The Journal of Systems and Software xxx (2013) xxx–xxx

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Generation and validation of traces between requirements and architecture based on formal trace semantics

Arda Goknil[a,*], Ivan Kurtev[b], Klaas Van Den Berg[c]

[a] AOSTE Research Team, INRIA, Sophia Antipolis, France
[b] Nspyre, Dillenburgstraat 25-3, 5652 AM Eindhoven, The Netherlands
[c] Software Engineering Group, University of Twente, 7500 AE Enschede, The Netherlands

ARTICLE INFO

Article history:
Received 2 August 2012
Received in revised form 2 October 2013
Accepted 2 October 2013
Available online xxx

Keywords:
Trace generation and validation
Requirements metamodel
Trace metamodel
Software architecture

ABSTRACT

The size and complexity of software systems make integration of the new/modified requirements to the software system costly and time consuming. The impact of requirements changes on other requirements, design elements and source code should be traced to determine parts of the software to be changed. Considerable research has been devoted to relating requirements and design artifacts with source code. Less attention has been paid to relating requirements (R) with architecture (A) by using well-defined semantics of traces. Traces between R&A might be manually assigned. This is time-consuming, and error prone. Traces might be incomplete and invalid. In this paper, we present an approach for automatic trace generation and validation of traces between requirements (R) and architecture (A). Requirements relations and architecture verification techniques are used. A trace metamodel is defined with commonly used trace types between R&A. We use the semantics of traces and requirements relations for generating and validating traces with a tool support. The tool provides the following: (1) generation and validation of traces by using requirements relations and/or verification of architecture, (2) generation and validation of requirements relations by using traces. The tool is based on model transformation in ATL and term-rewriting logic in Maude.

1. Introduction

Today's software systems are complex artifacts often deployed in a highly dynamic context. The requirements of software systems change continuously and new requirements emerge frequently. In order to reduce the cost for implementing the required system changes, it is important to apply change management as early as possible in the software development cycle. *Requirements traceability* is considered crucial in change management for establishing and maintaining consistency between software development artifacts. When changes in the requirements are proposed, the traceability information is used to calculate the impact of these changes on other software artifacts such as requirements, architecture, source code and test cases.

Traceability is the ability to follow the usage of an artifact throughout the software development process. For example, a requirement may be traced *forward* to the architectural design, detailed design, and source code components that implement it and *backward* to its stakeholders and business context. Traceability has been recognized as an important quality property (Ramesh and Jarke, 2001) which, however, is expensive and difficult to achieve in practice. It relies on the availability of a high volume of trace relations (simply called *traces* or *trace links* in this paper) between artifacts. Most approaches focus on achieving traceability between requirements and source code or between design and source code (Antoniol et al., 2002; Egyed, 2003; Grechanik et al., 2007; Hayes et al., 2006). Less attention has been paid to relating requirements with software architecture.

Establishing traces between requirements and architecture for change management may be beneficial due to several reasons. Determining the changes in source code can be difficult because code contains many details and there is an abstraction gap between the requirements, the rationale for change and the impacted code. In contrast, the architecture is at a higher level of abstraction and is generally of a smaller size than the code. Software architects base their architectural design decisions on the essential system requirements thus making direct relations between the requirements and the architecture. In addition to that, in many cases a software architecture expressed in a precise architectural description language (ADL) can be used for automatic code generation. This supports automatic traceability from architecture to code.

The design of software architectures is a highly creative and complex engineering task often performed manually. In the

* Corresponding author. Tel.: +33 761447052.
E-mail addresses: arda.goknil@inria.fr, ardagoknil@gmail.com (A. Goknil), ivan.kurtev@nspyre.nl (I. Kurtev), vdberg.nl@gmail.com (K. Van Den Berg).

0164-1212/$ – see front matter © 2013 Elsevier Inc. All rights reserved.
http://dx.doi.org/10.1016/j.jss.2013.10.006

current practices the trace information emerging during architectural design is either completely lost or only partially collected. Manual trace assignment is time-consuming and may lead to invalid and incomplete traces. The process of establishing and validating traces should be automated as much as possible in order to reduce the overall change management costs.

