

A Graphical Notation for Probabilistic Specifications

Position Paper

Sebastian Nielebock

Chair of Software Engineering
Institute of Distributed Systems
Faculty of Computer Science
Otto-von-Guericke University
39106 Magdeburg, Germany
sebastian.nielebock@ovgu.de

Tim Gonschorek

Chair of Software Engineering
Institute of Distributed Systems
Faculty of Computer Science
Otto-von-Guericke University
39106 Magdeburg, Germany
tim.gonschorek@st.ovgu.de

Frank Ortmeier

Chair of Software Engineering
Institute of Distributed Systems
Faculty of Computer Science
Otto-von-Guericke University
39106 Magdeburg, Germany
frank.ortmeier@ovgu.de

ABSTRACT

Nowadays formal methods represent a powerful but in practice not well supported way for verification. One reason among others for this is that such methods are often conceived of being too theoretical, can likely be misunderstood by non-experts, and notations are therefore misapplied.

For example to specify formal specifications, complex logics are state-of-the-art, especially for probabilistic verification properties. It is not only that users are unable to construct correct specifications but they also miss to understand what is meant by the verification results based on the specification.

In this paper we address this problem and endorse the usage of a graphical notation to construct such specification properties, especially probabilistic ones. We present how such a notation may look and emphasize that one single notation is sufficient to solve both lack of comprehensibility of the specification as well as of the verification results. Moreover, we extract some future research directions.

Categories and Subject Descriptors

F.4.1 [Mathematical Logic and Formal Languages]: Mathematical Logic—*Modal logic, Temporal logic*; F.3.1 [Logics and Meanings of Programs]: Specifying and Verifying and Reasoning about Programs—*Specification techniques*; D.2.1 [Software Engineering]: Requirements/Specifications—*Tools*

General Terms

Verification, Design, Theory

Keywords

verification, specification, specification patterns, graphical specification patterns

1. INTRODUCTION

Verification ensures desirable properties of technical systems by utilizing well-known techniques. Regarding safety properties, which describe the absence of human or environmental harm by a system, these techniques are often required by norms[13]. Another aspect is to guarantee specific qualities (so called liveness properties) of software or technical systems by verification.

With respect to such properties one has to verify qualitative as well as quantitative aspects of a system. A qualitative (safety) aspect may be that a train will stop in front of railroad crossing, if its barriers are still opened. The quantitative aspect enriches the qualitative one by not just asking *if* but *with which probability* this situation will/will not appear.

The general procedure of verification can be divided in the following three steps:

1. Description of the system behavior and its components by a formal description language e.g. Automata[12], Petri nets[17] or SAML (System Analysis Modeling Language)[11]
2. Specification of desirable properties to be verified by using a formal language e.g. temporal/stochastic logic like CTL (Computation Tree Logic) or PCTL (probabilistic CTL)[4]
3. Verification of these specified properties checked by an appropriate method e.g. symbolic model checking by using different tools (e.g. NuSMV,Prism)[6, 14, 4] or interactive verification by using e.g. KIV[19] or PVS[18].

To facilitate this process for non-experts in software and system verification, we developed a tool named VECS (Verification Environment for Critical Systems)¹, which supports the modeling as well as the specification and verification process. Its main design goal is to hide the formal and tedious verification part from users without loss of soundness of the verification results. In general this tool automatically translates SAML system descriptions to input languages of appropriate verification tools, in particular PRISM and NuSMV[15].

¹<https://cse.cs.ovgu.de/cse/vecs-info.html>

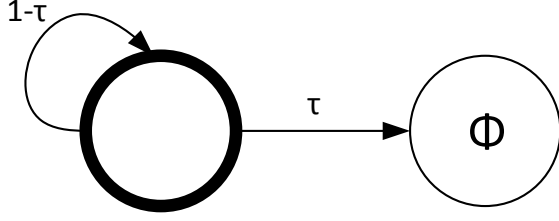


Figure 1: Example System whose verification results may lead to misunderstanding

Nevertheless, VECS does not overcome all current problems of the verification and specification process. A specific problem occurs, if one wants to specify a property to be verified on a system description. For users who are not familiar with e.g. temporal or stochastic logic it is likely that the resulting specification is not correctly representing intended behavior. This is critical, in particular for safety aspects, because in this case the verification results are based on the wrong properties.

