Combining Model-Based Testing and Automated Analysis of Behavioural Models using GraphWalker and UPPAAL

Saurabh Tiwari
DA-IICT Gandhinagar, India
saurabh_t@daiict.ac.in

Kumar Iyer
DA-IICT Gandhinagar, India
201911016@daiict.ac.in

Eduard Paul Enoiu
Mälardalen University, Sweden
eduard.paul.enoiu@mdh.se

***Abstract*—Model-based Testing (MBT) has been proposed to create test cases more efficiently and effectively. In contrast, analysis techniques (e.g., model checking) have been used separately from testing and have shown great potential when applied early in the development process. Still, these are confronted by applicability and scalability issues and work on specific modeling languages. The combined use of MBT and analysis techniques can support engineers in using both dynamic and static techniques. This paper proposes a hybrid approach by combining MBT using GraphWalker (GW) with Model-Based Analysis using UPPAAL by transforming the GW model into UPPAAL timed automata and supporting a combined analysis and testing process. The approach enables the automatic verification of both reachability and deadlock freedom properties to exploit the results obtained from this analysis step to improve the test model before generating and executing test cases on the system under test. The proposed approach can improve the combination of analysis and testing using a promising open-source MBT tool and is currently being evaluated in the context of actual use cases.**

***Index Terms*—Model-Based Testing, analysis, behavioural models, model checking, GraphWalker, UPPAAL**

I. Background and Motivation

Model-Based Testing (MBT) has been proposed to allow test cases to be created with less effort. MBT involves the creation of test suites from a behavioural model generated from functional or non-functional requirements. This model is often represented as a state transition diagram [1]. One significant benefit of model-based testing is developing a specification model in the early phase of the life cycle during the requirement specification. During test generation, different execution paths of the model would lead to different test cases that can be used to find new faults or for regression [2][3][4]. An MBT tool will traverse the different paths of a model, generating abstract and executable test cases as well as executing these test cases against the system to evaluate the results.

In contrast, analysis techniques such as model checking are used to check if the model meets the specified requirements [5]. Given a model and a verification property, the model checker verifies if the model satisfies the property or not [6]. Additionally, it also generates the paths as witnesses traces and counterexamples. These traces can also be used for generating test cases [7]. Model-checking can help to validate the specifications early in the software development and thereby obtain a better test model from which test cases can be generated as well as reduce the overall cost of system development [8].

The idea of combining MBT with model checking motivated us to come up with the following research question: *"Can we connect an MBT tool with a state-of-the-art model checker?"*. For this purpose, we developed a hybrid approach that transforms a model obtained from an MBT automation tool into a model compatible with a state-of-the-art model checker. In addition, we automate the process of model checking by generating some queries to verify the model. To achieve this, we used a well known open-source MBT tool named GraphWalker (GW)[1]. Models in GW are created in the form of directed graphs. This tool lacks the capability to automatically analyze and verify if the model corresponds to certain requirements. For this purpose, we are using a state-of-the-art model checker called UPPAAL[2] to perform model checking. This integrated tool environment allows developing models as a network of timed automata and can verify specific properties on these models.

In this paper, we propose a hybrid approach that can perform an automated analysis of a GW model by transforming it into a UPPAAL model and generating queries that are automatically verified by running "verifyta" without actually running the GUI of UPPAAL to perform model checking. An initial evaluation shows that the time taken by our tool for transforming these models is consistently lower when compared to manually creating the model and properties in UPPAAL and checking these using the GUI. However, checking other properties corresponding to software requirements requires manual intervention to generate queries. Thus, our proposed approach is the first step toward combining model-based testing with automated analysis and verification tools, which can be further modified to create a more realistic and complex set of properties.

