Tool Support for Generation and Validation of Traces between Requirements and Architecture*

Arda Goknil
Software Engineering Group,
University of Twente
7500AE, Enschede, the Netherlands
a.goknil@ewi.utwente.nl

Ivan Kurtev
Software Engineering Group,
University of Twente
7500AE, Enschede, the Netherlands
kurtev@ewi.utwente.nl

Klaas van den Berg
Software Engineering Group,
University of Twente
7500AE, Enschede, the Netherlands
k.g.vandenberg@ewi.utwente.nl

ABSTRACT
Traceability is considered crucial for establishing and maintaining consistency between software development artifacts. Although considerable research has been devoted to relating requirements and design artifacts with source code, less attention has been paid to relating requirements with architecture by using well-defined semantics of traces. We present a tool that provides trace establishment by using semantics of traces between R&A (Requirements and Architecture). The tool provides the following: (1) generation/validation of traces by using requirements relations and/or verification of architecture, (2) generation/validation of requirements relations by using traces. The tool uses the semantics of traces together with requirements relations and verification results for generating and validating traces. It is based on model transformations in ATL and term-rewriting logic in Maude.

Categories and Subject Descriptors
D.2.1 [**Requirements/Specifications**]: Tools; D.2.2 [**Design Tools and Techniques**]: Computer-aided software engineering (CASE)

General Terms
Documentation, Design, Languages, Verification

Keywords
Tools, Traceability, Generation and Validation of Traces, Architecture Verification, Requirements and Architectural Models

1. INTRODUCTION
Traceability is considered crucial for establishing and maintaining consistency between software development artifacts such as requirements documents, architectural design, detailed design, source code and test cases. The benefits of traceability are widely acknowledged today and there are tools to record and manage trace information. Despite many advances in tools, traceability remains a widely reported problem area in industry [14]. Some traceability approaches aim at generating trace information automatically [9] [10]. Egyed et al. [10] proposes an automated traceability approach that uses a small number of traces as input. In addition to that, some heuristics helping to define the meaning of trace dependencies are proposed.

Considerable research has been devoted to relating requirements and design artifacts with source code. Less attention has been paid to relating requirements with architecture by using well-defined semantics of traces. Most approaches focus on generating traces between requirements and source code or between design and source code. In most tools and approaches, there is a lack of precise definition of traces between requirements and architecture. This lack may cause incomplete and invalid trace generation for requirements and architecture.

Traceability influences a number of software development activities such as release planning, change impact analysis, testing, and requirements reuse [8]. In this respect, these activities may produce deficient results because of invalid and incomplete traces. For instance, change impact analysis may produce high number of false positive and false negative impacted elements. Consequently, the cost of implementing a change may become several times higher than expected [8].

In this paper, we present a tool that provides trace establishment by using semantics of traces between R&A (Requirements and Architecture). The tool provides the following: (1) generation/validation of traces by using requirements relations and/or verification of architecture, (2) generation/validation of requirements relations by using traces. *Generating traces* is the activity of deducing traces between requirements and architecture based solely on verification of architecture and/or the requirements relations. Alternatively, the software architect can provide an initial set of traces as input. *Validating traces* is the activity of identifying the traces which do not obey trace semantics. Our tool checks if the relations between requirements are preserved in their implementation in the architecture. This preservation is also used in the concept of *software reflexion models* where relations between elements in high-level models are preserved in their implementations [24].

In the implementation of the tool, we bridge three technical spaces [19]: *Semantic Web*, *Term rewriting*, and *Model-Driven*

ECMFA-TW'10, June 15, 2010, Paris, France.

* This research is part of the QuadREAD project (http://quadread.ewi.utwente.nl).

Engineering. In [12], we provide *tool support* for consistency checking and inferencing based on the semantics of relations for requirements. The requirements metamodel together with semantics of requirements relations in first-order logic (FOL) are given in [12]. The tool in [12] uses the semantic web technologies OWL and Jena for inferencing and consistency checking (*Semantic Web technical space*). The output of the tool in [12] is the requirements model which is used as an input by our tool for trace generation and validation. The architecture is expressed in Architecture Design and Analysis Language (AADL) [2]. We have defined dynamic semantics for part of AADL in terms of rewriting logic supported by the Maude language and tools [6] [7]. This enables performing simulation and verification of AADL models [29] (*Term rewriting technical space*). The result of the verification, which might be a counter example or execution trace, is one of the inputs to generate and validate traces. We use a trace metamodel with commonly used trace types. The semantics of the traces is provided with a formalization [13] in first-order logic. We use the formalization of traces together with requirements relations and verification results for generating and validating traces. The core part of the tool using verification results, requirements relations and traces as input is implemented as model transformations in ATL [18] (*Model-Driven Engineering technical space*).

The paper is structured as follows. Section 2 describes the overview. Section 3 presents the features of the tool. In Section 4, we explain the architecture of the tool. Section 5 describes the evaluation of the tool. Section 6 describes the related work, and in Section 7 we conclude the paper.

