Module Interactions for Model-Driven Engineering of Complex Behaviour of Autonomous Robots

Vladimir Estivill-Castro
School of ICT / IIIS
Griffith University, Nathan Campus
Brisbane, Australia
Email: v.estivill-castro@griffith.edu.au

René Hexel
School of ICT / IIIS
Griffith University, Nathan Campus
Brisbane, Australia
Email: r.hexel@griffith.edu.au

***Abstract*—In this paper, we describe a model-driven engineering approach that enables the complete description, validation, verification and deployment of behaviour to autonomous robots, directly, and automatically from the models. This realises the promises and benefits of model-driven engineering, such as platform-independent development and behaviour traceability. However, such a top-down approach of modelling by finite-state machines and sub-machines creates a conceptual challenge to the behaviour designer due to the complex interaction of independent modules. Simply finding which modules are necessary for other modules can be a challenge. We also describe here our solution to this. Interestingly, our approach goes in the opposite direction of Object Oriented Software Engineering as currently represented by the Unified Modeling Language and corresponding software processes. That is, typically, the static models are derived first (and in particular class diagrams), while dynamic modelling follows later with behaviour diagrams and interactions diagrams. We actually start with the description of behaviour in finite state machines and we complement this by static information provided by logics that describe concepts and by our dependencies diagrams that show static dependencies between modules.**

***Index Terms*—Automation of Software Design and Implementation. Software Modeling. Model-Driven Engineering. Visual Modeling.**

I. Introduction

Model-driven engineering raises the level of abstraction in software engineering so that engineers no longer have to be concerned with programming language details or the specifics of execution platforms. We show here an approach where executable software is generated automatically from models. We show that we can easily adapt to new platforms and behaviour requirements and illustrate this with the development of the complex software that constitutes the RoboCup challenge. The Mi-Pal team, qualified for RoboCup-2011, uses this approach to compose the programs that constitute the behaviour and execute on the humanoid autonomous robot platform.

We aim at systems at higher levels of abstraction. Our first toolset for a higher level of abstraction are logics, and in particular logics that emulate common reasoning. We argue for logics that describe a context by iterative refinement and are natural and analogous to how humans describe a context, starting from the most general case, then proving extensions or refinements. Similarly, our second tool is behaviour captured by a hierarchy of finite state machines (FSMs). This enables iterative refinement, describing the most general behaviour, which is then refined by a finite state sub-machine (sub-FSM).

For this reason, we use models at different levels of abstraction. From a high-level, platform independent model, it is possible to generate a working program without manual intervention. We describe this approach but we focus here on the technologies and infrastructure to facilitate design, verification and validation of inter-module communication. Other research publications expand on the details and technologies that have enabled this approach. In particular, we have discussed [1][2] the advantages of using non-monotonic reasoning to express in logic what otherwise becomes laborious and error-prone in an imperative programming language. For example, sanity checks on the landmarks reported by a vision system significantly benefit from their abstraction into logic rules. In fact, logic and iterative refinement are common in expressing and describing a concept. The *off-side rule* in soccer is an example that starts with "Usually a player is not off-side" (a default situation); then progressively some exceptions are presented. For example, "Unless two opponent players are between [a player] and the opponents' goal line", but then exceptions of the exception continue, forming the definition [3].

Modelling by FSMs, where the labels for transitions can be statements in a logic that demand proof, has been contrasted with plain FSMs, Petri nets, and Behavior trees (relevant behaviour modelling techniques in software engineering) using the very prominent example of modelling the behaviour of a microwave oven [4]. Our approach produces smaller models, clarifies requirements and we can generate implementations for diverse platforms and programming languages, e.g., the same models can generate code in Java for a Lego Mindstorm (`www.youtube.com/watch?v=iEkCHqSfMco`) as well as C++ for a Nao (`www.youtube.com/watch?v=Dm3SP3q9_VE`). The modelling of a microwave is a classical example in the literature of software engineering [5][4] as well as model-checking [6, Page 39] as the safety feature of *disabling radiation when the door is open* is an analogous requirement to the famous case of faulty software on the Therac-25 radiation machine that caused harm to patients [7, Page 2].

We have illustrated [8] the power of non-monotonic logic to describe and complement the descriptions of Behavior

