Modelling Behaviour Requirements for Automatic Interpretation, Simulation and Deployment

David Billington, Vladimir Estivill-Castro, René Hexel, and Andrew Rock
Griffith University, Nathan 4111, QLD, Australia
<http://griffith.edu.au/mipal>

Abstract. In this paper we propose a high level approach to capture the behaviour of an autonomous robotic or embedded system. Using requirements engineering, we construct models of the behaviour where system activities are captured mainly by collaborating state machines while the domain knowledge is captured by a non-monotonic logic. We explain our infrastructure that enables interpretation, simulation, automatic deployment, and testing of the models, minimising the need for developers to code. The approach also minimises faults introduced in the software development cycle and ensures a large part of the software is independent of the particular robotic platform.

Keywords: requirements engineering, interpretation of models, knowledge representation.

1 Introduction

In many robotic platforms and embedded systems, the production of software is complicated by the fact that creating correct code for the final deployment environment is costly. This provides a strong incentive for simulation, as the software can be verified and/or tested and brought to required levels of quality without the significant costs and risks of using the actual platform.

We propose here that the production of software for robots should be taken one step further and follow the software engineering postulations of requirements engineering and behaviour engineering. Namely, if we capture the required behaviour into unambiguous and descriptive models, then faults are not introduced into the software in later phases of the software development cycle. Moreover, if verified models are directly translated or interpreted in simulation environments and/or the corresponding target platform, software faults will be non-existent. The simulation will enable direct validation, and the modelling will enable traceability all the way from requirements to coded behaviours. This not only improves the design of behaviours and the compliance with requirements, but it also produces far more maintainable software. This way, software models also become instruments to document the implementation and to assist in the usage or training of people interacting with such autonomous systems.

Finite state machines (and their variants) are ubiquitous in modelling the behaviour of single threaded embedded systems. They appear in one form or another to capture the states that computer-based systems incarnate and the transitions they perform. Finite state machines have been used in many software engineering methodologies, tools, and developing approaches and continue to be core to widely supported software modelling standards such as UML. Moreover,

the semantics of the model can be made unambiguous, they can be represented diagrammatically, and they can also be simulated and interpreted in accordance with their semantics. We propose here to use this approach to represent the activities, transitions and states of the different modules of the software for a robot. However, we enhance the model by allowing a richer language for the labelling of transitions. We propose a non-monotonic logic that enables gradual capturing of the requirements and design of the behaviour with smooth, iterative refinement. This high-level descriptive tool allows a declarative description of domain knowledge. Moreover, the reactive nature of state-machines is complemented by the reasoning nature of the inference engine of the logic. Therefore, our modelling enables the diversity of architectures of multi-agent systems.

However, a robotic system usually demands the simultaneous control and co-ordination of several modules. We propose to enable several finite state-machine processes or threads and to model their message passing using Behavior Trees [23]. Behavior Trees have been proven to be fruitful in many complex applications of embedded system as a tool for requirements engineering.

Our models can not only be simulated, they can also be executed on board the robots and be migrated across platforms. We use a publish/subscribe architecture for what we call a *whiteboard* [3] to facilitate the communication between the multi-treaded machines and also to isolate generic software components from particular hardware platforms. Because we can enable priorities in the whiteboard, we can also adopt subsumption architectures [7]. Moreover, it is possible to simulate none, a few, or all of the modules and execute none, some, or all of the rest aboard the robot. Our whiteboard can operate on a single system as well as across a computer network. Because of this distribution capability, it is possible to execute several modules of a system for simulation and even execute some of the others on one or several robots. We can create simulations in which not all sensors and actuators are on the same robot. A situation where this may come handy is when we want to, for example, test a cruise control on a remote vehicle or robot, while using a brake pedal that is locally placed in a laboratory.

The aim of this paper is to reverse the belief among some authors that automatic code generation and simulation with state diagrams is impossible.

“Many people understand automatic code synthesis of a program to solve a problem from a statement of the problem specification. State chart-based tools cannot provide this because a state chart is just a higher-level (mostly visual) solution rather than a statement of the problem.” [18]

By using state diagrams in conjunction with an inference engine (for reasoning about transitions) for requirements engineering, we can ensure the correctness of the fully automatic, deployed system derived directly from the state diagrams.

