NON-MONOTONIC REASONING FOR REQUIREMENTS ENGINEERING
State Diagrams Driven by Plausible Logic

David Billington, Vladimir Estivill-Castro, Rene Hexel and Andrew Rock
School of ICT, Griffith University, Nathan Campus, Nathan, 4111 Queensland, Australia

Keywords: Automata and logic for system analysis and verification, Petri nets, Requirements engineering.

Abstract: We extend the state diagrams used for dynamic modelling in object-oriented analysis and design. We suggest that the events which label the state transitions be replaced with plausible logic expressions. The result is a very effective descriptive and declarative mechanism for specifying requirements that can be applied to requirements engineering of robotic and embedded systems. The declarative model can automatically be translated and requirements are traceable to implementation and validation, minimising faults from the perspective of software engineering. We compare our approach with *Petri Nets* and *Behavior Trees* using the well-known example of the one-minute microwave oven.

1 INTRODUCTION

We extend state transition diagrams in that we allow transitions to be labelled by statements of a non-monotonic logic, in particular Plausible Logic. We show that this has several benefits. First, it facilitates requirements engineering. Namely, we show this approach can be more transparent, clear, and succinct than other alternatives. Therefore, it enables better capture of requirements and this leads to much more effective system development. Furthermore, we show that such diagrams can be directly, and automatically translated into executable code, i.e. no introduction of failures in the software development process.

Finite automata have a long history of modelling dynamic systems and consequently have been a strong influence in the modelling of the behaviour of computer systems (Rumbaugh et al., 1991, Bibliographical notes, p. 113-114). System analysis and design uses diagrams that represent behaviour of components or classes. State diagrams (or state machines) constitute the core behavior modeling tool of object-oriented methodologies. In the early 90s state machines became the instrument of choice to model the behaviour of all the objects of a class. The Object-Modeling Methodology (OMT) of Rumbaugh et al. (1991, chapter 5) established state diagrams as the primary dynamic model. The Shlaer-Mellor approach established state models to capture the life cycle of objects of a given class (Shlaer and Mellor, 1992). Class diagrams capture the static information of all objects of the same class (what they know, what they store), but behaviour is essentially described in models using states and transitions. The prominence of OMT and Shlaer-Mellor has permeated into the most popular modelling language for object-orientation, the Unified Modeling Language (UML). "A state diagram describes the behavior of a single class of objects" (Rumbaugh et al., 1991, p. 90). Although the state diagrams for each class do not describe the interactions and behaviour of several objects of diverse classes in action (for this, UML has collaboration diagrams and sequence diagrams), they constitute a central modelling tool for software engineering.

Although UML and its variants have different levels of formality, in the sense of having a very clear syntax and semantics, they aim for the highest formality possible. This is because their aim is to remove ambiguity and be the communication vehicle between requesters, stakeholders, designers, implementors, testers, and users of a software system. Thus, they offer constraints very similar to the formal finite state machine models. For example, in a deterministic finite state machine, no two transitions out of the same state can be labelled with the same symbol. This is because formally, a deterministic finite state machine consists of a finite set of states, an input language (for events), and a transition function. The transition function indicates what the new state will be, given an input and the current state. Other

Billington D., Estivill-Castro V., Hexel R. and Rock A. (2010).
NON-MONOTONIC REASONING FOR REQUIREMENTS ENGINEERING - State Diagrams Driven by Plausible Logic.
In *Proceedings of the Fifth International Conference on Evaluation of Novel Approaches to Software Engineering*, pages 68-77
DOI: 10.5220/0002998300680077

Table 1: The Transition Function as a Table.

s_1	c_u	s_i
s_1	c_v	s_j
	...	
s_i	c_x	s_p

adornments include signalling some states as *initial* and some as *final*. However, a fundamental aspect of finite state machines is that the transition function is just that, a function (mathematically, a function provides only one value of the codomain for each value in the domain).

Granted that the model can be extended to a non-deterministic machine, where given an input and a state, a set of possible states is the outcome of the transition. In this case, the semantics of the behaviour has several interpretations. For example, in the theory of computation, the so-called *power set construction* shows that non-deterministic and deterministic finite state machines are equivalent. However, other semantics are possible, such as multi-threaded behaviour. Therefore, as a modelling instrument in software engineering, it is typically expected that the conditions emanating from a state are mutually exclusive and exhaustive. "All the transitions leaving a state must correspond to different events" (Rumbaugh et al., 1991, p. 89). Namely, if the symbol c_i is a Boolean expression representing the guard of the transition, then $\bigvee_{i=1}^{n} c_i = \mathsf{true}$ (the exhaustive condition), and $c_i \wedge c_j = \mathsf{false}$, $\forall\, i \neq j$ (the exclusivity condition). In fact, Shlaer-Mellor also suggest the analysis should make use of the *State Transition Table* (STT) (Shlaer and Mellor, 1992). Table 1 is the tabular representation of the transition function "to prevent one from making inconsistent statements" (Shlaer and Mellor, 1992, p. 52) and they provide an illustration where two transitions out of the same state and labelled by the same event are corrected using the table.