In this paper, we present an approach for generation and validation of traces between R&A (requirements and architecture). Our goal is to improve the currently observed practices by providing a degree of automation that allows faster trace generation and improves the precision of traces by validating them.

The traces, the requirements and the architecture descriptions are uniformly represented as models. We built on our previous work (Goknil et al., 2011) on modeling software requirements. Requirements specifications are captured in models that contain requirements and typed relations among them. The semantics of these relations is formalized in First-order logic (FOL).

Architecture Description Languages (ADLs) gained significant attention in the research community and a number of languages were proposed. Their current industrial adoption is reported to be low with some exceptions, for example, in the embedded systems domain. Languages Koala (van Ommering et al., 2000), EAST-ADL (Cuenot et al., 2011) and Architecture Analysis and Design Language (AADL) (SAE, 2010) are used for specifying and analyzing architectures of embedded systems. In particular, EAST-ADL and AADL are industry driven standards proposed by companies in the automotive and avionics domain.

In this paper we use AADL because of the availability of formal dynamic semantics of the language. The semantics allows verification and simulation of architectures which is a main enabling factor in our approach. We use dynamic semantics for part of AADL (Ölveczky et al., 2010a,b) expressed in rewriting logic supported by the Maude language and tools (Clavel et al., 2002, 2007).

We propose two mechanisms to *generate traces* between R&A. The first mechanism uses architecture verification techniques. A given requirement is formulated as a property in terms of the architecture. The architecture is executed and a state space is produced. If the property is satisfied, the elements in the execution trace after the verification are collectively responsible for the satisfaction of the given requirement. Trace links are established between the requirement and the architectural elements. Here, the term 'execution trace' refers to the sequence of states in the successful execution of the architecture. It differs from the terms 'trace link' and 'trace relation' that are used to denote a traceability connection between two model elements (e.g. a requirement and an architectural element). The generation process is fully automatic once the property to be verified is provided by the architect. We limit ourselves to verification of functional requirements only. Non-functional requirements such as performance, real-time properties, and others often require formalisms and verification tools different than the ones used in our work.

The second mechanism uses the requirements relations together with existing trace links. We ensure that the relations between requirements are properly 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 (Murphy et al., 1995). Requirements relations are reflected in the relations among the traced architectural elements. Based on an initial set of traces new traces are automatically inferred by using the requirements relations.

The *validation of traces* also uses architecture verification and relation preservation. If the verification of a requirement fails then the assigned traces (if any) may be invalid. Similarly, invalid traces may be present if there is a difference between the manually assigned and the generated traces.

The approach is supported by a tool based on Eclipse Modeling Framework, the ATL model transformation language (Jouault et al., 2008; Jouault and Kurtev, 2006), and the Maude tool set (Clavel et al., 2002, 2007). The tool provides: (1) generation and validation of *traces* by using requirements relations and/or verification of architecture, (2) generation and validation of *requirements relations* by using traces. One part of the approach is manual (uses manually assigned traces) and the other part is semi-automatic because, while it still generates traces automatically – these then need to be resolved by human users if contradictions exist between the manually assigned and automatically generated traces.

This paper is structured as follows. Section 2 describes the approach. Section 3 briefly introduces the elements of requirements models. This is needed for understanding the trace generation and validation process. Section 4 presents the trace metamodel and definitions of the trace types. In Section 5, we provide the formalization of the trace types. Section 6 introduces an illustrative example used in the following sections. Section 7 describes the details of generating and validating traces. Section 8 explains the tool support. In Section 9 we give a brief discussion for the overall approach. Section 10 presents a survey of the related work. Section 11 concludes the paper.

2. Overview of the approach

As we already mentioned the manual process of establishing traces is time consuming and error prone. The idea behind our approach is to start with a set of initial traces and to gradually validate them and infer new traces. The validation and inference can be applied iteratively until a sufficient amount of traceability information is achieved. The set of initial traces can be either automatically derived by executing the architecture or manually assigned by the architect.

While there are many research approaches that structure and formalize requirements, the practices of many software companies are still based on textual requirements specifications. Furthermore, the architects often design the software architecture based on the requirements without documenting the major design decisions and design alternatives. In this way, valuable trace information is lost or remains incomplete and unreliable. This hampers the maintenance and evolution of the software system.