This means not only that user cannot interpret written specification properties, but they are not able to correctly deal with the verification results.

For example, consider the system model depicted in Figure 1. Within this model (state automaton) the circles denotes states and arrows state transitions labeled with a probability τ and $1-\tau$, respectively, which denote how probable this state transition will be used. Moreover, the bold lined circle denotes the initial state of the system. Using stochastic logic, one may ask how probable it is that the state labeled with ϕ will be eventually reached. If $\tau > 0$ then the resulting probability of eventually reaching the state labeled by ϕ is one, because there exist a transition between the initial state and this interesting state. Non-experts might be confused with that because they assume the rate (probability per time) rather than the reaching of the state.

To overcome these current drawbacks of the verification process, we suppose the usage of a graphical notation among others for the specification task. To that end, we will concentrate on the mentioned problems. We do not provide a solution for other common practical problems e.g. the validity of the system model or the exponential increasing of the search space during model checking (so-called state space explosion problem). Note that these problems have to be addressed too, to increase usability of verification methods. Moreover, just an extract of our proposed graphical notation will be presented which is sufficient to support our position.

The remainder of the paper is structured as follows: In Section 2 some background knowledge and common approaches concerning presenting specifications are explained. In the following Section 3, we introduce our notion of a graphical specification language. Based on this, we will argue the pros and cons of our notation (see Section 4). Finally, in Section 5

we conclude the discussion and extract some future research activities.

2. BASICS AND RELATED WORK

This section introduces common state-of-art approaches to specify system properties to be verified. To that end, we consider if and how these approaches increase comprehensibility of the specified characteristic as well as of the verified result for non-experts. We discuss these approaches in more depth in Section 4.

2.1 Meaning of CTL/PCTL Properties

The current technique for specification is to directly apply temporal and/or stochastic logics e.g. CTL/PCTL. In the following some short informal introduction to CTL and PCTL is given. For a more formal explanation, interested readers are referred to further literature (e.g. [4]). For the purpose of this paper the informal explanation is sufficient.

CTL or PCTL consider the set of possible states, the system might be in. System Models are state transition systems. For each state there has to be at least one successor state. One distinguishes transitions into deterministic (i.e. exactly one successor) and non-deterministic ones (multiple successors). Then the semantics is the set of all traces of states compatible with the transition relations. This summarizes the possible behavior of a system as a tree of states connected by its state transitions and starting at one initial state. A *path* on this tree is one single (possible infinite) trace of states starting at a single state (usually the initial state).

CTL describes properties based on this tree. Therefore, a CTL formula consist of path quantifier (**A** "on all paths", **E** "there exists a path"), temporal operators (e.g. **F** "eventually in the future", **G** "globally, now and forever in the future", etc.) as well as a state formula (e.g. a Boolean formula) [8]. A path quantifier specifies whether the specification has to hold on all possible paths (**A**) or at least on one path (**E**). Temporal operators specifies the behavior on a single path, e.g. that the state formula has to be true in at least on state (**F**) or in all states (**G**) on the path. A state formula is an expression that, in combination with a given state, either evaluates to true or to false. Eventually, the combination of a path quantifier, a temporal operator and a state formula yield a CTL formula.

For example the formula **EF** ϕ means that there exist (**E**) a path on which there is finally (**F**) a state where the formula ϕ is true. In other words, there has to be at least one state on at least one path in the tree which satisfies the formula. In order to characterize probabilistic behavior, probabilities can be added to the state transition with multiple successor states. In particular, this means to get from one state into another one has to step either deterministically or non-deterministically to an intermediate state from which the successor state can be reached by a given probability.

An example for this is depicted in Figure 2. To that end, it is possible to get from state A (states are denoted as grey circles) to state B1 either by choosing non-deterministically the upper intermediate state (denoted as black circle) with a probability of 0.1 or by the lower one with a probability

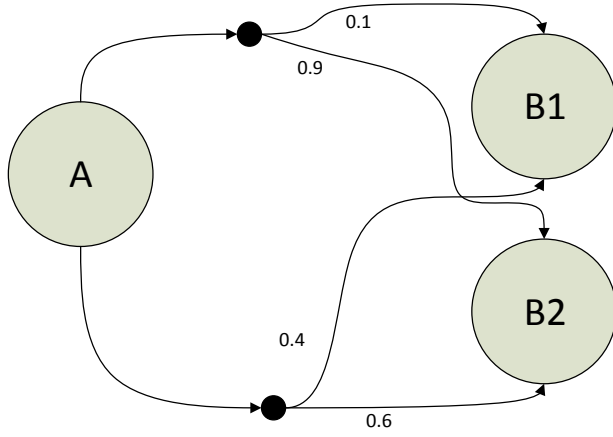


Figure 2: Example of non-deterministic, probabilistic state transitions

of 0.4. Each state has always a successor state, so that the sum of all probability distributions at the intermediate state has to be one. A path on this tree is described by a single trace of states with the chosen probabilities between these states.