2 Non-monotonic logic in Finite State Machines

Our approach uses one fundamental modelling technique, namely finite state machines (FSMs).¹ FSMs are standard models of automata. In the early 90s,

¹ Our approach also has been applied with the asynchronous modelling provided by Petri nets, but this will not be described here.

state machines with a single thread of control became the instrument of choice to model the behaviour of objects. Rumbaugh’s Object-Modeling Methodology (OMT) [17, Chapter 5] established state diagrams as the primary dynamic model. The so-called Shlaer-Mellor approach established state models to capture the life cycle of objects of a given class [19]. They were also the instrument of choice to model reactive systems [11]. Today, besides being a part of UML, FSMs are used with Web technologies and can be defined in XML (the W3C has a working draft specification “State Chart XML (SCXML): State Machine Notation for Control Abstraction,” specifying semantics and reference implementations). Parallel FSMs and Petri nets enable the modelling of multi-threaded architectures including client-server architectures. Both formalisms are usually represented as diagrams when modelling software systems. We will add reasoning capability to the reactive nature of these modelling tools, resulting in a very expressive mechanism for robotic behaviour.

A FSM is, in fact, a formal mathematical object. It consists of a finite set of states, an input language (for events), and a transition function. The transition function indicates the new state, given an input and the current state. Other adornments include signalling some states as initial and some as final or accepting. Actions can be associated with entering, and/or leaving a state, as well as activities that are performed while in a state. This model is used in formal ways in the theory of computation for describing regular languages. It has historically been used to formally describe computer networking protocols [6]. As a software-modelling instrument, it remains the main artefact for modelling the behaviour² of each object in a class in UML and has penetration in many approaches for the description of object behaviour in object-oriented systems as well as in agent systems. This model has been extended to describe behaviours in groups of agents and also appears in the literature in Computer Supported Collaborative Work (CSCW), namely *Team Automata* [20, 10] (also fmt.isti.cnr.it/~mtbeek/TA.html). State machines have also been used as the foundation for languages to model agent behaviour in RoboCup [12]. In summary, FSMs remain a core tool for modelling software behaviour.

However, a fundamental aspect of FSMs is that the transition function is just that, a mathematical function (it provides only one value of the codomain for each value in the domain). That is, given an input and a state, only one transition fires and the machine moves to only one new state. Granted that the model can be extended to a non-deterministic automaton, where given an input and a state, a set of possible states is the outcome of the transition. However, in this case, the semantics of the behaviour has several interpretations. Therefore, as a modelling instrument in software engineering, it is typically expected that the conditions emanating from a state are mutually exclusive and exhaustive. Namely, if the symbol c_i is a Boolean expression representing the guard of the transition, then it must be the case that $\bigvee_{i=1}^n c_i = \text{true}$ (the exhaustive condition), and $c_j \wedge c_i = \text{false} \quad \forall j \neq i$ (the exclusivity condition). When a finite-state machine is used

² Conversely, the class diagram is the artefact to model the static structure of an object-oriented system.

for language recognition, if the automaton has a state where $\bigvee_{i=1}^n c_i \neq \text{true}$, it is said to be input-incomplete. Input-incomplete automata can be made input-complete by adding a transition from the state to itself labelled with $\neg \bigvee_{i=1}^n c_i$. When the number n of transitions out of a state is large, this may not be what the behaviour designer attempted to capture. While, formally, our logic will achieve something similar, the priority relation of the logic can better aid the system engineer in identifying the default cases.

Thus, the designer of behaviour faces the difficult challenge of describing and ensuring the validity of exclusive and exhaustive conditions that trigger the departure from each state. We automate such validation.³ We propose to use a non-monotonic logic to describe the conditions that trigger a change of state. Non-monotonic logic models are better suited for incremental elicitation of the conditions that leave a state and those that keep control in the same state [1]. Non-monotonic reasoning ensures that the behaviour has a single, exhaustive interpretation, even if the original description does not and requires common sense to avoid cumbersome, “obvious” flaws. We generalise the language that guards transitions from symbols of events to entire reasoning exercises expressed in logics. We emphasise that the modelling remains deterministic, but what rule fires is determined by logical deduction in a declarative model.