II. Related Work

Nielsen [9] presented an approach for combining analysis and testing to improve the quality of embedded systems. The author has also pointed out that it is challenging to define

[1]https://graphwalker.github.io/
[2]https://uppaal.org/

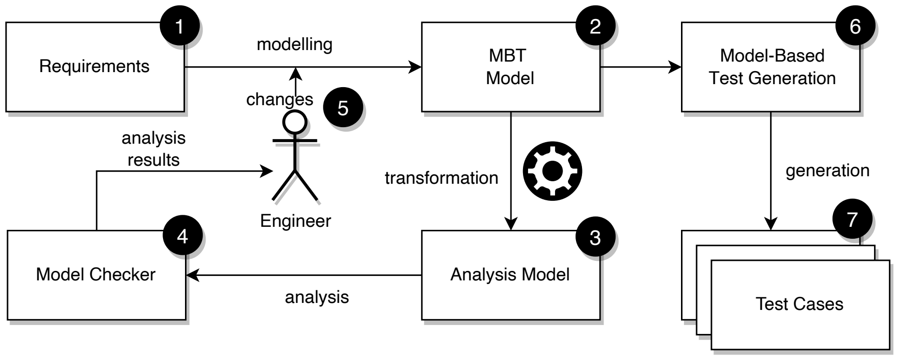

Fig. 1. The overall method for combined model-based testing and analysis using model-checking.

a common method applicable in industrial settings, but their approach seems promising. The proposed method is general and needs to be formalized for the specific combination of the analysis approaches and testing.

The combination of analysis and testing of architectural models has been considered broadly. The work by Bertolino et al. [10] reviewed the research in software architecture-based analysis and testing. Marinescu et al. [11] describes the verification of architectural models using the UPPAAL model checker. The author has introduced a framework supported by a tool called ViTAL, which captures the model as timed automata and verifies the model with UPPAAL. The input to the tool ViTaL is the EAST-ADL model. The tool converts it into a UPPAAL PORT model and supports the generation of queries to verify reachability and liveness properties.

Many studies report the use of MBT and model checking techniques separately [12], [13], [14], [15], [16]. Compared to such studies, we focus on practically combining these techniques by integrating two well-known tools.

III. Proposed Hybrid Approach

The process of combined model-based testing and model analysis using model checking aims to analyze the created model using model checking and based on the analysis results find suitable test cases using a description of the test objectives that guides towards a certain desirable property.

In Fig. 1, an overall method for model-based analysis and test generation is identified. A generic process of combined model-based analysis and test generation proceeds as follows:

- Step 1. Requirements artifacts are used or created for the purpose of guiding the test generation and analysis. In our case, the requirement artifact is either a specification of what the System-under-Test (SUT) should do in different forms (e.g., finite-state model).
- Step 2. A model is obtained using a testing tool that can be used for modelling an MBT model objective. The first step of modelling involves a human understanding the requirements and exploring the requirements specification document. For example, an FSM-based model consists of nodes and directed edges. The nodes represent the state of the system, whereas edges represent the requests/decisions when a certain event occurs.
- Step 3. In this step an automatic transformation is needed to map the test model to an analysis model used for model checking. For example, in the case of a finite-state model, guards, actions and variable declarations are used to generate a formal model.
- Step 4. Given a formal model of the system , a model-checker can be used to analyse the model given certain formalized requirements, for example, as a temporal logic formula. The model-checker returns an answer, and in some situations a model trace.
- Step 5. Based on the analysis results, the engineer could do certain changes to the original model and can continue using the MBT the model for test generation.
- Step 6. Using model-based test generation that encodes the test criteria and describes how the test generator should choose the resulting tests, one can generate test cases based on certain goals (e.g., model coverage, random test goals).
- Step 7. A test suite is generated by running the model over many possible executions using a certain model-based test generation tool.

Model-based test generation and analysis approaches can be quite different but all of them have common underlying dimensions (as shown in our hybrid approach) that can be quite helpful when implementing a combined model-based analysis and automated test generation in a certain software development project. In the following section, we instantiate this hybrid approach using specific tools.

