which are not at the heart of the challenge problem).

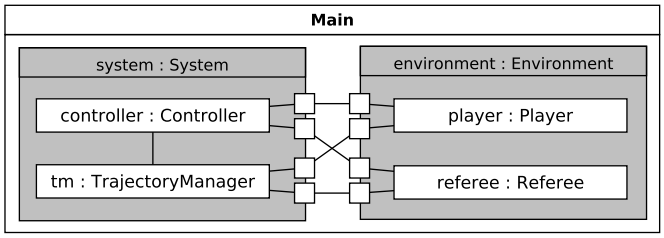

Figure 4: Composite structure diagram with the abstracted environment.

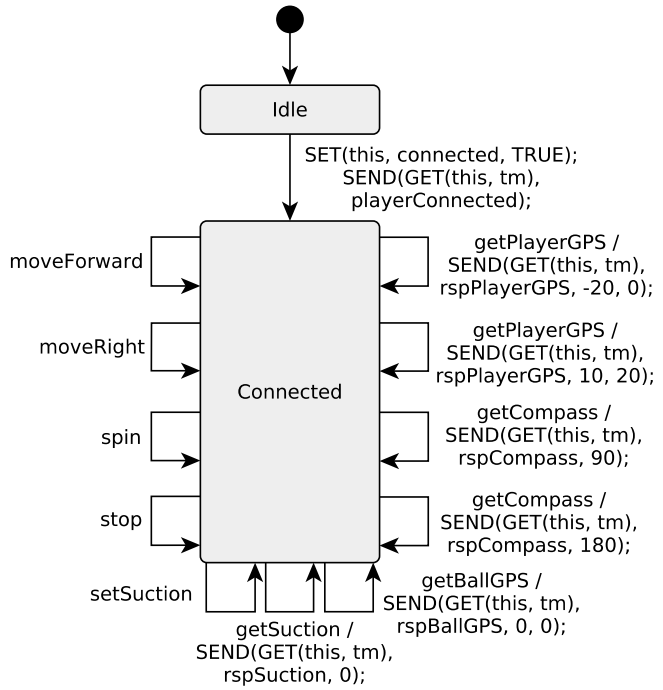

Figure 5: State machine of the abstracted Player.

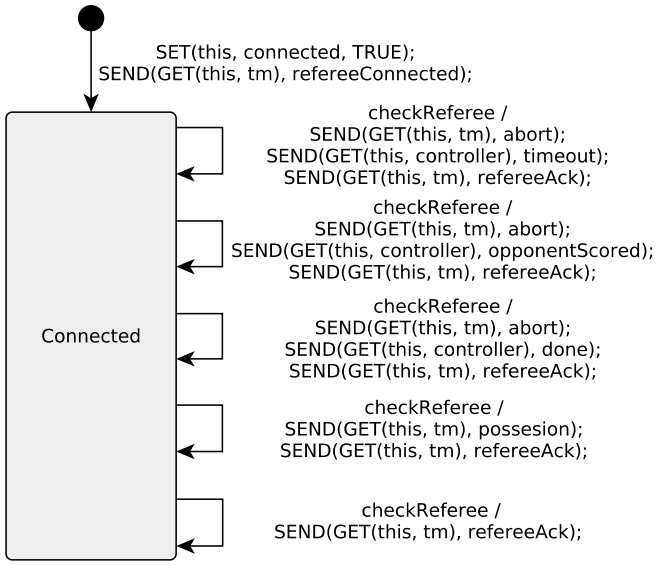

Figure 6: State machine of the abstracted Referee.

The *player* state machine for this abstracted environment is given in Figure 5. This state machine considers that the connection is always successful before reaching a *Connected* state. This state considers all events interleaving such that any received event can be consumed at any time. To render our environment more robust, we also consider that there is no links between sensors and actuators to consider a superset of all possible cases. For instance, no link is made between commands (e.g., *moveForward*) sent to the robot and its GPS position. Because our goal is not to verify control algorithms, signals sent to the system can take random parameters for position and orientation of the robot. The only important point is to detect that the robot is arrived on its shooting position and in the right shooting direction. For this purpose, we consider two cases for handling *getPlayerGPS* and *getCompass* signals: one case with the shooting position or direction, and one case with random values (different from the previous ones).

