Exploring the Twin Peaks using Probabilistic Verification Techniques[*]

Anitha Murugesan[†]
anitha@cs.umn.edu

Lu Feng[*]
lufeng@cis.upenn.edu

Mats P. E. Heimdahl[†]
heimdahl@cs.umn.edu

Sanjai Rayadurgam[†]
sanjai@cs.umn.edu

Michael W. Whalen[†]
whalen@cs.umn.edu

Insup Lee[*]
lee@cis.upenn.edu

[†]Department of Computer Science and Engineering
University of Minnesota
200 Union St. S.E., Minneapolis, MN 55455, USA

[*]Department of Computer and Information Science
University of Pennsylvania
3330 Walnut St., Philadelphia, PA 19104, USA

ABSTRACT

System requirements and system architecture/design co-evolve as the understanding of both the problem at hand as well as the solution to be deployed evolve—the Twin Peaks concept. Modeling of requirements and solution is a promising approach for exploring the Twin Peaks. Commonly, such models are deterministic because of the choice of modeling notation and available analysis tools. Unfortunately, most systems operate in an uncertain environment and contain physical components whose behaviors are stochastic. Although much can be learned from modeling and analysis with commonly used tools, e.g., Simulink/Stateflow and the Simulink Design Verifier, the SCADE toolset, etc., the results from the exploration of the Twin Peaks will—by necessity—be inaccurate and can be misleading; inclusion of the probabilistic behavior of the physical world provides crucial additional insight into the system's required behavior, its operational environment, and the solution proposed for its software. Here, we share our initial experiences with model-based deterministic and probabilistic verification approaches while exploring the Twin Peaks. The intent of this paper is to demonstrate how probabilistic reasoning helps illuminate weaknesses in system requirements, environmental assumptions, and the intended software solution, that could not be identified when using deterministic techniques. We illustrate our experience through a medical device subsystem, modeled and analyzed using the Simulink/Stateflow (deterministic) and PRISM (probabilistic) tools.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Model checking*

[*]This work has been partially supported by NSF grants CNS-0931931 and CNS-1035715.

General Terms

Verification

Keywords

Requirements, Architecture, Model based verification.

1. INTRODUCTION

As a development project progresses, is well known that deepening the understanding of both the problem at hand as well as the solution to be deployed leads to a co-evolution of the system requirements and system architecture/design—an exploration of the Twin Peaks [15].

Model based approaches, e.g., Simulink/Stateflow [8, 10], Rhapsody [1], or SCADE [3] are commonly used for this Twin Peaks exploration. Some approaches are supported by formal verification tools allowing an engineer to verify whether or not desirable properties (requirements) of a model are satisfied, e.g., the MathWorks Simulink Design Verifier (SLDV) [9] or the Gryphon toolsuite [13] developed at Rockwell Collins. These verification tools are generally based on some form of model-checking technology [2] and perform a deterministic determination of whether or not a property is satisfied by a model; either the property is true or it is false—there is no notion of the property being true "most of the time". Although such analysis can be extraordinarily useful in practice [11, 13], the results from the exploration of the Twin Peaks using traditional verification tools will—by necessity—be inaccurate and can be misleading. For example, we may perform the verification of our software models under a single-failure assumption for the various sensors and actuators interacting with our system. Given this assumption, we may perform a verification of the software demonstrating that—under this assumption—the requirements are met. Since we know that this assumption is wrong—there can be multiple failures, however unlikely—we do not know for certain how our software will really behave in the face of such unlikely events; it is quite possible that the "verified" software has a low likelihood of meeting its desired requirements under realistic operating conditions. Hence, inclusion of the stochastic behavior of the physical world provides crucial additional insight into the system's required behavior, its operational environment, and the solution proposed for its software.