IV. Proposed Hybrid Tooling

The combined MBT and analysis technique implemented in our tool is shown in Fig. 2. It is divided into the following five steps:

1) The test designer creates a GW model, which is exported as JSON.
2) In the next step, the JSON file is then imported into the tool by providing the file's location before executing the tool (JAR) in the command.
3) While running the GW2UPPAAL tool, the UPPAAL model in XML format is generated and used for analysis.
4) The generated UPPAAL model is then imported and executed by starting verifyta and providing the name of the generated model. This model also contains the

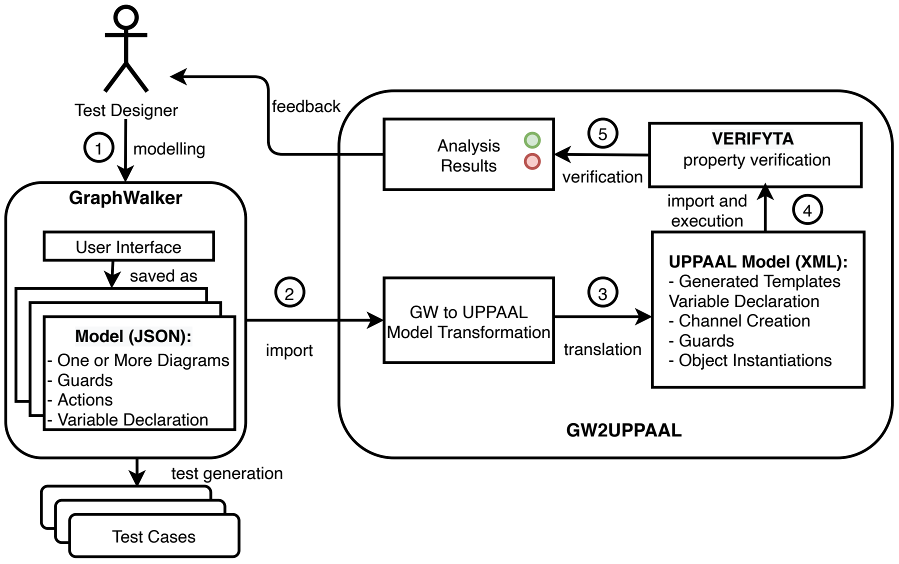

Fig. 2. The overall architecture of GW2UPPAAL tool and the interaction with GraphWalker for test generation.

queries to check the reachability and deadlock properties tested by executing "verifyta".

5) The results of this verification is then visually displayed and a test engineer will use these to analyze the model in the UPPAAL simulator to adapt the model in GW before test generation.

Apart from automated analysis, the generated model can also verify manually created queries to gain more confidence in the developed model before testing.

A. GW2UPPAAL: A Hybrid Approach

GW uses JSON (JavaScript Object Notation) as the model file format. JSON is a human-readable, lightweight, text-based data interchange format and is language-independent. On the other hand, UPPAAL exports a model in XML (Extensible Markup Language) files. Both JSON and XML are commonly used to transfer data back and forth between software programs through APIs.

Our tool transforms the GW JSON into UPPAAL compatible XML file to import the model in UPPAAL and perform model checking. The approach is divided into four steps (Section IV.B to Section IV.E). The programming language used for transformation is JAVA.

B. Importing the GW Model

The first step of GW2UPPAAL is to import the GW model (i.e., representing a state machine as shown in Fig. 3). This model is in JSON form, which is then parsed to extract the relevant data.

C. UPPAAL XML Layout Creation

Initially, we construct a basic UPPAAL template. This template is in XML form. The major challenge is to form the XML file compatible with UPPAAL and then populate it with the information needed to transform the model. In this step, we create a basic XML layout with elements specifically created

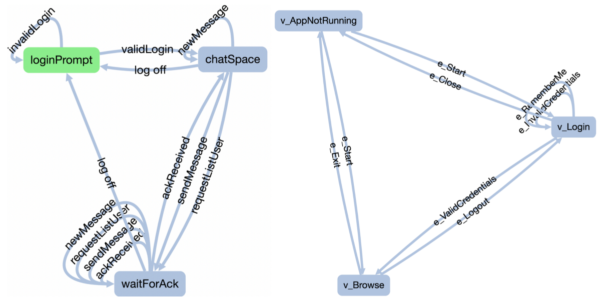

Fig. 3. Model Created in GraphWalker Studio

for UPPAAL. For example the root element ⟨nta⟩, ⟨declaration⟩ element which contains all the global declarations, XML version, the UPPAAL XML document type definition (DTD), etc. are created in this step.