Our approach aims at improving this situation observed in the current practices by providing a degree of automation that will make trace generation faster and will improve the precision of the traces. The precision of traces can be improved in two ways. First, by using architectural verification, we may discover traces that otherwise would be *missed* in case of manual assignment and informal reasoning. Second, by using trace validation, we may reveal traces that are *false positive* traces.

Fig. 1 gives the requirements and architectural models with traces in the approach. We assume that part of the structure of a requirements document is made explicit and represented in a requirements model. In this respect we rely on our previous work (Goknil et al., 2011) on requirements modeling. It gives the details on the process of extracting models from requirements documents and the tool TRIC that supports it. The requirements models consist of requirements and typed relations among them.

Traces are established (automatically or manually) between requirements models and architectural models. We assume that the architecture is expressed in a modeling language and we do not consider the process (if any) that may guide the design of the architecture and the transition from the requirements to the architecture. In the worst case there may be initially no trace information between R&A.

Please cite this article in press as: Goknil, A., et al., Generation and validation of traces between requirements and architecture based on formal trace semantics. J. Syst. Software (2013), http://dx.doi.org/10.1016/j.jss.2013.10.006

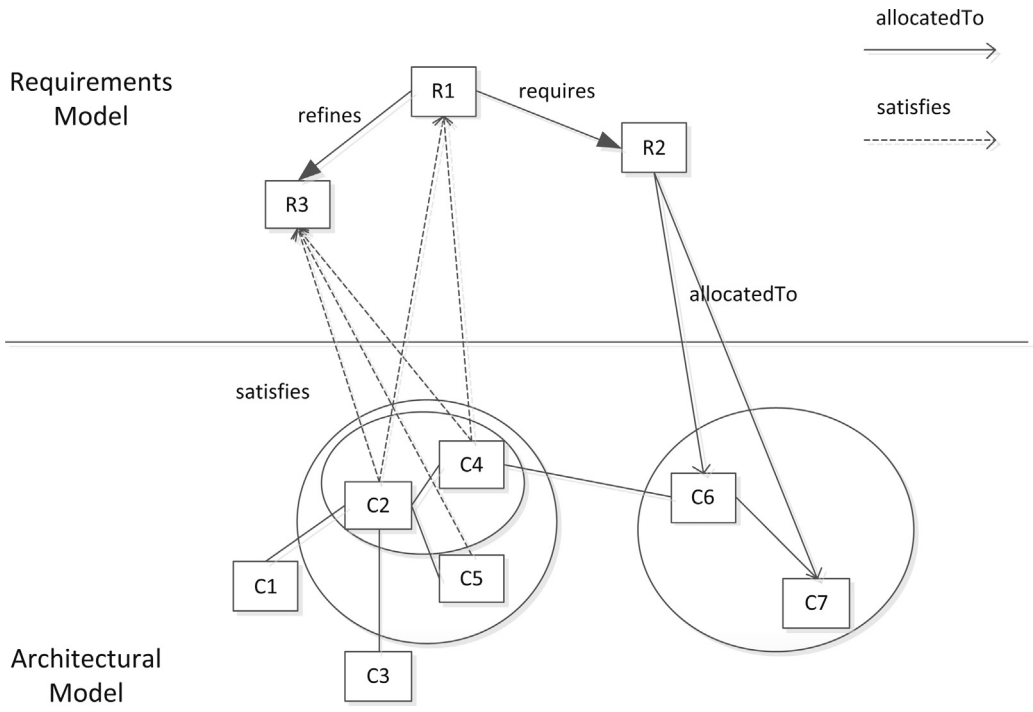


Fig. 1. Requirements and architectural models with traces.

We support two types of traces between R&A: *satisfies* and *allocatedTo*. The main difference between them is how they are established: *satisfies* traces are established automatically based on architecture verification and reasoning over existing traces. *allocatedTo* traces are usually assigned manually by the software architect. They may be derived from already existing traces collected during the architectural design phase.

A *satisfies* trace relates a set of architectural elements to a single requirement (In Fig. 1 we show individual links as arrows and the set of the traced elements are surrounded by an oval). The set of traced architectural elements is the minimal set that is responsible for fulfilling the requirement. The *satisfies* traces are established after verifying the requirement over the architectural model. The verification is done by reformulating the requirement as a formula in temporal logic and checking this formula in the Maude model checker. This is possible because the used architectural models are expressed in a subset of AADL that has executable semantics defined as Maude rewrite rules. If the formula evaluates to true then the elements from the execution trace of the model checker are considered to *satisfy* the requirement.