3 Modelling a Microwave Oven

We illustrate our approach with the requirements engineering of a microwave oven and its simulation in BECIE (a tool for behaviour engineering using Behavior Trees) and under Linux and Mac OS X. We also demonstrate its direct implementation on both a LEGO Mindstorm and an Aldebaran Nao.

The microwave oven has been a standard example in software engineering [13, 19, 21] and requirements engineering [14, 23, 9]. Table 1 shows the requirements as presented by Myers and Dromey [14, p. 27, Table 1]. Although this is in fact not exactly the same as the original by Shlaer and Mellor [19, p. 36], it illustrates the point that human specifications are ambiguous. We invite the reader to revise

Table 1: One-Minute Microwave Oven Requirements.

Req.	Description
R1	There is a single control button available for the use of the oven. If the oven is closed and you push the button, the oven will start cooking (that is, energise the power-tube) for one minute.
R2	If the button is pushed while the oven is cooking, it will cause the oven to cook for an extra minute.
R3	Pushing the button when the door is open has no effect.
R4	Whenever the oven is cooking or the door is open, the light in the oven will be on.
R5	Opening the door stops the cooking.
R6	Closing the door turns off the light. This is the normal idle state, prior to cooking when the user has placed food in the oven.
R7	If the oven times out, the light and the power-tube are turned off and then a beeper emits a warning beep to indicate that the cooking has finished.

these requirements, and because of the familiarity today with microwave ovens, most people would find Table 1 a complete and sufficient description of the

³ Validation here means that the design represents a well-defined, intended behaviour.

behaviour of such a system (please do not read further until you have reviewed the requirements again).

We take this opportunity to emphasise how humans reason in a non-monotonic fashion. In fact, most people find that R5 means *pausing* the cooking and although *opening the door stops the cooking*, we do not clear the timer. That is, closing the door resumes the cooking. This also implies that opening the door pauses the timer.

A distributed *whiteboard* [3] enables high level formalisation of requirements. Thus sensors, such as the microwave button, are hardware instruments that deposit a message on the whiteboard with the signature of the depositing module and a time stamp. Then, events like a door opening are detected on the whiteboard. Actions such as energising the microwave tube are communicated by simply placing a message on the whiteboard. Matters are a bit more complex, as messages on the whiteboard can expire and can have a priority, and thus actuators can be organised using “subsumption” [8].

Our methodology can be considered as building the score for an orchestra. We will have concurrent (parallel) FSMs. Each FSM controls the group of actuators that always behave the same by providing the cues for their state transitions. For example, in the microwave, energising the tube, running a fan, and spinning the plate all follow the same cues. However, the light does not and therefore would have a different FSM. We identify the states, and the transitions are labeled with a question to our inference engine. For example, the tube, the fan and the plate are at most in two possible states: a NOT_COOKING state or a COOKING state. We label the transition from NOT_COOKING to state COOKING with `cook` which

Fig. 1: Simple 2-state machines control most of the microwave.

is a predicate in the non-monotonic logic. It should be interpreted as the request to our deductive engine for an answer to “is it time to `cook`?”. The transition in the opposite direction is analogous. The label is `~cook` and again, we are asking “is it time to not `cook`?”. It is not hard to apply this to the light which also has two exclusive states: `LIGHT_OFF` or `LIGHT_ON`. Again, in the transition from `LIGHT_OFF` to `LIGHT_ON` we use the label `lightOn` to represent a query to the inference engine, “is it time to turn the `lightOn`?”. In general, our labels are predicates of the non-monotonic logic and stand for the question “is it time to `LABEL`?”. The state diagrams for these machines appear in Fig 1. We now turn our attention to the declarative module that answers these questions. The approach here is a iterative refinement. Consider the energising of the tube.

Behaviour 1: The default state is not to cook. However, if the timer indicates we have cooking time left, we should energise the microwave. Unless, of course, the door is open.

The above text directly corresponds to the logic program in Fig. 2a, depicting the code in the logic programming language DPL [16] which is an implementation

Fig. 2: Simple theories for 2-state machines.

of *Plausible Logic* [5, 15, 2]. Suffice it to say that Rule C0-C3 are the encoding of Behaviour 1. The rest of the code names the module, indicates that to make inferences with this logic program, the values of `timeLeft` and `doorOpen` are needed, and that the logic can make pronouncements with respect to `cook` and `~cook`. Fig 2b corresponds to the following behaviour.