In this paper, we share our inial experiences with model-based deterministic and probabilistic verification approaches while exploring the Twin Peaks. We illustrate our experiences using a Generic Patient Controlled Analgesic (GPCA) infusion pump's empty reservoir alarm system, modeled and analysed using the Simulink/Stateflow [8, 10] (deterministic) and PRISM [7] (probabilistic) modeling and verification approaches. The function of the sub-system considered in this paper is to raise an alarm if the remaining volume of drug in the reservoir is below a certain threshold. The possibility of sensor failures allows us to design multiple simple sensor filtering algorithms to reduce the false alarm rate (raising an alarm while there is still drug remaining in the reservoir) while maintaining the capability to detect and rase an alarm when the drug is running low. As will be shown, approaches that are seemingly sensible and can be verified to be correct using a traditional model checker may indeed have surprising behaviors when evaluated in a stochastic environment. Although we use a highly simplified version of the medical device subsystem, we show how exploration through traditional verification can give misleading results (or no useful results at all) and incorporating probabilistic reasoning can help illuminate weaknesses in system requirements, environmental assumptions, and the intended software solution.

We provide an overview of the case example in Section 2, followed by a brief description of the deterministic modeling and verification using Simulink/Stateflow tools in Section 3. In Section 4 we outline the probabilistic reasoning over the system using PRISM. We discuss our our findings and conclude in Section 5.

2. CASE EXAMPLE—THE GPCA

Infusion pumps are medical cyber physical systems used for controlled delivery of liquid drugs into a patient's body according to a physician's prescription (the set of instructions that governs infusion rates for a medication). In an typical infusion system, the patient receives the drug through a intravenous needle inserted into the patient's body and a clinician operates the device. Unfortunately, infusion pumps have been involved in numerous incidents [4] and hence there has been initiatives to improve their safety. To reduce the occurrence of critical accidents such as over- or under-infusion, modern infusion pumps are equipped with various kinds of sensors to detect and notify the clinician when hazards occur.

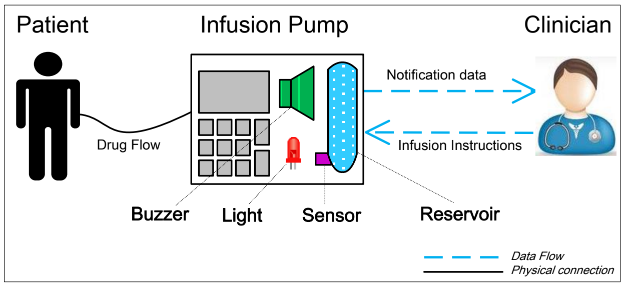

Figure 1: Infusion System Overview

Figure 1 shows a Generic Patient Controlled Analgesia (GPCA) infusion pump device in a typical usage environment, a hospital. In this paper we focus on the infusion pump's reservoir sensor sub-system. The infusion pump has a drug reservoir that stores liquid drug to be delivered to the patient. The pump is equipped with a reservoir sensor that senses the drug volume in the reservoir and reports if the volume is below a certain threshold. Based on the sensor report, the system's software commands aural and/or visual alarms to notify the clinician. The rationale for this sub-system is to avoid the hazard caused to a patient when the reservoir becomes empty while infusing a drug.

3. SIMULINK/STATEFLOW MODELING

As a part of a large ongoing initiative to analyse the complete infusion system[14], we modeled the system to better understand the requirements and verify that the requirements are met by our model. Our choice of modeling notation and tools was MathWork's Simulink/Stateflow, that allows graphical modeling and simulation of dynamic systems. The reason for our choice of Simulink/Stateflow tool is manyfold. First, its a widely used tool for modeling cyber-physical systems and since our effort was focused on providing a reference artifact that demonstrates effective modeling and verification for the infusion system as a whole, we choose a tool that is widely used. Second, the graphical modeling and expressibility provided by the notation is well suited to capture, analyse, and evaluate discrete and continuous control system behaviors. Finally, the tools provide a rich ecosystem of related tools, such as code generators, formal verification tools, test generators.