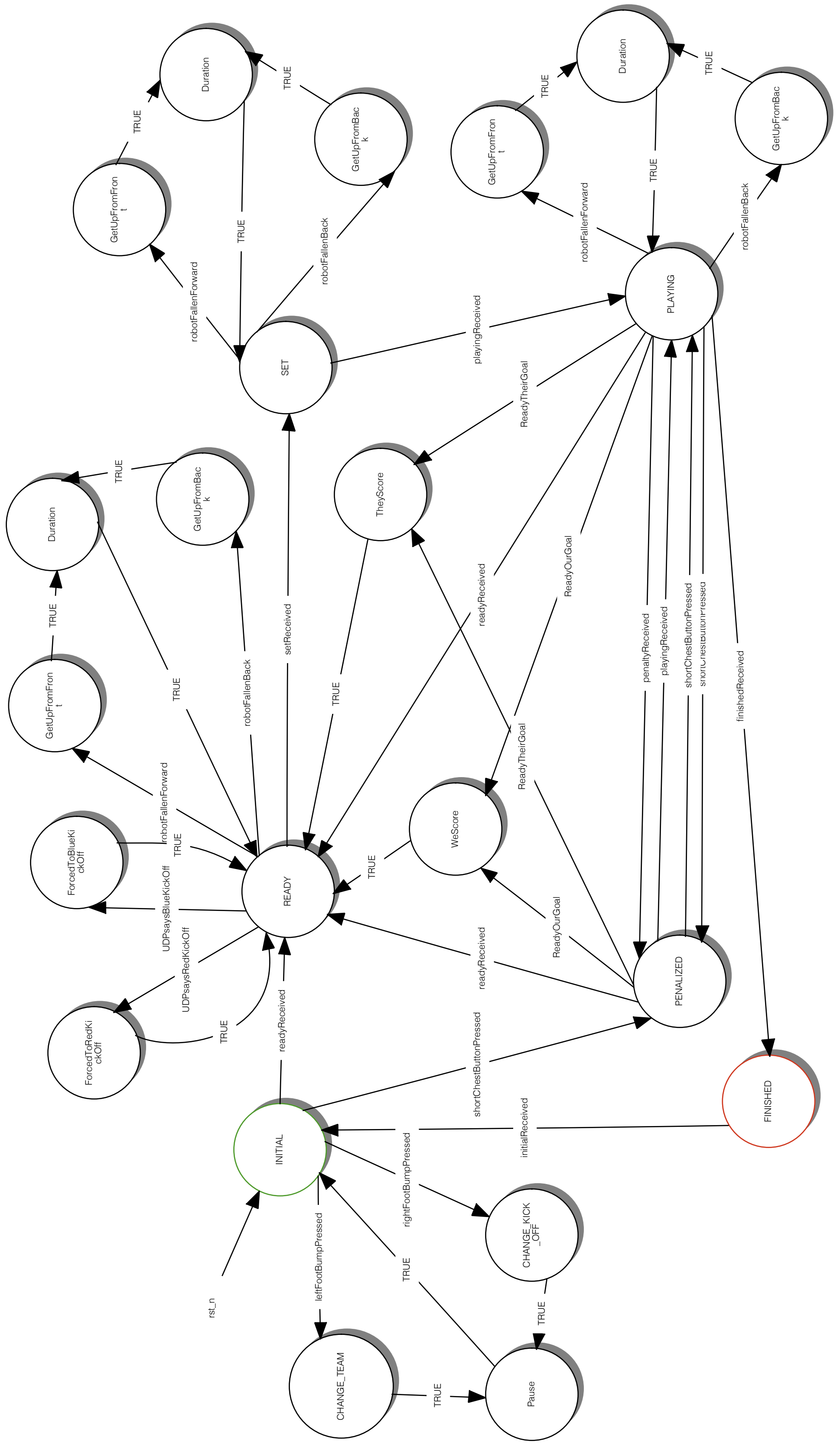


Fig. 1. The module `guGameController.fsm` that is interpreted on board the Nao's for participation in the SPL for RoboCup2011.

trees and of fine state machines for requirements engineering. Further illustrations [9] show the benefits of this idea in the context of embedded systems and robots. The software engineering architecture and the software design patterns that support our model-driven engineering are based on a whiteboard architecture [10][11]. This offers a cognitive architecture [12] or a *working memory* as well as a publisher-subscriber pattern for module communication, analogous to what others have called a repository architecture [5], or Data-Distribution Service [13]. Our whiteboard architecture is complementary to Aldebaran's inter-module communication and messaging architecture in the Red-Documentation.

Our interest for high-level modelling is that RoboCup, and in particular the Standard Platform League is an important benchmark for the deployment of legged robots in human environments (with RoboCup@Home also promoting this in the home or office). Therefore, there is a clear overlap with concerns in the field of software engineering, such as reliability, safety, human-computer interaction, requirements engineering, platform independence, composability, distribution, simplicity, and most importantly, model-driven engineering.

However, a challenging aspect of our approach is to model the interactions between modules, and to have a tool that enables the display of modules dependencies as behaviour designers integrate the behaviours of a complex system. Because we had shown an equivalence between FSMs and Behavior trees [8], we could translate our models to tools like BECCIE [14] that capture some of the module interactions, and this was sufficient for the already mentioned example of the micro-wave oven [9]. However, BECCIE's limitations do not enable this to scale further. Here we illustrate the new tools we have developed to achieve this.

The rest of this paper is structured as follows: Section II exemplifies the approach used. Section III shows how module interactions are modelled and what the consequences are for complex behaviour and iterative refinement. The paper is concluded with a discussion in Section V.

II. MODEL-DRIVEN ENGINEERING

We present a case study in the context of the SPL for RoboCup-2011 to illustrate our model-driven engineering approach, considering the FSM that playing robots are supposed to conform to. The model for this appears on page 7 of the SPL rules, and essentially indicates that the league's game controller would emit UDP packets (or a manual push of the chest button) for the playing robots to update their state. As in any requirements engineering scenario, the rules are under-specified and ambiguous – more seriously the actual SPL game controller (server) does not follow nor enforce the specified transitions. For example, Figure 2a and Figure 2b show the current activities for the state `INITIAL` and for the state `READY` (both corresponding to a state of the behaviour required by the competition). An `OnEntry` activity is to post (to the whiteboard) the message type `NaoMotionPlayer` (whose listener is `gunaomotion` with the message content `play get_up_anywhere`, which is a pre-loaded motion that stands up the Nao). Also in this state and also `OnEntry`, we post message type `LEDS` whose listener is `gunaoleds` to turn the ChestBoard off.