Behaviour 2: Usually the light is off, but when the door is open or there is time left, the light is on.

Note the ease with which we can develop this logic declaration. We start with the usual common case or default (Rule L0), then we produce refinements. In this case, the two new rules (L1 and L2) take precedence over L0. In a similar way, a finite state machine is developed for the behaviour of the button that can be used to add time. This button is in the state of **ENABLED** when pressed or **DISABLED** when not pressed. We chose to ask the inference engine (when the button is **DISABLED**) “when should we add?” The reverse transition is a question for when is time to `~add` and move out of the **ENABLED** state. Fig. 3 captures the requirements of the button. However, a button push is ignored when the

Fig. 3: The modelling of the button’s capability to add to the timer.

door is open. The modelling of the alarm is also similar. This appears in Fig. 4a The alarm should start in the **BELL_OFF** but if the timer indicates there is time,

the bell becomes armed and ready to ring. As soon as there is no time left it should move to the state `BELL_RINGING` and then back to `BELL_OFF`. Note that there is a transition with no label in Fig. 4a (such transition always fires as the null predicate evaluates to `true`). There is a simplification to this FSM by using actions in transitions. This occurs in our next case, the model of the timer (Fig. 4b).

Fig. 4: FSMs for the bell and the timer use the logic theories already developed.

This completes the requirements engineering process. Note that for the finite state machines we have used standard UML and OMT notations, where a transition may have *actions* that are performed *on entry* (illustrated in the timer model, where the initialisation sets the integer variable `currentTime` to zero) or *on exit* (also illustrated in the timer model Fig. 4b). Finally, we can handle *activities* that are performed while in a state (again, this appears in the model for the timer). For example, the button states (Fig. 3) have activities that post the state of the button to the whiteboard. The engine and the light (Fig. 1) post messages to the whiteboard that are read by the corresponding actuators.

We emphasise that with our architecture (and our implementations of FSM interpreters and inference engines), Figs 1, 2, 3, and 4 are all that is required for a fully functioning system! Provided that the C++ code for actions and

Fig. 5: Simulation of all 5 state machines in Ubuntu (top right: whiteboard; top left: CDL inference engine; top centre: button and door sensor simulation).

activities is correct and matches the whiteboard API, all is translated into

executable code. We simulated the full microwave in a POSIX environment (we have run the finite state machines simulating the microwave system on Ubuntu and on Mac OS X, see Fig. 5), and on a behaviour tree simulation tool (see Fig. 6). Moreover, the models are automatically translated and up-

Fig. 6: A simulation snapshot of all 5 state machines using BECIE [23].

loaded onto a robot. We have done this for a LEGO MindStorm microwave (see <http://www.youtube.com/watch?v=iEkCHqSfMco>). To illustrate this point further, we re-wired the actuators to other actions on a Nao humanoid robot, and uploaded the behaviour onto the robot, allowing it to simulate a microwave (see <http://www.youtube.com/watch?v=Dm3SP3q9%5FVE>; the video also demonstrates distribution, as inputs are provided from an iPhone or a PC, it demonstrates the whiteboard decoupling as Nao’s eyes are the light, moving the head is the engine, and making a sound is the alarm). We can also distribute the modules across a network. E.g., we can run the alarm logic on a desktop computer and have the sounds emitted from the speakers of the laptop, or even on an iPhone, while other behaviours execute on the robot or on simulators.

3.1 Modelling Inter-Machine Communication

We re-emphasise that all we need to execute the microwave system is the models described earlier. Not a single line of additional code is required. Only configuration files are modified to determine which subsystems execute which modules and where on the network to locate the whiteboard. However, from the software engineering point of view, it is important to visualise and represent the inter-module communication mediated by the whiteboard. Formally, this is not necessary, but enables further understanding of the overall system. For this, we suggest to use *Behavior Trees* [23, 24, 22]. Here, the system with all modules can be integrated into a single behavior tree [22] (Fig. 7). We have already demonstrated the equivalence between behavior trees and state diagrams [4] and also represented state machines with DPL. However, the behavior tree approach offers a methodology to obtain visualisation of interactions (what the methodology refers to as the *component interaction network* [23]). In our illustration with the microwave system, the obtained interaction network appears in Fig. 8.