For the *referee* state machine in Figure 6, we use the same principle to consider all events interleaving. Here, only one kind of events can be received (*checkReferee*) but different responses are possible (e.g., *timeout*, *done*). All these possibilities have been taken into account in our model. In all cases, the *refereeAck* signal is sent to the system for acknowledgement.

Using this setup, we have a generic environment model in

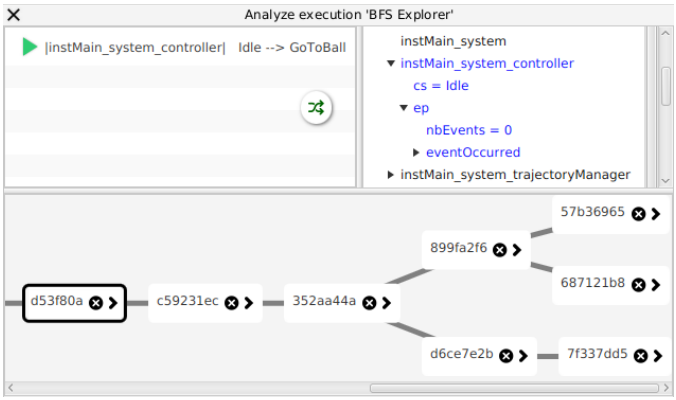

Figure 7: Simulation interface of the OBP2 tool.

terms of control flow as well as an abstracted representation of directions and GPS positions.

B. Analysis of the UML Model

With this abstracted environment model, we can now apply multiple V&V activities on our UML soccer player model. For this purpose, the UML model is loaded in EMI and the OBP2 model-checker is connected to it such that it can control and verify model execution. OBP2 is a explicit-state model-checker (still under development) that provides a modular infrastructure to verify models for various languages including UML.

Simulation. We can simulate our model using trace-based simulation facilities of OBP2 as shown in Figure 7. The simulator can be used to observe the current execution state and explore different execution traces. This is especially useful in early design phases to detect obvious errors or inconsistencies in the model.

Model-checking. OBP2 can also be used to verify formal properties through model-checking. For these experiments, we used an implementation of the "nested DFS" Büchi emptiness checking algorithm proposed by Gaiser and Schwoon in [GS09]. On this UML model, we have expressed eight properties about the system behavior. Each property has been specified in natural language and in LTL.

1) The player finally goes to the shooting position or aborts its action after having taken the ball.
   `"[] ((playerHasBall && goToBall) -> <> (goToGoal || listenReferee))"`
2) The player is never in the wrong direction when he shoots.
   `"[] !(inShootPos && !goalDirection)"`
3) If a timeout occurs, the player will finally move towards the ball after having listen to the referee.
   `"[] (timeout -> <> (listenReferee -> <> goToBall))"`
4) If the opponent scores, the player will finally move towards the ball after having listen to the referee.
   `"[] (opponentScored -> <> (listenReferee -> <> goToBall))"`
5) The controller finally ends after having received the *done* signal.
   `"[] (done -> <> end)"`
6) If the player is not in the *WaitConnection* state, both the player and the referee are connected.
   `"[] (notInWaitConnection -> (playerConnected && refereeConnected))"`
7) If the player tries to take the ball, it will finally succeed and send the *finish* signal to the *controller*, or it will continue indefinitely.
   `"[] ((ballTarget && !tmIdle && trajPos) -> <> ((finish && playerHasBall) || !finish))"`
8) If the player moves to a target position, it will finally reach this position and send the *finish* signal to the *controller*, or it will continue indefinitely.
   `"[] ((!ballTarget && !tmIdle && trajPos) -> <> ((finish && atTargetPos) || !finish))"`

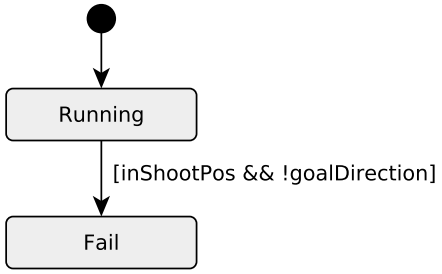

Figure 8: State machine of the UML Observer Automaton for property 2.