PCTL extends CTL by specifying properties with probabilities. For example the formula $\mathbf{P}_{<0.01}\mathbf{F}\phi$ means that the probability that there is finally a state which satisfies ϕ is lower than 0.01. Note that the probability on a path is calculated by multiplying all its probabilities. For the special case of the example in Figure 2, where probabilities are combined with non-deterministic behavior, the probability value is not a single number but a probability range with an upper and a lower bound.

In practice, time bounds, in particular bounded number of steps on a path, are often introduced. This is often done if probabilities on an infinite path do not converge to a useful value. For example the formula $\mathbf{P}_{<0.01}\mathbf{F}^{[t_1, t_2]}\phi$ means that the above explained property is reached between the time steps t_1 and t_2 .

Such properties are even less comprehensible for non-experts as well as for experts as is indicated during several studies[7, 10]. Multiple approaches to facilitate the usability of these logics were approached as it can be seen in the following section.

2.2 Related Work Concerning Comprehensive Specifications

Within this section we present approaches concerning more comprehensive specifications. Afterwards we refer to techniques to explain verification results.

A challenging problem for understanding verification properties, e.g. CTL or PCTL formulae, is that they have many possible constructs so that errors are likely integrated. To overcome this problems specification patterns were introduced, which represent common or well-studied solutions

for real-world specification problems. There exist domain-specific patterns e.g. safety patterns (cf. [5]) and general specification patterns based on statistical analysis (cf. [7, 10]). By composing these patterns more complex specifications can be achieved.

Nevertheless, these patterns decrease the number of possible forms of specifications, and there is still some space for misunderstanding these patterns[5]. Therefore, another direction is to explain patterns with natural words. Unfortunately, such an explanation could be ambiguous which is counterproductive to express unique specifications.

Another approach defines a specification languages based on natural words. In particular, this is a unique mapping of a specification pattern to a single word group[10]. By doing this, the ambiguous meaning is overcome but the resulting sentences sound too artificial and too similar to the logic they are based on[5].

Visual support was also introduced before as it can be seen by several approaches (especially but not restricted to by Symbolic Timing Diagrams, Life Sequence Charts or Message Sequence Charts cf.[5]). But as Bitsch reveals these graphical languages are more or less restricted in their expressiveness (at least for safety patterns). That is why he introduces the Graphical Notation for Specification with Safety Patterns (GNSS) which is similar to the well-known UML state charts. Unfortunately, this notation is only able to express qualitative properties i.e. CTL formulae.

Another challenge in working with specifications is interpreting the verification results and determining the possible erratic behavior.

One approach, presented by Aljazzar et. al., is to visualize the paths representing a counterexample for a given verification property[3]. A counterexample is a combination of states describing a behavior that violates a given property. By drawing the states involved in the counterexample as a directed graph, the user gets the violating state trace. Furthermore, if one wants to check a PCTL formula, it is also shown how probable the occurrence of each counterexample path is by mapping the probabilities to a representing color scale. The visualization provides the user with the opportunity to overlook the state transitions leading to the states violating the specifications. Nevertheless, it gives no specific statement which part of the property has been violated. This could improve the comprehensibility of the given counterexample, because the violated desired behavior is pointed out.

Other approaches do not focus on the visualization of a counterexample but on finding similar paths that do also or especially not lead into an erratic system behavior. For example one method searches for paths that are similar to the counterexample, i.e. which contain a specified control location, a state immediately preceding the error state[9]. Particularly, the difference of the paths, containing the control location state, but do not lead into an error state, can help understanding what causes the property to fail. Another notion is to change a given counterexample path so that the error state is not reached anymore [1, 2]. To do this, some state

transitions are changed so that the resulting path does not lead into an error state. The changes are done with respect to the requirement that the changed path stays as similar as possible to the given counterexample. The similarity is measured through specific distance metrics. This approach can lead the user to a state having an impact on whether the path reaches a state violating the given specification or not.

