Automatically Quantitative Analysis and Code Generator for Sensor Systems: The Example of Great Lakes Water Quality Monitoring

Bojan Nokovic and Emil Sekerinski

McMaster University, Computing and Software Department,
Hamilton, Canada
{nokovib,emil}@mcmaster.ca

Abstract. In model-driven development of embedded systems, one would ideally automate both the code generation from the model and the analysis of the model for functional correctness, liveness, timing guarantees, and quantitative properties. Characteristically for embedded systems, analyzing quantitative properties like resource consumption and performance requires a model of the environment as well. We use *pState* to analyze the power consumption of mote intended for water quality monitoring of recreational beaches in Lake Ontario. We show how system properties can be analyzed by model checking rather than by classical approach based on functional breakdown and spreadsheet calculation. From the same model, it is possible to generate a framework of executable code to be run on the sensor's microcontroller. The goal of model checking approach is an improvement of engineering efficiency.

Key words: Water Quality Monitoring; Probabilistic Model Checking; Validation; Verification

1 Introduction

In this work we build a model for and analyze the power consumption of water monitoring motes developed in the MacWater [1] project. The sensors are intended for water quality monitoring of beaches on Lake Ontario, to supplement and speed up the existing practice of manually taking water samples and analyzing them in a lab. For battery-powered motes, power consumption has a main impact on product usability. A shorter battery life requires more frequent battery replacements. As the motes are deployed in buoys (placed on a specific distance from the shore according to local regulations for testing water quality of beaches), there is a significant effort in battery replacements or any kind of maintenance.

The classical approach to power model design is based on a *functional breakdown*. First, power consumption is calculated following a design process similar to the one described in [2, 3]. Next, all activities that are possible sources of power consumption or *logical activities* [4] are identified. Finally power consumption is calculated manually by standard mathematical operations or with the help of standard tools, e.g. spreadsheet.

In our approach the system is first described by pCharts, a visual language for specifying reactive behaviour. Then, after specifying power consumption in relevant states,

input code for a probabilistic model checker is automatically created and power consumption calculated as a *cost* over probabilistic computational tree logic (PCTL) formula. On the example of Waspmotes, commercial Arduino-based motes by Libelium, we present the interaction between the environment and a device as a complex probabilistic timed automaton (PTA), on which it is still feasible to perform quantitative analysis by an off-the-shelf probabilistic model checker. In addition to the calculation of power consumption, we generate the framework of executable code to be run on the microcontroller. We model complex embedded systems, but the code here is executed on microcontrollers with restricted resources.

Application domains of embedded systems go from consumer electronics to telecommunication and transportation industries. Because of that, design and analysis of embedded systems have significant practical interest. The conventional software design process starts with gathering requirements. During the design, a system model is created and from the model, executable code is generated. Testing and performance measurements validate the design. If a problem is discovered, a step back in the design process needs to be taken, the system model is modified, and executable code is generated again. Our goal is to make the whole process less expensive and time-consuming. We present a process in which the whole validation is done on the system design model by model checking. Executable code is generated *after* validation and is guaranteed to satisfy the requirements. We developed a visual formalism for the design and analysis of complex embedded systems, pCharts, and its associated experimental tool, *pState*. Through the *pState* editor, the model, safety properties, and quantitative queries are entered. From the model, executable code for the software part of the system can be generated, safety properties can be verified, and quantitative properties can be analyzed.

In our previous works we introduced the basic features of *pState* [5] and described timed transitions [6]. In [7] we explained how the tool is designed, and we show how a communication protocol for radio-frequency identification (RFID) tags can be analyzed. In [8] we show on few simple examples how properties are specified in an intuitive way such that they can be written without knowledge of temporal logic. In this paper we we are focused on the methodological aspect and show that systems with tens of thousands states can be effectively analysed.