Recently, the software engineering community has been pushing for *Requirements Engineering* (RE) (Hull et al., 2005), concerned with identifying and communicating the objectives of a software system, and the context in which it will be used (Nuseibeh and Easterbrook, 2000). Hence, RE identifies and elicits the needs of users, customers, and other stakeholders in the domain of a software system. RE demands a careful systematic approach. Significant effort is to be placed on rigorous analysis and documented specification, especially for security or safety critical systems. RE is important because a requirement not captured early may result in a very large effort to re-engineer a deployed system.

2 NON-MONOTONIC LOGIC IN STATE DIAGRAMS

An ambition of both artificial intelligence (AI) and software engineering is to be able to only specify *what* we want, without having to detail *how* to achieve this. The motivation for our approach has a similar origin. We aim at producing a vehicle of communication that would enable the specification of behaviour without the need for imperative programming tools. Therefore, we want to use a declarative formalism (a similar ambition has lead to the introduction of logic programming and functional programming). Non-monotonic logic is regarded as quite compatible with the way humans reason and express the conditions and circumstances that lead to outcomes, as well as a way to express the refinements and even exceptions that polish a definition for a given concept. In fact, non-monotonic reasoning is regarded as one of the approaches to emulate common-sense reasoning (Russell and Norvig, 2002). We illustrate that the addition of this declarative capability to state transition diagrams for capturing requirements is beneficial because the models obtained are much simpler (a fact necessary to ensure that the natural language description has indeed been captured). This way, we only need to specify the *what* and can have all of the *how* within an embedded system generated automatically.

With our approach, modelling with state-diagrams is sufficient to develop and code behaviours. The semantics of a *state* is that it is lasting in time, while a *transition* is assumed to be instantaneous. The state-diagram corresponds closely to the formalism of finite state machines (defined by a set S of states, a transition function $t : S \times \Upsilon \to S$, where Υ denotes a possible alphabet of input symbols). In our case, we can specify the behaviour by the table that specifies the transition function t (Table 1).

We still use the notion of an initial state s_0, because, in our infrastructure that implements these ideas (Billington et al., 2009), in some *external-state* transition, behaviours must be able to reset themselves to the initial state. A final state is not required, but behaviours should be able to indicate completion of a task to other modules. Now, current practice for modelling with finite state machines assumes that transitions are labelled by events. In both the Shlaer-Mellor approach and OMT (Rumbaugh et al., 1991), transitions are labelled by events only. For example, in Fig. 1, a transition is labelled by the event `ball_visible` (e.g. a sensor has detected a ball). A slight extension is to allow labels to be a decidable Boolean condition (or expression) in a logic with values `true` or `false` (that is, it will always be possi-

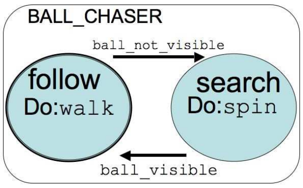


Figure 1: Simple Finite State Diagram.

ble to find the value of the condition guarding the transition). This easily captures the earlier model because rather than labelling by event *e*, we label by the Boolean expression *e_has_occured*. Our extension to behaviour modelling extends this further with the transition labels being any sentence in the non-monotonic logic. Replacing the guarding conditions with statements in the non-monotonic logic incorporates reasoning into the reactive[1] nature of state machines. Since our logics model reasoning (and beliefs like "in this frame vision believes there is no ball"), they are better suited to model these transitions (they may even fuse contradicting beliefs reported by many sensors and modules in a deterministic way), and gracefully handle situations with incomplete (or superfluous) information without increasing the cognitive load of the behaviour designer.

The designer can separate the logic model from the state-transition model. Moreover, the designer would not be required to ensure the exhaustive nature of the transitions leading out from a state; as priorities can indicate a default transition if conditions guarding other transitions cannot be decided.

"State diagrams have often been criticized because they allegedly lack expressive power and are impractical for large problems" (Rumbaugh et al., 1991, p. 95). However, several techniques such as nesting state diagrams, state generalisation, and event generalisation were used in OMT to resolve this issue. We have shown elsewhere (Billington et al., 2010) (1) how the technique of nested state diagrams (e.g. team automata (ter Beek et al., 2003; Ellis, 1997) handle complexity, and (2) that there is an equivalence between Behavior Trees and state machines, mitigating the problem of expressive power of state diagrams for larger systems. In fact our approach follows the very successful modelling by finite state machines (Wagner et al., 2006) that has resulted in *stateWORKS*, a product used for over decade in the engineering of embedded systems software (Wagner and Wolstenholme, 2003). In *stateWORKS*, transitions are labelled by a small subset of propositional logic, namely *positive logic algebra* (Wagner and Wolstenholme, 2003), which has no implication, and no negation (only OR and AND). Thus our use of a non-monotonic logic is a significant variation.

[1] In the agent model *reactive* systems are seen as an alternative to logic-based systems that perform planning and reasoning (Wooldridge, 2002).