2. OVERVIEW

Our tool supports trace generation and validation with different degrees of automation. Figure 1 gives the overview of the tool with inputs and outputs. The tool takes *requirements model*, *trace model*, *architecture model*, and *reformulated requirement* as input with the *constraints* derived from the semantics of traces and requirements relations.

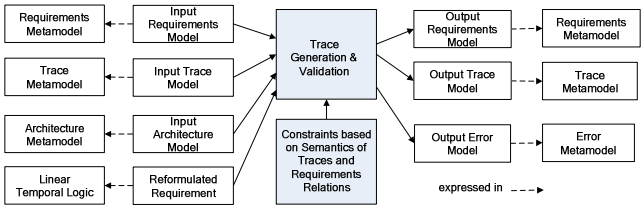


Figure 1 Overview of the Tool

The tool checks if the requirements are satisfied by the architecture. This is done by reformulating the requirements in terms of logical formulas over the architecture. To check the formulas we perform architecture simulation and verification in Maude by using input architecture model. According to the result of the verification, traces with different types (*AllocatedTo* or *Satisfies*) are generated between the reformulated requirement and the architecture. Traces are generated accordingly in the output trace model. In the validation, the tool compares the assigned traces in the input trace model with the architectural elements in the verification output. The invalid assigned traces are reported in the output error model.

The input requirements model contains the given and inferred requirements relations (see [12]). The constraints derived from the semantics of trace and requirements relations are used to deduce the new traces. Traces are generated for requirements which do not have any traces but which are related to requirements with traces. The output trace model contains the generated traces. The generation is vice versa. The same constraints are used to generate requirements relations from traces between requirements and architecture. The tool also uses the requirements relations and the constraints to check the validity of the assigned traces in the input trace model and the validity of requirements relations in the input requirements model. Invalid traces and requirements relations are reported in the output error model.

We have to note that all generated/invalid traces and requirements relations are candidates and suggestions for the software architect. They have to be checked by the architect for the final decision.

We depict the usage of the tool in a modeling process with trace generation and validation. This process is based on the analysis of activities during modeling of requirements, architecture and traces. Figure 2 gives a UML activity diagram of the process.

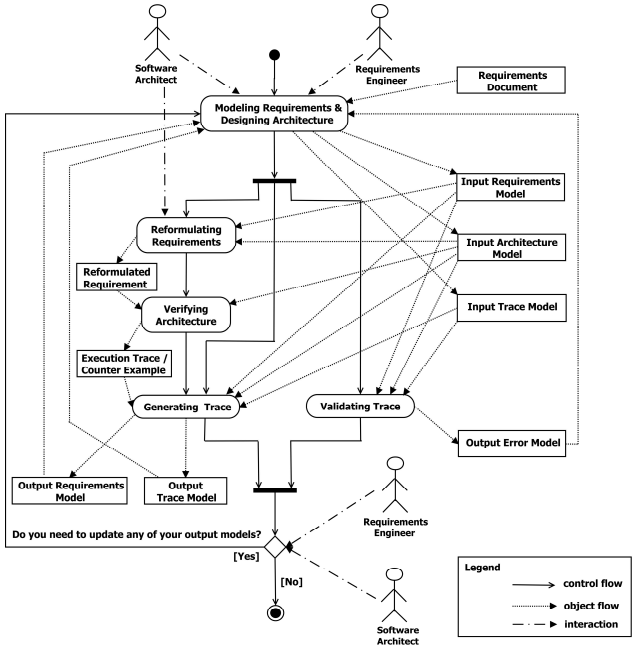


Figure 2 Modeling Process with Trace Generation and Validation

The process in Figure 2 consists of the following activities:

Modeling Requirements & Designing Architecture*:* This activity takes the requirements document as input and produces the input requirements model, input architecture model and input trace model for trace generation and validation. The requirements engineer models the requirements in the requirements document by assigning relations between them with tool support in [12]. The software architect designs the software architecture for the requirements and can assign some initial traces between requirements and architecture.

The modeling process is forked into three activities: *reformulating requirements*, *generating trace* and *validating trace*.

Reformulating Requirements: This activity takes the input requirements model and input architectural model as input and produces the reformulated requirement as output. The software architect reformulates the requirements in terms of logical formulas over the architecture.

Verifying Architecture*:* This activity takes the input architectural model and the reformulated requirement as input and produces execution trace or counter example (see Section 3.1). The activity checks whether the requirements are satisfied by the architecture. The activity is done in Maude automatically.

Generating Trace*:* This activity takes the input trace, requirements and architecture models with the output of verifying architecture as input and produces output trace and requirements models as output. If the activity uses only requirements relations in the requirements model and initial traces in the input trace model to generate traces and/or requirements relations, it is sufficient to perform this activity after modeling requirements & designing architecture activity only. The activity is automatic.