2 Related Work

Statecharts are a modelling notation which captures intuitively the requirements an embedded system has to meet. Formal methods typically address model correctness as they operate on a purely mathematical formalization. This makes it possible to prevent errors inexpensively at early design stages. Different variations of statecharts can be analyzed using Spin [9], NuSMV, SAL, and similar tools. Statecharts with timed transition constructs (clocks, timed guards, and invariants) have been analyzed by model-checking in [10, 11]. A formal semantics in terms of clocked transition systems is given in [10]. A translation of UML statecharts with a timed extension into the input language of the UPPAAL verification tool is given in [11]. The formalization of UML state machine in terms of an operational semantics presented in [12] is implemented in the tool for state machine model checking vUML. Baresi et al. describe model based set of UML

diagrams, called MADES UML diagrams for development of reactive, time critical embedded systems [13]. A formal semantics is presented using metric temporal logic. By a prototype verification tool, charts are translated into the input language of Zot [14], a bounded model/satisfiability checker.

pCharts are translated either into MDP or PTA [15] models and quantitatively verified over constructed PCTL formulas. MDP models are used for the verification of probabilistic and nondeterministic systems and PTA models can verify systems with real-time behaviour [16] by using *clock* variables whose values range over non-negative reals. Clock variables increase at the same rate as time and can be reset. pCharts can be augmented with transition or state quantitative information in the form of *costs*. Model with costs represent priced probabilistic (timed) automata and can be used to reason about properties like (1) minimum/maximum expected cost before some transition will take place, or (2) excepted cost to reach a particular state. To the best of our knowledge, *pState* is the first tool capable of generating input code for a probabilistic model checker from hierarchical states models with costs and stochastic transitions.

3 Chart Structure

A pChart is a hierarchical structure of states with transitions, expressions, types, and statements. The formal definition and a detailed description of pCharts syntax and semantics is given in [5, 6]. In this section, we briefly describe chart transitions, so that the specification in the following case studies will be easier to understand. The visual formalism of pCharts borrows hierarchical states, concurrent states, and broadcasting from statecharts [17] and adds *state invariants*, *probabilistic transitions*, and *costs (or rewards)* attached to both states and transitions.

Probabilistic transitions can be used to express randomized algorithms or to quantify the uncertainty of the environment. Probabilistic descriptions are useful for analyzing quality of service, response time, unreliable environments, and fault-tolerant systems. Quantitative queries can also be attached to any state in a state hierarchy. In general, quantitative queries are specified in the temporal logic PCTL [18] with operators for probabilities and costs/rewards. Due to the presence of nondeterminism, minimum and maximum probability and minimum and maximum cost/reward have to be distinguished. Analysis in *pState* proceeds by generating from the hierarchical, visual model a flattened, textual model together with quantitative queries that are passed to a probabilistic model checker. From charts without timed transitions, *pState* generates a Markov decision process (MDP) model, and from charts with timed transitions, *pState* generates a probabilistic timed automata (PTA) model. As the probabilistic model checker we use PRISM [19].

Transition label, $E[g]\$c = e$, consists of an event name E, an optional guard g with Boolean expression g, and optional cost specifications $\$c = e$, where c is a cost name and $e \geq 0$ is a numerical expression. Probabilistic alternative label, p_i/b_i, consists of a probability $p_i \in [0..1]$, an optional body b_i, where b_i is a statement without loops but possibly with broadcasts. The sum of the probabilities of all alternatives must be 1. If there is only one alternative, probability p_1 is left out.

State label, $S;E;C = e;i : l..u;b : bool;\ldots \mid inv\,\$c = e$, consists of state name S, a possibly empty list of integer and boolean declarations, an optional state invariant *inv*, a Boolean expression, and optional cost specifications $\$c = e$.