3 PLAUSIBLE LOGIC

Non-monotonic reasoning (Antoniou, 1997) is the capacity to make inferences from a database of beliefs and to correct those as new information arrives that makes previous conclusions invalid. Although several non-monotonic formalisms have been proposed (Antoniou, 1997), The β algorithm for PL uses the closed world assumption while the π algorithm uses the open world assumption. the family of non-monotonic logics called Defeasible Logics has the advantage of being designed to be implementable. The main members of this family, including *Plausible Logic* (PL), are compared in (Billington, 2008), listing their uses and desirable properties. Although the most recent member of this family, CDL (Billington, 2008), has some advantages over PL (Billington and Rock, 2001), the differences are not significant for the purposes this paper. We shall therefore use PL as its corresponding programming language, DPL, is more advanced. If only factual information is used, PL essentially becomes classical propositional logic. But when determining the provability[2] of a formula, the proving algorithms in PL can deliver three values (that is, it is a three-valued logic), $+1$ for a formula that has been proved, -1 for a formula that has been disproved, and 0 when the formula cannot be proved and attempting so would cause an infinite loop. Another very important aspect of PL is that it distinguishes between formulas proved using only factual information and those using plausible information. PL allows formulas to be proved using a variety of algorithms, each providing a certain degree of trust in the conclusion. Because PL uses different algorithms, it can handle a closed world assumption (where not telling a fact implies the fact is false) as well as the open world assumption in which not being told a fact means that nothing is known about that fact.

In PL all information is represented by three kinds of rules and a priority relation between those rules. The first type of rules are strict rules, denoted by the strict arrow $\rightarrow$ and used to model facts that are certain. For a rule $A \rightarrow l$ we should understand that if all literals in A are proved then we can deduce l (this is simply ordinary implication). A situation such as *Humans are mammals* will be encoded as $human(x) \rightarrow mammal(x)$. Plausible rules $A \Rightarrow l$ use

[2] Provability here means determining if the formula can be verified or proved.

the plausible arrow $\Rightarrow$ to represent a plausible situation. If we have no evidence against l, then A is sufficient evidence for concluding l. For example, we write *Birds usually fly* as $bird(x) \Rightarrow fly(x)$. This records that when we find a bird we may conclude that it flies unless there is evidence that it may not fly (e.g. if it is a penguin). Defeater rules $A \rightharpoonup \neg l$ say if A is not disproved, then it is risky to conclude l. An example is *Sick birds might not fly* which is encoded as $\{sick(x), bird(x)\} \rightharpoonup \neg fly(x)$. Defeater rules prevent conclusions that would otherwise be risky (e.g. from a chain of plausible conclusions).

Finally, a priority relation $>$ between plausible rules $R_1 > R_2$ indicates that R_1 should be used instead of R_2. The following example demonstrates the expressive power of this particular aspect of the formalism:

$\{\} \rightarrow quail(Quin)$	*Quin is a quail*
$quail(x) \rightarrow bird(x)$	*Quails are birds*
$R_1 : bird(x) \Rightarrow fly(x)$	*Birds usually fly*

From the rule R_1 above, one would logically accept that *Quin* flies (since *Quin* is a *bird*).

$\{\} \rightarrow quail(Quin)$	*Quin is a quail*
$quail(x) \rightarrow bird(x)$	*Quails are birds*
$R_2 : quail(x) \Rightarrow \neg fly(x)$	*Quails usually do not fly*

However, from R_2, we would reach the (correct) conclusion that Quin usually does not fly. But what if both knowledge bases are correct (both R_1 and R_2 are valid)? We perhaps can say that R_2 is more informative as it is more specific and so we add $R_2 > R_1$ to a knowledge base unifying both. Then PL allows the agent to reach the proper conclusion that Quin usually does not fly, while if it finds another bird that is not a quail, the agent would accept that it flies. What is important to note here is that if the rule set is consistent, all proofs withing PL will also be consistent. I.e. as long as any conflicts between plausible rules are properly resolved using priority relations, it will never be possible to prove both a literal l and its negation $\neg l$ at the same time.

Note that Asimov's famous Three Laws of Robotics are a good example of how humans describe a model. They define a general rule, and the next rule is a refinement. Further rules down the list continue to polish the description. This style of development is not only natural, but allows incremental refinement. Indeed, the knowledge elicitation mechanism known as *Ripple Down Rules* (Compton and Jansen, 1990) extracts knowledge from human experts by refining a previous model by identifying the rule that needs to be expanded by detailing it more.

Table 2: 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.

4 A CLASSICAL EXAMPLE

We proceed here to illustrate our approach with an example that has been repeatedly used by the software engineering community, e.g. (Dromey and Powell, 2005; Myers and Dromey, 2009; Shlaer and Mellor, 1992; Wen and Dromey, 2004; Mellor, 2007). This is the so called one-minute microwave oven (Shlaer and Mellor, 1992). Table 2 shows the requirements as presented by Myers and Dromey (Myers and Dromey, 2009, p. 27, Table 1). Although this is in fact not exactly the same as the original by Shlaer and Mellor (Shlaer and Mellor, 1992, p. 36), we have chosen the former rather than the latter because we will later compare with Behavior Trees regarding model size and direct code generation.