However, we do model our `guGameController` FSM for the behaviour executed by our robots for participation in RoboCup 2011. Figure 1 is produced with *Qfsm* (`qfsm.sf.net`), a graphical tool for designing FSMs. This produces XML files that our own tool, `qfsm2gu`, translates into to ASCII files. These files contain the *transition table* of the FSM and *activities* for each state of the automaton. Our FSMs are interpreted by our `gubehaviourinterpreter` module that, e.g., for `guGameController.fsm` reads the transition table from the file `TguGameController.txt` (transition files always start with the letter `T`), and the activities from `AguGameController.txt` (activity files start with `A`).

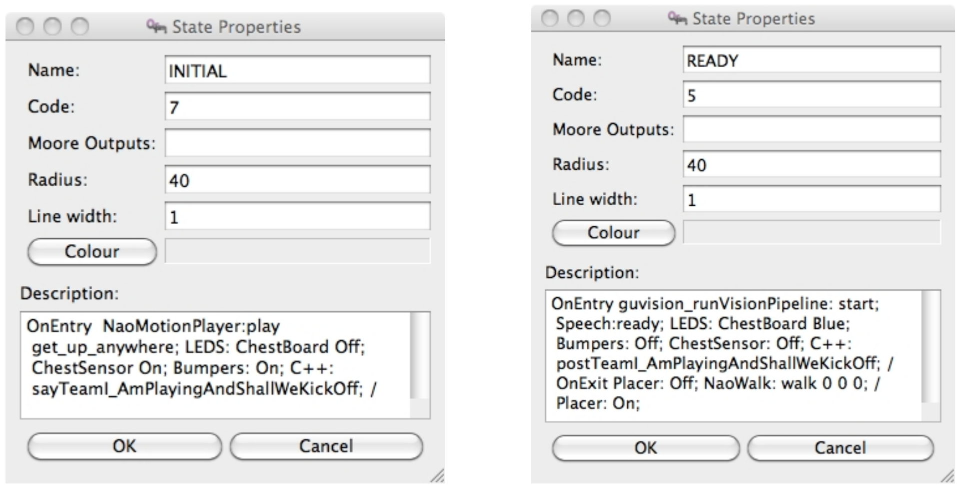
A. The semantics of our finite state machines

There are some important aspects of the interpreter of FSMs that represent behaviour. First, the transitions out of a state are not evaluated simultaneously, but they are evaluated in reverse order of their appearance in the transition file. Importantly, this liberates the behaviour designer of the concern of ensuring that only one transition can fire at any one time. In a sense, this provides a priority relation between the transitions and can be specified explicitly with *Qfsm* in the output field of a transition.

Second, the label of a transition is a query to an expert to make a proclamation about the truth value of that label. The interpreter will halt, waiting for a response on the *whiteboard* for this particular message type that indicates this proposition requires proof, typically by a logic inference engine – `gucdlmodule` that implements *Propositional Clausal Defeasible Logic* [15] (but, we have an implementation for standard prolog as well using `gnuprolog`). However, many times, the question is directly related to a sensor. That is, the best expert to ascertain the truth value of the transition label is a wrapper for a sensor (providing information about anything external to the system). For example, in the `guGameController.fsm` of Figure 1, a label `UDPSaysRedKickOff` is a query, but is answered by `guUDPreceiver`, which is the actual module connecting to the league's UDP server that can assert if the league's game controller is now broadcasting that the red team is to kick-off. There is a special label `TRUE` that always fires and causes a state transition.

It is important to highlight that the behaviour interpreter, the logic engine, and many of our modules are developed to conform to the POSIX standard (and therefore not only execute on the Nao but also, e.g., Linux, and MacOS). This enables module simulation, developing and testing independently of the platform. In particular, one can impersonate an expert by using our `testcdl` module and a FSM is oblivious to this.

We can use the example of `guGameController.fsm` to stress which of our modules provide the interface between the whiteboard and the Nao platform. In addition to `guUDPreceiver`, the following modules must run: `gunaobuttonsensor` for button-press events and



Name: INITIAL
Code: 7
Moore Outputs:
Radius: 40
Line width: 1
Description:
OnEntry NaoMotionPlayer:play get_up_anywhere; LEDS: ChestBoard Off; ChestSensor On; Bumpers: On; C++: sayTeamI_AmPlayingAndShallWeKickOff; /

Name: READY
Code: 5
Moore Outputs:
Radius: 40
Line width: 1
Description:
OnEntry guvision_runVisionPipeline: start; Speech:ready; LEDS: ChestBoard Blue; Bumpers: Off; ChestSensor: Off; C++: postTeamI_AmPlayingAndShallWeKickOff; / OnExit Placer: Off; NaoWalk: walk 0 0 0; / Placer: On;

(a) The activities of `INITIAL`. (b) The activities of `READY`.

Fig. 2. Display of activities in two states of `guGameController.fsm` using `qfsm`.

`gupositionsensor` to detect if (and which way) the robot has fallen.