4 Waspmote Sensor Power Consumption

We show how pCharts can be used to model the power consumption of the *end-unit* devices, and how to generate the framework for device-executable code. In a collaborative research effort, new sensor types for water quality indicators are developed. For the purpose of this paper, we use commercially available sensors to measure pH of lake water, to read the geographic position of the sensor by GPS, and to transmit data the ZigBee protocol. In our experiment we also use sensors to measure water conductivity, dissolved oxygen, and dissolved ions, which we leave out here for brevity. We show how to specify the impact of the environment on the working device, and how to quantitatively verify that impact. The model in Figure 1 has three concurrent states *Device*, *Environmnet* and *Test*. The state *Device* has itself four concurrent composite states *Board*, *pH*, *ZigBee*, and *GPS*. The state *Device* represents behaviour of

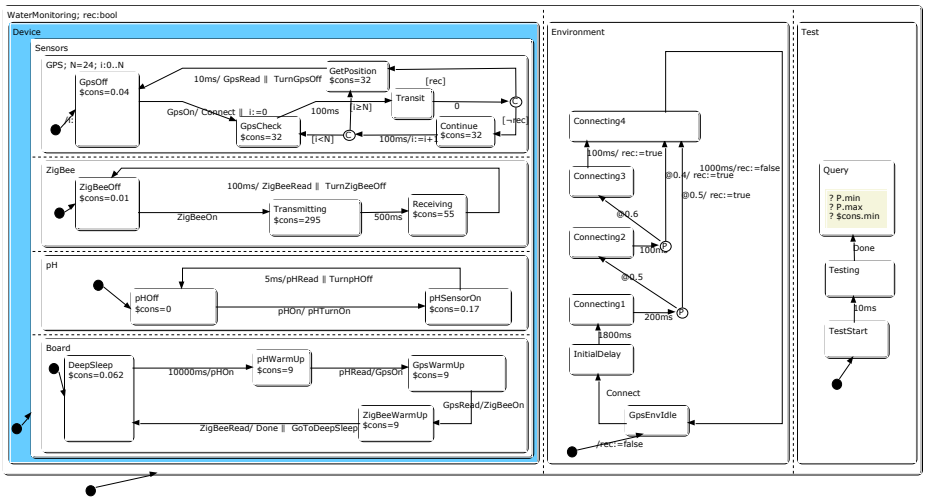

Fig. 1. Wireless Sensor Power Model in pCharts

the Waspmote [22] water monitoring mote. State *Environment* represents the impact of the environment on GPS communication. We add state *Test* to specify queries to be quantitatively verified by the model checker.

Initially, the state *Board* is in *DeepSleep*, state *pH* is in *pHOff*, state *ZigBee* in *ZigBeeOff*, and state *GPS* in the *GPSOff*. Every 10 seconds, *Board* wakes up, and broadcasts the event *pHOn*. On this event *pH* state goes from *pHOff* to *pHSensorOn* and executes the command *pHTurnOn*. This command is a separately written external function. For model checking, it is ignored, but it is used for executable code generation.

In the state *pHSensorOn*, *pH* stays only 5*ms*, to measure water acidity, and then goes back to *pHOff* state. During this process it broadcasts *pHRead* event and call *TurnpHOff* command. On the event *pHRead*, *Board* goes from *pHWarmUp* to *GpsWarmUp*, and broadcasts event *GpsOn*.

On the event *GpsOn*, state *GPS* goes from initial state *GpsOff* to *GpsCheck* and broadcasts event *Connect*. On this event, *Environment* moves form *GpsEnvIdle* to *InitialDelay*. The GPS is used to read a position of the device. In normal operation, based on our measurements, is takes between 1.8*s* and 2.2*s* for GPS to get connected. No connection is possible in less than 1.8*s*, in 50% of the time a connection is establishes between 1.8*s* and 2*s*, in 60% of time between 2*s* and 2.1*s*. By 2.2*s* the connection is always established. This is modelled by probabilistic transitions between *GpsEvnIdle* state and *Connecting4*. When the connection is established, boolean variable *rec* is set to *true*. In our model GPS tries to acquire signal for 4.8*s*, or every 200*ms* for 24 times. Once the connection is established, GPS goes into *GetPosition* state. Consumption in GPS depends on how fast the connection is established, and that is modelled by probabilistic transitions in the *Environment* state. From state *GetPosition*, GPS goes back into *GPSOff*, broadcasts *GpsRead* event and executes the *TurnGpsOff* command. On broadcasted event *GpsRead*, *Board* goes from *GpsWarmUp* to *ZigBeeWarmUp* and broadcasts event *ZigBeeOn*.