4.1 Microwave in Plausible Logic

Because we have a software architecture that handles communication between modules through a decoupling mechanism named the *whiteboard* (Billington et al., 2009), we can proceed at a very high level. We assume that 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. Thus, events like a button push or actions such as energising the microwave tube are communicated by simply placing a message on the whiteboard.[3] Thus, knowledge of an event like a button push simply exists because a corresponding message has appeared on the whiteboard. Similarly, an action like energising the microwave tube is trig-

[3]Matters are a bit more complex, as messages on the whiteboard expire or are consumed, and for actuators, they could have a priority and thus actuators can be organised with "subsumption" (Brooks, 1991).

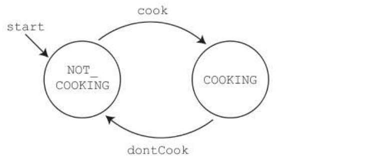
```
% MicrowaveCook.d

name{MicrowaveCook}.

input{timeLeft}.
input{doorOpen}.

C0: {}       => ~cook.
C1: timeLeft =>  cook. C1 > C0.
C2: doorOpen => ~cook. C2 > C1.

output{b cook, "cook"}.
output{b ~cook, "dontCook"}.
```

(a) DPL for 2-state machine controlling tube, fan, and plate.

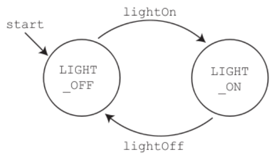
```
% MicrowaveLight.d

name{MicrowaveLight}.

input{timeLeft}.
input{doorOpen}.

L0: {}       => ~lightOn.
L1: timeLeft =>  lightOn. L1 > L0.
L2: doorOpen =>  lightOn. L2 > L0.

output{b lightOn, "lightOn"}.
output{b ~lightOn, "lightOff"}.
```

(b) DPL for 2-state machine controlling the light.

Figure 2: Simple theories for 2-state machines.



(a) A 2-state machine for controlling tube, fan, and plate.



(b) A 2-state machine for controlling the light.

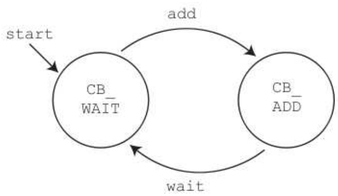
Figure 3: Simple 2-state machines control most of the microwave.

gered by placing a different message on the whiteboard. The driver for the corresponding actuator then performs an action for this particular message as soon as it appears on the whiteboard.

However, here the label `cook` for the transition of the state NOT_COOKING to the state COOKING and the label `dontCook` from COOKING to NOT_COOKING are not necessarily events. They are consequents in a logic model. For example, `dontCook` is an output of such a model that acts as the cue to halt the cooking. The logic model will specify the conditions by which this cue is issued. Fig. 2a shows the logic model in the logic programming language DPL that implements PL. In the case of the microwave oven requirements, for the purposes of building a model, a system analyst or software engineer would first identify that there are two states for various actuators. When the oven is cooking, the fan is operating, the tube is energised and the plate is rotating. When the oven is not cooking, all these actuators are off. The approach can be likened to arranging the score for an orchestra: all these actuators will need the same cues from the conductor (the control) and, in this example, all switch together from the state of COOKING to the state of NOT_COOKING and vice versa. They will all synchronously consume the message to be off or to be on. Thus, we have a simple state diagram to model this (Fig. 3a). By default, we do not have the conditions to cook. This is Rule `C0` in the logic model (called a *theory*) relevant to the cooking actuators. However, if there is time left for cooking, then we have the conditions to cook (Rule `C1`) and this rule takes priority over `C0`. However, when the door opens, then we do not cook: `C2` takes priority over `C1`.

The light in the microwave is on when the door is open as well as when the microwave is cooking. So, the cues for the light are not the same as those for energising the tube. However, the light is in only one of two states LIGHT_OFF or LIGHT_ON. The default state is that the light is off. This is Rule `L0` in the *theory* for the light (see Fig. 2b). However, when cooking the light is on. So Rule `L1` has priority over `L0`. There is a further condition that overwrites the state of the light being off, and that is when the door is open (Rule `L2`). Note that in this model, the two rules `L1` and `L2` override the default Rule `L0`, while in the model for cooking the priorities caused each new rule to refine the previous rule. The control button (Fig. 4) also has two states. In one state, CB_ADD, the time left can be incremented, while in the other state, pushing the button has no effect. Again, between these two states we place transitions labelled by an expression of PL (in all cases, simple outputs of a theory). The control button does not add time unless the button is pushed. This is reflected by Rule `CB0` and Rule `CB1` below and the priority that `CB1` has over `CB0`. When the door is open, pushing the button has no effect; this is Requirement R3 and expressed by Rule `CB2` and its preference over `CB1`. Because we already have defined that a push of a button adds time to the timer (except for the conditions already captured), if the button is not pushed, then we do not add time. This is a strict rule that in DPL is expressed by a disjunction.

