ABSTRACT
In this paper we extend finite state machines to allow expressions in Plausible Logic for labelling transitions. As a result, we enable the design of behaviours that incorporate non-monotonic reasoning with a high-level software development tool. Using a cognitive software architecture that supports the efficient implementation of a developing/programming environment, we automatically translate graphical designs of behaviour into executables that run on board autonomous robots. The graphical designs are obtained by demonstrating the transformation of the state machine into a Behavior Tree does not lose information and enhances modularisation of logic descriptions. We illustrate this with a description of the rapid development of the behaviour of a friendly poker player on an Aibo that interacts with humans.

KEY WORDS
Software Architecture, Programming Tools and Languages, Software for Robotics, Software Agents.

1 Introduction
The design of the behaviour of an autonomous robot is a delicate and sophisticated engineering task. One would like to define and document such behaviour with high-level tools that enable graphical models as well as secure and correct code generation. In fact, this has been the motivation for Computer Assisted Software Engineering (CASE) tools. We propose the labelling of transitions in state diagrams with predicates of a non-monotonic logic. Our approach enables the designer to focus on the conditions that will require the robot to move from one state to a new state using the advantages of a description formulated in non-monotonic logic. In particular, the reasoning component of the logic will resolve conflicts in the description, while the descriptive nature of non-monotonic logic relieves the designer from many implementation concerns, particularly the burdens and pitfalls of procedural implementation. This is similar to the motivation that gave birth to logic programming where the software engineer’s focus was on what is to be performed, rather than detailing how it is to be achieved.

We enable the design of robotic behaviours in terms and formalisms that are accessible to humans. This will become significantly more relevant as collaborative applications for teams of autonomous robots in human environments will emerge in the near future. Today’s technologies only handle the complexity of human environments to a very limited extent, but it is expected that in the near future, intelligent integrated systems with the capacity to act within such a complex environment will collaborate with their users in many tasks [24].

We will argue that non-monotonic logics can and should be incorporated into formalisms for the specification, analysis, and design of behaviour. Namely, we incorporate them into the central behavioural artefacts provided by state machines. In this paper, we will describe how developers define behaviours. We give an illustration with an application where robots have multi-modal interactions with humans in a game of poker. We believe this will demonstrate the benefits of our approach for intelligent integrated systems, combining capabilities such as reasoning and planning, voice recognition, image analysis, and motion control. Games are considered a suitable methodology for evaluating robot-human interaction [26] while general game playing is the new frontier of artificial intelligence and agent technology [18]. Our illustrations consider Robotic Soccer and our case study considers an autonomous robot interacting with a human in a competitive environment with incomplete information.1

2 Engineering the behaviour
Most of the talent and know-how of a software engineer is in the construction, analysis, design, and refinement of models. Models help to communicate the requirements, the architecture, the specifications, and naturally the behaviour of a software system. The use of diagrams is fundamental to capture and convey characteristics and descriptions of software. CASE tools enable most of the design by visualisation of models, and in some cases, automatically generate significant amounts of code. In combination with integrated development environments, software development as a whole is enhanced once these models are in place.

1 In game theory, making a decision without knowledge of all the values of variables that determine the state of the environment is labelled as incomplete information; in the literature of agents, this is referred to as an inaccessible environment.
We suggest that, for a robotic system interacting with humans, the description of the behaviour should incorporate a formal logic, in particular, non-monotonic logic. We also suggest how these logic descriptions can be integrated with a large family of methodologies for describing and visualising software system behaviour. For this purpose, we will introduce diagrams that use logic to describe behaviour. We introduce these diagrams into artefacts that are well accepted within the software engineering community. Our aim is that the introduction of these logics into software engineering modelling tools will facilitate a seamless integration of our approach into current practices.

We describe our approach using one fundamental modelling technique, namely finite states machines2. Finite state machines are standard models of automata. In the early 90s state machines with a single thread of control became the instrument of choice to model the behaviour of each object in the same class. The Object-Modeling Methodology (OMT) of Rumbaugh et al. [16, 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]. Parallel finite state machines 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. This paper does not address modelling communication in multi-threaded systems. We will focus only on declarative models for those components of a system that have a single thread of control. Thus, software engineering approaches for concurrency and synchronous languages [2] are discussed elsewhere.