ZigBee, a low-power secure networking protocol, is used to transmit the collected readings to a base station, from where data is further transmitted by a 3G connection to a database. We modelled the power consumption in the transmitting and receiving states, for data transmission and acknowledge reception. Once this process is finished *ZigBee* goes back to *ZigBeeOff*, broadcasts *ZigBeeRead* event and executes command *TurnZigBeeOff*. On the event *ZigBeeRead*, *Board* goes from *ZigBeeWarmUp* to *DeepSleep*, broadcast the event *Done* and executes the command *GoToDeepSleep*. The broadcasted event *Done* moves *Test* from *Testing* to *Query* state, where the queries

$$\text{"?}P.min\text{"},\ \text{"?}P.max\text{"},\ \text{"?\$}cons.min\text{"}$$

for *min* and *max* probabilities (*P*), and *min* costs of the consumption (\$*cons*) are specified. They are used for the calculation of the probability that the *Board* will go from initial *DeepSleep* back to *DeepSleep* mode, and to calculate the *consumption* in one cycle. Current consumption is specified in *mA* values according to the specification from Waspmote technical documentation [22]. From the pChart in Figure 1, a PTA model is automatically generated by flattening the hierarchical structure and creating PRISM input code in the form of guarded commands. We outline the translation scheme.

Constants and variables To each Basic state an unique numerical value on its scope is assigned. For instance, on the scope *ZigBee*, states *ZigBeeOff*, *Receiving*, and *Transmitting* are assigned values $0, 1, 2$ respectively. Those values are generated automatically. The value of constant *N* is specified by the designer of the model. Variables are automatically generated for each *XOR* state. For instance *XOR* state Board has five substates, and variable *board* is specified as integer with range 0..3, and initial value is *DeepSleep*, or 2. On the scope of each *XOR* state a special clock variable is generated. Variables *rec* and *i* are defined by the user as boolean and integer in the range 0..*N*; all integer variables must be given a range

Module From the pCharts representation, which consists of three concurrent processes, only one module of PRISM is created. This is achieved by the translation of parallel states into (nested) guarded commands with multiple assignments according to the rules described in [6].

Invariant Timed transition is enabled when clock variable reach value specified on the timed transition. For instance in *Environment*, transition from *Connecting3* to *Connecting4* state happens in 100*ms*. That is specified by clock invariant

$$(environment = Connecting3 => environmentclk <= 100)$$

which indicates that while in *Connecting3*, clock on the environment scope should be less or equal to 100. For each timed transition, an invariant is automatically generated. These invariants are not seen by the user and serve a different purpose than the invariants the user can specify to check the correctness of transitions.

Guarded Command The behaviour is described by commands. By the command

$$[](environment = Connecting3)\&(environmentclk = 100) ->$$
$$(rec' = true)\&(environment' = Connecting4)\&(environmentclk' = 0);$$

the transition form state *Connecting3* to the *Connecting4* happens when *environment* is in *Connecting3* and *environmentclk* is equal to 100. On the transition, *environment* is assigned *Connecting4* and *environmentclk* is reset. The cction on this transition is the assignment of *true* to the variable *rec*. For each event at least one guarded command is created.