Validating Trace*:* This activity takes the input trace model, input requirements model, input architecture model as input and produces an output error model as output. This activity is automatic. However, the interpretation of the output of this activity (the output error model) with the trace model should be done by the software architect manually.

Iterating: The process given in Figure 2 is iterative. The requirements engineer and/or the software architect may return to the requirements modeling & designing architecture activity in order to fix requirements relations and/or traces in the output models. If there is no need to update the models, the process is terminated.

In order to implement the tool, we successively provide the followings:

- *Trace Metamodel*. We use a trace metamodel to provide a structure to traces. The metamodel includes most commonly found entities in trace metamodel in literature, and requirements & architecture specific traces. We use two types of traces between requirements and architecture: *AllocatedTo* and *Satisfies*.
- *Semantics of Traces.* In the literature, traces in the trace metamodel are informally defined. We formalize requirements, architecture and traces between them by using FOL.

The tool uses traces with requirements relations. Therefore, we need semantics of requirements relations.

- *Semantics of Requirements Relations.* We identified five types of requirements relations: *requires*, *refines*, *partially refines*, *contains*, and *conflicts*. The requirements metamodel together with semantics of requirements relations in FOL are already given in [12].
- *Architecture and Verification.* The software architecture is expressed in Architecture Design and Analysis Language (AADL) [2]. We have defined dynamic semantics for part of AADL in terms of rewriting logic supported by the Maude language and tools [6] [7]. This enables performing simulation and verification of AADL models [29]. For the verification, architectural significant functional requirements are reformulated as formalized scenarios and then linear temporal properties are checked using linear temporal logic (LTL) [3]. Applying verification techniques for requirements is not the main focus of our paper. The details can be found in [28].
- *Generating and Validating Traces.* We use semantics of traces and requirements relations with architecture verification techniques for generating and validating traces.

3. FEATURES OF THE TOOL

3.1 Verification of Architecture for Functional Requirements

We limit ourselves to verification of functional requirements only. The purpose of the verification is to check if requirements are correctly implemented in the architecture. The tool uses verification results in both trace generation and trace validation as an input. Figure 3 illustrates the verification of architecture for functional requirements.

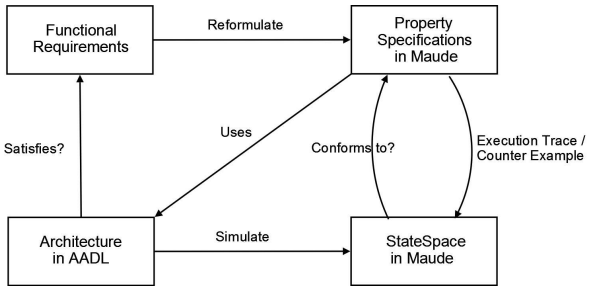


Figure 3 Verification of Architecture for Functional Requirements

The verification is represented by the *Satisfies* and *ConformsTo* relations in Figure 3. *ConformsTo* implies that the state space captures the specified properties. We have the following artifacts in the verification of architecture:

- *Functional Requirements*. Requirements which describe the functions that the system is to execute; for example, formatting some text or receiving a data.
- *Architecture in AADL*. The architecture of the system to be built. It plays the role of the solution for the problem defined by the requirements.
- *Property Specifications in Maude*. The formal description of the required behavior of the architecture. The requirements are reformulated as properties in terms of the solution, which is the architecture (*reformulate* and *uses* in Figure 3). These properties are checked for the architecture by the model checker. The requirement is first described as a formalized scenario, and then described as property specification [28]. The property specification uses any logic such as *Linear Temporal Logic (LTL)*, *First-Order Logic (FOL)*, or *Computation-Tree Logic (CTL)*. In the tool, we use the formal analysis features of Maude.
- *State Space in Maude*. The presence of a dynamic semantics specification of AADL in Maude makes the architectural models executable. The architecture is executed and a state space is produced (*simulate* in Figure 3). This execution simulates the behavior of the system on the architecture level

so that it can be studied to see how the system will work. Discrete event simulation, which introduces the notion of events, states, and state space, is used. A state describes the loci of data values within the architecture. Two states are connected by a transition and all states are captured by the state space. The result of the verification, which might be a counter example or execution trace, is used to generate and validate traces. An execution trace is the ordered set of states which are generated where the reformulated requirement is satisfied. A counter example is the ordered set of states which are generated where the reformulated requirement is not satisfied.

Since the focus of this paper is not verification and simulation, we do not give details of the AADL semantics used in the tool. This is itself a non-trivial topic and subject of another work.

3.2 Generation of Traces

Generating traces aims at deducing traces between requirements and architecture based solely on verification of architecture and/or the requirements relations in the requirements model. Our tool does not need initial traces to generate new traces.

The tool uses the result of the verification of architecture. If the verification is successful, the architecture satisfies the requirement. According to the semantics of traces in [13], the *Satisfies* trace is generated between the architectural elements in the execution trace and the requirement. These elements collectively satisfy the requirement and form the part of the architecture to which the requirement is traced. A counter example means that although the requirement is *allocated to* the architectural elements, the architecture does not satisfy it. The *AllocatedTo* trace can be generated but the *Satisfies* does not hold.