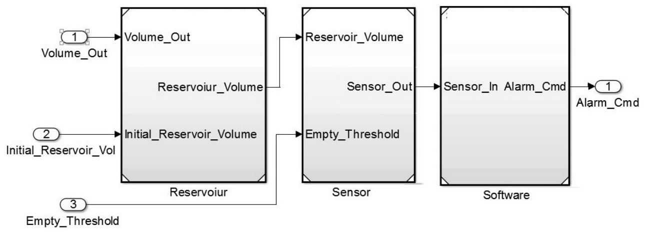

Figure 2: System Architecture in Simulink

As mentioned earlier, we in this paper include a small (and highly simplified) subset of the GPCA behaviors—the empty reservoir alarm sub-system. The simple sub-system architecture with three components: a Reservoir, a sensor and the alarm software, is shown in Figure 2. Each component is modeled as a Simulink block that has a state machine describing its functions. The high level description of the components are:

Reservoir: The reservoir holds the drug. The initial volume of the drug in the reservoir is set to some initial volume and the drug volume is then modeled as a non-increasing function with a floor of zero. In our simple model, there is no provision for refilling the reservoir.

Sensor: The sensor monitors the drug volume in the reservoir. When the volume drops below a certain threshold[1], the sensor outputs a boolean value: true representing that the volume is below the threshold and false otherwise. In our initial model, we assumed a perfect sensor with no failures.

[1]To avoid hazard of reservoir volume becoming zero before an alarm is raised—considering the delays of the hardware and software—the threshold value is set slightly above zero.

Table 1: Deterministic Modeling and Analysis

Case	Component Behavior	Requirement	Results
1	**Sensor:** Perfect. **Software:** Raise alarm on first "empty" reading.	vol <= threshold ⇒ Alarm	Both Satisfied
		vol > threshold ⇒ No Alarm	
2	**Sensor:** Single bit upset. **Software:** Raise alarm on two consecutive "empty" reading.	vol <= threshold ⇒ Alarm in next steps	Neither Satisfied. **Counter example:** Sensor fails every other reading.
		vol > threshold ⇒ No Alarm	
3	**Sensor :** Random failures. **Software :** Raise alarm on two consecutive "empty" reading.	vol <= threshold ⇒ Alarm in next steps	Neither Satisfied. **Counter example:** Sensor outputs always incorrect
		vol > threshold ⇒ No Alarm	

Software: The software issues an alarm based on the sensor output. For the initial system—since we assume a perfect sensor—the software issues an alarm command if the sensor output is true. Again, in this simplified model there is no provision to reset the alarm. The clinician is expected to refill the drug and restart the system and that will clear the alarm.

In our work with the full GPCA models, we verified numerous required properties (formalized requirements) using MathWorks' Simulink Design Verifier [9]. Two system requirements of interest in this paper are:

Req 1 *If there is insufficient drug in the reservoir (drug volume <= "threshold"), the alarm shall be raised.* and

Req 2 *If there is sufficient drug in the reservoir (drug volume > "threshold") the alarm shall not be raised.*

In short, raise the alarm when needed and do not provide nuisance alarms. Although the properties verified trivially for the (extremely) simple initial model, it is quite obvious that the model is an oversimplification of the actual behaviors of the system. In reality, the sensors have failures.

Let us consider two simple failure models for the sensor: (1) a single bit upset were the sensor might report empty when the reservoir in fact is not, but the next reading will always be correct, and (2) arbitrary failures of the sensor, it can report an erroneous value any time and as many times as it pleases. Since the verification was done in SLDV, there was no provision to model the failure in probabilistic terms, it was simply modeled as an arbitrary choice—fail or do not fail. Given sensor failures, simply raising an alarm as soon as the sensor reports "empty" is no longer feasible, we may miss situation where we should raise an alarm and/or the false alarm rate may be unacceptable. The various cases we investigated with SLDV are summarized in Table 1. Case 1 is the simplistic case discussed above.