These properties have been expressed in LTL by linking atomic propositions (e.g., `playerHasBall`, `goToBall`) with LTL operators: not (`!`), and (`&&`), or (`||`), globally (`[]`), and eventually (`<>`). Atomic propositions are boolean expressions that depends on objects of the model. These predicates are expressed using our C action language (also used to describe effects and guards of state machine transitions). Atomic propositions will not be further details in this paper but description of these predicates can be found on our repository. All these LTL properties have been verified with OBP2 on the model state-space composed of 16,844 configurations linked together with 31,370 transitions. For comparison purposes, the state-space exploration on a desktop computer (8 CPU cores 4 GHz, 16 GB RAM) running a Linux OS takes in average 4.3 seconds and 28 MB of memory.

Monitoring. Our model interpreter also provides the possibility to monitor safety properties at runtime using observer automata. Among the eight formal properties verified through model-checking, two of them are safety properties (properties 2 and 6). They can be encoded into observer automata designed with UML state machines. These observer automata can be deployed with EMI on the actual system to make runtime monitoring. For this purpose, these observer automata are synchronously composed with the system all along model execution. If monitoring can not prove the absence of an error, this activity is complementary to formal verification. Even if model-checking performs an exhaustive verification, it may miss some real execution cases due to incorrect environment

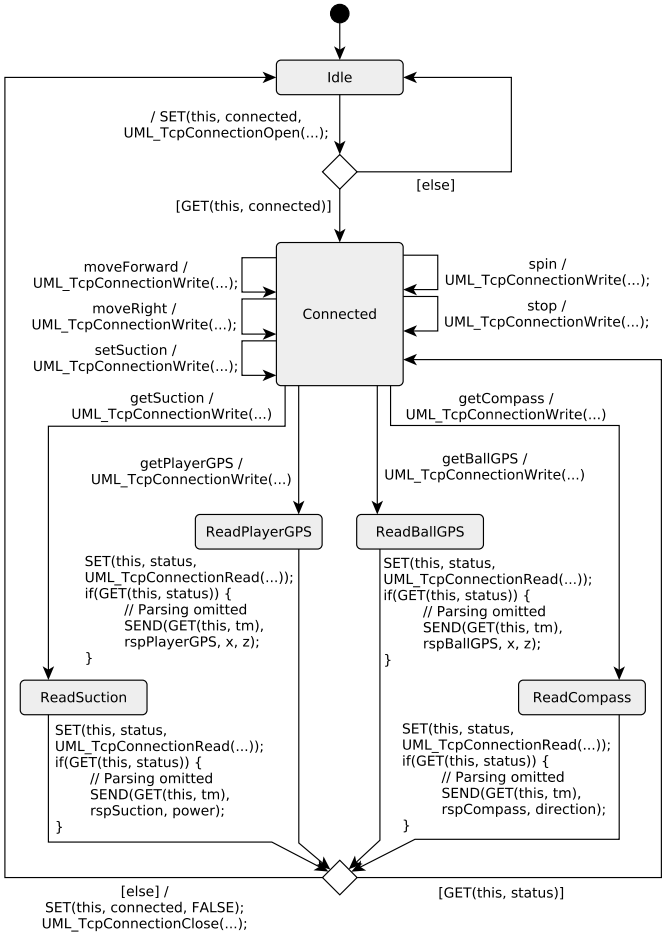

Figure 9: State machine of the concrete Player.

abstractions. Such failures can be detected at runtime through monitoring.

As an example, the observer automaton corresponding to property 2 is illustrated in Figure 8. The transition guard of this automaton is expressed with the same atomic propositions as the LTL property. If a failure is detected, the automaton reaches the *Fail* state. In this case, the problem can be notify to the user (e.g., by printing an error message in logs) or it can trigger appropriate error recovery mechanisms.

Therefore, all these activities contribute to improve our UML model and get more confidence in its behavior.

V. Execution of the UML Model with EMI

To execute the UML model with the actual robot, we design a concrete environment model to connect our UML model to the soccer simulator.

