pState: A Probabilistic Statecharts Translator

Bojan Nokovic
Computing and Software Department
McMaster University
Hamilton, Ontario, Canada
Email: nokovib@mcmaster.ca

Emil Sekerinski
Computing and Software Department
McMaster University
Hamilton, Ontario, Canada
Email: emil@mcmaster.ca

Abstract—**We describe *pState*, an experimental software toolkit for the design, validation and formal verification of complex systems. Classical statecharts are extended with probabilistic transitions, costs/rewards, and state invariants. Probabilistic choice can be used to model randomized algorithms or unreliable systems. Costs/rewards can be used to compute quantitative properties such as expected power consumption or expected number of lost messages in model of some communication protocol. State invariants are used to express safety conditions or consistency constraints. The charts are validated and transformed into an intermediate representation, from which code for various languages can be generated.**

Keywords - verification; statecharts; quantitative properties; model-checking; invariants

I. Introduction

The work reports on *pState*, a tool for the *holistic* modeling of complex systems; *pState* allows the correctness of a design to be evaluated, quantitative properties to be analyzed, and executable code to be generated.

Statecharts are used to define the behavior of a system by specifying how it reacts to external events. The formalism is an extension of finite state machine by *hierarchy*, *concurrency* and *broadcasting* [1]. Hierarchy is a structuring method that allows the developer to maintain an overview of large and complex applications by allowing to zoom in and out and reveal as much detail as needed; hierarchy allows the design to start with an outline and functionality to be added step by step. Concurrency and broadcasting allow the concurrent nature of complex systems to be naturally modeled. Statecharts are used as a graphical specification tool for reactive systems, but they are also executable and compilable [2].

pState supports *pCharts*, an extension of statecharts with *state invariants*, *probabilistic transitions,* and *costs/rewards*.

State invariants are conditions that are attached to individual states and specify what has to hold in that state [5]. Every incoming transition to the state must ensure that state invariant holds, and every outgoing transition can assume that invariant holds which gives a method for checking a chart against an annotation consisting of invariants attached to states in the state hierarchy. State invariants can express safety of an embedded system or consistency of a software system.

Probabilistic transitions can quantify the amount of certainty and quantitatively describe the randomness of the system and the randomness of its environment. Probabilistic descriptions are useful in requirements engineering and specification of software systems: quality of service, varying workload, randomized algorithms, unreliable environments, and fault-tolerant systems [4]. A probabilistic transition leads from a single state to one of several states depending on a probability distribution [8]. Following our earlier work on *iState* [3], *pState* implements an *event-centric* semantic in which events are procedures, unlike the state-centric semantic of UML and Statemate, in which events are data (in queues) [4, 10, 11, 12]. This is suitable as in our application, design and analysis of low-power wireless systems, events are processed quickly enough so that no queuing of events is necessary. The first contribution of this work is to implement an event-centric semantic of charts with probabilistic transition; the theory is fully described in [9]; here we report on the tool *pState*.

Different types of costs or, equivalently, rewards can be specified, like power consumption, number of failed transmissions, or elapsed time. A theory for costs is given by priced probabilistic automata; a recent overview with model-checking procedures is given in [13]. The second contribution of this work is extending statecharts with costs. The overall architecture of *pState* is shown in Fig. 1.

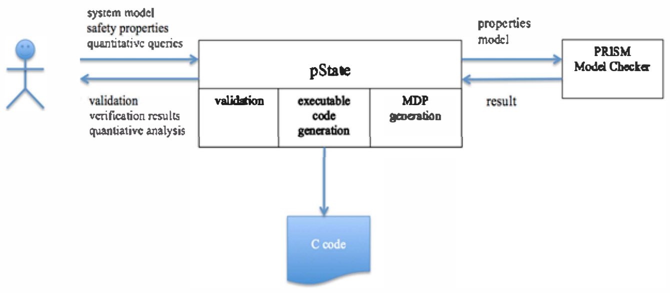
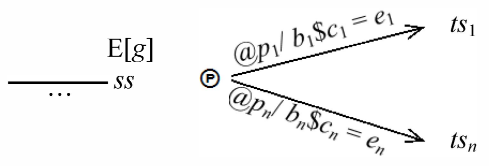


Fig. 1. *pState* Architecture

Quantitative queries properties are specified as temporal logic PCTL [6] formulae. The system model representation is validated and if it is well-formed, executable code can be generated. For verification of state invariants and analysis of quantitative queries, a model file and a properties file are passed to PRISM, a tool for model checking probabilistic systems [7]. The result is displayed in a separate window.