In Case 2, we introduce a sensor that can exhibit a single bit (and single step) fault. We also refine the software detection algorithm to require two consecutive readings of "empty" to raise the alarm. This change in our assumption about the sensors necessitates a change in the required behavior of the system; we can no longer require that an alarm be raised when the volume becomes too low, we must allow for a small delay to accommodate our simple sensor filtering algorithm. Thus, the required behavior (after appropriate negotiation with the stakeholder of course) is modified to require that the alarm shall be raised within two time steps after the volume drops below the threshold. With this change, we eliminate any false positives. Unfortunately, there is a possibility for a false negative - if the sensor fails in every other step after the volume has dropped below the threshold.

Case 3 introduces a sensor that can fail in any arbitrary step and none of our properties can be made to hold with just one sensor. To address this failure mode, we would have to introduce several redundant sensors and make assumptions on common model failures; a proposition our stakeholder are not too keen on since it would drive up the cost of our device and make maintenance more difficult. At this point, the benefits of formal verification are limited; we know we cannot prove that our requirements will hold under any realistic failure model (sensors will fail, but the really damaging combinations of failures may be exceedingly rare), but we would still like to determine if our system, e.g., the system with the two consecutive sensor inputs filtering algorithm, is good enough to be fielded. Talking to our imaginary stakeholder, a reliability rate in the high 90% seems acceptable. Since there are several viable sensors on the market and some of them seem to acceptable failure behaviors, we would like to explore if we can achieve this performance without excessive increases in the cost of our device. To perform this exploration, we must move away from the "traditional" verification techniques such as model-checking and theorem proving and continue our exploration using probabilistic techniques.

4. PROBABILISTIC ANALYSIS

Probabilistic model checking is a formal verification technique used for modelling and analysing systems that exhibit probabilistic behavior.

We use PRISM [7], the currently leading software tool in the area of probabilistic model checking, to continue our exploration and analysis of the reservoir sensor system. PRISM is a free and open-source probabilistic model checker developed at the University of Birmingham and the University of Oxford. Several types of probabilistic models including discrete-time Markov chains (DTMCs), continuous-time Markov chains (CTMCs), Markov decision processes (MDPs), probabilistic automata (PAs) and probabilistic timed automata (PTAs) can be built and analysed using PRISM. All of these models are specified using the PRISM modeling language, a simple high-level language for model description based on a guarded command notation. We model the reservoir sensor system as a DTMC.

The move from Simulink/Stateflow to PRISM was in this initial study done manually. Given our experience with extraction of models from Simulink and Stateflow, automating this process would be relatively straight forward (but quite time consuming and labor intensive). When we transfer to a probabilistic modeling tool, we have more freedom in how we model the various components of the system. For example, where we previously modeled the reservoir as a non-increasing function, we can now model it as a non-increasing function with a probability of 0.95 that the volume decreases (there is an infusion in progress) and a probability of 0.05 that the volume stays the same (the infusion is paused for some reason)[2]. Similarly, we can model the failure behavior of the sensor with more fidelity. In this case, we can assign a probability of x that the sensor will signal "empty" when, in fact, the reservoir is not empty, and a probability of y that it will signal "not empty" when the drug volume is, in fact, below the threshold. The question now, of course, is what effect does this kind of probabilistic behavior have on the satisfaction of the system level requirements given a specific sensor filtering algorithm.

```
const double pr =0.99;  // prob. that we will infuse
const double pe =0.99; // prob. of true positive ‘‘empty’’
const double pne =0.95;  // prob. of true negative ‘‘not empty’’

module reservoir
    vol: [0..inVol] init inVol;
    //The infusion can progress (or pause) until volume is 0
    [a] (vol>0) -> pr:(vol’=vol-drip) + (1-pr):(vol’=vol);
    [a] (vol<=0) -> (vol’=0);
endmodule

module sensor
    sensor_out: bool init false;
    [a] (vol<=empty)-> pe:(sensor_out’=true)
                        + (1-pe):(sensor_out’=false);
    [a] (vol>empty) -> pne:(sensor_out’=false)
                        + (1-pne):(sensor_out’=true);
endmodule

module software
    alarm: bool init false;
    [a] true -> (alarm’=sensor_out | alarm);
endmodule
```