Other modules are *actuators* that send a message or produce and effect on the environment external to the system. Actuators are typically subscribers through the whiteboard to postings by FSMs. It is important to understand that a state has essentially two types of *activities*, postings to the whiteboard or execution of some C++ code. The possibility to integrate C++ code means that any behaviour that we do not represent as FSMs can also be integrated into our modelling. The *activities* in our state machines are classified into three different execution steps (following very much the conventions of state machines for modelling Object Oriented Systems in OMT [16], UML, and may other standards for state machines).

- **On Entry:** These activities are executed at least once, and always just once and before any other activity upon arriving at the state.
- **On Exit:** These activities are executed at least once, and always just once and after any other activity upon leaving the state.
- **Internal activities:** These activities may not be executed at all. They are executed once, every time the entire set of leaving transitions has been tested (against the corresponding expert) and determined no transition fires.

Evaluation of leaving transitions and execution of internal activities is repeated until a transition fires that moves the machine to a new state.

Actuators that listen to messages posted by the `guGameController` state machine include the following.

- `gunaoleds`: The interface to illuminate Nao's ears, face, feet and the chest button.
- `gunaospeechmodule`: The robot speaks to identify itself.
- `gunaomotion`: The interface to actions like to get up if the robot is lying down.

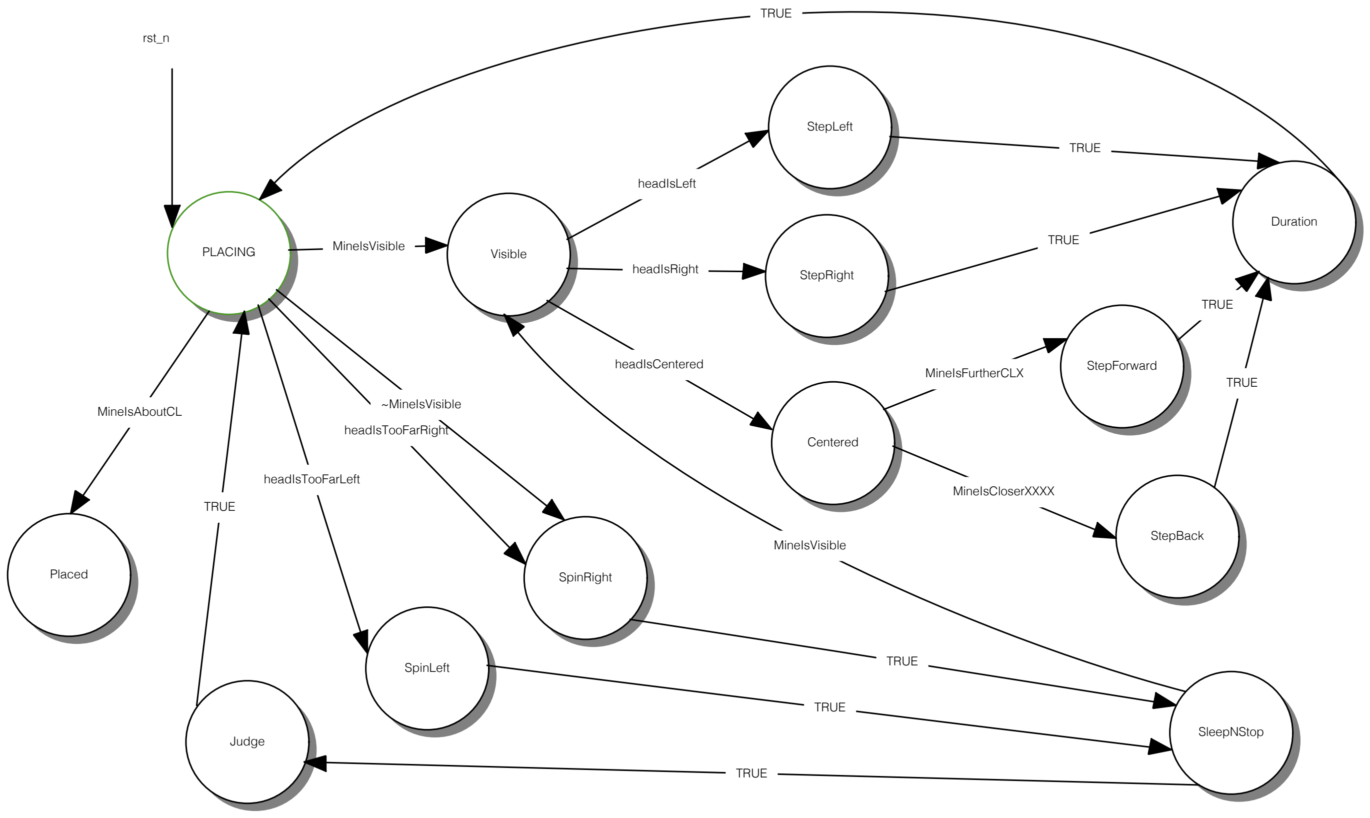
In the C++ code, there is a method named `sayTeamI_AmPlayingAndShallWeKickOff`. This routine uses C++ variables that record the integer number (player number) and the team as red or blue. This could also be modelled by states, but the state machine would basically be a clone of itself for playing red and for playing blue. Thus, this illustrates that sometimes clarity (and generality) is achieved with some algorithmic C++ code (rather than duplicating all the states). The values of the borrowed code are initially supplied on the command line but are updated by the `guGameController` state machine as the event from the league game controller demands via UDP.

