# Probabilistic UML Statecharts
# for Specification and Verification
## a case study

David N. Jansen

Universiteit Twente, Information Systems Group.
Postbus 217, 7500 AE Enschede, The Netherlands.
`dnjansen@cs.utwente.nl`

**Abstract.** This paper introduces a probabilistic extension of UML state-charts. A requirements-level semantics of statecharts is extended to include probabilistic elements. Desired properties for probabilistic state-charts are expressed in the probabilistic logic PCTL, and verified using the model checker Prism. The extension simplifies the verification of critical systems with probabilistic elements, e.g. fault-tolerant systems. The extension is illustrated using a case study: a gambling machine.
The theory behind this extension is explained in detail in a paper published recently [9]; this article concentrates on the case study.

## 1 Introduction

The UML hardly needs any justification at this conference. It is being used to describe more and more systems, and an increasing number of engineers get used to it. I believe that the UML will be used even in domains it was never designed for; accordingly, it is not far fetched to predict that the following years shall see substantial efforts to extend the UML towards soft real-time, fault-tolerance, quality of service and the like. First work in this direction has been undertaken, e. g., in [2, 5, 10, 12].

One of the principal modelling paradigms needed to express such aspects is the concept of *probability,* allowing one to quantitatively describe the randomness the system is exposed to, the randomness the system itself exhibits, or both.

We have defined an extension of statecharts with probability to describe the randomness the system itself exhibits and call the extended formalism *P-statecharts.* They can be used to describe, e. g., dependability aspects, fault-tolerance, or also randomised algorithms. Furthermore, probability is an abstraction means: it allows one to hide data dependencies by just representing the likelihood of particular branches to be taken.

Our extension of statecharts is coupled to a formal statechart semantics. This allows for formal verification of a behaviour specified by a number of statecharts, ensuring that the system has some desired properties. Formal verification is one way to minimize the risk that some critical system malfunctions and causes loss of life or property. We have chosen the requirements-level semantics of Eshuis and

Wieringa [4], which is based on the semantics by Damm et al. [3], because it is simple and close to the most frequently used semantics for UML. Requirements-level semantics mostly use the perfect technology assumption, which abstracts from limitations (in speed and memory) imposed by an implementation. A detailed justification of this semantics and comparisons to other semantics can be found in [4]. The setup of our probabilistic extension, however, is independent from the UML basis we take. This means that other formal statechart semantics can equally well be enhanced with a similar probabilistic extension.

Model checking is a formal technique which is relatively easy to use (compared to other formal techniques), and as such, it has been used successfully in several projects. Model checking compares the behaviour of (a model of) a system to desired properties or requirements to that system. Recently, model checking has been extended to probabilistic systems [1, 11]. With our formal P-statechart semantics, we can apply probabilistic model checking to a finite collection of P-statecharts. Using this technique, we can automatically check properties like: "The probability that a system crashes within 130 steps without ever visiting certain states is at most $10^{-5}$."

This article bases the presentation of P-statecharts on a case study. I describe the software in a so-called *fruit machine,* a kind of gambling machine. A fruit machine can be seen as a performance-critical system because there are several legal requirements on it, e. g., requirements on the average wins and losses: a manufacturer is required to show that its product meets the legal requirements by means of experimentation or calculation (i. e., formal verification). Formal verification, in this case, assures that the manufacturer will not lose the investment to develop a new gambling machine during legal approval. The example will show the advantages of P-statecharts over a formalism without hierarchy or parallelism.

Besides probabilistic choice, a system may also contain nondeterministic behaviour, e. g., where the probability distribution between several behaviours is unknown, depends on external factors, or is deliberately left unspecified. P-statecharts allow to model both probabilistic and nondeterministic behaviours.

*Organisation of the paper.* Section 2 introduces syntax and informal semantics of P-statecharts. Section 3 presents the case study. Section 4 describes probabilistic model checking, the property language used and how I have applied this to the case. Section 5 concludes the article.