A. Design of a Concrete Environment Model

With our modular UML model, we only need to replace the environment component to connect our UML model to the soccer simulator provided by the MDETools contest. For this purpose, we design a concrete environment model that aims at communicating with both the controlled player and

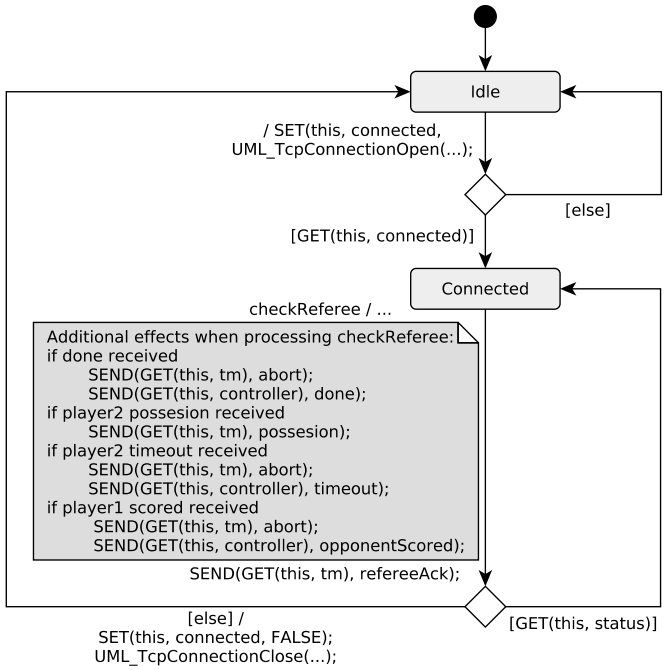

Figure 10: State machine of the concrete Referee.

the referee using TCP connections. The application layer used is a text-based protocol described by the MDETools challenge.

Our concrete environment model is composed of two objects: a *player* and a *referee*. In this case study, environment models have the same structure (Figure 4), only state machines are different. The *player* state machine is given in Figure 9. The first step is to open the connection with the robot. When the connection is made, the state machine reaches the *Connected* state. In this state, we can send requests to actuators (e.g., *moveForward*, *moveRight*) or to sensors (e.g., *getPlayerGPS*, *getCompass*). In the latter case, we need to wait a response of the soccer simulator to be able to send a signal to the system with the returned values. To communicate via TCP, we use the Application Programming Interface (API) of EMI that provides the following functions:

- `UML_TcpConnectionOpen` to open the connection
- `UML_TcpConnectionWrite` to write data
- `UML_TcpConnectionRead` to read data
- `UML_TcpConnectionClose` to close the connection.

For sake of simplicity, parameters of these functions and parsing of TCP responses have been omitted in Figure 9.

The state machine of the *referee* is illustrated in Figure 10. In the same way, the *Connected* state is reached when the connection is established. Contrary to the player, the referee port is used by the soccer simulator only to send notifications. For this reason, this TCP connection is not blocking and used only when needed. When the *checkReferee* signal is received, we check if some data have been received. If so, the received string is parsed and signals corresponding to useful notifications are sent to the system. In all cases, the state machine sends the *refereeAck* signal for acknowledgement when

processing is finished. The parsing has also been omitted on this state machine but a UML note explains which notifications are taken into account.

B. Execution with the Soccer Simulator

Once the concrete environment model designed, the UML model has been deployed with EMI on a host computer running a Linux OS. The execution with the actual environment achieves connection to the soccer simulator. The model successfully controls movements of the soccer player as well as suction and shooting of the ball. The controlled player scores in average 10 goals per match when the opponent stays in its initial position. The monitor of property 2 has also been embed with EMI at runtime. No failure has been detected on all execution traces performed until now. This means that for all shots achieved, the robot was oriented in the right direction.

VI. Discussions

In this section, we discuss strengths and limitations of our approach, as well as further possible improvements of our model.

A. Strengths

Our approach provides several benefits for executing and verifying UML models. The verification is directly applied on the design model with the same implementation of the language semantics as the one used for actual execution. Therefore, the behavior of the model that has been analyzed during the verification step is the same as the one used at runtime. Even if our interpreter has a bug, if verification tools ensure that system requirements are verified, runtime execution will also satisfy these requirements.