Figure 3: PRISM model for Case 4 in Table 2

An example of our model in PRISM is shown in Figure 3. At the top we define the probabilities used in the model: the probability that we will infuse in a specific time-step(`pr`), the probability of a true positive reading of the sensor (`pe`, the probability of reservoir below the threshold and the sensor reports it), and the probability of a true negative (`pne`, the probability of reservoir not empty and sensor reports it). Note that the probabilities of the sensor providing false negative as opposed to a false positive are different; this allows us to provide a far more accurate model of the environment than what is possible in our Simulink/Stateflow models.

Our PRISM model has the same components as discussed previously. The `reservoir` contains an initial volume of drug. If the volume of remaining drug is larger than zero, the volume will be decreased by one with probability `pr` and remain the same (infusion is paused) with probability `(1-pr)`. The `sensor` will sense the reservoir and if the remaining drug volume is less than or equal to the threshold, the sensor will accurately report that the tank is "empty" with a probability of `pe` and provide a false negative with probability `(1-pe)`. Similarly, the sensor may report a false positive if the reservoir is not empty. The `software` will, in our simplest model, simply latch the alarm to true when an empty reservoir is detected.

Given the sensor model that has a uniform probability of failure in every step, the probability of a false alarm will be predicated on the length of the infusion interval. For example, if the infusion interval is long, the sensor will be given many opportunities to provide two consecutive false positive readings leading to a violation of our "no false positives" requirement. Of course, the likelihood of a long infusion is dependent on both the rate of infusion, the initial volume in the reservoir, and the probability that the infusion is paused, all of which will affect the probably that our requirements are actually met.

When running our analysis in PRISM, we will use a sensor model that allows arbitrary failures as in Case 3 in Section 3; the case where none of our requirements could be verified to always hold in our system. Given the new probabilistic behavior of the sensors and the reservoir, however, there is a *chance* that our requirements are met even with the simplistic software defined in Figure 3. Case 4 in Table 2 shows the results over our simplest model. In this analysis, we assume the initial drug volume in the reservoir to be 20 units and the infusion rate to be one unit per step (`drip=1` in the model) when the pump is infusing. Further, we assume that the pump is paused 5% of the time, the probability of a false positive sensor reading is 0.05, and a false negative reading is 0.01.

As can be seen, our requirements may not hold all the time (as proved in the previous section), but we have a reasonable probability that the requirement will actually hold during operation (if our assumptions about the failure rates and infusion rates are reasonable). Given our assumptions on the environment, if the reservoir is empty, there will be an alarm with a probability of 95% and we will not raise a false alarm with a probability of 85%. This behavior, unfortunately, is deemed unacceptable and we need to improve our solution. Thus, we again explore various solutions (and relaxed requirements) as discussed in Section 3. The results of our probabilistic analysis can be seen in Cases 4-6 in Table 2 (we will discuss Case 6 shortly). All results in Table 2 use the probabilities discussed above.

To decrease the probability of false positives, we move to our solution of using two consecutive readings of "empty" to raise the alarm (Case 5). Again, we need to relax our requirements to allow for a slight delay in rasing the alarm after the remaining volume becomes too low. As can be seen, we now reduce the false alarm rate to an acceptable level (4% chance of a false alarm) while boosting the probability of meeting our requirement to 98%. Of course, a direct comparison in performance with Case 4 cannot be made since we there used a different requirement (immediate action when the drug level is low). If we use our relaxed requirement and our Case 4 software, we increase the probability of correctly raising an alarm when the drug volume is low to 99%, but the false positive rate is still too high since this requirement did not change (see the two last rows for Case 4). Thus, we consult our stakeholder and the performance is deemed acceptable.