D. Data Extraction from GW and the Generation of the UPPAAL Model

Each GW model contains the data that has to be extracted. For example, the model includes guards, actions, and variable declarations. In this step, this information is extracted, stored and used while generating the UPPAAL model.

For every model in GW, a model in UPPAAL is generated. If multiple diagrams have shared vertices, a single model is created in UPPAAL by flattening such diagrams. For example, the ⟨template⟩ tag is created which corresponds to an UPPAAL model, a ⟨name⟩ tag is created and populated, which describes the name of the model. Subsequent data of vertices are extracted from the GW model and added to the UPPAAL model. We also need to extract edge/transition information for generating the UPPAAL model. The information about guards and variables/actions is also extracted. The tool adds them to corresponding transitions and also initializes them with the original values. The reference to the initial state is included by adding the ⟨init⟩ tag and populating the start vertex id from GW. In the end, the tool also includes queries for verifying reachability and deadlock properties.

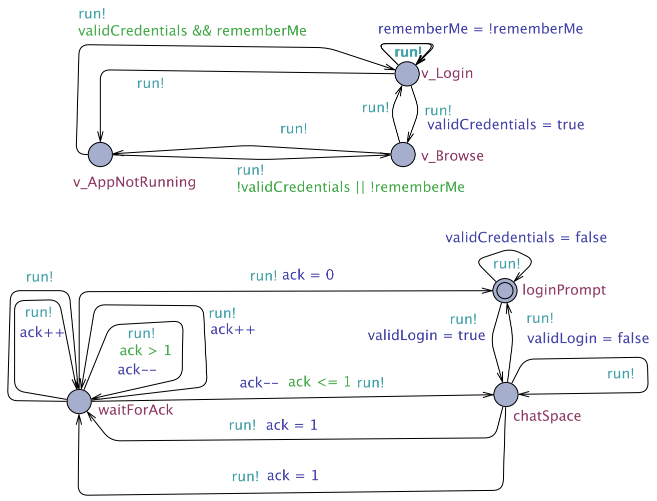

Fig. 4. UPPAAL model generated through GW2UPPAAL

E. UPPAAL Model Export

The last step is to export the UPPAAL XML. After all the information is extracted and added to the UPPAAL XML, the model is finally exported and used for model analysis and verification.

F. Demonstration

Fig. 3 shows the model created in GraphWalker studio. It is a Communication system example. Our tool generates the model shown in Fig. 4. We note here that the edge names are not included in the transformed UPPAAL model because UPPAAL needs an environment to execute the model to check these properties. By providing a common transition synchronization (i.e., run!), we create an environment that contains only one vertex with a self-loop and a synchronization channel (i.e., run?). Using this approach, the environment model can trigger transitions to all outgoing and incoming edges to use the verifier.
Demonstration video: https://youtu.be/EtsJ-GNSjJM
Code & artifacts: https://github.com/iyerkumar/GW2UPPAAL

V. Experimental Analysis and Results

We evaluated the tool with models containing single as well as multiple diagrams. Multiple diagrams are essentially part of a single model but are divided into multiple diagrams for better understanding and ease of use. These diagrams are connected by using shared vertices. A shared vertex is a vertex in the diagram that can be shared between multiple diagrams so that the test runner can execute the whole model, where the shared state is defined. This tool transforms it into a single model by flattening all the multiple diagrams. This is needed since UPPAAL does not directly support multiple diagrams in the same way as GW does.

Therefore, the GW2UPPAAL tool identifies every shared vertex and its corresponding shared vertices in all the diagrams. It creates one vertex representation for all of them and creates outgoing transitions to all the vertices that the shared vertex contains. The queries are generated in every model based on the number of vertices present in the model. Furthermore, for every vertex, the tool checks whether the vertex is reachable from the initial vertex. "Verifyta" is used to verify those properties.

Table I shows the comparison between the GW2UPPAAL approach for generating the model and a manual translation performed by the developers on ten available models. The results revealed that the time taken by GW2UPPAAL to create and execute the model is negligible compared to a manual transformation. We also found that the model generated by the GW2UPPAAL about guarantees correctness between the GW model to the UPPAAL model. Also, as the complexity of the model increases, the time taken to transform these models manually increases with the number of vertices and edges. In contrast, GW2UPPAAL can automate this modelling step and can reduce the time needed to verify the created models.