A finite state machine 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 protocols in computer networking [8]. As a software-modelling instrument, it is used to describe the behaviour of a class in the Object-Oriented Methodology (OMT) [16, Chapter 5]. As a result, it remains the main artefact for modelling the behaviour of a class3 in the Unified Modelling Language (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 with very specific formal semantics, and also appears in the literature in Computer Supported Collaborative Work (CSCW), namely Team Automata [21, 9] (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 [14]. In summary, finite state machines remain a core tool for modelling software behaviour.

However, a fundamental aspect of finite-state machines 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 machine, 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 $c_i \land c_i = \text{false} \lor j \neq i$ (the exclusivity condition). When a finite-state machine is used for language recognition, if the automaton has a state where $\bigvee_{i=1}^n c_i = \text{true}$, is 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 $\neg \bigvee_{i=1}^n c_i$. While formally our logic will achieve something similar, the priority relation of the logic can aid the system engineer with identifying the default cases. When the number $n$ of transitions out of a state is large, the above automatic completion may not be what the behaviour designer attempted to capture. Non-monotonic logic models are better suited for incremental elicitation of which conditions leave a state and which keep control in the same state.

Thus, the designer of behaviour faces the difficult challenge of describing and ensuring the validity of exclusive and exhaustive conditions that trigger the departure of each state. We automate such validation4. We propose to use a non-monotonic logic to describe the conditions that trigger a change of state. Common-sense reasoning of non-monotonic logics will ensure that the behaviour has an exhaustive and exclusive interpretation even if the description does not seem so (recall how instructions issued by human beings always require common sense in interpretation to avoid cumbersome “obvious” details). We generalise the language that guards transitions from symbols of events to entire reasoning exercises expressed in logics.

3 Behaviour design

The strategy we suggest to achieve the design of complex behaviour is to design simple, decomposable behaviours. These can be refined by sub-behaviours (the details of automatic translation into an object oriented language, such as Java or C++, are described elsewhere [6]).

---

2Our approach also has been applied with the asynchronous modelling tools provided by Petri nets, but this will not be described here.

3Conversely, the class diagram is the artefact to model the static structure of an object-oriented system.

4Validation is used in the sense that the design represents a well-defined behaviour.
Consider the state-diagram in Fig. 1. Suffice it to say that the circles (or ellipses) represent states, and the arrows represent transitions. Transitions are guarded by events. In the states, we carry out actions (motions) or invoke sub-behaviours. In UML, a state-diagram models the behaviour of only one class (it is the behaviour for all objects of that class). We do not intend to model interactions with other objects. For such protocols, UML uses Sequence Diagrams and Collaboration Diagrams. Our architecture [6, 10] is a cognitive architecture [17] where a working memory or whiteboard [12] enables to migrate from one state to the next, analogously to production systems. For the SONY Aibo, this is implemented explicitly in order to mediate between concurrent modules, while the Aldebaran Nao system architecture supports complex distributed object status exposure.

We have developed visual tools to design behaviours and reasoning mechanisms that control such behaviours. A table representation can be used to serialise a behaviour and to communicate it to robots. In fact, the robots can have a behaviour execution engine. That is, rather than compile behaviours statically, it is entirely possible to describe behaviours in ASCII files.

3.1 Modelling complex behaviours by team automata

The fact that the behaviours we expect to deploy are very complex demands a mechanism (a method) to manage this complexity. Although closely related, our objective here is behaviour specification and not software requirements specification. Therefore, we will not use formalisms for inter-component preconditions and postconditions. More importantly, the use of finite state machines (as the fundamental artefacts) has shown that the end result has the same expressive power. We will briefly mention “Behavior Trees” [23], because this formalism for capturing requirements has a visual hierarchical approach for focusing on only one state at a time and its possible subsequent states. Behavior Trees combine the expressive power of finite state machines and Petri nets. For every finite state machine there is a Behavior Tree. For every Petri net, there is a Behavior Tree with threads. Also, for every Behavior Tree without threads there is a state machine, and for every Behavior Tree with threads there is a Petri net.