The final requirement to model is the bell, which is armed while cooking, and rings when there is no time left. This is the transition `noTimeLeft` from

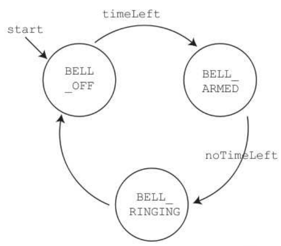


(a) State machine

```
% MicrowaveButton.d

name{MicrowaveButton}.

input{doorOpen}.
input{buttonPushed}.

CB0: {}            => ~add.
CB1: buttonPushed =>  add. CB1 > CB0.
CB2: doorOpen     => ~add. CB2 > CB1.

\/{buttonPushed, wait}.

output{b add, "add"}.
output{b wait, "wait"}.
```

(b) DPL theory

Figure 4: The modelling of the button's capability to add to the timer.



(a) Bell state machine

```
% MicrowaveBell.d

name{MicrowaveBell}.

input{timeLeft}.

output{b ~timeLeft, "noTimeLeft"}.
```

(b) DPL theory for the bell.

Figure 5: The modelling of the bell's capability to ring when the time expires.

BELL_ARMED to BELL_RINGING in Fig. 5. After ringing, the bell is off, and when cooking time is added to the timer, it becomes armed. The logical model is extremely simple, because the condition that departs from BELL_ARMED to BELL_RINGING is the negation of the one that moves from to BELL_OFF to BELL_ARMED. Moreover, we always move from BELL_RINGING to BELL_OFF. The most important aspect of this approach is that this is *all* the software analysis required in order to obtain the working program.

```
#define dontCook ( \
      doorOpen \
   || !timeLeft \
)

#define cook ( \
      !doorOpen && timeLeft \
)
```

(a) Tube, fan and plate

```
#define lightOff ( \
      !doorOpen && !timeLeft \
)

#define lightOn ( \
      doorOpen \
   || timeLeft \
)
```

(b) The light

```
#define add ( \
      buttonPushed && !doorOpen \
)

#define wait ( \
      !buttonPushed \
)
```

(c) Button and timer

```
#define noTimeLeft ( \
      !timeLeft \
)
```

(d) The bell

Figure 6: Translated C expressions for transitions.

4.2 Translation into Code

Once the high level model has been established in DPL, translation into code is straightforward. A Haskell proof engine implementation of DPL allows the interpretation and formal verification of the developed rule sets (Billington and Rock, 2001). This implementation was extended to include a translator that generates code that can be used in C, C++, Objective-C, C# and Java. The Haskell translator creates optimised Boolean expressions through the truth tables generated from the DPL rules. These Boolean expressions are then written out as C code that can directly be compiled and linked with libraries and application code. Incidentially, the C syntax for Boolean expressions is not only the same in supersets of C (such as C++ and Objective-C), but also in modern, related programming languages such as Java[4].

Figure 6 shows the DPL theories for the state diagram transitions translated into C by the Haskell proof engine. This code can then directly be used as a header file for a generic embedded system state machine to test the transition conditions. Subsequent refinements of the rules do not require any changes to the generic state machine code. A simple recompilation against the updated, generated header files is sufficient to update the behaviour of the state machine.

5 EVALUATION

The original approaches to modelling behaviour with finite state diagrams (Rumbaugh et al., 1991; Shlaer

[4]At this stage, expressions are generated using the `#define` preprocessor syntax, that is not supported directly in Java, but can easily be extracted using a script or even copy and paste.

and Mellor, 1992) had little expectation that the models would directly translate to implementations without the involvement of programmers using imperative object-oriented programming languages. However, the software development V-model (Wiegers, 2003) has moved the focus to requirements modelling, and then directly obtaining a working implementation, because this collapses the requirements analysis phase with the verification phase. There are typically two approaches. First, emulating or simulating the model, which has the advantage that software analysts can validate the model and implementation by running as many scenarios as possible. The disadvantage is the overhead incurred through the interpretation of the model, rather than its compilation. The second approach consists of generating code directly from the model (Wagner and Wolstenholme, 2003; Wagner et al., 2006). This removes the overhead of interpreting at run-time the modelling constructs. Approaches to the automatic execution or translation of models for the behaviour of software have included the use of UML state diagrams for generating code (Mellor, 2007), the automatic emulation or code generation from Petri nets (Girault and Valk, 2001), and the automatic emulation or code generation from *Behavior Trees* (Wen and Dromey, 2004; Wen et al., 2007b; Wen et al., 2007a). A more recent trend is *models@run.time*, where "there is a clear pressure arising for mirroring the problem space for more declarative models" (Blair et al., 2009).