[2]Note here that all numbers are entirely fictional and are included for discussion purposes only.

Table 2: Probabilistic Modeling and Analysis

Case	Component Behavior	Requirement	Results
4	**Software:** Raise alarm on first "empty" reading.	vol <= threshold ⇒ Alarm	Satisfied with P= 0.95
		volume > empty ⇒ No Alarm	Satisfied with P=0.85
		vol <= threshold ⇒ Alarm in next step	Satisfied with P= 0.99
		volume > empty ⇒ No Alarm	Satisfied with P=0.85
5	**Software:** Raise alarm on two consecutive "empty" reading.	vol <= threshold ⇒ Alarm in next step	Satisfied with P= 0.98
		vol > empty ⇒ No Alarm	Satisfied with P= 0.96
6	**Sensors:** Three redundant **Software:** Raise alarm when two out of three sensors indicate "empty".	vol <= threshold ⇒ Alarm	Satisfied with P= 0.99
		vol > threshold ⇒ No Alarm	Satisfied with P=0.89

At this point, the exploration of possible solutions and acceptable requirement could be over; one declares victory and moves on. Luckily, stakeholders and engineers closely inspect all assumptions made in the analysis and discover that the initial drug volume of 20 units is unrealistically low (infusion time is generally quite long) and the probability that the pump is paused is far more likely than the 5% we used in our analysis. To see if this had any impact on our analysis, we modify the initial conditions for our analysis and run our analysis with an initial drug volume of 100 units and a probability of the pump being paused at 50%. We are happy to see that under these environmental assumptions, the probability of meeting our "raise the alarm" requirement goes to 99%—we have the performance we seek. Unfortunately, given the long infusion time enabled by the larger drug volume and extensive pausing of the infusion, the probability of nuisance alarms goes from 4% per infusion session to 37%—there is simply more time for sensor failures to manifest themselves. In fact, the increase in the probability of having an alarm raised when the volume become too low is entirely caused by the increase in the false positive rate (if we erroneously raise an alarm and leave it on, it will still be on when the reservoir goes empty).

Table 3: Probabilistic analysis with varying initial conditions for Case 5

Parameters		Verification Results	
Initial Volume	Infusion Probability	Requirement_1 (True Positive)	Requirement_2 (True Negative)
20	0.95	0.98	0.96
100	0.95	0.98	0.78
100	0.50	0.98	0.63
1000	0.95	0.99	0.08
1000	0.50	0.99	0.008

Table 3 contains the results for various initial conditions. In the case of very long infusions (1000 units initial drug volume and a 50% chance of a paused infusion), we are more or less certain of getting a false alarm during every infusion (the probably of meeting our "no false alarms" requirement goes to zero as infusion time increases). Thus, our exploration of the Twin Peaks must continue.

At this point, more radical changes to our proposed solution may be warranted. For example, one could consider an alternate system architecture with three redundant sensors and simple majority voting (two out of three raises the alarm). The performance of such an architecture and voting mechanism is included as Case 6 in Table 2. Unfortunately, the false positive rate is not acceptable and our exploration must continue. Combinations of triple redundance and sensor filtering may yield the performance we seek with the existing sensors. Possibly, we may need to rethink our choice of sensor and aim for something far more reliable.

5. DISCUSSION AND CONCLUSION

The process of discovering and refining requirements, and exploring the architecture, design, and behavior of a system is an iterative, interactive, and highly non-trivial task. In this paper we illustrated how model based developments and verification techniques can help (and hinder) in this exploration.