A. Features and Limitations of the Tool

Here, we report some of the benefits of using the GW2UPPAAL and, consequently, model checking via UPPAAL on a GW model. The main advantage of using UPPAAL is that it allows us to check synchronous and asynchronous flows in the system, which GW tool support lacks. We can check several requirements by manually creating multiple objects of the model or by using clocks and the updates on variables. Also, after applying the transformation through our tool, we can add manual queries. By updating the model based on the tester's need, we can check the different execution flows represented in the model. For example, we can check that while a vertex is reached, the value of a specific variable must be true. This is an example of a requirement that cannot be checked against the model directly in GW.

It is important to note that the time taken by the tool to generate the UPPAAL model is roughly between 140 and 300 ms. As expected, this time is relatively low compared to manually creating the same model. It took between 5 to 15 minutes to create the same models manually. The integration of verifyta in the tool automates the execution of the generated queries. Out of the 15 models we evaluated through the tool, we found two models where verifyta did not execute, but the model and queries were generated correctly. The tester has to manually import the generated file in UPPAAL and run the queries to check the result. There were certain cases where GW ran indefinitely, whereas UPPAAL stopped the execution environment and generated an error promptly for a disconnected graph. The generation of queries and execution through "verifyta" is the first step toward the fully automated model analysis. Nonetheless, the transformation itself is highly beneficial for the tester to obtain confidence in the test model before generating test cases. The four types of properties supported by UPPAAL are reachability, liveness, safety, and deadlock. The safety and liveness properties are dependent on the requirements, and to express these, we need to analyze the requirements and generate queries from them.

GW2UPPAAL is based on adding model checking benefits to GW. So, in this case, we are not analyzing the requirements but focusing on the transformation. Hence, GW2UPPAAL

TABLE I
COMPARISON BETWEEN THE GW2UPPAAL APPROACH FOR GENERATING THE MODEL AND A MANUAL TRANSLATION

Model Name	Type of model	#Vertices	#Edges	Time by GW2UPPAAL	Manual transformation
Amazon shopping cart	Single	6	11	0.202 s	243 s
Can deposit machine	Single	3	8	0.200 s	187 s
Coffee machine	Single	20	33	0.220 s	789 s
Communication system	Single	15	37	0.178 s	772 s
Login system	Multiple	6	20	0.176 s	642 s
Door system	Single	4	6	0.140 s	182 s
Online chat application	Single	3	10	0.174 s	202 s
Server	Multiple	14	28	0.184 s	842 s
Spotify login	Single	3	9	0.177 s	172 s
Pet clinic	Multiple	11	25	0.182 s	844 s

does not support automatically checking safety and liveness properties, but one can check these by manually creating queries. It is worth noting that the tester must have a good understanding of UPPAAL and also the query language (TCTL) supported by UPPAAL to create and execute queries manually.

VI. CONCLUSIONS AND FUTURE WORK

This paper presents an approach used for combining MBT and model-based analysis. This approach is instantiated using GraphWalker and UPPAAL tools. The GW2UPPAAL tool-suite identifies all the modeling information from the GraphWalker model. Subsequently, it transforms it into the UPPAAL model and generates queries that can also be checked automatically. Thus, GW2UPPAAL not only automates the transformation process but also bridges the gap between model checking and an MBT tool.

We have evaluated our approach and tool with models containing both single as well as multiple diagrams. We found that the transformation of the model from GraphWalker to UPPAAL is applicable for several models available in the GraphWalker documentation and models created by industrial practitioners. Also, the average time taken to generate and execute the model using GW2UPPAAL is negligible compared to the manual approach. As a part of the extension, in future, we aim to evaluate how close are the models to the ones that are automatically translated using GW2UPPAAL by conducting an experimental study. Also, the usefulness of the proposed hybrid approach in terms of how good the final model is for performing model-based testing.

The approach and its tooling automatically create queries to check reachability and deadlock properties. However, since safety and liveness properties are specific requirements, the test designer currently has to check these manually on the model generated by the tool. In the future, the gap between requirements and model checking of certain properties can be minimized by identifying key requirements from the specification and generating queries.