5.1 Contrast with State Diagrams

Simulation and direct generation of code from a state diagram is clearly possible, since one only needs to produce generic code that reads the transition table (encoded in some standard form), then deploy and interpret that repeatedly by analysing the events received as well as the current state, and moving to the proper subsequent state. This has been suggested for UML (Mellor, 2007) and is the basis of the design pattern `state` (Larman, 1995, p. 406). However, while Finite State Machines continue to enjoy tremendous success (Wagner and Wolstenholme, 2003), "there is no authoritative source for the formal semantics of dynamic behavior in UML" (Winter et al., 2009). The best example for the success of Finite State Machines is *stateWorks* (`www.stateworks.com`) and its methodology (Wagner et al., 2006). We have downloaded the 60-day free license of *stateWorks Studio* and the *SWLab* simulator (Fig. **??**). Note that this finite state machine has only 5 states (the documentation of this example with *stateWorks* admits the model has issues, e.g. "to reset the system for the next start, we have to open and then close the door" and "the control system always starts even when the timeout value is 0"). These issues can be fixed but additional infrastructure is necessary, including 'counters' and 'switch points' as well as usage of the 'real time database (RTDN)'. The *stateWorks* example describes generic of microwave behaviour and needs some more polishing to capture more detailed requirements e.g. those outlined in Table 2. Since we argue in favour of using finite state machines for modelling behaviour, this tool concurs with that approach. However, our evaluation confirms that using PL is more powerful and closer to the original specification than the "positive-logic" transitions in *stateWorks*.

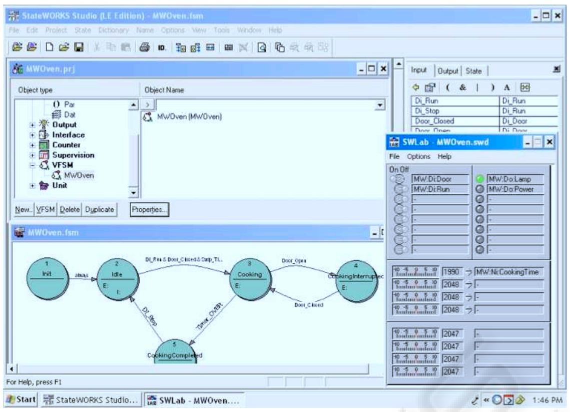


Figure 7: The execution of the example model provided in the demo version of *stateWorks* for a microwave oven.

5.2 Contrast with Petri Nets

Petri Nets (Peterson, 1977; Holloway et al., 1997) provide a formal model for concurrency and synchronisation that is not readily available in state diagrams. Thus, they offer the possibility of modelling multi-threaded systems that support requirements for concurrency. Early in the modelling effort Petri Nets were dismissed: "Although they succeed well as an abstract conceptual model, they are too low level and inexpressive to be useful to specify large systems" (Rumbaugh et al., 1991, p. 144). However, because it is quite possible to simulate or interpret a Petri Net (or to generate code directly from it), they continue to be suggested as an approach to directly obtain implementations from the coding of the requirements (Gold, 2004; Lian et al., 2008; Lakos, 2001; Saldhana and Shatz, 2000).

We have used PIPE 2.5 (`pipe2.sf.net`) to construct a model of the microwave as per the requirements in Table 2 (see Fig. 8). It becomes rapidly apparent that the synchronisation of states between com-

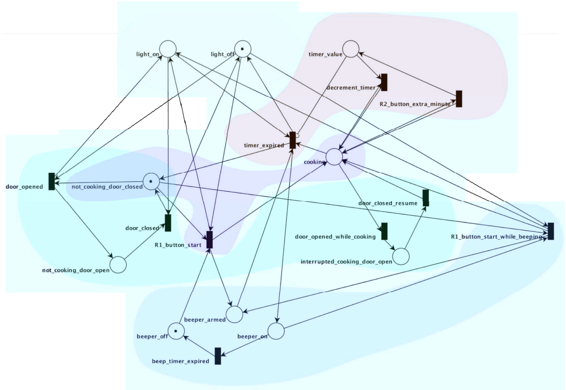


Figure 8: Capturing the requirements in Table 2 as a Petri Net, grouped by the various components of the microwave.

ponents of the system forces to display connectors among many parts in the layout, making the model hard to grasp.Even if we consider incremental development, each new requirement adds at least one place and several transitions from/to existing places. Therefore, we tend to agree that even for this small case of the one-minute microwave, the Petri Net approach seems too low level and models are not providing a level of abstraction to assist in the behaviour engineering of the system. One advantage of Petri Net models is that there are many tools and algorithms for different aspects of their validation. For example, once a network is built with PIPE, this software has algorithms to perform GSPN Analysis, FSM analysis, and Invariant Analysis. However, even for this example, considered simple and illustrative in many circles, the model is not a suitable input for any of the verification analysis in PIPE. On a positive note, some first-order logics have been included in transitions to create *Predicate/Transition Nets* (Genrich and Lautenbach, 1979; Genrich, 1991). Consequently, we see no reason why our approach to use a non-monotonic logic cannot be applied to Petri Nets.