II. Chart Structure

A probabilistic transition consists of a non-empty set *ss* of source states, an event name *E*, an optional guard *g* with Boolean expression *g*, and a non-empty set of probabilistic alternatives. Each probabilistic alternative consists of a probability p_i □ [0..1], a optional body b_i, where each b_i is a statement without loops but possibly with broadcasts, optional cost specifications $c_i = e_i$, where each c_i is a cost name and $e_i \geq 0$ is a real expression, and a non-empty set of target states ts_i. If the source or target consists of more than one state, these originate from or go to concurrent states. Furthermore, the sum of the probabilities of all alternatives must be 1. We use following notation, where the Ⓟ symbol is left out if there is only one probabilistic alternative.



S; E; $C = e$; $i : l..u$; $b : bool$;...
| $i\$c = e$

A state consists of an optional state name *S*, a possibly empty list of declarations, an optional state invariant *i*, a Boolean expression, and optional cost specifications $\$c = e$, where *c* is a cost name and $e \geq 0$ is a real expression. A declaration either declares a local event *E*, a constant $C = e$, where *e* is a constant expression, an integer subrange variable $i : l..u$, where *l*, *u* are constant expressions with $u \geq l$, or a Boolean variable *b* : *bool*. Costs attached to transitions are "one-time" costs, like the decrease of life expectancy of a component when switching on and off or the count of the number of message transmissions. Costs attached to a state depend on the time spent in that state, like the power consumption in standby state and transmitting state.

III. pState Editor

Figure 2 gives a view of the *pState*[1] graphical interface with the example of a TV set. The TV control activity is partitioned into two states, the *Basic* state *Standby* and the *AND* (concurrent) state *Working*. The initial state is *Standby*. When the chart is in *Working* state it is in both the *Picture* and *Sound* states. Within *XOR* (hierarchical) state *Picture* the chart is in one of the *Basic* states *WarmingUp* or *Displaying*, within *XOR* state *Sound*, the system is in one of the *Basic* states *Waiting*, *On*, or *Off*. The invariant of *Working* is that whenever *Picture* is in *Displaying*, *Sound* must not be in *Waiting*, i.e. must be either in *On* or *Off*. The invariant of *Sound* states that the sound level *lev* must be between 1 and 10. The event *power* causes the chart to flip between *Standby* and *Working*, no matter in which substates of *Working* the chart is. The transition on event *warm* broadcasts event *soundOn*. The transition on events *down* can only be taken if *lev* > 1 and when taken, will decrement *lev*. The transition on power to Working sets Picture and Sound to the default initial states *WarmingUp* and *Waiting* and sets *lev* to 5.

The structured editor uses the JHotDraw7.6 open-source framework [14]. The design tool in Fig. 2 consists of the drawing action, state figure, transition figure, initial state figure, probabilistic state figure, concurrency line, formulae box, and comment figure. To make the design more self-explanatory, semi-transparent text box comments can be added. The standard attribute bar with all selections from the JHotDraw framework is also provided, allowing for example color to be added or lines emphasized. Visual elements are added in drag-and-drop fashion using icons in the toolbar. The editor supports the *AND* and *XOR* hierarchy when editing. Unicode characters are used for Boolean and relational operators for readability. An XML based format is used for storing charts. Printing to PDF is also supported, see Fig. 3 for an example of a generated chart.

