Towards Integrated Correctness Analysis and Performance Evaluation of Software Systems (Doctoral Forum Paper)

Ioannis Stefanakos(✉)

Department of Computer Science, University of York, York, UK
is742@york.ac.uk

Abstract. In recent times, the involvement of computer systems in our lives has been drastically increasing, as has the need of improving the resilience of these systems, e.g. so they can withstand errors and changes in their environment. Techniques such as testing and simulation are often used to ensure this, but in the case of complex, real-time systems, these techniques can only provide coverage for a limited set of possible system behaviours. Software model checking and stochastic verification are alternative techniques that formally and exhaustively verify whether software meets its functional requirements and establish the performance and dependability properties of software, respectively. The two formal techniques are often used in isolation, yet software must simultaneously ensure a combination of functional and non-functional requirements. The doctoral project described in this paper aims to bring these two areas of software verification together by enabling the joint analysis of functional and non-functional properties of software systems.

1 Introduction

It has long been known that computer systems, both hardware and software, exhibit errors. In order to increase the reliability of these systems, software engineers may devote a substantial amount of time on testing and debugging. There has always been research focusing on developing new or improving the existing verification methods [1]. Verification is the area that includes all the techniques aiming to improve software quality and its main focus is to provide evidence that the final product conforms to the specified requirements during all of its life cycle processes [2]. Some of the techniques subsumed under verification are formal verification, testing and simulation.

Testing is considered an essential activity in software engineering. It is defined as the process of validation of the system's intended behaviour and identification of potential malfunctions [3]. With the increase of involvement of software and hardware systems in our everyday lives, testing has become more complex but at the same time necessary to ensure the correct functionality of these systems.

Work supported by Microsoft Research through its PhD Scholarship Programme.

Although successful testing identifies a significant amount of errors, it is still impossible to capture all of them [4], especially in dynamic environments. As a result, computer systems can still be afflicted by errors which could potentially lead to severe consequences, e.g. safety critical systems.

Stochastic verification and model checking are techniques able to address issues that testing is not able to identify. Both of these techniques are also known as formal methods, which is a line of study that depends on the fact that computer systems can be depicted as mathematical objects whose behaviour is in principle well-determined [1].

Model checking provides an automated method for verifying concurrent finite-state systems in which the system's intended behaviour is represented by a state-graph model checked to confirm it satisfies properties formalized in temporal logic [5]. The system semantics are given by means of a Kripke structure and the specification is expressed using temporal logic. Temporal logic is a formalism for reasoning about time without introducing it explicitly [6].

A Kripke structure (KS) is a labeled graph that contains all the possible states of a system and the transitions between them. The states are represented by the vertices of the graph and the transitions by its edges. In more detail, a tuple $M = (S, S_0, R, AP, L)$ is the representation of a Kripke structure where S is a set of states, $S_0 \subseteq S$ is a set of the initial states, $R \subseteq S \times S$ is a transition relation, AP is a set of atomic propositions and $L : S \rightarrow 2^{AP}$ is a labeling function that maps each state to the set of propositional variables that hold in it [7]. The states represent the different states of a system. The main difference between Kripke structures and labelled transition systems (LTS), which is another popular basis for many formal modelling languages, is that transitions in LTS are labelled to describe the actions which cause a state change while the states in KS are labelled to describe how they are modified by the transitions [8].

Stochastic verification (also known as probabilistic model checking) is a formal verification method that can establish quality properties of software systems that exhibit stochastic behaviour [9]. Software systems of this nature can be found in applications used in aircrafts and vehicles, as well as in personal devices such as mobile phones. In order to be able to verify the correctness of the systems operating under uncertain environments, it is necessary to analyze quantitative properties such as performance and reliability.

Probabilistic model checking uses models that can be categorized as continuous and discrete time, deterministic and non-deterministic, and compositional. The simplest type of all probabilistic models is the Discrete-Time Markov Chains (DTMCs) [10]. DTMCs are Kripke structures that all their transitions are linked to a specific probability. The sum of all out-going transition probabilities, that each state has, is equal to one.