5.3 Contrast with Behavior Trees

Behavior Trees (Wen et al., 2007a) is another powerful visual approach for *Behaviour Engineering* (the systematic progression from requirements to the software of embedded systems). The approach provides a modelling tool that constructs acyclic graphs (usually displayed as rooted tree diagrams) as well as a Behavior Modelling Process (Dromey and Powell, 2005; Myers and Dromey, 2009) to transform natural language requirements into a formal set of requirements. The *Embedded Behavior Runtime Environment (eBRE)* (Myers and Dromey, 2009) executes Behavior Tree models by applying transformations and generating C source code. The tool *Behavior Engineering Component Integration Environment (BECIE)* allows Behavior Trees to be drawn and simulated. Proponents of Behavior Trees argue that these diagrams enable requirements to be developed incrementally and that specifications of requirements can be captured incrementally (Wen et al., 2007b; Wen and Dromey, 2004). The classic example of the one-minute microwave has also been extensively used by the Behavior Tree community (Wen and Dromey, 2004; Dromey and Powell, 2005; Myers and Dromey, 2009). Unfortunately, for this example, Behavior Trees by comparison are disappointing. In the initial phase of the method, requirements R1, R2, R5, R6 in Table 2 use five boxes (Wen and Dromey, 2004; Dromey and Powell, 2005), while R3 and R4 demand four. Six boxes are needed for requirement R7 (Wen and Dromey, 2004; Dromey and Powell, 2005). Then, the *Integration Design Behavior Tree (DBT)* demands 30 nodes (see (Dromey and Powell, 2005, p. 9) and (Wen and Dromey, 2004, Fig. 5)). By the time it becomes a model for *eBRE* the microwave has 60 nodes and 59 links! (Myers and Dromey, 2009, Fig. 6) and the *Design Behavior Tree* (Myers and Dromey, 2009, Fig. 8) does not fit legibly on an A4 page. Sadly, the approach seems to defeat its purpose, because on consideration of the system boundaries (Myers and Dromey, 2009, Fig. 7), outputs to the alarm are overlooked. Moreover, the language for logic tests in the tools for Behavior Trees is far more limited than even the ‘positive-logic’ of *stateWORKS*. The equivalence between finite state machines and Behavior Trees (Billington et al., 2010) is based on the observation that Behavior Trees correspond to the depth-first search through the treads of the states of the system behaviour control. It is not surprising that the notion needs far more nodes and connections than the corresponding finite state machine.

6 FINAL REMARKS

We stress two more aspects of the comparison. First, the approaches above attempt, in one way or another, to construct the control unit of the embedded system, and from it the behaviour of all of its components. This implies that the control unit has a state space that is a subset of the Cartesian product of the states of the components. Our approach is more succinct not only because of a more powerful logic to describe state transition, but because our software architecture decouples control into descriptions for the behaviour of components. Second, our experience with this approach and the development of non-monotonic mod-

Figure 9: Hardware running Java generated code.

els show that capturing requirements is structured and incremental, enabling iterative refinement. That is, one can proceed from the most general case, and produce rules and conditions for more special cases. We ensured that our method delivers executable embedded systems directly from the modelling by implementing an oven where the hardware is constructed from LEGO Mindstorm pieces, sensors, and actuators (see Fig. 9). As with *eBRE*, we output Java source code but execute a finite state machine. The execution then is verified because of the clear connection between the model and the source code (as well as testing it on the hardware).[5]

REFERENCES

Antoniou, G. (1997). *Nonmonotonic Reasoning*. MIT Press, Cambridge, Mass. ISBN 0-262-01157-3.

Billington, D. (2008). Propositional clausal defeasible logic. In Holldobler, S., Lutz, C., and Wansing, H., editors, *Logics in Artificial Intelligence*, volume 5293 of *Lecture Notes in Artificial Intelligence*, pages 34–47, Dresden, Germany. 11th European Conference on Logics in Artificial Intelligence (JELIA2008), Springer.

Billington, D., Estivill-Castro, V., Hexel, R., and Rock, A. (2009). Architecture for hybrid robotic behavior. In Corchado, E., Wu, X., Oja, E., Herrero, A., and Baruque, B., editors, *4th International Conference on Hybrid Artificial Intelligence Systems HAIS*, volume 5572, pages 145–156. Springer-Verlag Lecture Notes in Computer Science.

Billington, D., Estivill-Castro, V., Hexel, R., and Rock, A. (2010). Plausible logic facilitates engineering the behavior of autonomous robots. In *Proceedings of the IASTED Software Engineering Conference*.

[5] See www.youtube.com/watch?v=iEkCHqSfMco for the system in operation. The corresponding Java sources and incremental Petri net stages for Fig. 8 are at vladestivill-castro.net/additions.tar.gz as well as material from (Billington et al., 2010).

Billington, D. and Rock, A. (2001). Propositional plausible logic: Introduction and implementation. *Studia Logica*, 67:243–269. ISSN 1572-8730.

Blair, G., Bencomo, N., and Frnce, R. B. (2009). Models@run.time. *IEEE Computer*, 42(10):22–27.

Brooks, R. (1991). Intelligence without reason. In Mylopoulos, R. and Reiter, R., editors, *Proceedings of the 12th International Joint Conference on Artificial Intelligence*, pages 569–595, San Mateo, CA. ICJAI-91, Morgan Kaufmann Publishers. Sydney, Australia. ISBN 1-55860-160-0.

Compton, P. and Jansen, R. (1990). A philosophical basis for knowledge acquisition. *Knowledge Acquisition*, 2(3):241–257. ISSN 0001-2998.

Dromey, R. G. and Powell, D. (2005). Early requirements defect detection. *TickIT Journal*, 4Q05:3–13.

Ellis, C. (1997). Team automata for groupware systems. In *GROUP '97: Proceedings of the international ACM SIGGROUP conference on Supporting group work*, pages 415–424, New York, NY, USA. ACM.