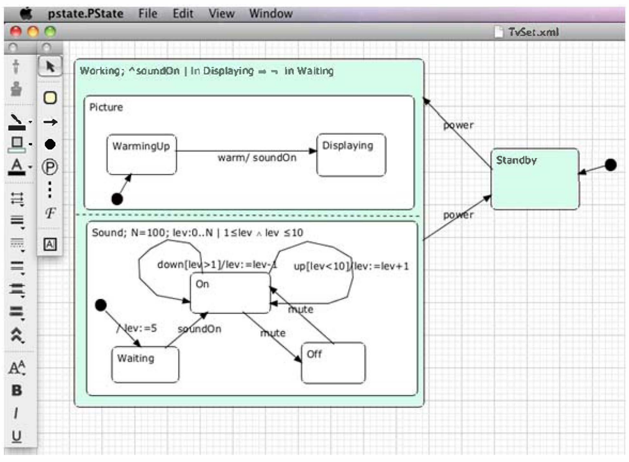


Fig. 2. Statecharts with invariants for TV set

IV. Verifying pCharts

To verify pCharts we use PRISM, a model checker that supports timing, probabilities, and costs [7]. The *pState* constructs for probabilities and costs were designed to allow a translation to PRISM, based on earlier work of translating statecharts with invariants [3, 5]. That work shows how the hierarchical statechart structure can be flattened into a guarded command language like B [15]. Statements in PRISM are a form of probabilistic guarded commands, with *updates* being multiple assignment statements:

$$[]guard \rightarrow prob_1 : update_1 + \ldots + prob_n : update_n;$$

The PRISM models that we use are defined as Markov Decision Processes (MDP) [16]. Currently *pState* generates

[1] pState can be downloaded at http://pstate.mcmaster.ca

textual model file and properties file. A property file is created form state invariants and formulae specified in formulae boxes.

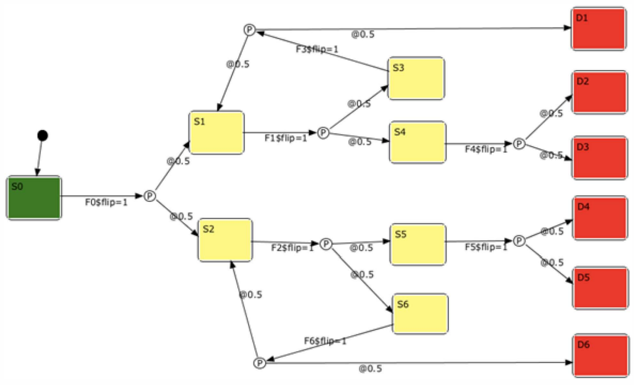


Fig. 3. Casting of a die using fair coin flips

Temporal logic formulae for the model checker are generated from state invariants. The editor allows costs to be specified for each alternative of a probabilistic transition or only once for all alternatives, as in Fig. 3. The cost specifications are extracted from all transitions and a *rewards structure* is generated as part of the PRISM model. The example of simulating the behavior of a standard six-sided die by a fair coin is taken from [6]. The initial state is S0, and states D1, ..., D6 are possible die outcomes. If the outcome for tossing a fair coin is *heads*, the left branch determines the next state. If the outcome is tails, the right branch determines the next state. Tossing the coin in S2 leads with equal probability to either state S5 (from which the die-outcomes D4 or D5 are possible with equal probability) or to state S6. From state S6, a coin flipping yields with probability 1/2 the outcome D6, or with a probability 1/2 return to state S2. When we run the code generated by *pState* with PRISM we get an MDP model with 13 states and 20 transitions. All six final states are equally likely, which can be shown with temporal formulae for the probability of *eventually* reaching D1, ..., D6. The probability formulae

$$Pmin = ?[F\,(root = D4)] \qquad Pmax = ?[F\,(root = D4)] \qquad (1)$$

Both result in 0.1666665; the minimum and maximum probability may differ in presence of nondeterminism.