Due to the fact that our approach is transformation-free, analysis results are directly expressed in terms of design concepts which facilitate their understanding by engineers.

Another advantage is that our approach can be used to apply formal verification tools on models designed with semi-formal languages like UML. Indeed, the operational semantics of our model interpreter is used as the reference. For instance, regarding semantic variation points, the interpreter implementation make choices that are used for both verification and execution.

The concept of modular UML model provides facilities for designing models. Only the environment component need to be replaced to apply verification activities or to execute the system in its actual environment.

Our approach is well-suited for designing UML models of embedded systems with state machines. Our C action language enriched with macros enables to access UML objects as well as low-level peripherals through the EMI API (e.g., TCP functions).

B. Limitations.

Our model interpreter has also drawbacks that limit its usability. One limitation is that our model interpreter only supports a subset of UML. For instance, EMI does not provide execution support for hierarchical states that would have been useful to better define state machines, or reference attributes, which would have been helpful to store socket references rather than ids. Currently, only integer and boolean attributes are managed.

Another drawback is that our tool handles neither timing nor real-time constraints. To verify such constraints, a symbolic representation of time is needed to be able to apply model-checking but this is currently not supported.

The design of EMI focuses on the ability to execute UML models of embedded systems and the possibility to deploy them on bare-metal [BBJ+18b]. For this purpose, EMI does not use threads, which are usually provided by the OS layer. This can be seen as a drawback for this case study because we are not able to listen TCP sockets all the time. Instead of threads, EMI relies on cooperative scheduling such that each object can execute a step each time the scheduler asks it. The order on which steps are executed depends on the scheduling policy.

In terms of model-checking, we only focus on verifying the control flow but it would have been interesting to also verify control algorithms. Some additional refinements may be needed to take that into account. However, this is in general a difficult task that requires the help of domain experts to define appropriate abstractions. These abstractions should be sufficiently generic to cover all possible cases and sufficiently specific to avoid state-space explosion.

C. Experience Feedback

From our experience, the design of the first prototype of the UML soccer player takes a couple of hours. This prototype was a simple model, without the modular architecture, that has been used just for design assessment. Then, we have implemented a second version of the model with the modular model architecture and the two environment models. During the design process, we have also tried different kinds of trajectories and different optimizations (e.g., shooting distance, moving speed). Therefore, we spend approximately one week to get the final version of the model presented in this paper.

Model simulation has been used quite early in the development process to analyze the model behavior during the design phase. This has been useful to identify obvious design or programming mistakes in our model. To analyze the model behavior in details, we have expressed different system requirements and verify them via model-checking. It results that several LTL properties were not verified at the beginning. The analysis of counterexamples has been useful for improving our model. Indeed, in some very intricate cases, we notice that some event pools were full because some useless events have not been consumed. Hence, interesting events that come after could not be added to these event pools. As a result, some deadlocks or bad behaviors occurred.

Therefore, the use of EMI for analyzing this model has been beneficial to detect design faults and improve our UML soccer player model.

D. Improvements of the Model.

Our model has been successfully verified and executed with EMI but we consider here some improvements that can be made. It would have been interesting to implement a pathfind algorithm for computing trajectories that avoid the opponent robot. This could prevent some potential blocking situations and increase the number of goals. Another idea is to enhance the high-level strategy of our model to take into account several shooting positions and to reduce displacements. Finally, due to the fact that EMI does not support timing constraints, our model does not handle the ball possession rule. This rule mentions that a player cannot keep possession of the ball during too much time otherwise he will receive a penalty.

VII. Related Work

Our model interpreter focuses on analyzing and executing UML models but other tools provide such capabilities.

Previous editions of the MDETools challenge problem show interesting solutions (e.g., [HM18], [LA18]) based on various languages and technologies. Among them, ThingML [HFMH16] is a textual modeling language based on a subset of UML with a first-class action language. ThingML also provides the possibility to integrate platform-specific code or to use existing librairies and frameworks. ThingML models can then be transformed into different programming languages including C and Go. Another tool, called Umple, has been used in the last MDETools challenge problem. Umple [oO] is a textual modeling language based on a subset of UML concepts as well as other pinciples like mixins, traits, aspects, mixsets, and filters. It can also incorporate programming-language code like Java or C++. An Umple model is transformed into executable code using the Umple compiler.