## 2 P-statecharts

This section gives an informal introduction to P-statecharts. For a formal definition and the formal semantics, see our article [9].
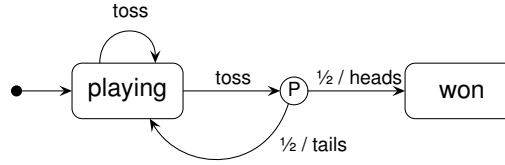
**Fig. 1.** Example P-statechart: unreliable, but fair coin

Statecharts are a graphic language introduced by Harel [7] to describe the behaviour of some system or component. They have provisions for state refinement (hierarchy) and parallelism of several subcomponents in the system. Therefore, a system's state (also called a configuration) may consist of several states in the statechart (also called nodes). Possible transitions are indicated by edges from one set of states to another; when a transition is taken, a part or all of the configuration may change.

P-statecharts look like a simple extension of the statechart syntax: instead of simple edges, one draws *P-edges,* where the action and the target depend on the outcome of a probabilistic experiment. A traditional edge can be seen as a P-edge where a single outcome gets probability 1.

A P-edge with a non-trivial probability distribution is drawn in two parts: first an arrow with event and guard $\xrightarrow{e[g]}$ that points to a symbol Ⓟ (a so-called P-pseudonode), then several arrows emanating from the P-pseudonode, each with a probability and an action set $\xrightarrow{p/A}$. This notation is inspired by C-pseudonodes Ⓒ, used for case selection purposes e. g., in [8].

A simple example is given in Fig. 1. It depicts a P-statechart which shows the behaviour when playing with an *unreliable, but fair coin:* the event "toss" may or may not be ignored. If the system reacts, it outputs "heads" or "tails", each with 50 % chance. If the output is heads, the system stops playing. It is unspecified how (un)reliable the system is: the choice between ignoring and reacting is nondeterministic.

Formally, a P-statechart is a quadruple, consisting of a set of nodes *Nodes*, a set of events *Events*, a set of variables, and a set of P-edges. A P-edge is a tuple $(X, e, g, P)$, where $X \subseteq Nodes$ is a non-empty set of source state nodes, $e \in Events$ is an event, $g$ is a guard (a boolean combination of expressions over the variables), and $P : \mathbf{P}(Actions) \times \mathbf{P}(Nodes) \to [0, 1]$ is a probability measure. (The set *Actions* contains the possible actions, which consist of sending events to other P-statecharts and assignments to variables.) The function $P$ can be considered as a hyperedge with multiple possible targets $(A, Y)$. A target is an action set $A \subset Actions$ together with a non-empty set $Y$ of successor nodes. Once the P-edge is triggered by event $e$ and guard $g$ holds in node(s) $X$, a target $(A, Y)$ is selected with probability $P(A, Y)$.
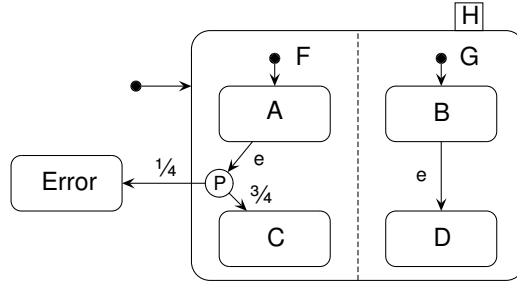
**Fig. 2.** Example: consistency depends on the chosen target

## 2.1 Semantics

We have chosen to extend the requirements-level semantics of Eshuis and Wieringa [4] to a probabilistic semantics. In a similar way, one could extend other semantics to include probabilistic elements.

A system consists of a finite collection of P-statecharts. The behaviour of a P-statechart can be described intuitively as follows. The statechart is always in some state or configuration (which consists of one or several nodes, as in traditional statechart semantics). A P-edge is taken if the P-statechart is in the source node(s), the event of the edge happens and its guard holds. Then, the system chooses one of the possible results (probabilistically and nondeterministically); it leaves the source nodes, executes the chosen action and enters the chosen target nodes of the P-edge. More than one edge may be taken simultaneously, if the event triggers several nonconflicting P-edges.

*Some aspects of the formal semantics.* We have defined a formal semantics of a fixed finite collection of P-statecharts in terms of a Markov decision process (MDP). A MDP is a structure that allows for both probabilistic and nondeterministic elements in a transition. The model checker PRISM [11] supports MDPs as an input language.

The interplay between probabilistic and nondeterministic elements has to be designed carefully. The P-statechart in Fig. 2 describes a system which reacts to an event e in two independent components, of which one causes an error with probability $\frac{1}{4}$. The transition A → Error is inconsistent with B → D. But whether these transitions become both enabled depends on the probabilistic choice. This forces us to resolve the probabilism *prior* to nondeterministic choice between inconsistent transitions. This is done by introducing intermediate states in the MDP where necessary.

