Vahid RAFE

Arak University

Formal Analysis of Service-oriented Architectures

Abstract*. Even though model checking is one of the most accurate analyses techniques to verify software systems, the problem of model checking is that it is not feasible for large and complex software systems, which their state spaces are too large. In these situations one can use scenario-based model checking techniques. In this paper, we present an approach to analyze large and complex systems specified by graph transformation systems. To do so, we propose the scenario-based model checking techniques. We explain how the approach will affect to the size of the state space by focusing on the most important occurring scenarios in a system.*

Streszczenie. *Problem sprawdzania modelu software jest istotny jeśli badamy duży i złożony system. W artykule zaprezentowano metodę umożliwiającą taką analizę systemu opisanego przez transformację grafu. Zaproponowano technikę sprawdzania opierającą się na modelu bazującym na scenariuszu. (****Analiza formalna architektury typu service-oriented****)*

Keywords: Graph Transformation, SoA, Verification, Bogor.
Słowa kluczowe: software, transformacja grafu.

Introduction

For the most of the modern software systems it is desirable to operate “correctly” (especially for safety-critical systems) [1]. By “correctly” we mean that in these systems there must be no bug and all the requirements must be covered by the developed software system. Finding the bugs and defects in the early stages of development process has less cost. So, it is better to do verification on the model in the design stage and before implementation.

Another important issue is that some times it is desirable to analyze different critical application scenarios and operations (like database access, user interaction, communication with remote components, etc.) [2] on the model before implementation in order to judge the realizability of a given application scenario. Especially, in distributed applications it is very important to choose the right platform middleware before implementation to reduce the cost.

Model checking techniques are one of the most accurate and automatic techniques to not only verify safety-critical systems, but also to analyze different critical application scenarios. But as a precondition for using model checking techniques, it is necessary to specify the system through a proper formalism like graph transformation systems.

Graphs and diagrams are a very natural means to describe complex structures and systems and to model concepts and ideas in a direct and intuitive way [3]. For example, the structure of an object-oriented system or the execution flow of a program can be seen as a graph. Graph transformation [3, 4] is very popular as a high-level and expressive specification formalism (e.g. to formally capture software requirements). Therefore using graphs and graph transformation systems as a formal background for software modelling is a natural choice. But even using graph transformation for modeling, it is necessary to find a proper solution for verification. There exist different approaches to verify graph transformation through model checking like [5,6,7]. All these approaches can deal with priori bounded transition systems because it is the nature of the model checking techniques generally.

In this paper we follow the line of works presented in [8] and [6,9]. In [8], the authors present an style-based approach for modeling service oriented architectures. Then the authors explain the importance of analyzing application scenarios in distributed applications. But for the lack of a proper approach, they cannot analyze the desired scenarios efficiently and they just do some experiments by simulation. In fact, the most important part of their work is proposing an approach for modeling service oriented applications through an architectural style. But modeling is not enough since users want to be able to discover the interesting properties behind their models. Thus the modeling must be complemented with proper analysis capabilities. In [6,9] the authors present an efficient solution for model checking graph transformation using a model checker called Bogor [10]. The presented approach has some key characteristics like supporting layered graphs [9], verification of attributed and typed graph transformations and supporting different kinds of properties for verification (e.g. safety, reachability, liveness and deadlock freeness). In this paper we use this approach to verify different scenarios on the systems which are modeled using style presented in [8]. Note that we use models presented in [8] for analysis, because we believe that it is a practical approach can be used to develop large, dynamic, loosely coupled and service oriented systems. But in general, the proposed approach can be used for analyzing different systems which are specified by graph transformation systems.

The rest of the paper is organized as follows: in the section 2, the required background i.e. Bogor and graph transformation will be discussed. In Section 3, the proposed approach will be explained. In section 4, the related work will be presented and section 5 concludes the paper.

Bogor and Graph Transformation

We use the approach presented in [6,9] to analyze the scenarios. In [6,9], we have proposed a method for encoding GTS along with different properties to BIR[1] –the input language of Bogor-. So, it is necessary to briefly introduce the main characteristics of the presented approach.