Fig. 7: Behavior tree with all components for finding the component diagram.

Fig. 8: The resulting component interaction network after the behavior tree analysis (white boxes: inputs/sensors; dark boxes: outputs/actuators; highlighted: the five state machines). The microwave timer plays a central coordination role.

4 Modelling Agent Behaviour – a Spider

We now illustrate the design process using our approach in a more complex example of animal-like behaviour or an agent for a simple autonomous robot. Fig. 9 describes the behaviour in natural language. We start with some sim-

1. Charlotte is a nocturnal female spider. At dusk she wakes up, spins a web, and then goes about the spidery business of waiting, eating, mating, and laying eggs. At dawn, she eats her web, then sleeps through the day (hiding away, lest she gets eaten by a bird). Eating her web ensures she is not hungry during the day, and has enough resources to spin another tomorrow evening.
2. Spinning her web makes Charlotte very very hungry. She will stay hungry until she has eaten three flies or equivalent (smaller male spiders). Hungry or not, Charlotte never turns down a chance to eat something. Charlotte is receptive to mating if she has not mated yet, or has laid eggs since her last mating, and is not tired and not hungry. Mating and laying eggs make her tired, and only a good day's sleep will refresh her.
3. Male spiders come along at night with the intent to mate. If Charlotte is receptive, they might get lucky. If Charlotte is hungry, "too bad", the male gets eaten immediately. Clever males bring a fly gift for her to eat, or don't come alone, so that it is more likely that her hunger can be satisfied. Luckily, flies are preferred as food. If Charlotte is not too hungry, she will mate, and only then eat the (evolutionarily-speaking) lucky male.
4. Charlotte lays eggs at the first opportunity, but only at night, when she is not hungry, and not tired.

Fig. 9: The description of Charlotte's behaviour.

ple finite state machines. We assume there is a light sensor that can detect day/night changes, and thus, using composite states we can obtain a high level state diagram for Charlotte (Fig. 10). Consequently, **dusk** and **dawn** are simple predicates of the logic that test the sensors' input placed on the whiteboard. However, the detailed behaviour of Charlotte's night business is more complicated, but solvable with our approach. The finite state machine (Fig. 11) is composed of sub-states. Every time a fly lands on the web, the fly count is incremented and similarly, every time a male spider appears on the web, the male

Fig. 10: High level spider behaviour with C++ actions and composite states. count is incremented. Again, Charlotte’s sensors can detect captured flies, plac-

Fig. 11: Sub-Behaviours of Charlotte’s business behaviour with C++ actions.

ing a simple message on the whiteboard that set the predicate **fly landed** to **true** without need for anything else besides direct inference. The same applies when a male appears. Whenever Charlotte eats one of the flies on her web, the fly count is decremented (note that as in UML or OMT an action in a transition is placed after a slash “/” and a transition with a null predicate is always **true**). Similarly, every time she eats a male, the male count decrements. After mating, the “lucky male” also ensures that Charlotte’s hunger is reduced through his sacrifice, resulting in a transition with no label from **MATING** to **EATING.MALE**.