## 3 Case study: a fruit machine

The following case study illustrates the definition and use of P-statecharts: A fruit machine contains three reels that show fruit symbols. When the user starts
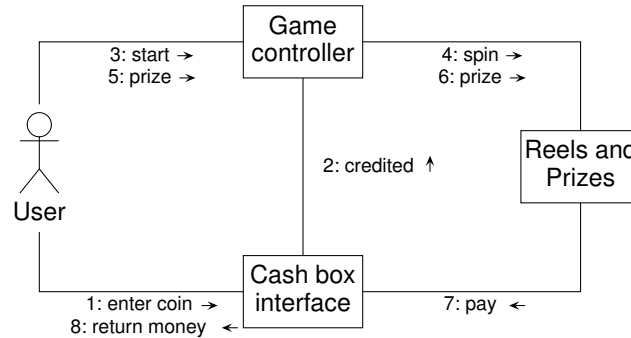
**Fig. 3.** Collaboration diagram for a typical game

| Combination | | | Prize |
|---|---|---|---|
| reel 1 | reel 2 | reel 3 | |
| bar | bar | bar | 10 |
| cherry | cherry | cherry | 5 |
| grapes | grapes | grapes | 5 |
| ? | bar | bar | 5 |
| cherry | ? | cherry | 2 |
| grapes | grapes | ? | 2 |
| ? | ? | bar | 2 |
| ? | ? | cherry | 1 |

**Table 1.** Rewards for specific game results

a game, the reels spin until some timeout. If the visible parts of the reels show some specific combination of symbols (e. g., three times cherry), the player gets a prize after pressing the "prize" button. For simplicity, I do not consider the display.

The fruit machine contains a combination of probabilistic and nondeterministic elements; for example, the outcome of a single reel is probabilistic, but which of the reels stops first is nondeterministic.

The system contains two important objects: the *reels and prizes* object describes the reels' outcome and which prize the user gets; the *game controller* describes the sequence of games on a higher level. In addition, there is a *cash box interface.*

A typical interaction of the objects is shown in the collaboration diagram in Fig. 3. It depicts the following scenario: a user enters a single coin and then presses the start button. The user waits until the reels stop, then presses the "prize" button to get his prize.

The statechart for the reels and prizes object is shown in Fig. 4. Here, the statechart syntax allows us to model the three reels independently; it is not necessary to draw all possible combinations explicitly. This enables engineers to create models of complex systems. Note that the state of the reels is only
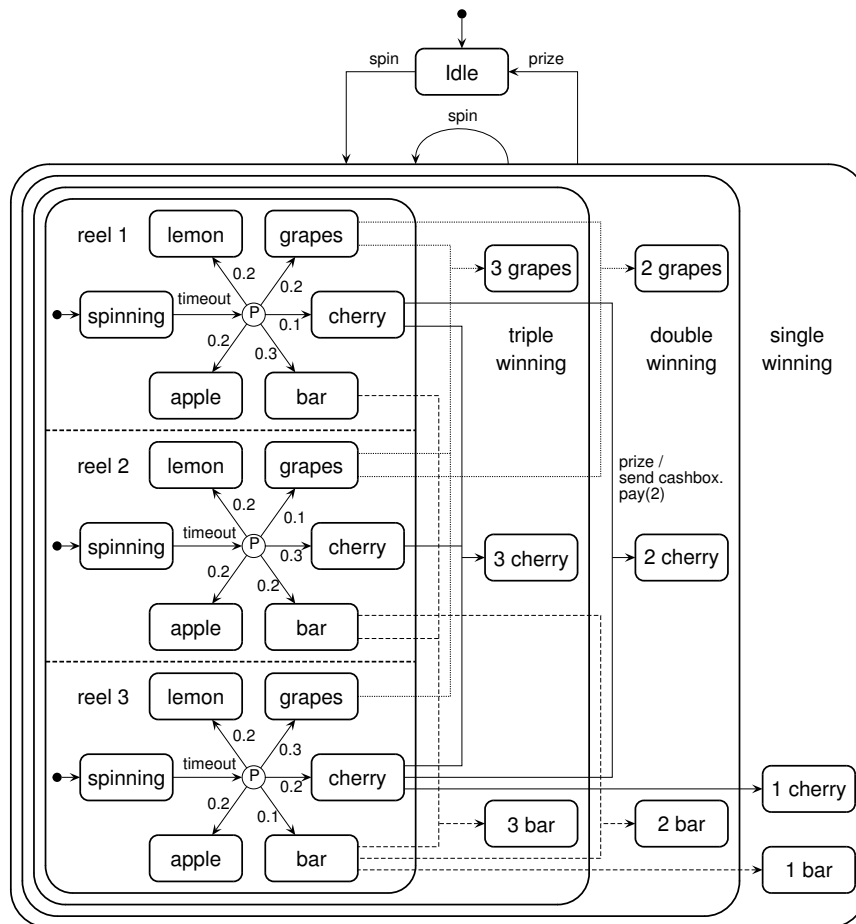
**Fig. 4.** The reels and prizes' behaviour. To avoid cluttering, some P-edge labels have been omitted and some P-edges are drawn dashed or dotted.
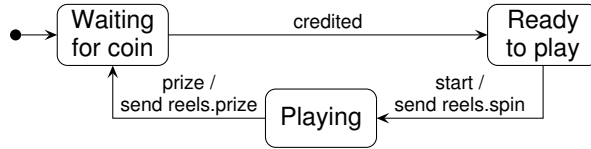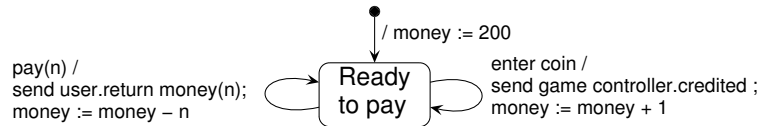
**Fig. 5.** The game controller's behaviour