More focused on UML, different kinds of tools can be used to perform model execution. Rhapsody [GHP02] and Rational Software Architect [LNH06] are modeling tools that can be used to design UML models. Model execution, simulation, and debugging can be applied on resulting models using code generation capabilities. Moka [RDCT18] and Moliz [ML12] are two model interpreters of fUML [OMG17a] models. Such tools can be integrated with Papyrus [LTE+09] and provide execution, simulation, debugging, and testing facilities. GUML [CSMB12] and UniComp [Cic18] are two model compilers that can be used to compile directly UML models into efficient executable code. These compilers can perform optimizations at model-level and enforce predictability of the resulting executable code. More tools dedicated to the execution of UML models can be found in the following literature review [CMS18]. This work defines a classification framework to characterize research studies based on UML model execution. Among its findings, this review notices that very few tools support model-level debugging and the use of formal specification languages. However, model simulation is supported by more than half of the selected approaches, which suggests that model execution is valuable during early design phases.

Other tools provide more capabilities to analyze model execution. GEMOC Studio [BDV+16] is both a language and a modeling framework that can be used to design domain-specific languages and models conforming to these languages. Models can be executed with execution engines [DDC+14], [MLWK13], [VLDCM15] and analyzed with V&V tools (e.g., trace execution manager, omniscient debugger, graphical animator). Mbeddr [VRSK12] is another interesting initiative for designing embedded systems using a set of integrated and extensible languages. The design model can be transformed into a formal model for being verified with a model-checker, and into C executable code for being executed on actual systems. Mbeddr also provides other analysis facilities including testing or debugging. More research studies have been identified and evaluated in the systematic review about model execution tracing in [HMZ+19]. Among the approaches under evaluation, 36% were based on UML models. This review shows that traces are used for different V&V activities including debugging, testing, model-checking, and semantic differencing.

In comparison to all these tools, our approach enables to apply model-checking directly on the design model with the same implementation of the language semantics than the one used for model execution. As a result, our model interpreter unifies verification and execution of UML models.

VIII. Conclusion

In this paper, we have proposed a solution to the MDETools'19 challenge problem by verifying and executing a UML model of a robot soccer player with EMI.

This model has been designed as a modular UML model such that different environment models can be connected to the system for different purposes. For this work, we have designed two environment models: an abstracted environment model used to close the system for analysis activities, and a concrete environment model that implements TCP connections for connecting our model to the soccer simulator.

Our UML design model has been analyzed through several V&V activities. For this purpose, the OBP2 model-checker has been connected to EMI to control model execution and reuse the operational semantics of our model interpreter. Using this setup, we have performed trace-based simulation to explore different execution traces and scenarios. We have also expressed eight system requirements as LTL properties and verified them with model-checking. Even if model-checking is based on an exhaustive verification of the model state-space, some real execution cases could have been missed due to incorrect environment abstraction. To exploit the complementarity between monitoring and model-checking, we expressed one safety property into a UML observer automaton. This automaton has been deployed with the model interpreter to perform runtime monitoring.

To execute our UML model with the actual soccer simulator, we replace the environment component with a concrete model that implements the two TCP connections (player and referee). The deployment of this model show successful control of the robot with reasonably good performances.

In this paper, we show that our tool provides some benefits for model verification but its has also some limitations mainly due to the limited UML subset supported. Indeed, our model interpreter can only handle primitive attributes and time management is not supported. Therefore, the ball possession rule could not be implemented.

We also believe that several improvements can be achieved on our model to enhance its quality and its performances. Among them, the implementation of a pathfind algorithm could be useful to choose optimized trajectories and avoid the opponent. The abstracted environment model may also be improved to take in consideration control algorithms with additional assumptions for refining this model.

To improve our model interpreter, we plan to support more UML concepts to increase the UML coverage of our tool. Further work also includes to evaluate the resource overhead of our tool in comparison with other model execution engines.