Bogor is an extensible software model checking framework developed at Kansas State University. It has novel capabilities which make it a proper framework to use in domain-specific analysis approaches.

In Bogor, control-flow and actions are stated in a guarded command format: guard expression evaluates conditions, while actions (commands) modify the states of the system. The guarded commands are placed into one or more locations. The locations are created by labeling parts of the code using ‘loc’ in BIR. It is also possible to modify the control-flow by explicitly jumping among locations using ‘goto’ in BIR.

For example, consider the BIR model of Fig 1, it comprises a global variable x initialized to 20 and a thread

[1] Bandera Intermediate Representation

(MAIN) containing two locations (loc0 and loc1). The execution of a BIR model, always start from the thread MAIN. In the thread MAIN of Fig 5, at the beginning, the guards in the first loc (loc0) are checked. Both guards are evaluated to true, so Bogor chooses among them non-deterministically. The first guard from the loc0 checks whether x is even, and if it is the case, computes its new value (x:=x/2). Then statement goto loc0 shows that loc0 is the next location to reach. The second guard in loc0, checks whether x is a factor of 5 and, if it is the case, the new value is x:=x/5. then the command goto loc1 shows that the next location is loc1. During the verification, Bogor creates an automaton whose states represent the result of the execution of the whole program. Hence, each time that the value of x is changed, it generates a new state and it keeps doing so as long as it does not detect any new state.

```
system example{
  int x:=20;
 main thread MAIN(){
  loc loc0:
      when x%2==0 do{ x:=x/2;} goto loc0;
      when x%5==0 do{x:=x/5;} goto loc1;
    loc loc1:
      when x%2==0 do{x:=x/2;} goto loc1;
      when x%2!=0 do {x=:=3*x+1;} goto loc1;
  }
 }
```

Fig. 1. An example BIR model

Now, after a short description of Bogor, we can represent our approach to verify graph transformation systems through Bogor. The main steps of the proposed approach to encode graph transformation systems are summarized as follows: (for more details, interested readers can refer to [9])

- The first step is encoding type graph into BIR to define the date structures needed later. Each note type is translated to a 'record' contains all the information in the node (include attributes and associations), then we add a further record to represent the graph itself.
- The second step is instantiating the host graph. To do so, a variable of type 'graph' must be declared in BIR. Then in the first loc (loc0) using only one true guard we initialize the host graph. Note that this guard does not need any condition.
- In the third step, the rules must be rendered into BIR. This process is done in the second loc (loc1). Each LHS (Left Hand Side) of the rule (and NAC –the Negative Application Conditions- if present) are encoded into one or more guards while the RHS (Right Hand Side) is rendered into the actions.
- In the end, the properties must be translated into BIR. To do so, we use special graph rules in which LHS shows the positive conditions, NAC shows the negative ones and the RHS is similar to the LHS while it does not change the states.

Our Proposed Approach

There is a growing need for large, but distributed, flexible and loosely coupled software systems. To support flexibility, modern distributed applications are usually deployed on top of a platform middleware like CORBA, EJB or Web Services.

However, the costs and the level of support provided may differ among the platforms, as well as the precise rules of interaction between components. Choosing the right platform for a given application is therefore a mission-critical question, especially as later migration may be costly, if not impossible [2].

In order to reduce this risk, it is common to produce a prototype implementation covering all critical application scenarios and operations, like database access, user interaction, communication with remote components, etc. Following a model-driven development approach, application scenarios are expressed at a conceptual level, for example, in terms of UML [11] use case and sequence diagrams. This representation allows to reason about requirements at a high level of abstraction, without considering

implementation-specific details and it has a direct impression on the costs.

Our proposal is to check the reachability of consecutive configurations obtained from the sequence diagrams by horizontal cuts after abstracting from application specific details of business messages.

Here we are interested in showing that starting from the initial configuration, applying a proper sequence of graph transformation rules, the configuration obtained after sending the business message in the first cut of the sequence diagram is reachable (in the cases in which the scenario is realizable). But the problem is that we do not know the elementary operations how and in which order are performed.