**Fig. 6.** The cash box interface's behaviour

registered as long as it is needed. I use the UML priority scheme from [8]: smaller scopes have higher priority. This implies, e. g., that P-edges to subnodes of triple winning have priority over P-edges to subnodes of double winning, and so, the highest prize is found and paid. Some P-edge labels have been omitted: each edge that leads to a prize has the label prize / send cashbox.pay(*amount*) with the appropriate amount from Table 1 filled in. I have shown one example.

The statecharts of the other objects are shown in Figs. 5 and 6.

## 4   Model checking

I have composed the above statecharts with a simple user simulation and constructed the MDP which is their semantics. The MDP has been fed into the model checker PRISM [11], a probabilistic model checker for MDPs and similar structures.

I had to put a bound on the integer variable which represents the stock of return money (as in a real automaton, there is only room for a finite amount of coins). The automaton starts with 200 coins. I have added a game counter to express properties for a series of games. After some simplifications, the MDP had 1 789 057 reachable states.

### 4.1   Property language

As a property specification language, we propose to use the probabilistic branching time logic PCTL, which extends CTL with probabilistic features. PCTL was first designed for fully probabilistic systems without nondeterminism [6]. We use the interpretation of PCTL over MDPs defined by Baier and Kwiatkowska [1, 11]. PCTL allows one to express properties such as:

$\Psi$ The probability that a system crashes within 130 steps without ever visiting certain states is at most $10^{-5}$.

In order to decide these properties, the nondeterminism is resolved by means of schedulers (also known as adversaries or policies). Here, we restrict ourselves to the fragment of PCTL for which actual model-checking tool-support is available; i. e., we only consider properties interpreted for all fair schedulers.

*Syntax and informal semantics.* The syntax of PCTL is given by the following grammar, where $a$ denotes an atomic proposition, $\mathsf{v}$ denotes a variable, $k \in \mathbb{Z}$ an integer constant, $p \in [0,1]$ denotes a probability and $\sqsupseteq$ is a placeholder for a comparison operator $<, \leq, =, \geq, >$:

$$\varphi, \psi ::= \textbf{true} \mid a \mid \mathsf{v} \leq k \mid \mathsf{v} \geq k \mid \varphi \wedge \psi \mid \neg\varphi \mid \mathcal{P}_{\sqsupseteq p}[\varphi\,\mathcal{U}^{\leq k}\,\psi] \mid \mathcal{P}_{\sqsupseteq p}[\varphi\,\mathcal{U}\,\psi]$$