These approaches have one main drawback. They do not reflect the given specification from which the determined counterexamples have been generated. They only give support where to find states violating the property but not showing the property that is violated. We suppose that linking generated counterexamples with the specifications, by which they are produced from, can support understanding what the erratic behavior is or help to detect wrong specification properties.

Summarizing, there exist a few approaches that either try to support users phrasing their specifications without having knowledge of formal specification languages or gives the user more information about paths representing the model checking results. But there is no approach leading the user through the whole process of writing the specifications and interpreting the results, especially based on the input specifications. Moreover, we think that a graphical notation is easier to understand and less ambiguous than a semi-formal written notation. That is why, we want to provide a graphical notation that supports the user in formulating formal specifications and serves as an initial point for understanding and retracing the erratic behavior.

3. NOTION OF A GRAPHICAL NOTATION FOR PROBABILISTIC SPECIFICATIONS

In the following, we introduce our notion of a graphical specification language. The basic idea refers to GNSS or UML state charts which are adapted concerning their current drawbacks to express probabilistic specification properties. Moreover, this notation is based on the probabilistic specification patterns of Grunske[10] due to its relevance to real world problems. In particular, we contribute a unique mapping of graphical representation to these patterns.

For the purpose of this paper, we just introduce a graphical representation of the “Probabilistic Existence”-pattern, which is sufficient for the subsequent discussion. The formula of this pattern is denoted as $\mathbf{P}_{\bowtie p} \mathbf{F}^{[t_1, t_2]} \phi$, where $\bowtie \in \{<, \leq, \geq, >\}$. For simplification we will use the probability condition < 0.01 . Then, this formula is similar to that explained in Section 2.1.

First, to explain how our graphical specification for this formula looks, we present the graphical representation of the corresponding formula $\mathbf{EF}\phi$ in Figure 3.

Note that every pattern is surrounded with a rectangle with radiused corners. The meaning of the notation’s elements is as follows:

- the three arrows on the left border represent the path quantifier (one arrow pointing to the right stands for

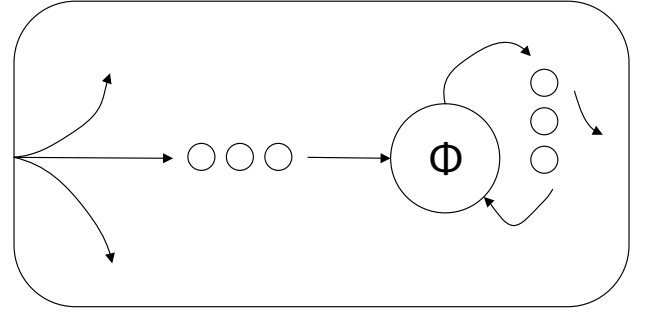


Figure 3: Graphical Representation of the CTL formula $\mathbf{EF}\phi$

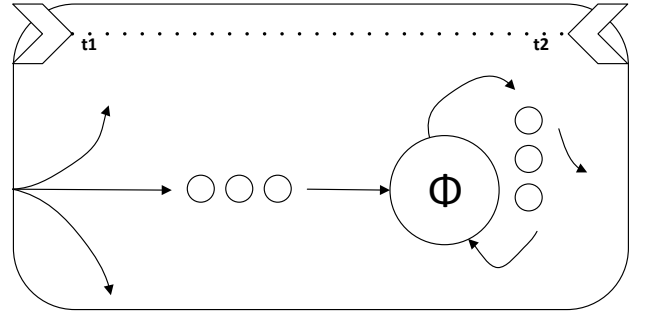


Figure 4: Graphical Representation of the CTL formula $\mathbf{EF}^{[t_1, t_2]} \phi$

E, all three pointing to the right represent **A** and two of three arrows pointing to the right are a special case for PCTL)

- arrows denote state transitions
- circles express potential states
- textual notation inside a state is referred to as a state formula e.g. ϕ
- three not connected states in between two arrows mean that there may be arbitrary states, especially not a single state, in between
- arrows that do not have a subsequent state symbolize that the subsequent behavior is not interesting for the satisfaction of this pattern

Based on this notation the representation means that the formula is satisfied if there is (in the non-deterministic case at least) one path with arbitrary intermediate states that finally reaches a state in which the formula ϕ is satisfied. Moreover, it is possible that such a state can be reached in the future. If the formula contains time bounds e.g. $\mathbf{EF}^{[t_1, t_2]} \phi$ (depicted in Figure 4) the time bound is symbolized by two chevrons connected with a dotted line. Note that the dotted line is applied because usually these models are time discrete.