Rewards Properties based on *costs* are specified on states or transition. In our example, for each state of *Device*, the cost of consumption *cons* is specified. That is passed to PRISM in the module `rewards ... endrewards`; for instance when the *board* is in *DeepSleep*, the current is only 0.062 *mA*. *Snippet of Generated PRISM Code*[1]

```
pta

const N=24; const ZigBeeOff=0; const Receiving=1; const Transmitting=2;
const GpsWarmUp=0; const ZigBeeWarmUp=1; const DeepSleep=2; const pHWarmUp=3;
...
module watermonitor2

    environment:[0..5] init GpsEnvIdle; environmentclk : clock;
    board:[0..3] init DeepSleep; boardclk : clock;
    gps:[0..4] init GpsOff; gpsclk : clock;
    ...
    rec: bool init false;
    i:[0..N] init 0;

    invariant
```

[1] Full generated code is published on the pState web site `http://pstate.mcmaster.ca/`.

```
        (environment=Connecting3=>environmentclk<=100)
        & (environment=Connecting1=>environmentclk<=200)
        & (environment=InitialDelay=>environmentclk<=1800)
        ...
    endinvariant

    [] (gps=Transit)&(gpsclk=0)->
       (gps'=rec?GetPosition:Continue)&(gpsclk'=0);
    [] (zigbee=Receiving)&(zigbeeclk=100)&(board=ZigBeeWarmUp)
       &(test=Testing)->
       (test'=Query)&(board'=DeepSleep)&(boardclk'=0)&(testclk'=0)&
       (zigbee'=ZigBeeOff)&(zigbeeclk'=0);
     ...
endmodule

rewards "cons"
    (board=GpsWarmUp): 9;
    (board=DeepSleep): 0.062;
     ...
endrewards
```

Results The verification is done by the PRISM *Digital Clock* engine. The created model has 17221 states and 17232 transitions. The calculated minimal and maximal probabilities (*P.min* and *P.max*) to reach *Query* are the same and 1, which means that the test always terminates. There are no nondeterministic transitions, so the *min* and *max* probabilities are equal. The calculated expected minimal consumption (*cons.min*) is 248581.78*mAms*, and it took 126.65s to do calculation. The maximum time of one cycle is a simple sum of deep sleep time (10000*ms*) and the times to read pH (5*ms*), get position of GPS (2210*ms*), and send data by ZigBee (600*ms*) which is 12815*ms*. So, the average current consumption is $248581.78mAms/12815ms = 19.39mA$. Waspmote devices are usually powered by the battery of 6600*mAh*, so according to our calculation the battery can last for approximately 340 *hours*, or 14.1 *days*. Thus we are able to predict automatically the battery life from the model. All properties are verified on Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz, 12.0GB of RAM and on 64-bit Operating System.

4.1 Waspmote Sensor C Code

The executable code is generated only for the part of the model, which does not have probabilistic transitions. We need to selecting *Device* by mouse click on blue state and then we select *View C Code* from main menu.

Snippet of Generated C Code

```
/* Variables */
enum board_status {DeepSleep, pHWarmUp, ZigBeeWarmUp, GpsWarmUp} board;
enum zigbee_status {ZigBeeOff, Receiving, Transmitting} zigbee;
enum ph_status {pHSensorOn, pHOff} ph;
enum gps_status {Continue, GetPosition, GpsOff, Transit, GpsCheck} gps;
```

```
int i;
#define MAX_TIME_EVENTS 8
void exactly4 (long t);
void exactly5 (long t);
void exactly6 (long t);
void exactly7 (long t);
void exactly0 (long t);
void exactly1 (long t);
void exactly2 (long t);
void exactly3 (long t);

int main(void){

    pthread_t threadRun;  // thread variable

    /* initialize data to pass to thread */
    eventdata.n = 0;
    eventdata.tr = false;
    eventdata.ir = true;

    /* create schedule run thread */
    pthread_create (&threadRun, NULL, (void *) &run, (void *) &eventdata);

    /* Initialization */

    board = DeepSleep;
    schedule( &exactly0, 10000, 1);

    zigbee = ZigBeeOff;
    ph = pHOff;
    gps = GpsOff;
    i=0;

    return 0;
}
void exactly0(long t){
    pHOn();
    board = pHWarmUp;
}
void pHOn(long t){
    if ((ph == pHOff)) {
        pHTurnOn();
        ph = pHSensorOn;
        schedule( &exactly3, 5, 1);
    }
}

...
```