The second way to generate traces is to use the requirements relations. The constraints about traces are derived from the semantics of traces and the semantics of requirements relations (see [13]). The constraints ensure that requirements relations are preserved in the architecture by the satisfying elements. Our tool uses the constraints also to generate requirements relations from traces.

The verification result, and therefore the traces, depends on the reformulation of the requirement to be checked. The software architect needs to consider potential false positive and missed traces. Such traces are defined in relation to the set of actual traces, which is the *golden standard* for a pair of requirements and architecture.

3.3 Validation of Traces

Validation aims at identifying the traces which do not obey the trace semantics. Our tool uses the semantics of requirements relations together with the trace semantics to validate traces which are already generated or assigned by the architect. Checking is performed according to the constraints derived from the semantics of traces and requirements relations. The following is an example constraint to be checked in the trace model by the tool:

$E_{A1} \supseteq E_{A2}$ if (R_1 Contains R_2) $\wedge$ (E_{A1} Satisfies R_1) $\wedge$ (E_{A2} Satisfies R_2)

where E_{A1} and E_{A2} are sets of architectural elements, and R_1 and R_2 are requirements.

The tool identifies traces or requirements relations which violate the constraints. Validation using requirements relations can be used in two ways. First, the software architect may decide that an invalid trace is a true positive and then he reconsiders the requirements relations. Second, the software architect decides that requirements relations are all valid, then, he identifies the invalid traces.

Our tool also provides validation of traces by using verification results. The architect assigns some *AllocatedTo* traces while creating the architecture. In order to ensure that the architecture satisfies the requirements, the verification of architecture is processed. For the requirements satisfied by the architecture, the *Satisfies* traces are generated by the tool. The assigned *AllocatedTo* traces and the generated *Satisfies* traces are compared by the tool. These traces are validated based on the comparison.

The tool finds the differences and intersections of the sets of the traces. The software architect should check especially the difference of the sets and decide about the validity of traces.

For the requirements which are not satisfied by the architecture, the *AllocatedTo* traces are generated from the counter example by the tool. The assigned and generated *AllocatedTo* traces are validated based on the comparison of trace sets.

The requirement may describe a complex system property amenable to decomposition. The tool can not trace to the part of the requirement responsible for a failure. The requirements engineer should decompose the requirement into sub-parts (*Contains* relation) until each requirement describes only one property.

4. ARCHITECTURE OF THE TOOL

The tool contains five components (rounded boxes in Figure 4).

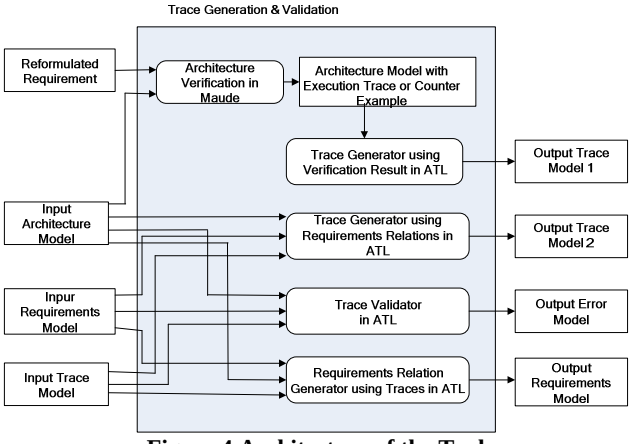


Figure 4 Architecture of the Tool

Architecture Verification in Maude: The input for architecture verification component is the input architecture model and the requirement(s) reformulated as LTL. This component is used in the trace generation. The verification and simulation are performed by the model checker and the rule execution engine of Maude. The architectural model originally expressed in AADL is transformed to Maude terms. The AADL metamodel is encoded as a set of sorts. The dynamic semantics of AADL is given in rewriting rules. Requirements are reformulated as LTL formulas, the language supported by Maude checker.

Trace Generator using Verification Result in ATL: The input of the component is the execution trace and counter example. The

component is implemented as an ATL transformation. If the verification result is an execution trace, the *Satisfies* traces are generated between the checked requirement(s) and the architectural elements in the execution trace. If the verification result is a counter example, the *AllocatedTo* traces are generated between the checked requirement(s) and the architectural elements marked in the counter example. The output is given in the output trace model 1.

Trace Generator using Requirements Relations in ATL: The input of the component is the input architecture model, input trace model, and input requirements model. The component is used in the trace generation. It is implemented as an ATL model transformation. The component generates new traces based on requirements relations in the input requirements model and the constraints in Figure 1. The output is given in the output trace model 2.