Generated PRISM code for the die by a fair coin example:

```
mdp

const D3=0; const S6=1; const D6=2; const S2=3; const S3=4;
const D4=5; const S1=6; const S4=7; const D1=8; const S0=9;
const S5=10; const D5=11; const D2=12;

module diebyfaircoin

root :[0..12] init S0;

// Guarded commands
[F1](root=S1) -> 0.5:(root'=S4) + 0.5:(root'=S3);
[F0](root=S0) -> 0.5:(root'=S2) + 0.5:(root'=S1);
[F5](root=S5) -> 0.5:(root'=D4) + 0.5:(root'=D5);
[F4](root=S4) -> 0.5:(root'=D3) + 0.5:(root'=D2);
[F3](root=S3) -> 0.5:(root'=D1) + 0.5:(root'=S1);
[F2](root=S2) -> 0.5:(root'=S5) + 0.5:(root'=S6);
[F6](root=S6) -> 0.5:(root'=S2) + 0.5:(root'=D6);

endmodule

rewards "flip"

[F1]root=S1: 1; [F0]root=S0: 1; [F5]root=S5: 1; [F4]root=S4: 1;
[F3]root=S3: 1; [F2]root=S2: 1; [F6]root=S6: 1;

endrewards
```

Each transition flipping the coin has a flip cost of 1 attached to it. This allows to determine the expected number of coin flips for reaching a final state. The reward formula

$$\begin{aligned} Rmin = ?[F\,(root = D1) \mid (root = D2) \mid (root = D3)| \\ (root = D4) \mid (root = D5) \mid (root = D6)] \end{aligned} \qquad (2)$$

result in 3.666665 for both *Rmin* and *Rmax* which is calculated using same formula (2). Formulae (1) and (2) are easy to write once the reward structures are generated. On the other hand, formulae for the correctness of invariants are automatically generated, for example for the TV chart:

$$\begin{aligned} &P >= 1[G((1 <= lev)\ \&\ (lev <= 10))] \\ &P >= 1[G((picture = Displaying) => \\ &\qquad !((\,sound = Waiting)))] \end{aligned} \qquad (3)$$

When we run on PRISM code generated by *pState* for TV example, we get an MDP model with 60 states and 108 transitions. Formulae (3) are verified in 0.0010 seconds, on MacBook Pro 1.83GHz Intel Code Duo, 2GB SDRAM.

Generated PRISM code for TV set example:

```
mdp

const N=100; const Standby=0; const Working=1; const WarmingUp=0;
const Displaying=1; const Waiting=0; const Off=1; const On=2;

module tvset

root :[0..1] init Standby;
picture :[0..1] init WarmingUp;
sound :[0..2] init Waiting;
lev :[0.. N] init 5;

// Guarded commands
[warm](root=Working)&(picture=WarmingUp)&(sound=Waiting) ->
        (sound'=On)&(picture'=Displaying);
[warm](root=Working)&(picture=WarmingUp)&(sound!=Waiting) ->
        (picture'=Displaying);
[power](root=Working) -> (root'=Standby);
[power](root=Standby) ->
        ( root'=Working)&(picture'=WarmingUp)&(sound'=Waiting);
[down](root=Working)&(sound=On)&(lev>1) ->
        ( lev '=( lev -1))&(sound'=On);
[up](root=Working)&(sound=On)&(lev<10) ->
        (lev'=(lev+1))&(sound'=On);
[mute](root=Working)&(sound=On) -> (sound'=Off);
[mute](root=Working)&(sound=Off) -> (sound'=On);


endmodule
```

V. Compiling pCharts to C Code

The algorithm described in [5] is used to generate executable code. We generate an IF statement for single operation events, and a CASE statement if there is more than one operation on an event. Implementation of parallel composition as sequential composition is automated as it is described in [5]. Validation of pCharts is done in three steps; (1) We check that composite states are not childless. *AND* state must have at least two children, and each child of *AND* state must be *XOR* state. (2) Then we check that all *XOR* states have initial transitions, and (3) we validate concurrent transitions by detecting transitions between concurrent states which are not allowed. Portion of C code generated from TV Set pCharts is shown bellow.

```
enum root_status {Working, Standby} root;
enum picture_status {WarmingUp, Displaying} picture;
enum sound_status {On, Waiting, Off} sound;
int lev;

int main(void){
    /▯ Initialization ▯/
    picture = WarmingUp;
    sound = Waiting;
    lev = 5;
    root = Standby;
    return 0;
}
soundOn(){
    if (( root == Working)) {
            if ((sound == Waiting)) {
              sound = On;
            }
    }
}
warm(){
    if (( root == Working)) {
         if (( picture == WarmingUp)) {
              soundOn();
              picture = Displaying;
         }
    }
}
down(){
    if (( root == Working)) {
         if (( sound == On)&&(lev>1)) {
              lev =( lev −1);
              sound = On;
         }
    }
}
power(){
     switch (root) {
         case (Standby) :
              root = Working;
              picture = WarmingUp;
              sound = Waiting;
              lev = 5;
         break;
         case (Working):
              root = Standby;
         break;
     }
}
...
```