It has to be noted that some approaches for requirements traceability consider the relations within a single model as traces. For example, the requirements relations *refines* and *requires* are considered as in-model traces. In this paper we deal only with traces between R&A.

In many real life projects the number of requirements is large and it is not feasible to verify every requirement in the described manner. The software architect may choose to verify only the most critical requirements and to manually assign the traces for the rest. Manually assigned traces are of type *allocatedTo*. They express the expectation of the software architect that a certain set of architectural elements is responsible for fulfilling a given requirement. Since the *allocatedTo* traces are manually assigned they may be invalid and/or incomplete.

Our approach supports validation of traces by using relations among requirements. Fig. 1 shows three requirements and two relations among them: *refines* and *requires*. The meaning of the supported relations among requirements is explained in detail in Section 3. The relations among requirements have to be reflected by the relations among the traced architectural elements. For example, if requirement R_1 refines R_3 (see Fig. 1) the architectural elements that satisfy R_1 must be a subset of the architectural elements that satisfy R_3. Indeed, Fig. 1 shows that this is the case. If this constraint is not fulfilled then the established traces are considered as invalid. In a similar way if one requirement *requires* another requirement then the sets of the traced architectural elements must have a non-empty intersection. The complete definitions of the constraints imposed by the relations among requirements on the traced architectural elements are given in Section 7.2.

These constraints can also be used to infer new traces on the basis of a set of initial traces and the given relations among requirements.

In case of detection of invalid traces the software architect needs to revisit the manually assigned traces and eventually to verify them. Another reason for invalid traces is that the requirements model is not correct and some of the assigned requirements relations do not hold. In this way our approach also supports corrections in the requirements at a later stage when the architecture and the traces may reveal invalid or non-detected relations among requirements.

These basic mechanisms of automatic *generation* of *satisfies* traces, *trace validation*, and *inference* based on requirements relations can be combined in various scenarios that the software architect can follow. Fig. 2 gives the overview of the approach with inputs and outputs in the scenarios.

Scenario 1. *Generating traces by using verification of architecture.*

This scenario takes the reformulated requirement(s) and the architecture as input. The reformulated requirements are logical formulas over the architecture. The formulas are checked by the Maude model checker. If the result is positive, *satisfies* traces are generated. If a counter example is found, all the architectural elements used in the counter example are considered to be related to the requirement with the *allocatedTo* traces. The software architect should inspect the input models for errors.

Please cite this article in press as: Goknil, A., et al., Generation and validation of traces between requirements and architecture based on formal trace semantics. J. Syst. Software (2013), http://dx.doi.org/10.1016/j.jss.2013.10.006