For output of the two trace generator components, we use two different output trace models in order to state that the outputs do not have to be the same. In the generation part of Figure 1 which is *generating traces by using requirements relations and verification of architecture*, the three components above are used. First, the traces are generated in the output trace model 1 by the component *trace generator by using verification result*. Then the output trace model 1 is used as an input trace model by the component *trace generator by using requirements relations* to generate traces based on requirements relations in the input requirements model.

Trace Validator in ATL: The input of the component is the input architecture model, input trace model, and input requirements model. The component is used in the trace validation part of all scenarios. It is implemented as an ATL transformation. The component checks the validity of assigned traces between R&A by using verification output or requirements relations. It can also check the validity of requirements relations by using traces between R&A. The output is the output error model which contains invalid traces and invalid requirements relations.

Requirements Relation Generator using Traces in ATL: The input of the component is the input architecture model, input trace model, and input requirements model. The component is used in the trace generation part of Figure 1. It is implemented as an ATL model transformation. The output is given in the output requirements model which contains only the generated requirements relations.

5. EVALUATION OF THE TOOL

Our tool can be evaluated from different perspectives like *usability*, *performance* and *scalability*. The usability of the tool mainly relies on the Eclipse environment. For the output of the counterexample and execution traces, no GUI is provided. For the prototype we consider this to be acceptable. In this section, we conduct performance and scalability tests of the tool for generating and validating traces. Our tool uses model checking techniques in verification of architecture for functional requirements. Since these techniques may have problems in handling large amounts of model elements and states, the performance and scalability of our tool mainly depends on the scalability of the model checking algorithms in Maude. Therefore, we focus on the model checking part of our tool in our performance and scalability tests.

Performance testing is conducted to evaluate the compliance of a system or component with specified performance requirements [31]. The requirement in our test is that the tool performs in reasonable time (say less than one minute) with average number of architecture elements. We base our estimate for the average number of architectural elements on a report by MacCormack et al. [22]. They characterize the differences in design structure between complex software products like Mozilla and Linux. The report shows that the architectural model of a real system contains around 2000 model elements. We take this finding as a base for our performance tests.

Scalability testing is a performance testing focused on ensuring the application under test gracefully handles increases in workload [31]. The workload in our performance test is the number of states. Our interpretation of scalability for evaluating the tool is the following: *the tool scales if the time spent by the tool increases linearly when the number of generated states increases linearly*.

Our dependent variable in the performance and scalability tests is the time for simulation and verification (in seconds). The independent variable used in the performance tests is number of elements in the architecture. We define the number of elements as follows: *number of component instances + number of feature instances + number of port connections where component, feature and port connections are the architectural elements in AADL*. The independent variable used in the scalability test is the number of states generated in the simulation. We define the number of states in the simulation as follows: *the number of states the simulation is enforced to explore*. These two variables are closely related to each other. If the number of elements is increased, it is likely that the number of states required for simulation and verification also increases. However, this does not always have to be the case. For example, if we introduce a new system property to the architecture, not related to the existing system properties, we do not have to increase the number of states in the simulation and verification of architecture for existing system properties.

Memory consumption is not measured in the performance tests. The runs for each performance test are executed six times. The average for each run is derived from six executions. The performance tests are done with Intel Core 2 Duo i5 running at 2.67 GHz, and 2.99GB of memory, running Kubuntu 9.10. We use Core Maude 2.4 for Linux. The models used in the performance tests are artificially created to test the tool with certain number of elements and states.

Performance test. The test is set up as follows. We increase the number of elements by adding components, data ports and connections to the architecture. We start with 2000 architectural elements and end up with 3000 architectural elements. The number of states for each run is 500, 1000 and 2000. The results of the performance test are shown in Table 1. Since the results of the performance test might be different when the verification result is an execution trace or a counter example, the performance test is done for both cases (see Table 1(a) and Table 1(b)). The standard deviation of the data is approximately 0.3%.

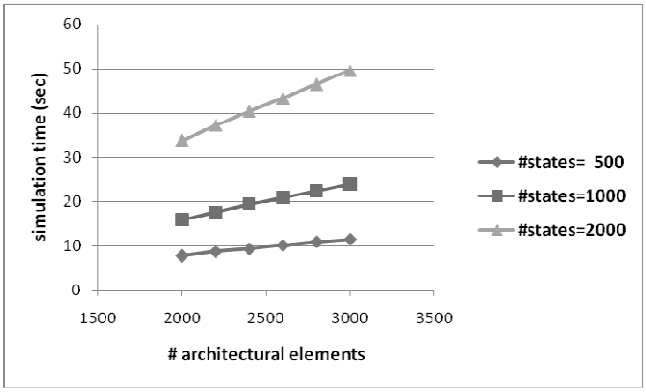
Table 1 Simulation Times in the Performance Test

	Simulation Time (sec) for the Execution Trace		
# elements	**# states = 500**	**# states = 1000**	**# states = 2000**
2000	7.8	15.9	33.8
2200	8.7	17.5	37.2
2400	9.3	19.4	40.4
2600	10.1	20.9	43.3
2800	10.9	22.4	46.5
3000	11.5	23.9	49.6