B. The abstraction power of sub-machines

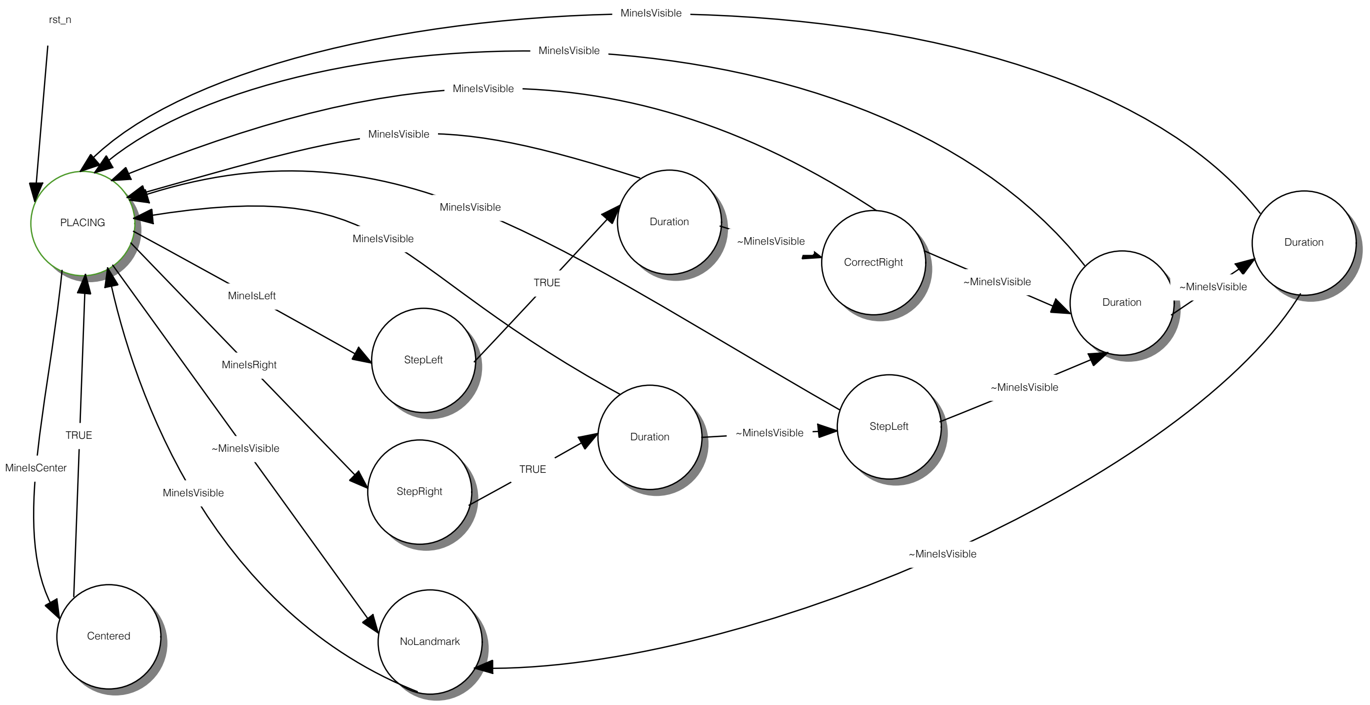
While everything that is required for the SPL in the `INITIAL` state is defined in the corresponding state of the `guGameController.fsm`, this is not the case for the `READY` state. There are many things that are done directly here in the `OnEntry` section, such as starting the vision pipeline in the module `gunaovision`. Also the ChestBoard LED is set to blue, and the sensitivity of the buttons is turned off (otherwise, the feet bumpers sense events just by walking).

So, how to achieve the behaviour that in state `READY` the robot is to find its correct position within the field before the state `SET`? The posting of the message with a type corresponding to the name of a sub-FSM starts a previously dormant automaton. In this case `Placer: On` (the message content is `On`). The `OnExit` activity is a posting of `Placer: Off` that makes this sub-machine dormant.

Sub-machines are a sub-class (in the C++ and object-oriented sense) of FSMs with the additional feature that they can be suspended or resumed. The `Placer.fsm` sub-machine (Fig. 3a) uses an implementation of a Kalman filer for localisation inside our module `gulocalizationfilter`. It uses walks from `gunaomotion` to walk until it is 150 cm from its own goal. The localisation module listens to he whiteboard for postings by `gunaovision` of landmark sightings (for its internal sensor model) and also to the `walk` commands for its internal motion model. `Placer.fsm` uses



(a) The `Placer.fsm` sub-machine.



(b) The `GoalTracker.fsm` sub-machine.

Fig. 3. *Qfsm* model of two `READY`-state sub-FSMs of `guGameController.fsm`.

`gunaosensor` (and `Naophysical`) to know if the head angle relative to the body is pointing straight or sideways (in order to post suitable commands to `motion`; the step to move in the direction of a landmark may be to walk forward or to walk to the side or even to spin). The `GoalTracker` uses only the `gulocalizationfilter` filtering to post commands to `gunaomotion` to turn the head in order to keep the target landmark in the centre of the vision frame.

III. Modelling Module Interactions

If the reader was able to remember what modules are required and which ones communicate with each other, even at this very top level of the behaviour of the robot, we would be surprised. Clearly, the description in the previous section needs some way to document, present, visualise, verify and validate the inter-module dependencies and the corresponding message passing and communication.