ACKNOWLEDGEMENT

This work has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No. 957212; from the Swedish Innovation Agency (Vinnova) through the SmartDelta project.

REFERENCES

[1] H. Robinson, "Finite state model-based testing on a shoestring," in *Proceedings of the 1999 International Conference on Software Testing Analysis and Review (STARWEST 1999)*, 1999.

[2] P. Akpinar, M. S. Aktas, A. B. Keles, Y. Balaman, Z. O. Guler, and O. Kalipsiz, "Web application testing with model based testing method: Case study," in *2020 International Conference on Electrical, Communication, and Computer Engineering (ICECCE)*, 2020, pp. 1–6.

[3] S. R. Dalal, A. Jain, N. Karunanithi, J. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz, "Model-based testing in practice," in *Proceedings of the 21st international conference on Software engineering*, 1999, pp. 285–294.

[4] A. Marques, F. Ramalho, and W. L. Andrade, "Comparing model-based testing with traditional testing strategies: An empirical study," in *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops*, 2014, pp. 264–273.

[5] M. M. Ben-Ari, "A primer on model checking," *ACM Inroads*, vol. 1, no. 1, p. 40–47, mar 2010.

[6] B. Das, D. Sarkar, and S. Chattopadhyay, "Model checking on state transition diagram," in *ASP-DAC 2004: Asia and South Pacific Design Automation Conference 2004 (IEEE Cat. No.04EX753)*, 2004, pp. 412–417.

[7] P. E. Ammann, P. E. Black, and W. Majurski, "Using model checking to generate tests from specifications," in *Proceedings second international conference on formal engineering methods (Cat. No. 98EX241)*. IEEE, 1998, pp. 46–54.

[8] W. Chan, R. J. Anderson, P. Beame, S. Burns, F. Modugno, D. Notkin, and J. D. Reese, "Model checking large software specifications," *IEEE Transactions on software Engineering*, vol. 24, no. 7, pp. 498–520, 1998.

[9] B. Nielsen, "Towards a method for combined model-based testing and analysis," in *Proceedings of the 2nd International Conference on Model-Driven Engineering and Software Development*, ser. MODELSWARD 2014. Setubal, PRT: SCITEPRESS - Science and Technology Publications, Lda, 2014, p. 609–618. [Online]. Available: https://doi.org/10.5220/0004873106090618

[10] A. Bertolino, P. Inverardi, and H. Muccini, "Software architecture-based analysis and testing: a look into achievements and future challenges," pp. 633–648, 2013.

[11] R. Marinescu, C. Seceleanu, and P. Pettersson, "An integrated framework for component-based analysis of architectural system models," in *ICTSS 2012 Ph. D. Workshop held in conjunction with the The 24th IFIP Int. Conference on Testing Software and Systems (ICTSS'12) in Aalborg, Denmark, November 19, 2012.*, 2012, pp. 1–6.

[12] M. Utting, A. Pretschner, and B. Legeard, "A taxonomy of model-based testing approaches," *Software testing, verification and reliability*, vol. 22, no. 5, pp. 297–312, 2012.

[13] T. Ahmad, J. Iqbal, A. Ashraf, D. Truscan, and I. Porres, "Model-based testing using uml activity diagrams: A systematic mapping study," *Computer Science Review*, vol. 33, pp. 98–112, 2019.

[14] L. Villalobos Arias, C. U. Quesada López, A. Martínez Porras, and M. Jenkins Coronas, "A tertiary study on model-based testing areas, tools and challenges: Preliminary results," 2018.

[15] A. K. Karna, Y. Chen, H. Yu, H. Zhong, and J. Zhao, "The role of model checking in software engineering," *Frontiers of Computer Science*, vol. 12, no. 4, pp. 642–668, 2018.

[16] M. Luckcuck, M. Farrell, L. A. Dennis, C. Dixon, and M. Fisher, "Formal specification and verification of autonomous robotic systems: A survey," *ACM Computing Surveys (CSUR)*, vol. 52, no. 5, pp. 1–41, 2019.