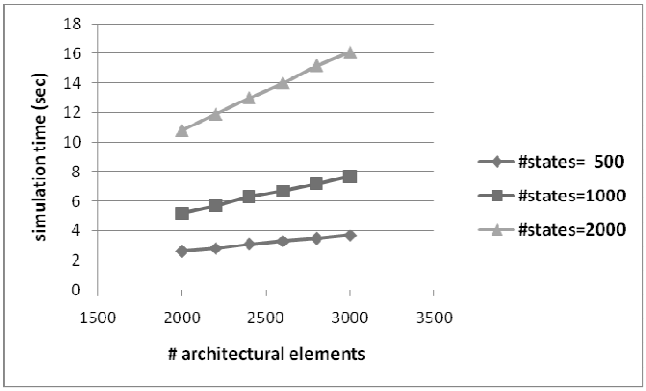
(a) Simulation with Execution Trace

	Simulation Time (sec) for the Counter Example		
# elements	**# states = 500**	**# states = 1000**	**# states = 2000**
2000	2.6	5.2	10.8
2200	2.8	5.7	11.9
2400	3.1	6.3	13.0
2600	3.3	6.7	14.0
2800	3.5	7.2	15.2
3000	3.7	7.7	16.1

(b) Simulation with Counter Example



(a) Simulation with Execution Trace



(b) Simulation with Counter Example

Figure 5 Simulation Time as function of the Number of Architectural Elements

According to these performance tests, the tool performs below one minute with average number of architecture elements in a real system. The increase in the simulation time is linear and up to 50 seconds for 2000 states (see Figure 5).

Scalability test. The goal of this test is to investigate how the tool handles increases in the number of states over several orders of magnitude. Our independent variable is the number of states. We also compare the scalability test results of the tool using Maude with the results of the tool using different simulation and verification environments such as Alloy [17]. Therefore, same execution semantics of AADL in Maude are encoded in Alloy. The first part of the performance test is done in Maude with 10000 architectural elements (see Table 2(a)). Then, the second part of the performance test is done in Alloy (see Table 2(b)). In [20], we investigated simulation and verification in Alloy. Based on our experience, we already know that Alloy is not suitable for large amounts of model elements and states. Therefore, we choose to run the second part of the performance test in Alloy with small number of architecture elements (38 elements) (see Table 2(b)).

Table 2 Simulation Times in the Scalability Test

Number of States	Simulation Time (sec)
10	1.5
100	9.5
1000	82.1
3000	265.4
4500	401.8
5000	-

(a) Simulation in Maude (# elements = 10000)

Number of States	Simulation Time (sec)
20	14.2
40	53.7
60	105.8
80	180.4
100	300.9

(b) Simulation in Alloy (# elements = 38)

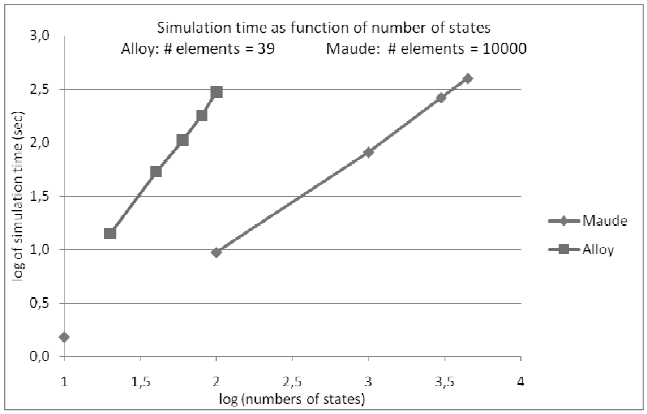


Figure 6 Simulation Time vs. Number of States in Alloy and Maude

According to the scalability test results of our tool using Maude, the simulation time increases linearly when the number of states increases linearly (see Figure 6). We ran out of memory in Maude when we try simulation for 10000 architectural elements with

5000 states. For Alloy, the simulation time also increases linearly when the number of states increases, however, for much smaller number of architectural elements and much smaller number of states.

According to these test results, we conclude that our tool scales much better over a broad range of states for more realistic architectures sizes with Maude than with Alloy.

We cover a subset of AADL semantics in our tool. Our results are valid for this subset. The performance test results may change with more AADL semantics encoded as state transitions in Maude. There is another tool [23] [27] for the representation of AADL models in Maude which covers more execution semantics of AADL. However, we needed our own state transitions for trace generation and validation purposes. As a future work, we plan to integrate the tool in [23] [27] with our tool.

6. RELATED WORK

A number of approaches with tool support describe generating and validating traces. Egyed et al. [9] [10] provides an automated traceability approach that uses a small number of traces as input. Similarly to this work, we use reformulation of requirements as scenarios. In [9] [10], the source code is executed according to the scenarios and then traces are generated between requirements and source code. Footprint graph is used to detect the incomplete and incorrect input to the approach. Dependencies between requirements can be detected based on overlaps among the lines of code implementing the requirements. However, there is only one type of traces and requirements relations in [9] [10]. In our tool, we have multiple trace and requirements relation types.