As Miller et al. have pointed out, the process of discovering and refining models and requirement in tandem is a powerful process for improving both [12]. Nevertheless, the tools used in traditional verification approaches rely on "yes"–"no" answers; either a requirement is met or it is not. Since all systems really on some level of cooperation from their environment to operate as intended [5, 6], assumptions regarding how this environment operates are crucial in the verification and exploration of the system requirements and the system architecture, design, and behavior. When exploring a systems architecture containing both logical (software) and physical (electrical or mechanical) components where the system requirement—as they should [5]— are expressed in terms of the physical system, not the software, capturing assumptions about the environment and the various components involved becomes a challenge. By their nature, all physical components will experience failures in various forms and modeling and analyzing these failure modes using traditional verification techniques such as model checking and theorem is difficult; either we make unrealistic assumptions about the environment and physical components (e.g., one failure assumption) and succeed in our verification, or we make realistic assumptions (e.g., unlimited Byzantine failures) and fail in our verification.

By moving to a verification framework where we can model the stochastic nature of many system components, we believe we will be able to more effectively perform the requirement-solution exploration needed to devise an effective engineered system meeting realistic requirement. Note here that we are not advocating abandoning traditional verification; such verification is invaluable when establishing desirable properties of many classes of models. We are advocating the inclusion of probabilistic reasoning in the process as a complement to better understand the tradeoffs involved.

The intent of this paper is to demonstrate the use of probabilistic verification techniques as a Twin Peaks exploration tool. Although the case example used was exceedingly simple and the choice of values used for analysis was random, it helps to illuminate the impact of stochastic sensor and environment behavior on the architectural choices and possibility of satisfying requirements. Our goal was to discuss the possibilities and set the stage for a broader initiative applying these stochastic techniques to explore and assess the requirements and solution spaces for Cyber Physical Systems.

6. REFERENCES

[1] IBM Rational Rhapsody. http://www.ibm.com/developerworks/rational/products/rhapsody/, 2014.

[2] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[3] Esterel-Technologies. SCADE Suite product description. http://www.esterel-technologies.com/v2/scadeSuiteForSafetyCriticalSoftwareDevelopment/index.html, 2004.

[4] U. Food and D. Administration. White Paper: Infusion Pump Improvement Initiative. April 2010.

[5] C. A. Gunter, E. L. Gunter, M. Jackson, and P. Zave. A reference model for requirements and specifications. *IEEE Software*, 17(3):37–43, May/June 2000.

[6] J. Hammond, R. Rawlings, and A. Hall. Will it work? [requirements engineering]. In *Requirements Engineering, 2001. Proceedings. Fifth IEEE International Symposium on*, pages 102 –109, 2001.

[7] M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In G. Gopalakrishnan and S. Qadeer, editors, *Proc. 23rd International Conference on Computer Aided Verification (CAV'11)*, volume 6806 of *LNCS*, pages 585–591. Springer, 2011.

[8] MathWorks Inc. Simulink. http://www.mathworks.com/products/simulink.

[9] MathWorks Inc. Simulink Design Verifier. http://www.mathworks.com/products/sldesignverifier.

[10] MathWorks Inc. Stateflow. http://www.mathworks.com/stateflow.

[11] S. P. Miller, M. P. Heimdahl, and A. Tribble. Proving the shalls. In *Proceedings of FM 2003: the 12th International FME Symposium*, September 2003.

[12] S. P. Miller, A. C. Tribble, M. W. Whalen, and M. P. E. Heimdahl. Proving the shalls: Early validation of requirements through formal methods. *Int. J. Softw. Tools Technol. Transf.*, 8(4):303–319, 2006.

[13] S. P. Miller, M. W. Whalen, and D. D. Cofer. Software model checking takes off. *Commun. ACM*, 53(2):58–64, 2010.

[14] A. Murugesan, S. Rayadurgam, and M. Heimdahl. Using models to address challenges in specifying requirements for medical cyber-physical systems. In *Fourth workshop on Medical Cyber-Physical Systems*, Apr. 2013.

[15] B. Nuseibeh. Weaving together requirements and architectures. *Computer*, 34:115–117, 2001.