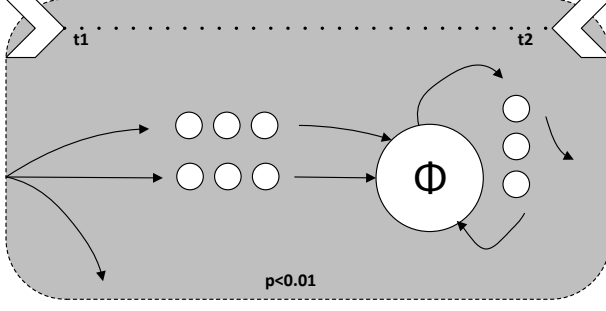


Figure 5: Graphical Representation of the PCTL formula $P_{<0.01} F^{[t_1, t_2]} \phi$

Finally the probability is added inside the graphical notation. The representation of the PCTL formula $P_{<0.01} F^{[t_1, t_2]} \phi$ is denoted in Figure 5. The stochastic condition is denoted by a grey background color of the surrounding box and a textual probability condition. If one wants to specify probabilistic specifications without any time bound, it is possible, to omit the time symbols.

Note that there exists a major difference between the visualized PCTL formula and the previous CTL formula. While for the satisfaction of the CTL formula, it is sufficient that at least one path reaches the desired state (in the example that, which satisfies ϕ), in the PCTL formula there may be multiple paths influencing the considered probability and therefore they satisfy the probability condition. The reason for this is, that the probability condition, based on the probability mentioned in Section 2.1 (sum all path probabilities), is used. This is why we depict two paths, which lead to the state that satisfies ϕ .

Note that by applying the patterns presented by Grunske[10] one is able to build up complex specifications by combining them. This benefit is also available with this notation due to unique mapping of each pattern to a graphical representation.

By using our proposed notation, one is able to visualize the verification results. Figure 6 is an example of the result visualization of a probabilistic specification pattern, where the PCTL formula $P_{<0.01} F^{[t_1, t_2]} \phi$ is not satisfied. To this end, elements of the pattern are graphically emphasized by increasing transparency (i.e. not or less contributing part for satisfaction/non-satisfaction of the pattern) or increasing thickness (i.e. more/most contributing part). This can be helpful if the user wants to interpret some nested specification. A simple example is a specification with an or-clause with two possible paths, e.g. $P_{<0.01} (F^{[t_1, t_2]} \phi \vee F^{[t_3, t_4]} \psi)$. If the specification holds, we increase the thickness of the part that the model fulfills (also both are possible) and decrease the visibility (increase the transparency) of the part that is not fulfilled. If the specification does not hold, we increase the thickness of the path that causes the violation e.g. that corresponds to the counterexample. In Figure 6 for example, the specification does not hold, because the probability of $F^{[t_1, t_2]} \phi$ is higher than 0.01. We emphasize the paths

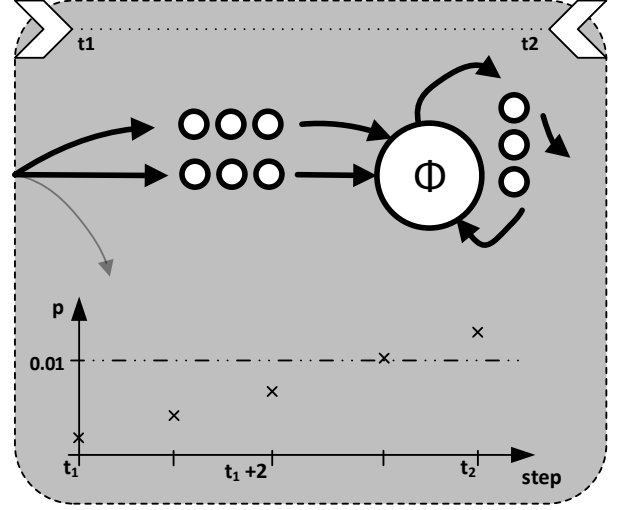


Figure 6: Graphical Representation of a PCTL Verification Result

leading to this, i.e. the paths fulfilling $F^{[t_1, t_2]} \phi$.