The System Modeling Language (SysML) [26] is a domain specific modeling language for system engineering. It provides modeling constructs to represent text-based requirements and relate them to other modeling elements such as architectural elements with stereotypes. Types of traces between requirements and architecture are given with informal textual definitions in SysML. However, the SysML standard does not provide any formal definitions of trace types.

Schwarz et al. [30] describe a graph-based traceability approach with tool support. Generation and maintenance of traces are handled by model transformations. The *Satisfies* trace is provided without any formal semantics or textual definition. Components, interfaces and ports in the architecture are created automatically from requirements and use cases by using heuristics. *Satisfies* traces are generated in result. In our approach, architecture is created manually and then the traces are generated and validated.

The tool by Grechanik et al. [15] supports generating traces between types and variables in Java programs and elements of use-case diagrams (UCD). The tool combines program analysis, run-time monitoring, and machine learning to generate traces. Initial traces are needed for trace generation. Relations between program entities are compared with corresponding relations between elements in UCDs only to validate traces.

Mader et al. [21] focus on maintaining traces between requirements and UML models for the changes in UML models. Patterns for maintaining traces are specified based on the classification of UML model changes. The tool in [21] is complementary to our tool.

Bonde et al. [4] describes an interoperability approach based on generating a trace model by using model transformations. The trace model in [4] consists of one type of trace (*Relationship*) and model transformation operation types (*Copy*, *Convert*, *Link*, and *Create*). The work in [4] focuses on traces between platform independent and platform specific models in MDA context.

Egyed [11] introduces the UML/Analyzer tool which does consistency checking based on model transformation. Abi-Antoun and Medvidovic [1] describes a semi-automatic approach to assist in refining a high-level architecture specified in an architecture description language into a design described with UML. The works in [1] and [11] are complementary in the sense that one describes the refinement and the other one is providing how to ensure the preservation of properties in this refinement. Heckel and Thone [16] propose a notion of refinement which requires the preservation of both structural and behavior at the lower level of abstraction. Based on formal refinement relationships between abstract and more concrete architectural models, they use model checking techniques to verify that abstract scenarios can also be realized in the more concrete architecture. Most of the works given above focuses on the generation and validation of traces between architectural and detailed design models.

7. CONCLUSIONS

In this paper, we presented a tool that provides trace establishment by using semantics of traces between R&A (Requirements and Architecture). The tool uses Maude, a formal language based on equational and rewriting logic, and MDE technologies such as Eclipse EMF and ATL.

There are some open issues in the usage of the tool. Reformulation of requirements in terms of solution domain is a part of design process and is hard to automate. The architect might still need to check the generated traces. In case of false positives the requirements model and relations should be checked. This suggests an iterative semiautomatic process of using our tool. In such a process, the software architect can gradually improve the quality of the traces and the requirements. Case studies conducted with the industry [5] shows that LTL/CTL is hard to reformulate and check requirements in the architecture. Domain-specific languages can be used for requirements of certain type that allow compilation of LTL/CTL formulas [5]. Starting from natural language text, Semantics Business Vocabulary and Rules (SBVR) [25] can support reformulation requirements in terms of LTL/CTL formulas. Extending our tool with this kind of languages will ease the reformulation of requirements.

Maintenance of traces is not covered in this paper. We focus on generating and validating traces as a first step. In case of changes in the requirements or in the architecture, some of the traces will be invalid and incomplete. Our tool needs to be extended for maintenance of traces.

We mainly focused on scalability issues in our tool for generating and validating traces. Since model checking techniques may have problems with big size of models and number of states, the scalability of our tool depends on the scalability of the model checking algorithms in Maude. Our tool needs further improvement for usability. The core parts of the tool are implemented. However, integration of these parts is currently done manually and we need a user interface to control all these parts.

8. REFERENCES

[1] Abi-Antoun, M., and Medvidovic, N. 1999. Enabling the Refinement of a Software Architecture into a Design. In: UML'99, LNCS, vol. 1723, pp. 17-31.

[2] Architecture Design and Analysis Language (AADL). http://www.aadl.info. Accessed 05 January 2010.

[3] Baier, C., and Katoen, J.P. 2008. Principles of Model Checking. MIT Press.

[4] Bonde, L., Boulet, P., and Dekeyser, J.L. 2005. Traceability and Interoperability at Different Levels of Abstraction in Model-Driven Engineering. In: FDL 2005, pp. 263-276.

[5] Ciraci, S. 2009. Graph based Verification of Software Evolution Requirements. PhD thesis, Univ. of Twente. CTIT Ph.D.-thesis series No. 09-162 ISBN 978-90-365-2956-3

[6] Clavel, M., Duran, F., Eker, S., Lincoln, P., Marti-Oliet, N., Meseguer, J., and Queseda, J.F. 2002. Maude: Specification and Programming in Rewriting Logic. Theoretical Computer Science, 285, 187-243, Elsevier.