The meaning of $\textbf{true}$, comparisons, conjunction and negation is standard. Formula $\mathcal{P}_{\sqsupseteq p}[\varphi\,\mathcal{U}^{\leq k}\,\psi]$ holds in a state if the probability of the set of paths that reach a $\psi$-state in at most $k$ steps while passing only through $\varphi$-states is $\sqsupseteq p$. $\mathcal{P}_{\sqsupseteq p}[\varphi\,\mathcal{U}\,\psi]$ has the same meaning, but does not put a bound on the number of steps needed to reach the $\psi$-state. (A formal interpretation on MDPs is omitted here, and can be found in [1].) Property $\Psi$, e. g., is expressed as $\mathcal{P}_{\leq 10^{-5}}[\neg\varphi\,\mathcal{U}^{\leq 130}\,crash]$ where $\varphi$ describes the states that should be avoided.

*Schedulers and fair schedulers.* The above explanation is ambiguous if nondeterminism is present, because the probability will (in general) depend on the resolution of nondeterminism. Non-determinism is resolved by schedulers. A *scheduler* selects, for each initial fragment of a path through the MDP, one of the possible continuations. It does not resolve probabilistic choices. Several types of schedulers do exist, see [1]. Here, we consider *fair* schedulers. A fair scheduler only selects fair paths. A path $\pi$ is *fair* if, for each state $s$ that appears infinitely often in $\pi$, each of the possible nondeterministic continuations in $s$ also appears infinitely often. PRISM checks whether a PCTL-formulas holds for all fair schedulers.

## 4.2 Desired properties checked

I have checked the MDP with some properties which resemble legal requirements to the automaton. Most of these requirements are conditions on the allowed minimal, maximal, or mean loss of the player. I have added a variable gamecount to the model to help in formulating requirements on a series of games. I have checked the following properties:

*The player doesn't lose too much.* I state this as: In 30 games, the player loses at most 70 % with probability at least 0.9. (A single game costs 1 coin, so the maximum loss is 30 coins.)

$$\mathcal{P}_{\geq 0.9}[\text{gamecount} < 30\,\mathcal{U}\,(\text{gamecount} = 30 \wedge \text{money} \leq 200 + 0.7 \cdot 30]$$

Remember that money represents the amount of money in the automaton; a loss for the player makes money rise. PRISM reported the property holds in 4' 20".

*The player doesn't win too much.* I state this as: In 30 games, the player wins less than 10 coins with probability greater than 0.99.

$$\mathcal{P}_{\geq 0.99}[\text{gamecount} < 30 \, \mathcal{U} \, (\text{gamecount} = 30 \wedge \text{money} > 200 - 10)]$$

Also here, PRISM reported the property holds in 5' 47".

*The median loss is less than 50 %.* This is formulated as: In 30 games, the probability to lose 15 or more coins is less than 50 %.

$$\mathcal{P}_{< 0.5}[\text{gamecount} < 30 \, \mathcal{U} \, (\text{gamecount} = 30 \wedge \text{money} \geq 200 + 15)]$$

PRISM reported that the property holds in 4' 45".

*The stock of coins is large enough.* This is not a legal requirement, but a practical one: how large should the stock of return money be that the chance a prize cannot be paid is less than $10^{-20}$? Let's check whether it is large enough:

$$\mathcal{P}_{\leq 10^{-20}}[\text{true} \, \mathcal{U} \, \text{money} < 1]$$

PRISM reported that the property is false. Experimentation with some other probabilities led to the conclusion that the probability that a prize cannot be paid some time is quite high. However, when I assume that the coins are refilled regularly (say, every 1000 steps), it is enough to prove

$$\mathcal{P}_{\leq 10^{-20}}[\text{true} \, \mathcal{U}^{\leq 1000} \, \text{money} < 1]$$

which is true. PRISM needed 16' 31" to check this; the check is slow because the tool unfolds the system to 1000 steps.

*Timed requirements.* Some of the legal requirements are real-time properties, e. g.: a game must last at least 4 seconds on average. Our modelling language does not include real-time, so it is impossible to check this requirement.

## 5  Conclusion

*Contributions.* This article is centered around a case study of a probabilistic system, a gambling machine. It illustrates our extension of a statechart dialect with probabilistic features [9]. This extension has a formal semantics, based on the semantics of [4], which uses Markov decision processes as semantic model. The case study shows how to use the probabilistic logic PCTL to specify and check desired properties of a probabilistic system.