Genrich, H. J. (1991). Predicate/transition nets. In Jensen, K. and Rozenberg, G., editors, *High-level Petri Nets, Theory and Applications*, pages 3–43. Springer-Verlag.

Genrich, H. J. and Lautenbach, K. (1979). The analysis of distributed systems by means of predicate/transition-nets. In Kahn, G., editor, *Semantics of Concurrent Computation, Proceedings of the International Symposium*, volume 70 of *Lecture Notes in Computer Science*, pages 123–147, Evian, France. Springer.

Girault, C. and Valk, R. (2001). *Petri Nets for System Engineering: A Guide to Modeling, Verification, and Applications*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

Gold, R. (2004). Petri nets in software engineering. Arbeitsberichte Working Papers. Fachhochschule Ingolstadt, University of Applied Sciences.

Holloway, L., Kroch, B., and Giua, A. (1997). A survey of Petri net methods for controlled discrete event systems. *Discrete Event Dynamic Systems: Theory and Applications*, 7:151–190.

Hull, E., Jackson, K., and Dick, J. (2005). *Requirements Engineering*. Springer, USA, second edition.

Lakos, C. (2001). Object oriented modeling with object petri nets. In Agha, G., de Cindio, F., and Rozenberg, G., editors, *Concurrent Object-Oriented Programming and Petri Nets, Advances in Petri Nets*, volume 2001 of *Lecture Notes in Computer Science*, pages 1–37. Springer.

Larman, C. (1995). *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*. Prentice-Hall, Inc., Englewood Cliffs, NJ.

Lian, J., Hu, Z., and Shatz, S. M. (2008). Simulation-based analysis of UML statechart diagrams: Methods and case studies. *The Software Quality Journal*, 16(1):45–78.

Mellor, S. J. (2007). Embedded systems in UML. OMG White paper. www.omg.org/news/whitepapers/ label: We can generate Systems Today.

Myers, T. and Dromey, R. G. (2009). From requirements to embedded software - formalising the key steps. In *20th Australian Software Engineering Conference (ASWEC)*, pages 23–33, Gold Cost, Australia. IEEE Computer Society.

Nuseibeh, B. and Easterbrook, S. M. (2000). Requirements engineering: a roadmap. In *ICSE - Future of SE Track*, pages 35–46.

Peterson, J. L. (1977). Petri nets. *Computer Surveys*, 9(3):223–252.

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

Russell, S. and Norvig, P. (2002). *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Inc., Englewood Cliffs, NJ, second edition. ISBN 0130803022.

Saldhana, J. A. and Shatz, S. M. (2000). Uml diagrams to object petri net models: An approach for modeling and analysis. In *International Conference on Software Engineering and Knowledge Engineering (SEKE)*, pages 103–110, Chicago.

Shlaer, S. and Mellor, S. J. (1992). *Object lifecycles : modeling the world in states*. Yourdon Press, Englewood Cliffs, N.J.

ter Beek, M., . Ellis, C., Kleijn, J., and Rozenberg, G. (2003). Synchronizations in team automata for groupware systems. *Computer Supported Cooperative Work (CSCW)*, 12(1):21–69.

Wagner, F., Schmuki, R., Wagner, T., and Wolstenholme, P. (2006). *Modeling Software with Finite State Machines: A Practical Approach*. CRC Press, NY.

Wagner, F. and Wolstenholme, P. (2003). Modeling and building reliable, re-useable software. In *Engineering of Computer-Based Systems (ECBS-03), IEEE International Conference on the*, pages 277–286, Los Alamitos, CA, USA. IEEE Computer Society.

Wen, L., Colvin, R., Lin, K., Seagrott, J., Yatapanage, N., and Dromey, R. (2007a). "Integrare", a collaborative environment for behavior-oriented design. In Luo, T., editor, *Cooperative Design, Visualization, and Engineering, 4th International Conference, CDVE*, volume 4674 of *Lecture Notes in Computer Science*, pages 122–131, Shanghai, China. Springer.

Wen, L. and Dromey, R. G. (2004). From requirements change to design change: A formal path. In *2nd International Conference on Software Engineering and Formal Methods (SEFM 2004)*, pages 104–113, Beijing, China. IEEE Computer Society.

Wen, L., Kirk, D., and Dromey, R. G. (2007b). A tool to visualize behavior and design evolution. In Di Penta, M. and Lanza, M., editors, *9th International Workshop on Principles of Software Evolution (IWPSE 2007), in conjunction with the 6th ESEC/FSE joint meeting*, pages 114–115, Dubrovnik, Croatia. ACM.

Wiegers, K. E. (2003). *Software Requirements*. Microsoft Press, second edition.

Winter, K., Colvin, R., and Dromey, R. G. (2009). Dynamic relational behaviour for large-scale systems. In *20th Australian Software Engineering Conference (ASWEC 2009)*, pages 173–182, Gold Cost, Australia. IEEE Computer Society.

Wooldridge, M. (2002). *An Introduction to MultiAgent Systems*. John Wiley & Sons, NY, USA. ISBN 047149691X.