VI. Conclusion

The development of *pState* was motivated by the analysis of a specific kind of RFID tags [17]. So far we have applied *pState* to smaller examples. We are working on including *timed* deterministic and stochastic transitions. This will allow us to verify requirements like "the RFID tag will not be excited more than five times per hour in 95% of cases".

Acknowledgment

We are grateful to the reviewers for the constructive comments.

References

[1] D. Harel, "Statecharts: A visual formalism for complex systems," Sci. Comput. Program., vol. 8, no. 3, pp. 231–274, Jun. 1987.

[2] ——, "Statecharts in the making: a personal account," in Proceedings of the third ACM SIGPLAN conference on History of programming languages, ser. HOPL III. New York, NY, USA: ACM, 2007

[3] E. Sekerinski and R. Zurob, "iState: A statechart translator," in UML 2001 – The Unified Modeling Language, 4th International Conference, ser. Lecture Notes in Computer Science, M. Gogolla and C. Kobryn, Eds., vol. 2185. Toronto, Canada: Springer-Verlag, 2001, pp. 376–390.

[4] D. N. Jansen, "Extensions of statecharts with probability, time, and stochastic timing," Ph.D. dissertation, University of Twente, Enschede, 2003. [Online]. Available: http://doc.utwente.nl/58230/

[5] E. Sekerinski, "Verifying statecharts with state invariants," in 13th IEEE International Conference on Engineering of Complex Computer Systems, K. Breitman, J. Woodcock, R. Sterritt, and M. Hinchey, Eds., IEEE Computer Society, March 2008, pp. 7–14.

[6] C. Baier and J. Katoen, Principles of Model Checking. The MIT Press, 2008.

[7] A. Hinton, M. Kwiatkowska, G. Norman, and D. Parker, "PRISM: A tool for automatic verification of probabilistic systems," in Proc. 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, ser. Lecture Notes in Computer Science, H. Hermanns and J. Palsberg, Eds., vol. 3920. Springer, 2006, pp. 441–444.

[8] R. Segala and N. Lynch, "Probabilistic simulations for probabilistic processes," Nordic J. of Computing, vol. 2, no. 2, pp. 250–273, June 1995.

[9] B. Nokovic and E. Sekerinski, "Analyzing pCharts with probabilistic model checker," April 2013, pp. 1–9, unpublished.

[10] R. Eshuis and R. Wieringa, "Requirements-level semantics for UML statecharts," in Fourth International Conference on Formal Methods for Open Object-Based Distributed Systems. S. Smith and C. Talcott, Eds., vol. 177. Kluwer Academic Publishers, 2000, pp. 121–140.

[11] D. Harel and A. Naamad, "The STATEMATE semantics of statecharts," ACM Trans. Softw. Eng. Methodol., vol. 5, pp. 293–333, October 1996.

[12] Y. Zhao, Z. Yang, J. Xie, and Q. Liu, "Quantitative analysis of system based on extended UML state diagrams and probabilistic model checking," Journal of Software, vol. 5, no. 7, pp. 793 – 800, 2010.

[13] G. Norman, D. Parker, and J. Sproston, "Model checking for probabilistic timed automata," Formal Methods in System Design, vol. 41, no. 2, pp. 1–27, October 2012.

[14] W. Randelshofer, "JHotDraw," http://www.randelshofer.ch/oop/jhotdraw/index.html, December 2012.

[15] J.-R. Abrial, The B Book: Assigning Programs to Meanings. Cambridge University Press, 1996.

[16] M. L. Puterman, Markov Decision Processes: Discrete Stochastic Dynamic Programming. New York, NY, USA: John Wiley & Sons, Inc., 1994.

[17] B. Nokovic and E. Sekerinski, "Analysis of interrogator-tag communication protocols," McMaster University, SQRL Report 60, November 2010.