ACKNOWLEDGMENT

This work is partially funded by Davidson Consulting. The authors especially thank David Olivier for his advice and industrial feedback.

REFERENCES

[BBD+17] Valentin Besnard, Matthias Brun, Philippe Dhaussy, Frédéric Jouault, David Olivier, and Ciprian Teodorov. Towards one Model Interpreter for Both Design and Deployment. In *3rd International Workshop on Executable Modeling (EXE 2017)*, Austin, United States, September 2017.

[BBJ+18a] Valentin Besnard, Matthias Brun, Frédéric Jouault, Ciprian Teodorov, and Philippe Dhaussy. Embedded UML Model Execution to Bridge the Gap Between Design and Runtime. In *MDE@DeRun 2018 : First International Workshop on Model-Driven Engineering for Design-Runtime Interaction in Complex Systems*, Toulouse, France, June 2018.

[BBJ+18b] Valentin Besnard, Matthias Brun, Frédéric Jouault, Ciprian Teodorov, and Philippe Dhaussy. Unified LTL Verification and Embedded Execution of UML Models. In *ACM/IEEE 21th International Conference on Model Driven Engineering Languages and Systems (MODELS '18)*, Copenhagen, Denmark, October 2018. doi:10.1145/3239372.3239395.

[BDV+16] Erwan Bousse, Thomas Degueule, Didier Vojtisek, Tanja Mayerhofer, Julien Deantoni, and Benoit Combemale. Execution Framework of the GEMOC Studio (Tool Demo). In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2016, pages 84–89, New York, NY, USA, 2016. ACM. doi:10.1145/2997364.2997384.

[BTJ+19] Valentin Besnard, Ciprian Teodorov, Frédéric Jouault, Matthias Brun, and Philippe Dhaussy. Verifying and Monitoring UML Models with Observer Automata. In *ACM/IEEE 22th International Conference on Model Driven Engineering Languages and Systems (MODELS '19)*, Munich, Germany, September 2019.

[Cic18] Federico Ciccozzi. Unicomp: A Semantics-aware Model Compiler for Optimised Predictable Software. In *Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results*, ICSE-NIER '18, pages 41–44, New York, NY, USA, 2018. ACM. doi:10.1145/3183399.3183406.

[CMS18] Federico Ciccozzi, Ivano Malavolta, and Bran Selic. Execution of UML models: a systematic review of research and practice. *Software & Systems Modeling*, April 2018. doi:10.1007/s10270-018-0675-4.

[CSMB12] Asma Charfi Smaoui, Chokri Mraidha, and Pierre Boulet. An Optimized Compilation of UML State Machines. In *ISORC - 15th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, Shenzhen, China, April 2012.

[DDC+14] Julien Deantoni, Papa Issa Diallo, Joël Champeau, Benoit Combemale, and Ciprian Teodorov. Operational Semantics of the Model of Concurrency and Communication Language. Research Report RR-8584, INRIA, September 2014.

[GHP02] Eran Gery, David Harel, and Eldad Palachi. Rhapsody: A Complete Life-Cycle Model-Based Development System. In Michael Butler, Luigia Petre, and Kaisa Sere, editors, *Integrated Formal Methods*, pages 1–10, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[GS09] Andreas Gaiser and Stefan Schwoon. Comparison of Algorithms for Checking Emptiness on Büchi Automata. In Petr Hlinený, Václav Matyáš, and Tomáš Vojnar, editors, *Annual Doctoral Workshop on Mathematical and Engineering Methods in Computer Science (MEMICS'09)*, volume 13 of *OpenAccess Series in Informatics (OASIcs)*, pages 18–26, Dagstuhl, Germany, 2009. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/DROPS.MEMICS.2009.2349.

[HFMH16] Nicolas Harrand, Franck Fleurey, Brice Morin, and Knut Eilif Husa. ThingML: A Language and Code Generation Framework for Heterogeneous Targets. In *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*, MODELS '16, pages 125–135, New York, NY, USA, 2016. ACM. doi:10.1145/2976767.2976812.

[HM18] Jakob Høgenes and Brice Morin. Implementing the MDETools' 18 challenge with ThingML. In *MODELS Workshops*, pages 346–355, 2018.