Model checking and probabilistic model checking are necessary to ensure that the produced software meets both functional and non-functional requirements. While their importance is recognized, software engineers often only consider one or the other during their software analysis and this has mainly to do with the

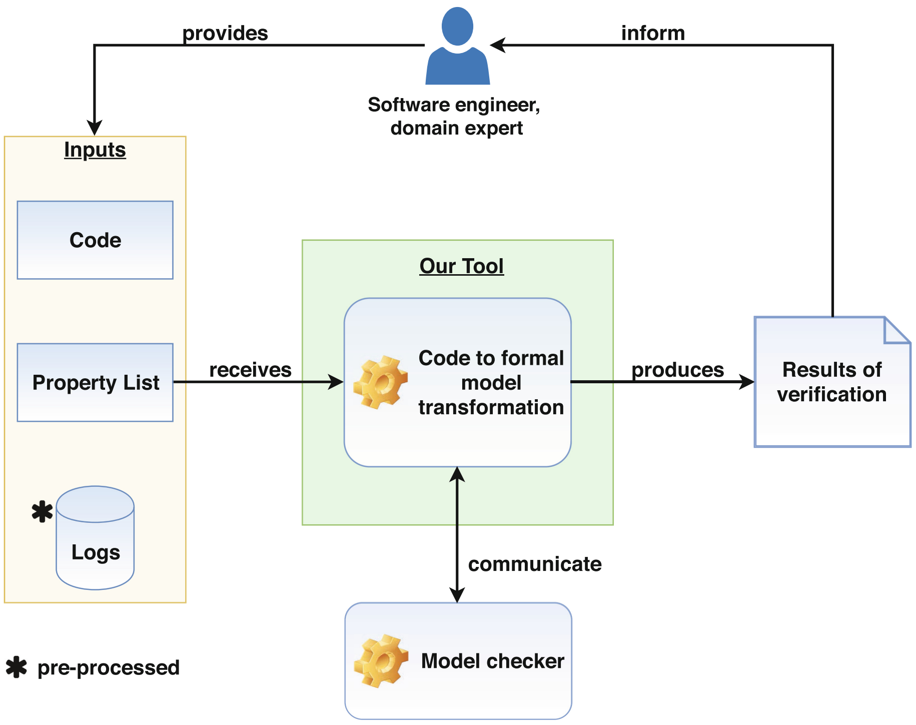


Fig. 1. High-level diagram of the proposed approach

fact that these techniques use disjoint models, different formalisms, etc. This project aims to bring together the two areas of verification by (a) extending existing modelling paradigms in order to integrate the verification of both functional and non-functional requirements, and (b) achieving this integration with an acceptable cost and good scalability that enables the application of the new verification techniques to real-world systems.

2 Objectives and Proposed Solution

The project focuses on constructing state-transition models (e.g., Kripke structures and discrete-time Markov chains or DTMCs [10]) of the source code under verification, through the implementation of a code-to-model transformation method. This method will be implemented as a hybrid verification tool. Additionally, a list of properties to be verified will be given as input to the tool. To enable the analysis of non-functional properties, the tool will also use as input preprocessed logs of the system. These logs will capture the operational profile of the software, and will be used to calculate the probability of executing different branches within the code, based on previous use. The resulting probabilities will be assigned to the respective state transitions of the generated DTMC model which, finally, together with the source code will be analyzed by model checkers, to verify both functional and non-functional properties of interest. When fully developed, the approach may use model checkers and verification tools such as Storm [11], NuSMV [12], Java PathFinder (JPF) [13], FACT [14,15], ePMC [16] and OMNI [17,18]. The joint analysis of functional (e.g. deadlock freeness, reachability) and non-functional requirements (e.g. response time, energy consumption) will provide insight to software engineers and the ability to inspect the impact of different changes to the system.

Moreover, our project was planned with an emphasis on resilience, which can be defined as the ability to provide required capability in the face of adversity [19, 20]. Specifically, the analysis provided by our approach will enable the selection, at design time or runtime, of method implementations (code) that can withstand the actual workload and other aspects of the environment of a system without violating the system requirements. To achieve the project goals, we propose an approach that comprises the following key components:

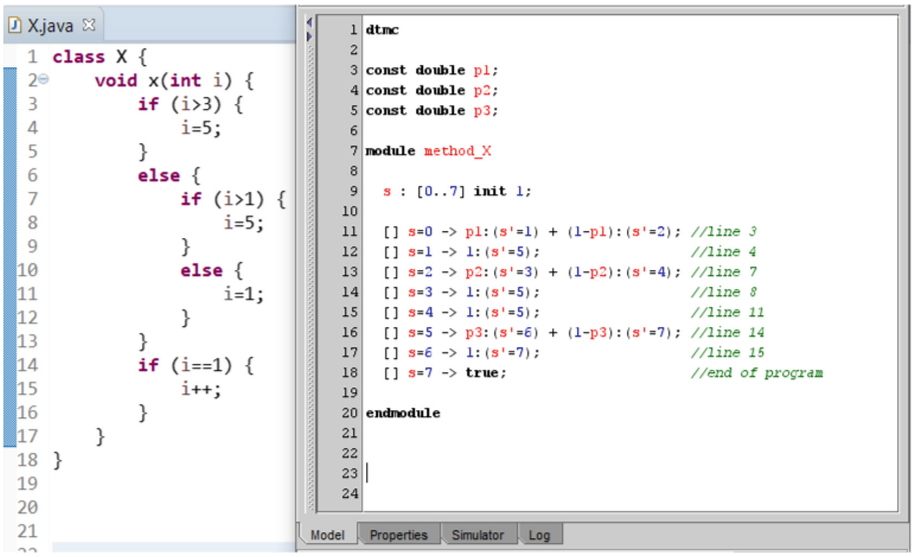
– The implementation of a code-to-model transformation method that will enable the conversion of Java source code into Kripke structures and DTMC models. For the implementation of this method, we used JavaParser[1], a set of libraries that supports code generation, code analysis and code refactoring.
– A list of input parameters for our tool, i.e. source code, properties of interest and pre-processed logs of user data.
– A communication channel between our tool and popular model checkers, some of them mentioned earlier, for the verification of both functional and non-functional requirements.

Figure 1 depicts the high-level architecture of our solution. In the first step, the software engineer/domain expert will submit the required inputs to our tool. In the next step, the tool will generate the formal models based on the input data, and will communicate with the model checkers to initiate the verification procedure. As soon as this process finishes, the tool will receive the results and will produce an output file, which can be used by the engineer to detect requirement violations.

To achieve the project objectives, we have organized the research work into the following tasks:

1. Review existing literature on joint verification of functional and non-functional requirements to learn the current state of research. The outcome of this literature review is summarized in Sect. 4.
2. Based on the identified limitations, form a theoretical method as a potential solution. A first version of this solution was described earlier in this section.
3. Proceed with the implementation of the proposed theoretical method. The preliminary implementation work carried out so far is presented in Sect. 3.
4. Evaluate the method using case studies taken from model checking benchmarks and real-world applications (e.g. Android code).

[1] https://javaparser.org.



```
 1 class X {
 2     void x(int i) {
 3         if (i>3) {
 4             i=5;
 5         }
 6         else {
 7             if (i>1) {
 8                 i=5;
 9             }
10             else {
11                 i=1;
12             }
13         }
14         if (i==1) {
15             i++;
16         }
17     }
18 }
19
20
21
```

```
dtmc

const double p1;
const double p2;
const double p3;

module method_X

  s : [0..7] init 1;

  [] s=0 -> p1:(s'=1) + (1-p1):(s'=2); //line 3
  [] s=1 -> 1:(s'=5);                  //line 4
  [] s=2 -> p2:(s'=3) + (1-p2):(s'=4); //line 7
  [] s=3 -> 1:(s'=5);                  //line 8
  [] s=4 -> 1:(s'=5);                  //line 11
  [] s=5 -> p3:(s'=6) + (1-p3):(s'=7); //line 14
  [] s=6 -> 1:(s'=7);                  //line 15
  [] s=7 -> true;                      //end of program

endmodule
```

Fig. 2. Example of input source code (left) and output DTMC model (right)

5. Extend the approach to support further automation of the method and apply additional techniques (e.g. parametric model checking).
6. Further evaluate and refine our framework, making any necessary improvements based on the evaluation results.

3 Preliminary Work

So far, the project has developed a preliminary version of the approach from Fig. 1. This version of our verification tool supports only the transformation of Java source code into a DTMC model. Thus, the verification process is currently performed manually, using the probabilistic model checker PRISM (www.prismmodelchecker.org) after the automated generation of the DTMC.

Figure 2 presents an example of the tool's input and output. At this stage, the tool is only able to extract a DTMC model from a given source code (with transition probabilities represented as parameters of the DTMC model). We still need the logs of user data to reason about them, which is one of the future steps of the project, along with the choice of appropriate model checkers (some mentioned earlier) to complete the verification process.

Focusing on this example, every variable assignment in the code is represented by a new state in the DTMC model and can lead to a another state with probability $P = 1$. The if statements on the other hand, also represented by a new state in the model, lead to two possible states. One with probability P if the condition is satisfied and a second one with probability $1 - P$, in the case the condition is not satisfied. In the latter case, we move to the else branch, if it exists, or to the next expression. For instance, consider the if statement starting in line 3 of the Java code from Fig. 2; the if branch of this conditional statement is executed for $i > 3$. Therefore, if the log of an application that uses this code shows that i is equally likely to take the values $0, 1, \ldots, 9$, the probability that the if branch is executed will be 0.7, and the probability that the else branch is executed will be 0.3. Of course, in real systems, the conditions are often more complex, and the values of variables are rarely uniformly distributed like in this simple example.