More importantly, SPL autonomous robots follow a design pattern that repeats what we have seen in the last forty years of desktop computer development. The hardware of the system and its Operating System becomes a commodity while the software on the system determines its ultimate behaviour. Here, we have a large number of sensors and actuators that can be utilised to navigate the robot's environment and to exhibit effective or intelligent behaviour. Sensor fusion needs to be performed to integrate (uncertain) input from numerous different input devices (such as buttons, cameras, etc.).

Therefore, while it can be argued that the static model of the robot remains the same throughout (and therefore conforms to traditional software engineering design processes as, for example, standardised in UML), this static structure is only marginally descriptive of the actual behaviour the robot is required to exhibit. In fact, a quite marginal change to behaviour can trigger a vastly different interaction pattern between modules or subsystems. For instance, switching from ultrasonic distance sensors to vision to detect nearby obstacles only requires a minute change to the corresponding behaviour state machine.

However, immediately, module dependencies and, consequently, the design of the overall static system structure changes. In a traditional approach, this would require a full redesign of the system and its composition.

To address this problem, we have created a tool that inspects a dictionary of strings that describes what messages a module is a listener for and what messages are posted. In reality, there are two types of paradigms for listeners. They may subscribe and wait for a message, and thus, as processes, such listeners are paused and then re-started by the whiteboard. The second module just queries the whiteboard if such a message of interest is present on the whiteboard and thus is non-blocking. Therefore, we also need to indicate the type of message, whether it is provided or required by that module (akin to inputs and outputs in UML composite structures [17]).

Moreover, many modules are platform independent and some modules may have alternatives. This is easily represented by specifying the same provisions and requirements. An example of an alternative module is a `guspeechmodule`. Such a module exists in two version, a MacOS version that generates speech from text on a laptop, and a Nao version that carries the same functionality but on a Nao robot. Even though these module need different compilers and run under different operating systems, from the perspective of supplier

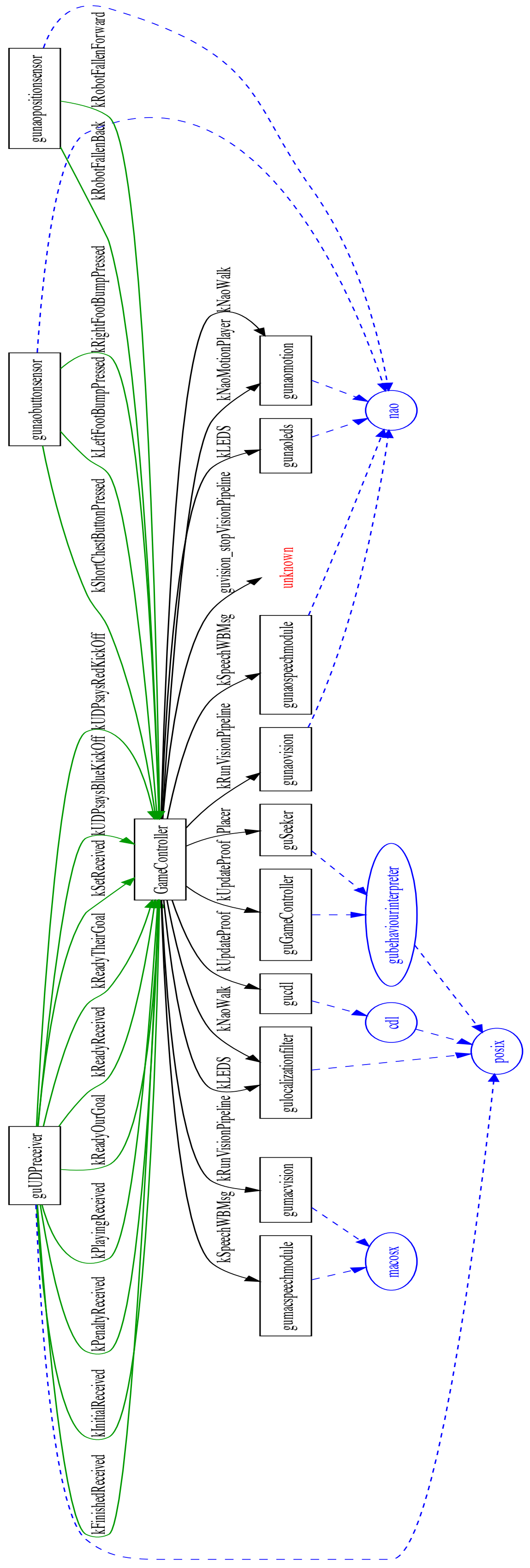


Fig. 4. The diagram that illustrates the module dependencies related to the module `guGameController.fsm` whose behaviour appears in Figure 1.

(provider) and consumer (listener) message types, the modules are completely analogous.