In UML, the state machine formalism is well adopted, and thus is perhaps more ubiquitous. In particular, it is possible to iteratively refine behaviours to become more detailed and more elaborate. This is the intention of the notion of sub-behaviour. We illustrate this by refining the simple behaviour of the Ball_Chaser presented in Fig. 1. Suppose that we want to carry out a slightly more sophisticated search for the ball when it is not visible. In RoboCup with the Sony AIBO, usually the ball is not visible because it just went below the robot’s head. So rather than making the robot spin until the ball becomes visible, we should take a small step back. If the ball still remains invisible, we would spin for two seconds (approximately a full circle). Then we take another step back, and so on (see Fig. 2). This ball searching behaviour may be useful in its own right in many other settings, so we actually define it as a sub-behaviour. We illustrate this by refining the simple behaviour of the Ball_Chaser presented in Fig. 1.

Table 1. The transition function as a table.

<table>
<thead>
<tr>
<th>$s_i$</th>
<th>$event_u$</th>
<th>$s_i$</th>
<th>$event_v$</th>
<th>$s_j$</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$s_i$</td>
<td>$event_x$</td>
<td>$s_p$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

We still use the notion of an initial state $s_0$, because, in our infrastructure behaviours must be able to reset to this state. A final state is not required, but behaviours should be able to indicate completion of a task to other modules.\(^5\)

\(^5\)Formally, our finite state machines can be viewed as Büchi automata [22, Page 136]. A Büchi automaton is the extension of a finite state automaton to infinite inputs. It accepts an infinite input sequence if and only if there exists a run of the automaton which visits (at least) one of the final states infinitely often.

\(^6\)Converting an object into a sequence of bits so that it can be stored or transmitted across a network, also called deflating or marshalling.

\(^7\)Behaviours can be loaded dynamically from disk. Even behaviour modification whilst the robot is running is possible by supplying a new description (the table specifying the transition function) over a WiFi link.
to consider all details. If the ball becomes visible, the robot will switch back to follow regardless of the current state in the sub-behaviour. As the Ball_Finder behaviour is improved and refined, the ball_chaser better handles the case when it misses the ball. We do not have the space here to describe other aspects of sub-behaviour composition, such as passing or returning parameters (e.g., an initial or a final state).

### 3.2 Introducing non-monotonic logic

Current practice for modelling with finite state-machines assumes that transitions are labelled by a Boolean condition in a logic with values true or false and that such condition will be decidable (that is, it will always be possible to find the value of the condition guarding the transition). For example, in Fig. 1 a transition is labelled by the event ball_visible. Even if the condition is decidable, the input may be noisy (for example, the ball may have just been missed in the current frame despite being visible for many frames in the past).

Our extension to behaviour modelling starts by extending the transition labels in the table representation of the state machine to any sentence in the non-monotonic logic. Replacing the guarding conditions with statements in non-monotonic logic incorporates reasoning into the reactive nature of state machines. Since our logic models reasoning and modality (e.g., “in this frame, vision believes there is no ball”), they are better suited to model state transitions (they may even fuse contradicting beliefs reported by many sensors and modules), and gracefully handle situations with incomplete (or superfluous) information without increasing the complexity in the mental map 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 out of a state; as priorities can indicate a default transition if conditions guarding other transitions cannot be decided.

The resulting state-diagrams (and the corresponding finite state machines) may be large and cumbersome to display. Here is where we make use of the transformation that shows the equivalence between Behavior Trees and state-diagrams (Behavior Trees can involve more than one component, that is more than one class from the object-oriented model). Consider picking one particular state of a state-diagram, designated as the root, placing that stage at the top level of a drawing. The transitions out of this state are drawn downwards. All those states directly reachable from the root state are placed in a row of states at the next level down. We continue this recursively. This is an embedding by breadth-first search. However, the recursion terminates when we find that we are back to a state already visited. In this case, Behavior Trees do not draw the back-arrow for the transition, but draw a down arrow to a clone of the already visited state (and no further exploration is made). The Behavior Tree corresponding to a state-diagram has the advantage that it is a hierarchical planar graph, and thus, it is drawn without any arrows crossing, and can be presented to users of a CASE tool in essentially the same way as the hierarchical structure of directories, subdirectories, and files. The only thing to remember is that leaves in these diagrams may be actually clones of previous states.

After this step, one can consider a partition of the table describing the transition function (recall Table 1) where the components of the partition are the sub-tables where the departing state is common. That is, one component comprises all the rows of the transition function that have $s_i$ as the departing state. Now we have as many functions as states (mathematical projections), where for each state $s_i \in S$, we have a function $t_{s_i} : \mathcal{Y} \rightarrow S$ given by $t_{s_i}(event_j) = t(s_i, event_j)$ (recall that $\mathcal{Y}$ is the universe of events\textsuperscript{8}). A component of the partition is also simple for the software engineer to conceptualise, because it just describes what conditions and events cause a change from the given state, corresponding to the analysis of one node in a state-diagram (or also one node in a Behavior Tree).