Now the logic’s power really shines through predicates like **receptive** which, again, translate to the question “shall we move to **MATING**?” Thus, the predicate **eatFly** is also a question to the inference engine to establish whether there is a fly on the web. Similarly, **eatMale** and **layEggs** are requests for deducing if Charlotte shall enter the states **EATING.MALE** or, respectively, **LAYING.EGGS**. We do not present the entire DPL program here, just the main rules (Fig. 12a) that establish the transition out of **WAITING** and their simulation (Fig. 12b). Other states return back to **WAITING** when their activity is completed.

```

% receptiveness to mating
Rec: {} => receptive.
NotRec1: "hunger > 0" => ~receptive. NotRec1 > Rec.
NotRec2: tired => ~receptive. NotRec2 > Rec.
NotRec3: eggLaden => ~receptive. NotRec3 > Rec.

% Default is to wait.
Wait: {} => wait.
Wait: {} => ~mate.
Wait: {} => ~eatFly.
Wait: {} => ~eatMale.
Wait: {} => ~layEggs.
Wait: {} => ~layEggs.

% Mating is less urgent than any other item of business.
Mate: "maleCount > 0" => mate. Mate > Wait.

% Eating beats mating if hungry.
EatMale: {"maleCount > 0", "receptive"} => eatMale. EatMale > Wait.
EatMale > Mate.

% Eating flies beats eating males or mating.
EatFly: "flyCount > 0" => eatFly. EatFly > Wait. EatFly > Mate.
EatFly > EatMale.

% Prefer to lay eggs as soon as possible.
Lay: {eggLaden, "tired", "hunger > 0"} => layEggs. Lay > Wait.
Lay > Mate. Lay > EatMale. Lay > EatFly.

% Overriding urgency at dawn is to eat web and hide.
EatWeb: dawn => eatWeb. EatWeb > Wait. EatWeb > Mate.
EatWeb > EatMale. EatWeb > EatFly. EatWeb > Lay.

```

(a) Simple theory for Charlotte.

The figure shows three sequential screenshots of the 'DefeasibleTest' application. Each window has a 'Logic' dropdown set to 'dpl', a 'cdl' button, and a 'Proof Level' dropdown with options: μ , p , π , β , ϵ , δ , α . Below these are two tables: 'Input' and 'Atom'. The 'Input' table has columns for 'State' and 'Atom'. The 'Atom' table has columns for 'Atom', 'Proof', and 'Negation'. Buttons for 'Add', 'Delete', and 'Do Proofs' are at the bottom of each window.

- Top Window:** All inputs are unchecked. The 'Atom' table shows 'eatFly' with a proof level of μ and 'wait' with a proof level of μ .
- Middle Window:** 'q_files_0' is checked. The 'Atom' table shows 'eatFly' with a proof level of μ and 'wait' with a proof level of μ .
- Bottom Window:** 'q_files_0' and 'dawn' are checked. The 'Atom' table shows 'eatFly' with a proof level of μ , 'eatMale' with a proof level of μ , 'eatWeb' with a proof level of μ , 'layEggs' with a proof level of μ , and 'wait' with a proof level of μ .

(b) Inference Simulation

Fig. 12: Spider Business DPL.

5 Conclusions

We have demonstrated the viability of our approach to capture the requirements of, model, simulate, and deploy a robotic system in a fully automated way. While we have no space here to describe other illustrative cases of our approach, suffice it to say that we have also modelled sophisticated embedded systems (such as the cruise control of a car) that have been described in the literature.

References

1. G. Antoniou. The role of nonmonotonic representations in requirements engineering. *Int. J. of Software Engineering and Knowledge Engineering*, 8(3):385–399, 1998.
2. D. Billington. The proof algorithms of plausible logic form a hierarchy. S. Zhang and R. Jarvis, editors, *Proc. 18th Australian Joint Conf. on Artificial Intelligence*, vol 3809, p. 796–799, Sydney, Australia, Dec. 5th–9th 2005. Springer Verlag LNAI.
3. D. Billington, V. Estivill-Castro, R. Hexel, and A. Rock. Architecture for hybrid robotic behavior. E. Corchado, X. Wu, E. Oja, A. Herrero, and B. Baruque, eds, *4th Int. Conf. on Hybrid Artificial Intelligence Systems HAIS*, vol 5572, p. 145–156. Springer-Verlag LNCS, June 2009.
4. D. Billington, V. Estivill-Castro, R. Hexel, and A. Rock. Plausible logic facilitates engineering the behavior of autonomous robots. R. Fox and W. Golubski, editors, *IASTED Conf. on Software Engineering (SE 2010)*, p. 41–48, Anaheim, California, USA, Feb. 16 - 18 2010. IASTED, ACTA Press. location: Innsbruck, Austria.
5. D. Billington and A. Rock. Propositional plausible logic: Introduction and implementation. *Studia Logica*, 67:243–269, 2001.
6. C. V. Bochmann and C. A. Sunshine. Formal methods in communication protocol design. *IEEE Transaction on Communications*, 28(4):624–631, April 1980.