The method shows some promising preliminary results. In preliminary testing, we allocated a time counter under specific branches of the source code and then performed simulation to calculate the average response time. Next, we added reward structures in the DTMC model (e.g. $rewards$ "*execTime*" $s = 1 : 2$; $s = 3 : 1$; $endrewards$), to assign the corresponding states with the same time values used in the source code. To deal with probabilities at this stage, we created a uniform distribution of values ranging between -1 and 1. The final results derived from the model were finally obtained by establishing and verifying properties in Probabilistic Computation Tree Logic (PCTL) [21], e.g. R {"*execTime*"}=? $[F\ s\ =\ 7]$. This property specification translates to what is the total execution time by the time we reach program termination. The simulation and probabilistic model checking produced the same results in these preliminary experiments.

4 Related Work

While several studies have been conducted in the areas of model checking and probabilistic model checking, with notable advances in both [6,9,22,23], there are significantly fewer approaches when it comes to combining these two techniques.

Cortellessa et al. [24] build an XML-based framework that consists of software models and formal relations among them, to support the integration of functional and non-functional analysis of software systems. The XML representation is translated by an Analysis filter as input of the desired analysis methodology. The two considered methodologies are CHARMY and TwoTowers. The former specifies software architectures and their behavioural properties by using state machines and scenarios as the source notation. Model checking is then performed to these notations in order to evaluate the consistency between the software architecture and the functional requirements. The latter supports the validation of performance requirements at an architectural level. It takes as input an AEmilia textual description (ADL), builds the corresponding Markov model and evaluates the performance indices of interest. Feedback is then provided whether the model should be modified. Despite the joint functional and non-functional analysis, this work is limited at architectural level and to the integration of only two methodologies.

Nostro et al. [25] present an approach for the automated synthesis of application layer connectors between heterogeneous networked systems, addressing functional and non-functional interoperability that takes place both at pre-deployment and run-time. During pre-deployment time, an analysis module receives the applications' specifications and through their analysis, synthesizes a mediator that enables the functional inter-operation among them. Following, a connector analysis module takes as input the synthesized mediator and the non-functional requirements and performs a stochastic model-based analysis to evaluate the desired non-functional properties. Feedback is provided back to the connector synthesis module about the system's expected operation and how to improve the synthesized mediator in the case that the non-functional requirements are not met. Pre-deployment time's output is a connector that satisfies both functional and non-functional requirements. At run-time, probes are used on the applications and the synthesized mediator to monitor the connected system. When a violation occurs, the probes identify it and trigger the adaptation process, re-evaluating the new specification. Similarly to the previous approach, this one is limited at architectural level and needs to address open issues such as analysis optimization and scalability aspects.

Filieri et al. [26,27] introduce a general methodology that uses symbolic execution of source code for extracting failure and success paths that can be used for probabilistic reliability assessment. The result of symbolic execution is a finite set of paths, each with a path condition. These paths can either lead to success, failure or can be interrupted by the bounded exploration. These approaches perform reliability analysis directly on source code, in contrast with most of the current approaches that are limited on architectural level. However, only reliability has been addressed, and the bounded exploration can potentially lead to loss of information necessary for non-functional property analysis. Our research aims to address the problem of bounded exploration of loops, and to consider additional non-functional properties, e.g. performance.

5 Conclusion

Model checking and probabilistic model checking are techniques widely used to verify functional properties of software systems and to establishing performance and dependability properties of these systems, respectively. However, the two techniques are often used in isolation. Their integration is difficult due to the different formalisms and models they use. Additionally, most of the current approaches are limited at architectural level, and the ones focused on source code have limitations in both exploration depth and variety of non-functional properties. In this doctoral paper, we proposed an approach that combines the two techniques at source code level, with the aim to provide insight to software engineers about violations of functional and non-functional requirements of software systems.