One of the non-monotonic logics that we use in our implementation is Plausible Logic [15, 3] (PL), which is the only non-monotonic logic with an efficient non-looping

\begin{figure}
\centering
\includegraphics[width=0.5\textwidth]{Figure2.png}
\caption{Simple example of a sub-behaviour and its use (composition).}
\end{figure}

\begin{figure}
\centering
\includegraphics[width=0.5\textwidth]{Figure3.png}
\caption{Illustration of a state diagram’s translation into a Behavior Tree.}
\end{figure}
algorithm [3]. Notably, PL distinguishes between formulas proved using only factual information and those using plausible information. The partition of the transition table of a finite state machine for a behaviour can always be codified into a module of the DPL programming language [7] for PL as follows. Suppose (without loss of generality by renumbering states and transitions) that one node of the state-diagram has a source state $s_0$ and that the destination states are $\{s_1, s_2, \ldots, s_k\}$ guarded by transition events $t_{s_0}(e_i) = s_i$, for $i = 1, \ldots, k$. Fig. 4 shows the corresponding PL program. The module name appears in Line 1 (this enables linking with other modules and scheduling execution in different modes). Line 2 describes the set of next states, while Line 3 and Line 4 indicate that only one state can be next. Line 5 lists the events that inputs to the module and Line 6 declares the initial state. Line 7 lists the transitions while Line 8 indicates that a transition takes precedence over remaining in the initial state. This proves that any behaviour described by finite state machines can be coded in PL (although this may not be advisable, as we mentioned, for reasons of efficiency). Nevertheless, the construction above is implemented in our visual tool that enables the development of behaviours as state-diagrams and the automatic generation of code in DPL.

Table 2. Non exclusive transitions.

| $s_0$ | $event_u$ | $s_i$ |
| $s_0$ | $event_v$ | $s_j$ |
| $s_0$ | $event_u \land event_v$ | $s_p$ |

Moreover, using logic enables flexibility. Consider a scenario where there are 3 transitions as in Table 2. Here, the third transition is a special case of the first two, and we do not have the exclusivity condition. In fact, if $event_u$ and $event_v$ are both true, the three transitions fire (for instance in the semantics of generic production systems). But it is possible that the third transition is the result of refining the model. That is, the first two really mean something like “we move from state $s_0$ to $s_1$ if $event_u$ happens and we do not know about $event_v$; and we move from $s_0$ to $s_j$ if $event_v$ happens and we do not know about $event_u$”. With our state-diagrams, we enable the bridging of the exclusivity condition; that is, we enable transitions that are labelled by conjunctions of other events (or labels of other transitions). In such a case, we use the priority relation of PL to resolve the conflict. In one approach, where the conjunction is considered more specific, the three transitions above translate to rules in the logic model as shown in Table 3 (where the priority relations in Line 7 and Line 8 resolve the possible conflict). However, using our validation tool, it is possible to check for each state if two transitions fire, and to request from the system designer to refine the logic model in order to remove the ambiguity (the tool for that would be the priorities among rules).

The visualisation tools for nodes can also handle disjunctions (Table 4 (a)). Since the expressions labelling the transitions can be composed of conjunctions and/or disjunctions, we can label transitions with statements in formal logic. In our case, we chose to label transitions with expressions in PL. This means that the modules in PL not only keep track of the overall finite state machine synchronous flow of control, but modules of PL are also linked in an asynchronous mode as there is absolute freedom in the order of evaluation of the conditions guarding transitions. Thus, for every pair of transitions there is no prescribed order for the evaluation of guarding expressions (in Table 4 (b), $expression_u$ and $expression_v$ are evaluated asynchronously). Moreover, the priority relation on rules in PL can be used by the designer to prescribe an order if so desired. We also enable designers to choose between the evaluation of all expressions, or evaluating only sufficient expressions to determine a follow-up state.

The logic can handle terms and predicates for which it has no information. Thus, the different modules in PL now constitute a rather loosely linked, decoupled software architecture. It emulates a collection or community of agents, each establishing facts if they can prove them from the information available, and the non-monotonic nature of the logic handles the potential conflict between facts.