[HMZ+19] Fazilat Hojaji, Tanja Mayerhofer, Bahman Zamani, Abdelwahab Hamou-Lhadj, and Erwan Bousse. Model Execution Tracing: A Systematic Mapping Study. *Software & Systems Modeling*, February 2019. doi:10.1007/s10270-019-00724-1.

[LA18] Timothy Lethbridge and Abdulaziz Algablan. Applying Umple to the rover control challenge problem: A case study in model-driven engineering. In *MODELS Workshops*, pages 386–395, 2018.

[LNH06] Daniel Leroux, Martin Nally, and Kenneth Hussey. Rational software architect: A tool for domain-specific modeling. *IBM systems journal*, 45(3):555–568, 2006.

[LTE+09] Agnes Lanusse, Yann Tanguy, Huascar Espinoza, Chokri Mraidha, Sebastien Gerard, Patrick Tessier, Remi Schnekenburger, Hubert Dubois, and François Terrier. Papyrus UML: an open source toolset for MDA. In *Proceedings of the Fifth European Conference on Model-Driven Architecture Foundations and Applications (ECMDA-FA 2009)*, pages 1–4, 2009.

[ML12] Tanja Mayerhofer and Philip Langer. Moliz: A Model Execution Framework for UML Models. In *Proceedings of the 2nd International Master Class on Model-Driven Engineering: Modeling Wizards*, MW '12, pages 3:1–3:2, New York, NY, USA, 2012. ACM. doi:10.1145/2448076.2448079.

[MLWK13] Tanja Mayerhofer, Philip Langer, Manuel Wimmer, and Gerti Kappel. xMOF: Executable DSMLs Based on fUML. In Martin Erwig, Richard F. Paige, and Eric Van Wyk, editors, *Software Language Engineering*, pages 56–75, Cham, 2013. Springer International Publishing. doi:10.1007/978-3-319-02654-1_4.

[OMG17a] OMG. Semantics of a Foundational Subset for Executable UML Models, October 2017. URL: https://www.omg.org/spec/FUML/1.3/PDF.

[OMG17b] OMG. Unified Modeling Language, December 2017. URL: https://www.omg.org/spec/UML/2.5.1/PDF.

[oO] University of Ottawa. Umple Home Page. URL: http://www.umple.org.

[RDCT18] Sebastien Revol, Géry Delog, Arnaud Cuccurru, and Jérémie Tatibouët. Papyrus: Moka Overview, 2018. URL: https://wiki.eclipse.org/Papyrus/UserGuide/ModelExecution.

[TDLR17] Ciprian Teodorov, Philippe Dhaussy, and Luka Le Roux. Environment-driven reachability for timed systems. *International Journal on Software Tools for Technology Transfer*, 19(2):229–245, April 2017. doi:10.1007/s10009-015-0401-2.

[TLRDD16] Ciprian Teodorov, Luka Le Roux, Zoé Drey, and Philippe Dhaussy. Past-Free[ze] reachability analysis: reaching further with DAG-directed exhaustive state-space analysis. *Software Testing, Verification and Reliability*, 26(7):516–542, 2016. stvr.1611. doi:10.1002/stvr.1611.

[VLDCM15] Matias Ezequiel Vara Larsen, Julien Deantoni, Benoit Combemale, and Frédéric Mallet. A Behavioral Coordination Operator Language (BCOoL). In Timothy Lethbridge, Jordi Cabot, and Alexander Egyed, editors, *International Conference on Model Driven Engineering Languages and Systems (MODELS)*, number 18, page 462, Ottawa, Canada, September 2015. ACM.

[VRSK12] Markus Voelter, Daniel Ratiu, Bernhard Schaetz, and Bernd Kolb. Mbeddr: An Extensible C-based Programming Language and IDE for Embedded Systems. In *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity*, SPLASH '12, pages 121–140, New York, NY, USA, 2012. ACM. doi:10.1145/2384716.2384767.

[Wor19] MDETools Workshop. Challenge Problem, 2019. URL: https://mdetools.github.io/mdetools19/challengeproblem.html.