Additionally, a diagram of the probability over the corresponding step is added. It shows the probability that the state formula ϕ is true within the corresponding step. Note the example in Figure 6. It shows that the specification $P_{<0.01} F^{[t_1, t_2]} \phi$ is not fulfilled, because the probability is higher than 0.01. But, furthermore, it provides the information that the specification holds up to step t_1+2 and why it is not for the following.

Another verification result is a counterexample. They show a sequence of states violating the specification. The visualization of a counterexample is emphasized by increasing the transparency of the elements in the pattern, which are not important. In contrast, the elements of the pattern symbolizing the counterexample are emphasized by using thicker lines and bold text.

After introducing the graphical notation, we discuss the drawbacks and benefits with respect to our notion in the next section.

4. DISCUSSION OF GRAPHICAL NOTATION SUPPORT

In this section we argue about the usage of this graphical notation for the specification task. To that end, we consider drawbacks and benefits of this support concerning comprehensibility of these specifications as well as of the results based on these specifications.

4.1 Drawbacks

There are several threats, which might undermine our concept of a graphical specification notation.

For example Bitsch emphasized that several graphical notations are not sufficient concerning their expressiveness of specification[5]. That means, one is not able to specify all typical real world problems (e.g. in the scope of safety). Nevertheless, the notion of our notation is based on specification patterns thus usually occurring real world problems. By relying on convincing studies, we believe that this risk can be diminished.

Another problem is that this new notation means that engineers or system designer have to learn a new notation. Actually if this is true, we believe that there is a major difference between learning a text-based notation (e.g. CTL or PCTL) instead of a graphical notation which symbolizes the behavior. Like Moody[16] is reasoning, visualization are processed in parallel in human brains instead of the sequential processing of text. Moreover, images are more likely to be remembered, which facilitates and speed up the learning process in our opinion.

Nevertheless, Moody underlines that introducing visualizations does not automatically lead to better comprehensibility[16]. For example, he argues that the combination of notation symbols increases the complexity of the notation, which therefore becomes more confusing. One of his major claims is that not only the semantics should be addressed but mainly the syntax of a notation. This is, because syntax has to support the semantics e.g. using graphical symbols that can be associated with the specified behavior. That is why he introduces several principles for creating comprehensive and supportive graphical notations.

4.2 Benefits

Although there are several drawbacks our notation has to overcome, it provides many advantages.

One main advantage is that it neither only supports writing specifications nor only analyzing the results. It can be used within the whole process. So the user can find out which part of the specification has been violated. We emphasize that this supports the user in understanding which particular part causes the erratic behavior.

Furthermore, a graphical notation, if it is not ambiguous, is easier to understand and to apply than an artificial sounding pattern language. Additionally, written patterns are made for one specific language. Especially non-native English speakers could benefit from a graphical notation by improved comprehensibility. Moreover, the psychological aspects of graphical notations, i.e. the characteristics a notation should have to increase its comprehensibility, based on well-studied knowledge base [16].

Another point is the complexity of our notation. With the usage of given specification patterns, we reduce the number of possible symbols to a, in our point of view, manageable minimum. By following the principles of Moody, especially concerning complexity management of models, this might increase the comprehensibility of the property bases just by the visual perception[16]. Due to [16], visual perceptions are easier to understand and to handle for the human brain. Furthermore, this increases the efficiency of using specifications, because a complex graphic is easier to comprehend

than a large written text on the same complexity level.

This shows that a graphical notation, if it is carefully developed, overcomes the drawbacks mentioned in the previous section. Moreover, it provides the opportunity to visualize verification results directly on the given specification.

5. CONCLUSION

As it can be seen, graphical notations can have several benefits versus other approaches. In our opinion, visualizations are much more human-oriented, which means they are much easier to learn, to be understood and to be applied than textual notations. Nevertheless, we stated that the semantics as well as the syntax has to be carefully and deliberately chosen.

We presented a first notion to address the problems of comprehensibility of typical specifications (so-called specification patterns) and of the verification results. One major benefit is that our graphical specification notation can be applied for the specification task as well as for the interpreting of the verification results.