9There are several alternatives in non-monotonic default logic for prioritising among rules [1, Chapter 8], so our approach applies to these other logics as well.
Table 3. PL code to handle conflict.

| Switch_S_0_to_S_i : \{“e_u”\} ⇒ State(S_j) | Switch_S_0_to_S_i ⇒ Default. |
|---------------------------------------------|
| Switch_S_0_to_S_j : \{“e_v”\} ⇒ State(S_j) | Switch_S_0_to_S_j ⇒ Default. |
| Switch_S_0_to_S_p : \{“e_u”, “e_v”\} ⇒ State(S_p). | Switch_S_0_to_S_p ⇒ Default. |
| Switch_S_0_to_S_p > Switch_S_0_to_S_i. | Switch_S_0_to_S_p > Switch_S_0_to_S_i. |

Table 5. PL expressions as inputs.

\[
\begin{array}{c|c|c}
S_1 & PL_{expression_u} & S_1 \\
S_1 & PL_{expression_v} & S_j \\
... & ... & ... \\
S_i & PL_{expression_x} & S_p \\
\end{array}
\]

3.3 Automation and execution of behaviours designed as diagrams.

We have shown constructively (with an algorithm) that any state diagram with transitions labelled by events (as in Table 1) can be represented by a series of modules in DPL (the programming language for PL) and this translation is performed automatically by our tools. Note that the time complexity of the translation is linear on the number of transitions of the original state-diagram (or the number of rows in the transition table format of the finite state machine, see Table 1). With our extension, the transition table has the format as in Table 5. Moreover, for the same state in column 1 of this table, more than one PL expression may evaluate to true. But then, the table can be additionally decorated by instances of the priority relation of the logic. It is here that we can profit from interpreting the information from sensors (or other reasoning modules) as beliefs in order to combine them for common-sense decision making. Surprisingly, for example, the ranking of the 169 possible initial hands of Texas-Hold’em Poker\(^{10}\) is subject to contention\(^{11}\). Therefore, perhaps it is better, after recognising its hold,\(^{12}\) that the robot represents this fact with formulas such as “vision believes our hand-strength pre-flop\(^{13}\) is in the top 50% of all possible hands”. Sensors may supply contradictory information. In soccer, vision on the robot regularly reports a distance to the ball different from what an IR sensor reports. Therefore, a fact such as “sensor X says that the temperature is above 20” can be modelled as a simple belief using the power of plausible rules. That is, we do not know the temperature of the environment, but we can have a rule that says “if sensor X says temperature is above 20, then usually the temperature is above 20”. So evaluations and measurements of the environment, and comparisons for specific constants and values, can be handled with more flexibility. We can incorporate sensor information on the same aspect of the environment from more than one sensor using plausible rules and priority relations between these rules. Similarly, we can incorporate two or more personalities into a poker player that give potentially contradictory advice on the next action.

For example, our poker strategy is by default a tight\(^{14}\) aggressive\(^{15}\) strategy. However, we monitor the opponent’s moves and collect statistics on their behaviour. If we learn that the current opponent’s behaviour is tight and passive, we become even tighter and more aggressive.

The example of the poker player also illustrates another aspect of our approach. This other aspect is that two modes of execution are derived automatically. Both are feasible when we have only a small number of final states. In particular, it is not uncommon to have stages within the state diagram leading to the same set of outputs. For example, the design may demand a behaviour that has only a few final options at its completion. This is common in games like poker or dominoes, where the external state of IN_MY_TURN has a very limited number of options for finishing the behaviour. For example, an active poker player can only call, raise, or fold (and in some situations, only a subset of these options). In dominoes\(^{20}\), there are also very few legal moves. This pictorially leads to Behavior Trees that start in the initial state of the behaviour IN_MY_TURN and whose leaves are all of the form PLAY_X where X is the legal move. The corresponding state-diagrams and Behavior Trees do not have any transitions leading back. Schematically, the state diagrams look like Fig. 3.

When this happens (no transitions that lead back), our tools that evaluate the large system of PL modules can operate in two modes that are analogous to the two types of

\(^{10}\)Head-on Texas-Hold’em has been an AAAI sponsored benchmark competition.

\(^{11}\)Although there is much agreement, different books, web-sites and digital games implementations, do not completely agree on the ranking.