At the initialization, the timed event *exactly1* is scheduled in 10000*ms*. When it runs, it broadcasts the event *pHOn* and moves *Board* into *pHWarmUp* state. Broadcasted event *pHOn* moves state *pH* from *pHOff* to *pHSensorOn*, sets a new timed event *exactly3* to be run in 5*ms*, and calls input-output action *PhTurnOn*. This action is hardware dependent and it is part of a prewritten input-output library. All actions which are not broadcasting events, should be defined in the input-output library. Thus we are able to predict automatically the battery life from the model.

5 Conclusions

This paper reports on ongoing work on the pChart formalism and its associated tool, *pState*. For quantitative verification we use the probabilistic model checker PRISM, but the tool architecture allows in principle other probabilistic model checkers like MRMC [20], or some tool from the MoDeSt [21] toolset to be added. The focus in *pState* is on code generation for embedded microprocessors. The goal is to have a seamless and automated approach from modelling and analysis to code generation that can be used by engineers to evaluate design alternatives and to generate trustworthy code. Our overall goal is to support a *holistic approach* in which qualitative properties, notably structural well-formedness, correctness with respect to invariants, and timing guarantees, can be verified together with quantitative properties, notably resource consumption, reliability, and performance. These properties cannot be analyzed by considering exclusively the computerized part; rather, its environment has to be considered to certain extent.

In the case study we presented numerical result of power consumption calculation for sensor motes. The same result can be calculated by some spreadsheet tools, but we believe that our approach is more convenient in a sense that calculation is automatic. On the model we can explore alternative designs and immediately validate the impact on the overall power consumption. Once the design is optimized, the framework of executable code is generated.

One of the principal challenges in quantitative and qualitative verification of real-life systems is scalability of probabilistic model checker. Model can be too big to be verified by analysis of a model's state space. One of the way to overcome state-space explosion is an abstraction in which parts of the model that are not relevant for a particular property are taken out. Another approach is statistical verification, in which property satisfiability is given with some probability. With the development of new generations of model checkers scalability improves, and we believe that both qualitative and quantitative verification will become an inevitable part of software modelling tools in near future.

References

1. McMaster, "MacWater," http://macwater.mcmaster.ca:8080/, June 2015.
2. L. Negri, M. Sami, Q. D. Tran, and D. Zanetti, "Flexible power modeling for wireless systems: Power modeling and optimization of two Bluetooth implementations," in *Proc. 6th*