Our models can now be generated completely automatically from the dictionary of messages, which in turn is generated directly from the code (in fact it is part of the code). The result is a series of diagrams that show dependencies for each module as well as overall module dependencies. For example, Figure 4 shows the module concerned at the centre of the diagram. Those modules that are suppliers to `guGameController.fsm` are show in the upper row of modules, and the green arrows show that these are experts, queried in labels of the FSM, about the change of state. In this case, `guGameController.fsm` will be making blocking calls to these modules requesting they make a proclamation on a particular proposition (evaluating to `true` or `false`) that labels the transition. The bottom row of modules are those to whom the module `guGameController.fsm` will be posting a message. The black arrows indicate also this is a non-blocking interaction while the direction of the arrow indicates who is the provider of a message and who is the listener. In this illustration we have chosen a faulty version of a FSM that posts a message not recognised in the dictionary, i.e., no listening module has been found. Therefore, we see the word "unknown" in red as a destination of a message. This warns the behaviour designer that there is a fault in the current design of interactions of the software, at least with the respect of the behaviour specified by this FSM.

Discussion

What additional advantages besides the correctness of module interactions does this provide? The behaviour designer can now configure particular testing, verification and validation plans, and the corresponding script can be generated automatically. For example, by looking at the corresponding diagram for a module, and indicating associated modules, a particular script can be rapidly configured for testing the chosen module on a particular platform. The script will only start those modules necessary for interaction and support of the module under scrutiny, and therefore significant resources of compilation, porting to the platform, and test configuration are saved. Lets recall the importance of testing [5, Chapter 7] and in particular testing automation and early validation; the sooner we verify a change and test that we have not introduced a fault or broken the current functionality the better. This leads to more traceability, to more reliability and to more robustness in the software process and the product itself.

Why not use UML's collaboration diagrams or UML's sequence diagrams (or some other sort of UML interaction diagram)? Simply because such UML diagrams are used to model the dynamic behaviour of the system. They represent a particular trace of execution. The order and time of message passing is the principal aspect. Our FSMs are already the dynamic model. In fact, our proposed diagrams here represent static information; they are a static model of the software on-board of the robot. This is precisely why they are so useful in configuring versions and identifying the modules that together integrate a module. Thus, the diagrams here are in fact more analogous to UML composite structures [17]. In fact, it is trivial to convert the dependency information on whiteboard message suppliers and listeners to corresponding ports therein. However, this would not capture the fact that the responsibility for such compositions are factored out from the individual modules (as we already mentioned, our software architecture is actually a repository architecture in the terminology of Sommerville [5, Chapter 6] or whiteboard architecture [10][11]).

IV. Other Aspects

Some features in our approach that enable further powerful, high-level control on the behaviours for the robots are

1) to dynamically load a behaviour (a FSM) at any time and not only at start-up, and
2) to dynamically modify vision pipelines, so the camera feed (upper or lower camera) is adjusted, based on FSM context.

We mentioned that the FSMs (or sub-machines) that model our behaviour are in fact encoded as two tables: the transitions table and the actions table. The capability to read, parse and have an internal representation of the FSM is not only used at start-up time, but can be used on demand. In the example discussed earlier regarding the model of the Game Controller, the robot can, during execution, re-load the transition table from the file `TguGameController.txt` and the activities from `AguGameController.txt`. Once the corresponding parsing and internal representation are ready for the interpreter, this refreshed behaviour can take over. This parsing and re-building of the internal representation is not a CPU-intensive operation. The grammar of the transition table and the activities table is very straightforward and the internal representation is not particularly different from a graph representation of the FSM as the diagrams we have been displaying. Namely, our class `fsmMachine` that represents a behaviour model is a vector of `fsmStates`. An object of the class `fsmState` has a `stateID`, `stateName`, a vector of `fsmTransitions` and an `fsmActivity` object. An `fsmActivity` object has postings and/or callbacks for each of three possibilities: OnEntry, OnExit and Internal. An object of the class `fsmTransition` can hold an expression to evaluate.

Granted, this parsing must be combined with the facilities that enable sub-machines. That is, sub-machines can be paused (and therefore become dormant), and later be resumed from their initial state. Therefore, a dormant sub-machine can be re-loaded without the need to halt the whole robot. Moreover, re-loading a sub-machine can be part of a behaviour. Therefore, this opens the door to the possibility of the robot learning or adapting its behaviour while operating, by simply modifying the behaviour model during execution (however, such a learning behaviour is not implemented yet).

Once the concept of a model being able to be loaded during runtime and not only during start-up is available it is not difficult to see that a linear software architecture, such as a pipeline (also known as a pipe and filter architecture [5, Section 6.3.4]), can easily be modified and adapted with specific commands during runtime. This is what enables the second aspect mentioned above.

The advantages provided by these facilities are many. For example, they can be used as a powerful mechanism for a faster and more reliable software development cycle for the the robot (and, in general, for embedded systems). To illustrate this, it is enough to consider what the testing of a behaviour demands if the robot needs to be shut down every time a new behaviour is loaded. Typically, re-booting a robot such as the Nao is quite time consuming, and requires placing the robot in a safe position, e.g., to physically prevent the robot from falling. The boot process is slow, because it is not only the operating system that needs to be loaded, but also all the middleware that enables the hardware subsystems, and any other modules that the behaviour uses and that are part of the system as a whole. As we alluded earlier, in the case of playing robotic soccer, these include many modules for motion, vision, sonar, actuators, etc. In general, which modules are required for a behaviour is determined by our new diagrams illustrating module dependencies. Dynamically loading a behaviour (or a sub-behaviour as a sub-machine) enables iterative refinement and testing of new behaviour, without the lengthy delay of re-booting the robot for every single modification of the behaviour model. This facilities and speeds up the testing of every behaviour. The more a behaviour is tested, the more reliable it becomes.