\(^{12}\)In Texas-Hold’em, a player holds two cards which are not disclosed to opponents and aims at defeat them by constructing a higher hand with any five cards from its hold and the 5 community cards, or forcing opponents to fold their hand by bets that are not matched.

\(^{13}\)Community cards are released in 3 stages, named the flop, the turn, and the river. Before any community cards are known, the betting round is called pre-flop.

\(^{14}\)A player is tight if it gets involved in the betting rarely, as opposed to a loose player who gets involved frequently.

\(^{15}\)A player is aggressive if it raises frequently when it has the opportunity, as opposed to a passive player that just matches (calls) another player’s bet.
automatic reasoning (namely forward chaining and backward chaining). That is, the designer can choose between two ways by which the state diagram translates into C++ code, compiles, and operates on the robot. In the forward chaining approach, the execution follows the execution of the corresponding finite state machine. That is, the PL expressions guarding transitions from the initial state are evaluated, and a new state is determined as a result. Subsequently, the new state is taken as the current state and the expressions guarding outgoing transitions are used to determine the next state. This is repeated until a final state is determined. In the example of our poker player, when it is the robot’s turn, it runs the PL module that determines if the type of opponents has to be changed, e.g. whether or not to switch to a new personality such as playing tighter. Then, this personality tosses a coin to perhaps bluff. The PL description corresponding to this personality is then used to make the final decision on whether to call, fold, or raise.

However, backward chaining is also possible, that is, to execute all the PL modules leading to final states completely, but independently (even asynchronously). Then modules on the previous level synthesise from the current level, and so on, traversing backwards in the transition chain until the initial state is reached. In the poker player example, we may run all styles (personalities) and get a series of “suggestions” on how to play (as if these were experts on the next move). Then, we synthesise back, so that now we can take the advice from the strategy that best counters the type of opponent we believe we are facing, and directly select among the suggested actions to call, raise, or fold. While backward chaining may seem more wasteful, this makes it possible to run the reasoning engine in a state-agnostic fashion. This is an option, as typically the run time of the embedded PL reasoning engine is negligible within the action cycle of the robot. We can therefore easily execute all PL modules, rather than selectively execute them, if so desired.

4 An illustrative example

The expectation that robots would interact with humans inspired Isaac Asimov to write “I Robot” as part of a series of books and to develop the character Susan Calvin who enunciated the “Three Laws of Robotics”:

1. A robot may not injure a human being, or, through inaction, allow a human to come to harm.
2. A robot must obey orders given to him by human beings except where such orders would conflict with the First Law.
3. A robot must protect its own existence as long as such protection does not conflict with the First or Second Law.

These laws illustrate how humans describe a behaviour: They define a general rule, and the next rule is a refinement. Rules further down the list improve the description. This style of development is not only natural, but allows incremental refinement. PL models developed using progressive refinement have successfully been used on board a mobile robot to reason about the objects reported by a vision module in a sequence of scenes [5], to represent the domain knowledge and reason about the off-side rule in soccer [4], and to model the behaviour of a robot raising alarms while considering information about its elderly owner [4].

However, the best prototype of our approach to diagram-based behaviour design was obtained from the development of a robotic poker player. Design and refinement of the strategies for playing was performed using a prototype of a case tool (Fig. 5). The visual designs representing an abstract model of state diagrams enabled us to automatically generate not only C++ code for the SONY Aibo and for a PL reasoning simulator, but we also obtained the equivalent Java code for the official AAAI Texas Hold’em competition. This way, we were able to produce a playing strategy that won against all the publicly available software poker players written in Java.

The robotic poker player was demonstrated at a public open day of the University and an expo for industrial partners and visitors.

5 Final Remarks

Our approach integrates well with key software engineering paradigms. First, the model of agency, proposed as a post-object-orientation paradigm for software development [25]. The agent model suggests negotiation, perhaps interaction through auctions or regulators. Since our behaviours are composed of loosely-coupled modules capable of non-monotonic reasoning, we can model and support several agents who may arrive at rather contradictory conclusions or bid for possibly incompatible actions. The overall system will mediate between them for a global, well-defined behaviour. This was illustrated with the example of a poker player modelled as several experts or personalities that would not necessarily suggest the same action in a particular game scenario. Nevertheless, all can execute and are mediated by a non-monotonic reasoning regulator (similarly, we can model discordant sensors in this way).

Our approach also integrates well with the notion of software patterns. We already suggested that behaviours could be loaded and executed dynamically. This is more
References