How ever, we can state the configuration after the sending message in the first cut, in terms of a reachability graph and then check it using Bogor according to the approach presented in [9].

Consider the first cut in scenario specified through a sequence diagram. We know after sending such a message a reachability graph must be reachable. If we call this reachability graph P1 we can simply check it by the LTL formulae: ◊(P1). Now, to verify other parts of the scenario, we can consider other cuts and define the reachability graphs P2,P3,..,Pk from the cuts 2,3,..,k respectively. Then using the LTL formula ◊(P1^ ◊(P2^◊(…))) one can check the realizability of the scenario.

Although the approach presented above seems sound theoretically, but in practice there are some limitations. For example, it is not practical to translate and verify the mentioned reachabilty property in Bogor all at once, because it can be complicated. In addition, all the state space (or a large part of it) must be searched by Bogor. Now we can use a better (more optimized) solution.

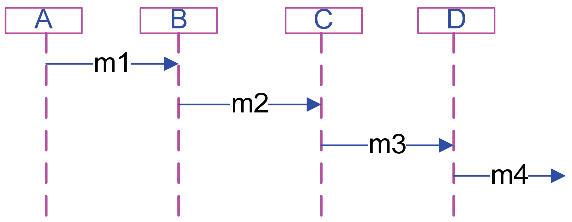

Fig. 2. A sequence diagram representing a general scenario

The idea: first, we must remove objects (nodes) in the host graph which are not involved in the scenario. Note that the deleted nodes must be from business related components and it cannot be a part of non-functional components. Consider a general scenario represented by a sequence diagram as shown in Fig. 2.

Our proposal is to check the reachability of consecutive configurations obtained from the sequence diagrams by horizontal cuts after abstracting from application specific details of business messages. Hence, in the first cut of the sequence diagram of Fig 2, we must define the configuration of the system after sending the message m1 from object A to B through a rechability property named P1. Now to verify it, we check ◊(P1), in fact, using the translator presented in [9], this property will be translated to a BIR function and then Bogor performs the verification. If the reachability property holds on the model, we can consider the second cut and so forth. But to increase the performance, we use another approach. We check the safety property ☝(¬P1) (the negation of the rechability property) to obtain the sequence of applied graph rules performed on the initial state. If the first cut is reachable from the initial configuration H1, the model checker must report an error along a counter-example when checking the safety property. Tracing the counter-example, we obtain a sequence of graph rules applied to the initial state H1. suppose this sequence consists of rules R1,R2,…,Rk. it means starting from the state H1, then applying rules R1, R2,…,Rk to H1 (with the same order) results the state H2: $H_1 \xrightarrow{R_1} G_1 \xrightarrow{R_2} G_2 ... \xrightarrow{R_k} H_2$, in which Gi's are intermediate generated host graphs (states). As we mentioned it before, for model checking we use AGG and Bogor. In fact, the system is modeled in AGG. Now, we make a very simple modification on the model in AGG. We add a node type called "RuleNo" to the type graph. This node has an attribute named "No" of type integer. In the host graph, we add a node of type RuleNo with the value equal to 1 for its attribute "No". For each rule in the set of R1, R2,…,Rk , we add two nodes of type RuleNo, one into the LHS and the other to the RHS. Then for the rule Ri (1≤i≤k), we set the value of "No" attribute in the LHS to i and the RHS to i+1. For other rules in the graph transformation system which are not in set (R1,R2,…,Rk), (i.e. k<i), we set the values to n+1 (n is the number of all the rules). Using this approach, only rules R1, R2,…,Rk will be applied to the host graph (in the same order). Using AGG and starting from the host graph H1, by opening the "Transform" tab and choosing "start" the host graph H2 will be generated after a while. Now, for the second cut in the sequence diagram, we use the host graph H2 for the starting state rather than H1. The reason is that (1) for the second cut, the analysis can continue from the state in the last step and the model checker does not need to generate the previous states, so it is possible to find reachable states (for the next cuts) sooner and (2) it is possible to change the rules (not always) in a way that the size of the transition system is reduced (without changing the semantics), as we will discussed more about it in this section.