*Observations.* The fruit machine example profited from the fact that the three reels could be modelled independently, using hierarchy and parallelism; the user needs not draw all possible combinations of reel outcomes explicitly. This makes P-statecharts a powerful language, suitable to describe complex probabilistic systems.

I think that probabilistic statecharts are easy to learn for people who know statecharts, as there is only one simple syntactic extension for non-trivial probabilistic edges. As statecharts are known widely among software engineers, it may simplify the use of probabilistic model checking: instead of a completely different formalism, an engineer can use an easy extension of a known language to create the model.

In the formal semantics for this simple-looking extension, one has to find a delicate balance between probabilism and nondeterminism. I have illustrated this by the example P-statechart in Fig. 2. (In the case study, I didn't use edges where consistency depends on the resolution of probabilistic choices. In this case, a simpler extension might be enough; but whether a simplification excludes other applications, needs further study.)

*Future work.* We plan to continue our research in two directions: On one hand, we would like to incorporate real-time elements into the formalism. This includes a stochastic extension, where the time between several events is distributed according to a probability distribution.

On the other hand, we would like to consider environmental randomness. This is randomness introduced from the outside, because users or other external entities behave according to some probability distribution. For example, users may have preferences; or sensors may fail with a specific error rate.

## References

1. Christel Baier and Marta Kwiatkowska. Model checking for a probabilistic branching time logic with fairness. *Distributed Computing*, 11(3):125–155, 1998.
2. M. Dal Cin, G. Huszerl, and K. Kosmidis. Quantitative evaluation of dependability critical systems based on guarded statechart models. In *Proceedings, 4th IEEE International Symposium on High-Assurance Systems Engineering : . . . HASE*, pages 37–45, Los Alamitos, 1999. IEEE.
3. Werner Damm, Bernhard Josko, Hardi Hungar, and Amir Pnueli. A compositional real-time semantics of statemate designs. In Willem-Paul de Roever, Hans Langmaack, and Amir Pnueli, editors, *Compositionality : the significant difference. COMPOS '97*, volume 1536 of *LNCS*, pages 186–238, Berlin, 1998. Springer.
4. Rik Eshuis and Roel Wieringa. Requirements-level semantics for UML statecharts. In Scott F. Smith and Carolyn L. Talcott, editors, *Formal Methods for Open Object-Based Distributed Systems IV : . . . FMOODS*, pages 121–140, Boston, 2000. Kluwer Academic Publishers.
5. Stefania Gnesi, Diego Latella, and Mieke Massink. A stochastic extension of a behavioural subset of UML statechart diagrams. In L. Palagi and R. Bilof, editors, *Fifth International Symposium on High-Assurance Systems Engineering (HASE)*, pages 55–64. IEEE Computer Society Press, 2000.

6. Hans Hansson and Bengt Jonsson. A logic for reasoning about time and reliability. *Formal Aspects of Computing*, 6:512–535, 1994.

7. D. Harel. Statecharts : a visual formalization for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

8. David Harel and Eran Gery. Executable object modeling with statecharts. *Computer*, 30(7):31–42, July 1997. IEEE.

9. David N. Jansen, Holger Hermanns, and Joost-Pieter Katoen. A probabilistic extension of UML statecharts : specification and verification. In Werner Damm and Ernst-Rüdiger Olderog, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems : 7th intl. symposium . . . proceedings*, volume 2469 of *LNCS*, Berlin, 2002. Springer.

10. Peter King and Rob Pooley. Derivation of Petri net performance models from UML specifications of communications software. In Boudewijn R. Haverkort, Henrik C. Bohnenkamp, and Connie U. Smith, editors, *Computer Performance Evaluation : Modelling Techniques and Tools ; . . . TOOLS 2000*, volume 1786 of *LNCS*, pages 262–276, Berlin, 2000. Springer.

11. Marta Kwiatkowska, Gethin Norman, and David Parker. Probabilistic symbolic model checking with prism: A hybrid approach. In Joost-Pieter Katoen and Perdita Stevens, editors, *Tools and Algorithms for the Construction and Analysis of Algorithms : . . . TACAS*, volume 2280 of *LNCS*, pages 52–66, Berlin, 2002. Springer.

12. C. Lindemann, A. Thümmler, A. Klemm, M. Lohmann, and O. P. Waldhorst. Quantitative system evaluation with DSPNexpress 2000. In *Workshop on Software and Performance (WOSP)*, pages 12–17. ACM, 2000.