References

1. Emerson, E.A.: The beginning of model checking: a personal perspective. In: 18th International Conference on Computer Aided Verification, pp. 27–45 (2008)

2. IEEE Standard for System and Software Verification and Validation. In: IEEE Std 1012–2012, pp. 1–223 (2012)
3. Bertolino, A.: Software testing research: achievements, challenges, dreams. In: FOSE 2007, pp. 85–103 (2007)
4. Lee, P., Verma, S., Harris, I.G.: A Comparison of Error Detection between Simulation-based Validation and Model Checking. University of California, Center for Embedded Computer Systems (2013)
5. Alur, R., Courcoubetis, C., Dill, D.: Model-checking for real-time systems. In: LICS 1990, pp. 414–425 (1990)
6. Clarke, E.M., Lerda, F.: Model Checking: Software and Beyond. J. Universal Computer Science **13**, 639–649 (2007)
7. Clarke, E.M.: The birth of model checking. In: Grumberg, O., Veith, H. (eds.) 25 Years of Model Checking. LNCS, vol. 5000, pp. 1–26. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-69850-0_1
8. De Nicola, R., Vaandrager, F.: Action versus state based logics for transition systems. In: Guessarian, I. (ed.), Semantics of Systems of Concurrent Processes, pp. 407–419 (1990)
9. Kwiatkowska, M., Norman, G., Parker, D.: Advances and challenges of probabilistic model checking. In: Allerton 2010, pp. 1691–1698 (2010)
10. Norman, G., Parker, D., Kwiatkowska, M., et al.: Using probabilistic model checking for dynamic power management. Formal Aspects Comput. **17**(2), 160–176 (2005)
11. Dehnert, C., Junges, S., Katoen, J.-P., Volk, M.: A storm is coming: a modern probabilistic model checker. In: CAV 2017, pp. 592–600, (2017)
12. Cimatti, A., et al.: NuSMV 2: an opensource tool for symbolic model checking. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 359–364. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45657-0_29
13. Brat, G., Havelund, K., Park, S., Visser, W.: Java PathFinder - second generation of a Java model checker, Advances in Verification Workshop (2000)
14. Calinescu, R., Ghezzi, C., Johnson, K., Pezzé, M., Rafiq, Y., Tamburrelli, G.: Formal verification with confidence intervals to establish quality of service properties of software systems. IEEE Transact. Reliab. **65**(1), 107–125 (2015)
15. Calinescu, R., Johnson, K., Paterson, C.: FACT: a probabilistic model checker for formal verification with confidence intervals. In: Chechik, M., Raskin, J.-F. (eds.) TACAS 2016. LNCS, vol. 9636, pp. 540–546. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49674-9_32
16. Calinescu, R., Paterson, C.A., Johnson, K.: Efficient parametric model checking using domain knowledge. In: IEEE Transactions on Software Engineering (2018). https://doi.org/10.1109/TSE.2019.2912958
17. Paterson, C.A., Calinescu, R.: Accurate analysis of quality properties of software with observation-based Markov chain refinement. In: ICSA 2017, pp. 121–130 (2017)
18. Paterson, C.A., Calinescu, R.: Observation-enhanced QoS analysis of component-based systems. In: IEEE Transactions on Software Engineering (2018). https://doi.org/10.1109/TSE.2018.2864159
19. INCOSE, Resilient Systems Homepage. https://www.incose.org/incose-member-resources/working-groups/analytic/resilient-systems. Accessed 11 June 2019
20. Bennaceur, A., et al.: Modelling and analysing resilient cyber-physical systems. In: SEAMS SEAMS 2019, pp. 70–76 (2019)
21. Hansson, H., Jonsson, B.: A logic for reasoning about time and reliability. Formal Aspects Comput. **6**(5), 512–535 (1994)

22. Calinescu, R., Kikuchi, S.: Formal methods @ runtime. In: Calinescu, R., Jackson, E. (eds.) Monterey Workshop 2010. LNCS, vol. 6662, pp. 122–135. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21292-5_7
23. Calinescu, R., et al.: Synthesis and verification of self-aware computing systems. Self-Aware Computing Systems, pp. 337–373. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-47474-8_11
24. Cortellessa, V., et al.: A framework for the integration of functional and non-functional analysis of software architectures. Electron. Notes Theoret. Comput. Sci. **116**, 31–44 (2005)
25. Nostro, N., et al.: Achieving functional and non functional interoperability through synthesized connectors. J. Syst. Softw. **111**, 185–199 (2016)
26. Filieri, A., Pasareanu, C.S., Yang, G.: Quantification of software changes through probabilistic symbolic execution. In: ASE 2015, pp. 703–708 (2016)
27. Filieri, A., Pasareanu, C.S., Visser, W.: Reliability analysis in symbolic pathfinder. In: ICSE 2013, pp. 622–631 (2013)