[7] Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., and Talcott, C. 2007. All about Maude - A High-Performance Logical Framework. Lecture Notes in Computer Science, Vol.4350, Springer.

[8] Dahlstedt, A.G., Persson, A. 2005. Requirements Interdependencies: State of the Art and Future Challenges. In: Aurum, A., Wohlin, C. (eds.) Engineering and Managing Software Requirements, pp. 95–116. Springer, Berlin.

[9] Egyed, A., and Grunbacher, P. 2005. Supporting Software Understanding with Automated Requirements Traceability. Int. J. Soft. Eng. Knowl. Eng. 15(5), 783-810.

[10] Egyed, A. 2003. A Scenario-Driven Approach to Trace Dependency Analysis. IEEE Trans. Software Eng. 29(2), 116-132.

[11] Egyed, A. 2000. Automatically Validating Model Consistency during Refinement. Technical report, Center for Software Engineering, University of Southern California.

[12] Goknil, A., Kurtev, I., van den Berg, K., and Veldhuis, J. W. 2009. Semantics of Trace Relations in Requirements Models for Consistency Checking and Inferencing. Theme Issue on Traceability, Software and System Modeling Journal.

[13] Goknil, A., Kurtev, I., and van den Berg, K. 2010. Generation and Validation of Traces between Requirements and Architecture based on Trace Semantics. Technical Report (To appear), University of Twente.

[14] Gotel, O.C.Z., and Finkelstein, C.W. 1994. An Analysis of the Requirements Traceability Problem. In: RE'94, pp. 94-101. IEEE Computer Society Press, Colorado, USA.

[15] Grechanik, M., McKinley, K.S., and Perry, D.E. 2007. Recovering and Using use-case-diagram-to-source-code traceability links. In: ESEC-FSE '07, pp. 95-104.

[16] Heckel, R., and Thone, S. 2005. Behavior-Preserving Refinement Relations between Dynamic Software Architectures. In: WADT'04, LNCS, vol. 3423, pp. 1-27.

[17] Jackson, D. 2002. Alloy: a Lightweight Object Modelling Notation. ACM Trans. Softw. Eng. Methodol. 11(2), 256-290.

[18] Jouault, F., and Kurtev, I. 2006. Transforming Models with ATL. In: Bruel, J.-M. (ed.) MoDELS 2005. LNCS, vol. 3844, pp. 128–138. Springer, Heidelberg.

[19] Kurtev, I., Bézivin, J., and Aksit, M. 2002. Technical Spaces: An Initial Appraisal. In: CoopIS, DOA'2002 Federated Conferences, Industrial track, Irvine.

[20] Looman, S.A.M. 2009. Impact Analysis of Changes in Functional Requirements in the Behavioral View of Software Architectures. M.Sc. Thesis, University of Twente.

[21] Mader, P., Gotel. O., and Philippow, I. 2009. Enabling Automated Traceability Maintenance through the Upkeep of Traceability Relations. In ECMDA-FA '09, LNCS, vol. 5562, pp. 174-189.

[22] McCormack, A., Rusnak, J., and Baldwin, C.Y. 2006. Exploring the Structure of Complex Software Designs: An Empirical Study of Open Source and Proprietary Code. Management Science, vol. 52, no. 7, pp. 1015-1030.

[23] Moment2-AADL. http://www.cs.le.ac.uk/people/aboronat/tools/moment2-aadl/

[24] Murphy, G.C., Notkin, D., and Sullivan, K. 1995. Software Reflexion Models: Bridging the Gap between Source and High-Level Models. In: SIGSOFT'95, pp. 18-28.

[25] OMG: Semantics of Business Vocabulary and Rules (SBVR). OMG Standard, v. 1.0.

[26] OMG: SysML Specification. OMG ptc/06-05-04, http://www.sysml.org/specs.htm. Accessed 05 January 2010

[27] Ölveczky, P.C., Boronat, A., Mesequer, J., and Pek, E. 2010. Formal Semantics and Analysis of Behavioral AADL Models in Real-Time Maude. Technical Report at UIUC (to appear).

[28] Post, H., Sinz, C., Merz, F., Gorges, T., and Kropf, T. 2009. Linking Functional Requirements and Software Verification. In: RE'09. IEEE Computer Society Press, 295-302, USA.

[29] Rivera, E.J., Guerra, E., de Lara, J., and Vallecillo, A. 2009. Analyzing Rule-Based Behavioral Semantics of Visual Modeling Languages with Maude. In: SLE 2008, LNCS 5452, pp. 54-73, France.

[30] Schwarz, H., Ebert, J., and Winter, A. 2009. Graph-based Traceability: a Comprehensive Approach. Theme Issue on Traceability, Software and System Modeling Journal.

[31] http://www.aptest.com/glossary.html#S