Now we consider the second cut in which the component B sends the message m2 to the component C in Fig 2. Suppose the configuration of the system after receiving m2 by B, can be defined through a reachability property P2. This time we start from H2 and then check the reachability of P2. If it is reachable, then we check its negation form as a safety property ☝(¬P2). If the second cut is reachable from the configuration H2, the model checker must report an error along with a counter-example. Tracing the counter-example, we obtain a sequence of graph rules applied to the state H2. suppose this sequence consists of rules R1',R2',…,Rk'. it means starting from the state H2, then applying rules R1', R2',…,Rk' to H2 (with same order) results the state H3. This procedure can continue until either an error occurs (the safety property holds on the model), in this case the overall property (scenario) is not satisfied (realizable) on the mode, or a finial state Hc is reached (c is the number of cuts) and it means that the overall property (scenario) is satisfied (realizable) on the model. In this case we have:

$$H_1 \xrightarrow{cut1} H_2 \xrightarrow{cut2} H_3 . \quad . \quad . \xrightarrow{cutc} H_c$$

Using the approach presented above, it is possible to verify larger models and scenarios because each time (after a cut) we use the state generated in the previous step as the staring state for the next step and it seems a branch and bound paradigm, because in each step we reduce the state space by eliminating those states which were generated in the previous step. Although using this approach is semi automated, but it is possible to analyze larger models and more complicated scenarios.

Related Work

In [12,13] the authors presents an architectural style to model service oriented architectures, but for the modeling they just mention that it is possible to analyze reachability properties using model checking, without explanation how it is possible. In fact, in these works, the authors have a more stress on modeling rather than analysis.

In [2] authors present a style-based approach for modeling service oriented architectures using styles. The styles are defined using graph transformation systems. For analyzing scenarios, they suggest model checking. They use MorØ [14] for this purpose. They encode graph transformations to MorØ code and then MorØ carry outs the model checking. But they did not present a general (automatic) approach to verify graph transformations using MorØ. In addition, in the cases in which all the transition system must be searched, the authors mention that the state space explosion will occur. Also, they emphasize "that unsuccessful preliminary model checking attempts were also useful from a validation point of view, as they revealed several unexpected side effects of graph transformation rules".

The same authors extend their work [8]. In this extension, they use UML stereo types to use by designers who are not familiar with graph transformation systems. For analysis, they use scenarios and model checking. They use CheckVML for model checking, as it is not suitable for dynamic models [15] (models in which nodes are added/deleted by transformation rules while the system evolves), so they do some experiments using simulation.

There are also other approaches which use different notations rather than graphs to state the software architecture; hence they use different approaches for analysis. The use of model checking techniques for verifying software architectures has been thoroughly studied by several proposals. vUML [16], veriUML [17], JACK [18], and HUGO [19] support the validation of distributed systems, where each statechart describes a component, but do not support any complex communication paradigm. So, it is not possible to analyze scenarios by these approaches.

Conclusion

In this paper, we have introduced scenario-based model checking, an efficient solution to analyze large and complicated software systems. The idea is rendering properties to scenarios. Then, one must check the reachability of consecutive configurations obtained from the sequence diagrams by horizontal cuts after abstracting from application specific details of business messages.

Using this approach, one can analyze the result of different critical application scenarios before implementation. In addition, considering the only components (nodes) involved in the scenario, and ignoring other components, the performance can be increased. Especially, if we could change the rules in a scenario-directed way, the performance will be considerably better.

We must notice that this approach is not a general approach applicable for analyzing every property or even every model because it is not always possible to render a property to a scenario or changing the model. But still it is a solution which can be used to analyze large and complex software models.