IEEE International Symposium on World of Wireless Mobile and Multimedia Networks, J. Cantarella, Ed. IEEE Computer Society, 2005, pp. 408–416.
3. L. Negri and A. Chiarini, "Power simulation of communication protocols with StateC," in *Applications of Specification and Design Languages for SoCs*, A. Vachoux, Ed. Springer, 2006, pp. 277–294.
4. M. Mura, M. Paolieri, F. Fabbri, L. Negri, and M. G. Sami, "Power modeling and power analysis for IEEE 802.15.4: a concurrent state machine approach," *Consumer Communications and Networking Conference*, pp. 660–664, 2007.
5. B. Nokovic and E. Sekerinski, "pState: A probabilistic statecharts translator," in *Embedded Computing (MECO), 2013 2nd Mediterranean Conference on*, 2013, pp. 29–32.
6. ——, "Verification and code generation for timed transitions in pcharts," in *Proceedings of the International C* Conference on Computer Science and Software Engineering*, ser. C3S2E '14. New York, NY, USA: ACM, 2014, to appear.
7. ——, "Analysis and implementation of embedded system models: Example of tags in item management application," in *W01 1st Workshop on Model-Implementation Fidelity (MiFi), Grenoble, France*, 2015, p. 10.
8. ——, "A holistic approach to embedded systems development," in *2nd Workshop on Formal-IDE, Oslo, Norway*, 2015, p. 14.
9. E. Mikk, Y. Lakhnech, M. Siegel, and G. J. Holzmann, "Implementing Statecharts in PROMELA/SPIN," in *Proceedings of the Second IEEE Workshop on Industrial Strength Formal Specification Techniques*, ser. WIFT '98. Washington, DC, USA: IEEE Computer Society, 1998, pp. 90–.
10. C. Petersohn and L. Urbina, "A timed semantics for the statemate implementation of statecharts," in *FME '97: Industrial Applications and Strengthened Foundations of Formal Methods*, ser. Lecture Notes in Computer Science, J. Fitzgerald, C. Jones, and P. Lucas, Eds. Springer Berlin Heidelberg, 1997, vol. 1313, pp. 553–572.
11. A. David, M. O. Möller, and W. Yi, "Formal verification of UML statecharts with real-time extensions," in *Fundamental Approaches to Software Engineering, 5th International Conference, FASE 2002*, ser. LNCS, R.-D. Kutsche and H. Weber, Eds., vol. 2306. Springer, 2002, pp. 218–232.
12. J. Lilius and I. P. Paltor, "Formalising UML state machines for model checking," in *Proceedings of the 2nd international conference on The unified modelling language: beyond the standard*, ser. UML'99. Berlin, Heidelberg: Springer-Verlag, 1999, pp. 430–444.
13. L. Baresi, A. Morzenti, A. Motta, and M. Rossi, "A logic-based semantics for the verification of multi-diagram uml models," *SIGSOFT Softw. Eng. Notes*, vol. 37, no. 4, pp. 1–8, Jul. 2012.
14. M. Pradella, A. Morzenti, and P. San Pietro, "Refining real-time system specifications through bounded model- and satisfiability-checking," in *Automated Software Engineering, 2008. ASE 2008. 23rd IEEE/ACM International Conference on*, 2008, pp. 119–127.
15. G. Norman, D. Parker, and J. Sproston, "Model checking for probabilistic timed automata," *Formal Methods in System Design*, pp. 1–27, 2012.
16. M. Kwiatkowska, G. Norman, and D. Parker, "Stochastic games for verification of probabilistic timed automata," in *Proc. 7th International Conference on Formal Modelling and Analysis of Timed Systems (FORMATS'09)*, ser. LNCS, J. Ouaknine and F. Vaandrager, Eds., vol. 5813. Springer, 2009, pp. 212–227.
17. D. Harel, "Statecharts: A visual formalism for complex systems," *Sci. Comput. Program.*, vol. 8, no. 3, pp. 231–274, Jun. 1987.
18. C. Baier and J. Katoen, *Principles of Model Checking*. The MIT Press, 2008.
19. M. Kwiatkowska, G. Norman, and D. Parker, "Prism 4.0: Verification of probabilistic real-time systems," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, G. Gopalakrishnan and S. Qadeer, Eds., vol. 6806. Springer Berlin Heidelberg, 2011, pp. 585–591.

20. J.-P. Katoen, I. S. Zapreev, E. M. Hahn, H. Hermanns, and D. N. Jansen, "The ins and outs of the probabilistic model checker MRMC," in *Proceedings of the Sixth International Conference on the Quantitative Evaluation of Systems, QEST '09*. Los Alamitos: IEEE Computer Society Press, September 2009, pp. 167–176.
21. A. Hartmanns, "Modest - a unified language for quantitative models," in *Specification and Design Languages (FDL), 2012 Forum on*, Sept 2012, pp. 44–51.
22. L. C. D. S.L., "Waspmote," http://www.libelium.com/, July 2014.