To address current drawbacks of many notations, we will utilize the principles of Moody[16] on our graphical notation. By doing this, we hypothesize that our notation is much more effective and efficient to construct specifications. That means one is able to specify real-world properties on real-word system models without expertise in temporal and stochastic logic. This hypothesis has to be evaluated with real users. So we plan to conduct further studies with students and engineers in our future research.

Furthermore, we will investigate the benefits of interactivity for user support, e.g. by clicking on the elements symbolizing the counterexample, the user gets a graphical graph representation of the counterexample. Additionally, other paths, not violating the specification (if any exist) are shown. If using the VECS simulation tool, the user can go into the counterexample and step forward and backward through the path. Thereby, we suppose that this supports the understanding on how the system behaves and what leads to the counterexample.

Another research direction might be that this or a similar notation might be used to model a system. Within this modeling task other problems like validity of the model to the real system which is usually addressed by traceability has to be considered.

6. ACKNOWLEDGMENTS

This work was developed within the Probabilistic Models for Safety Analysis (ProMoSA) project funded by the German Research Foundation (DFG).

7. REFERENCES

- [1] D. K. O. S. Alex Groce, Sagar Chaki. Error explanation with distance metrics. *International Journal on Software Tools for Technology Transfer*, 8:229 – 247, 2006.
- [2] F. L. Alex Groce, Daniel Kroening. Understanding counterexamples with explain. In *Lecture Notes in*

- Computer Science*, volume Volume 3114, pages 453–456, 2004.
- [3] H. Aljazzar and S. Leue. Debugging of dependability models using interactive visualization of counterexamples. In *Proceedings of the fifth International Conference on Quantitative Evaluation of Systems*, pages 189 – 198. IEEE, 2008.
 - [4] C. Baier and J.-P. Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.
 - [5] F. Bitsch. *Verfahren zur Spezifikation funktionaler Sicherheitsanforderungen fuer Automatisierungssysteme in Temporallogik*. PhD thesis, University Stuttgart, 2006.
 - [6] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. In *Proc. International Conference on Computer-Aided Verification (CAV 2002)*, volume 2404 of *LNCS*, Copenhagen, Denmark, July 2002. Springer.
 - [7] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in Property Specifications for Finite-state Verification. In *Proceedings of the 21st International Conference on Software Engineering, ICSE '99*, pages 411–420, New York, NY, USA, 1999. ACM.
 - [8] O. G. E. M. Clarke and D. Peled. *Model Checking*. The MIT Press, 1999.
 - [9] A. Groce and W. Visser. What went wrong: Explaining counterexamples. In *SPIN Workshop on Model Checking of Software*, 2003.
 - [10] L. Grunske. Specification Patterns for Probabilistic Quality Properties. In *Proceedings of the 30th International Conference on Software Engineering, ICSE '08*, pages 31–40, New York, NY, USA, 2008. ACM.
 - [11] M. Gdemann. *Qualitative and Quantitative Formal Model-Based Safety Analysis*. PhD thesis, Otto-von-Guericke University Magdeburg, 2011.
 - [12] J. E. Hopcroft and J. D. Ullman. *Formal languages and their relation to automata*. Addison-Wesley Longman Publishing Co., Inc, Boston, MA, USA, 1969.
 - [13] IEC SC 65A. Functional safety of electrical/electronic/programmable electronic safety-related systems. Technical Report IEC 61508, The International Electrotechnical Commission, 3, rue de Varembe, Case postale 131, CH-1211 Genve 20, Switzerland, 1998.
 - [14] 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.
 - [15] M. Lipaczewski, S. Struck, and F. Ortmeier. SAML goes Eclipse - Combining Model-Based Safety Analysis and High-LevelEditor Support. In *Proceedings of the 2nd International Workshop on Developing Toolas Plug-Ins (TOPI)*, pages 67–72. IEEE, 2012.
 - [16] D. Moody. The “Physics” of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering. *IEEE Trans. Softw. Eng.*, 35(6):756–779, Nov. 2009.
 - [17] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77:541 – 580, 1989.
 - [18] R. J. M. S. N. Owre, S. Pvs: A prototype verification system. In *CADE-11 Proceedings of the 11th International Conference on Automated Deduction: Automated Deduction*, pages 748 – 752. Springer-Verlag, 1992.
 - [19] S. G. S. K. Reif, W. Interactive correctness proofs for software modules using kiv. In *Proceedings of the Tenth Annual Conference on Computer Assurance*, pages 151 – 162. IEEE, 1995.