REFERENCES

[1] C. baier and J.P. Katoen. Principals of Model Checking. The MIT Press, (2008)

[2] L. Baresi, R. Heckel, S. Thöne, and D. Varró. Modeling and validation of service-oriented architectures: Application vs. style. In Proc. ESEC/FSE 03 European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering, pages 68–77. ACM Press, (2003)

[3] L. Baresi, R. Heckel. Tutorial introduction to graph transformation: a software engineering perspective. Proc. First Int. Conf. Graph Transformation (ICGT), (LNCS, 2505), pp. 402–429, (2002)

[4] H. Ehrig, K. Ehrig, U. Prange and G. Taentzer: Fundamentals of Algebraic Graph Transformations, Springer, (2006)

[5] A. Rensink. The GROOVE Simulator: A Tool for State Space Generation, In Applications of Graph Transformations with Industrial Relevance (AGTIVE), vol. 3062 of Lecture Notes in Computer Science, 479–485, (2004)

[6] L. Baresi, V. Rafe, A. T. Rahmani and P. Spoletini: "An Efficient Solution for Model Checking Graph Transformation Systems", Electronic Notes in Theoretical Computer Science (ENTCS), Vol. 213, Elsevier Science B.V., ISSN: 1571-0661, PP. 3-21 (2008)

[7] A. Schmidt and D. Varró, CheckVML: A Tool for Model Checking Visual Modeling Languages, In Proc. of 6th International Conference on the Unified Modeling Language (UML), vol. 2863 of LNCS, 92–95, (2003)

[8] L. Baresi, R. Heckel, S. Thöne, and D. Varró. Style-based modeling and refinement of service-oriented architectures. Int. Journal on Software and Systems Modeling, SoSyM, (2006)

[9] V. Rafe, A. T. Rahmani, L. Baresi, P. Spoletini: "Towards Automated Verification of Layered Graph Transformation Specifications", journal of IET Software, Vol.3 No.4, pp. 276-291 (2009).

[10] Robby, M. Dwyer, and J. Hatcliff. Bogor: An Extensible and Highly-Modular Software Model Checking Framework, In Proc. of the 9th European software engineering Confference, 267–276, (2003)

[11] Object Management Group. UML specification version 1.4, (2001). http://www.omg.org/uml/.

[12] L. Baresi, R. Heckel, S. Thöne, and D. Varró. Style-based refinement of dynamic software architectures. In Proc. of the 4th Working IEEE/IFIP Conference on Software Architecture, WICSA 2004, pages 155–164. IEEE Computer Society, (2004)

[13] L. Baresi, R. Heckel, S. Thöne, and D. Varró. Modeling and analysis of architectural styles based on graph transformation. In Proc. 6th ICSE Workshop on Component-Based Software Engineering (CBSE6): Automated Reasoning and Prediction, pages 67–72, (2003)

[14] The MurØ. Model Checker: http://verify.stanford.edu/dill/murphi.html

[15] Rensink A., Schmidt A´ ., Varro D.: 'Model checking graph transformations: a comparison of two approaches'. Proc. Second Int. Conf. Graph Transformation (ICGT), (LNCS, 3256), pp. 226–241, (2004)

[16] J. Lilius and I.P. Paltor. vUML: a tool for verifying UML models. In Proceedings of the 14th IEEE International Conference on Automated Software Engineering (ASE), pages 255–258, (1999)

[17] K. Compton, Y. Gurevich, J. Huggins, and W. Shen. An automatic verification tool for UML. Technical Report CSE-TR-423-00, University of Michigan, EECS Department, (2000)

[18] S. Gnesi, D. Latella, and M. Massink. Model checking UML statecharts diagrams using JACK. In Proceedings of the 4th IEEE International Symposium on High Assuarance Systems Enginering (HASE), pages 46–55. IEEE Press, (1999)

[19] T. Schäfer, A. Knapp, and S. Merz. Model checking UML state machines and collaborations. Electronic Notes in Theoretical Computer Science, 55(3):13 pages, (2001)

***Authors**: Dr. Vahid Rafe, Department of Computer Engineering, Faculty of Engineering, Arak University, Arak 38156-8-8349, Iran, v-rafe@araku.ac.ir*