V. Conclusion

We have described our model-driven engineering approach to software development. We can completely develop the behaviour of autonomous humanoids robots through models that consist of

1) models for logics that describe the domain knowledge and the declarative part of the system,
2) models for the action part of the system, that are visualised by finite state machines, or state diagrams.

However, understanding the interactions, the service available, and the request that will be made to service providers needs validation and visualisation. We have described the mechanisms to obtain such diagrams and the benefits they provide to software development.

Nevertheless, there are also some aspects of our infrastructure that constitute immediate targets for further work;

- to expand even further the vocabulary of messages the behaviour interpreter can use when requesting a proof so we can use other inference engines,
- to add priorities to the messages on the whiteboard, so we can have a subsumption architecture, and
- to add a planning module (so we can apply the infrastructure to other environments besides soccer, that demand more planning and are less reactive).

Acknowledgements

The authors would like to thank Andrew Rock and David Billington for fruitful discussions and collaboration in the conceptual idea of model-driven engineering of behaviour of autonomous robots.

The authors also thank Carl Lusty, Steven Kuok, and Vitor Bottazzi who helped significantly in the programming of many of the modules and tools used in the practical illustration of this approach, which is the large software environment and system that is the code for the RoboCup Standard Platform League.

References

[1] D. Billington, V. Estivill-Castro, R. Hexel, and A. Rock, "Non-monotonic reasoning for localisation in robocup," in *Australasian Conference on Robotics and Automation*, C. Sammut, Ed. Sydney: Australian Robotics and Automation Association, December 5-6 2005.

[2] ——, "Using temporal consistency to improve robot localisation," in *RoboCup 2006: Robot Soccer World Cup X*, ser. Lecture Notes in Computer Science, G. Lakemeyer, E. Sklar, D. G. Sorrenti, and T. Takahashi, Eds., vol. 4434. Springer, 2006, pp. 232–244.

[3] ——, "Chapter 3: Non-monotonic reasoning on board a sony AIBO," in *Robotic Soccer*, P. Lima, Ed. Vienna, Austria: I-Tech Education and Publishing, 2007, pp. 45–70.

[4] ——, "Non-monotonic reasoning for requirements engineering," in *Proceedings of the 5th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE)*. Athens, Greece: SciTePress (Portugal), 22-24 July 2010, pp. 68–77.

[5] I. Sommerville, *Software engineering (9th ed.)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2010.

[6] E. M. Clarke, O. Grumberg, and D. Peled, *Model checking*. MIT Press, 2001.

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

[8] D. Billington, V. Estivill-Castro, R. Hexel, and A. Rock, "Plausible logic facilitates engineering the behavior of autonomous robots," in *IASTED Conference on Software Engineering (SE 2010)*, R. Fox and W. Golubski, Eds. Anaheim, USA: ACTA Press, February 16 - 18 2010, pp. 41–48, location: Innsbruck, Austria.

[9] ——, "Modelling behaviour requirements for automatic interpretation, simulation and deployment," in *SIMPAR 2010 Second International Conference on Simulation, Modeling and Programming for Autonomous Robots*, ser. Lecture Notes in Computer Science, N. Ando, S. Balakirsky, T. Hemker, M. Reggiani, and O. von Stryk, Eds., vol. 6472. Darmstadt, Germany: Springer, November 15th-18th 2010, pp. 204–216.

[10] B. Hayes-Roth, "A blackboard architecture for control," in *Distributed Artificial Intelligence*, A. H. Bond and L. Gasser, Eds. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1988, pp. 505–540.

[11] D. Billington, V. Estivill-Castro, R. Hexel, and A. Rock, "Architecture for hybrid robotic behavior," in *Hybrid Artificial Intelligence Systems, 4th International Conference, HAIS 2009, Salamanca, Spain, June 10-12, 2009. Proceedings*, ser. Lecture Notes in Computer Science, E. Corchado, X. Wu, E. Oja, Á. Herrero, and B. Baruque, Eds., vol. 5572. Springer, 2009, pp. 145–156.

[12] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 2nd ed. Englewood Cliffs, NJ: Prentice-Hall, Inc., 2002.

[13] C. H. Wu, W. H. Ip, and C. Y. Chan, "Real-time distributed vision-based network system for logistics applications," *Int. J. Intell. Syst. Technol. Appl.*, vol. 6, pp. 309–322, March 2009. [Online]. Available: http://portal.acm.org/citation.cfm?id=1521389.1521397

[14] L. Wen and R. G. Dromey, "From requirements change to design change: A formal path," in *2nd International Conference on Software Engineering and Formal Methods (SEFM 2004)*. Beijing, China: IEEE Computer Society, 28-30 September 2004, pp. 104–113.

[15] D. Billington, "Propositional clausal defeasible logic," in *Proceedings of the 11th European conference on Logics in Artificial Intelligence*, ser. JELIA '08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 34–47. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-87803-2_5

[16] J. Rumbaugh, M. R. Blaha, W. Lorensen, F. Eddy, and W. Premerlani, *Object-Oriented Modelling and Design*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1991.

[17] *OMG Unified Modeling Language (OMG UML), Superstructure, V2.3*. Object Management Group, May 2010, ch. 9, Composite Structures, pp. 167–198.