7. R.A. Brooks. How to build complete creatures rather than isolated cognitive simulators. In K. VanLehn, editor, *Architectures for Intelligence*, p. 225–239, Hillsdale, NJ, 1991. Lawrence Erlbaum Associates.
8. R.A. Brooks. Intelligence without reason. R. Myopoulos and R. Reiter, eds, *Proc. 12th Int. Joint Conference on Artificial Intelligence*, p. 569–595, San Mateo, CA, 1991. ICJAI-91, Morgan Kaufmann Publishers. Sydney, Australia.
9. R. G. Dromey and D. Powell. Early requirements defect detection. *TickIT Journal*, 4Q05:3–13, 2005.
10. C. Ellis. Team automata for groupware systems. *GROUP '97: Proc. of the Int. ACM SIGGROUP Conf. on Supporting group work*, p. 415–424, New York, NY, USA, 1997. ACM.
11. D. Harel and M. Politi. *Modeling Reactive Systems with Statecharts: The STATE-MATE Approach*. McGraw-Hill, 1998.
12. M. Löttsch, J. Bach, H.-D. Burkhard, and M. Jüngel. Designing agent behavior with the extensible agent behavior specification language XABSL. *7th Int. Workshop on RoboCup 2003 (Robot World Cup Soccer Games and Conferences)*, vol 3020 *LNAI*, p. 114–124. Springer, 2004.
13. S. J. Mellor. Embedded systems in UML. OMG White paper, 2007. www.omg.org/news/whitepapers/ label: We can generate Systems Today.
14. T. Myers and R. G. Dromey. From requirements to embedded software - formalising the key steps. *20th Australian Software Engineering Conf. (ASWEC)*, p. 23–33, Gold Coast, Australia, 14-17 April 2009. IEEE Computer Soc.
15. A. Rock and D. Billington. An implementation of propositional plausible logic. J. Edwards, editor, *23rd Australasian Computer Science Conf.*, vol. 22(1) of *Australian Computer Science Communications*, p. 204–210, Canberra, Jan. 2000. IEEE Computer Soc., Los Alamitos.
16. Andrew Rock. The DPL (decisive Plausible Logic) tool. Technical report, (continually) in preparation, available at www.cit.gu.edu.au/~arock/.
17. J. Rumbaugh, M. R. Blaha, W. Lorensen, F. Eddy, and W. Premerlani. *Object-Oriented Modelling and Design*. Prentice-Hall, Englewood Cliffs, NJ, 1991.
18. M. Samek. *Practical UML Statecharts in C/C++: Event-Driven Programming for Embedded Sytsems*. Newnes, second edition, 2008.
19. S. Shlaer and S. J. Mellor. *Object lifecycles : modeling the world in states*. Yourdon Press, Englewood Cliffs, N.J., 1992.
20. M.H. ter Beek, C.A. . Ellis, J. Kleijn, and G. Rozenberg. Synchronizations in team automata for groupware systems. *Computer Supported Cooperative Work (CSCW)*, 12(1):21–69, Feb. 2003.
21. F. Wagner, R. Schmuki, T. Wagner, and P. Wolstenholme. *Modeling Software with Finite State Machines: A Practical Approach*. CRC Press, NY, 2006.
22. L. Wen, R. Colvin, K. Lin, J. Seagrott, N. Yatapanage, and R.G. Dromey. “Integrare”, a collaborative environment for behavior-oriented design. T. Luo, ed, *Cooperative Design, Visualization, and Engineering*, *4th Int. Conf., CDVE*, vol 4674 of *LNCS*, p. 122–131, Shanghai, China, Sep. 16-20 2007. Springer.
23. L. Wen and R. G. Dromey. From requirements change to design change: A formal path. *2nd Int. Conf. on Software Engineering and Formal Methods (SEFM 2004)*, p. 104–113, Beijing, China, 28-30 Sep. 2004. IEEE Computer Soc.
24. L. Wen, D. Kirk, and R. G. Dromey. A tool to visualize behavior and design evolution. M. Di Penta and M. Lanza, eds, *9th Int. Workshop on Principles of Software Evolution (IWPSE 2007)*, in conjunction with the 6th ESEC/FSE joint meeting, p. 114–115, Dubrovnik, Croatia, Sep. 3-4 2007. ACM.