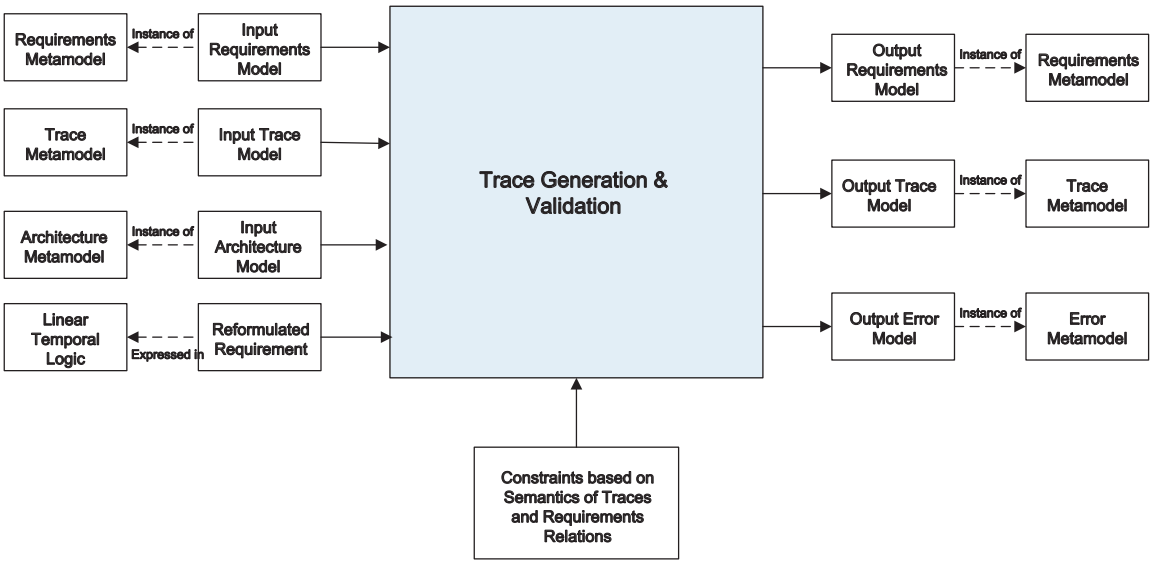


Fig. 2. Overview of the approach.

Scenario 2. *Validating traces by using verification of architecture.*

In this scenario, a manually assigned set of *allocatedTo* traces is checked. The reformulated requirement(s) is verified in the way described in Scenario 1. The validation phase compares the assigned traces with the architectural elements in the verification output. The invalid assigned traces are reported in the output error model.

Scenario 3. *Generating/validating traces by using requirements relations.*

This scenario takes the requirements model, an initial trace model and constraints in Fig. 2 as input. The constraints specify how the relations among requirements are reflected in the architecture. New traces are inferred for requirements without traces that are related to requirements with assigned traces. The requirements relations are used to infer new traces and to validate the existing traces. The output trace model contains the generated traces. Invalid traces are reported in an output error model.

Scenario 4. *Generating/Validating requirements relations by using traces between R&A.* This scenario is concerned only with analysis of the requirements model based on traceability information. The relations among architectural elements may reveal new requirements relations, or the lack of such relations between the traced

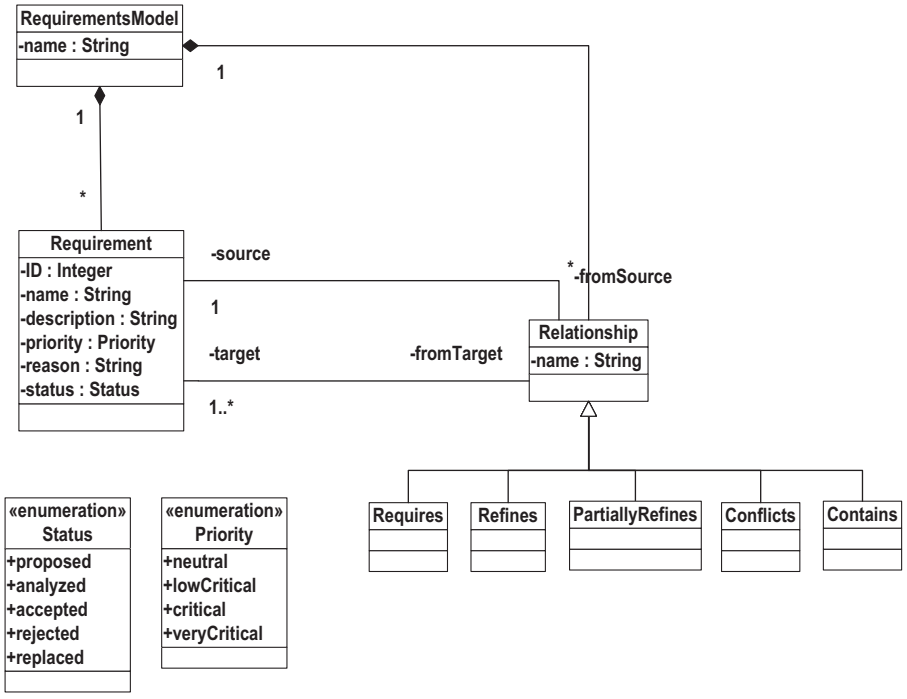


Fig. 3. Requirements metamodel.

Please cite this article in press as: Goknil, A., et al., Generation and validation of traces between requirements and architecture based on formal trace semantics. J. Syst. Software (2013), http://dx.doi.org/10.1016/j.jss.2013.10.006

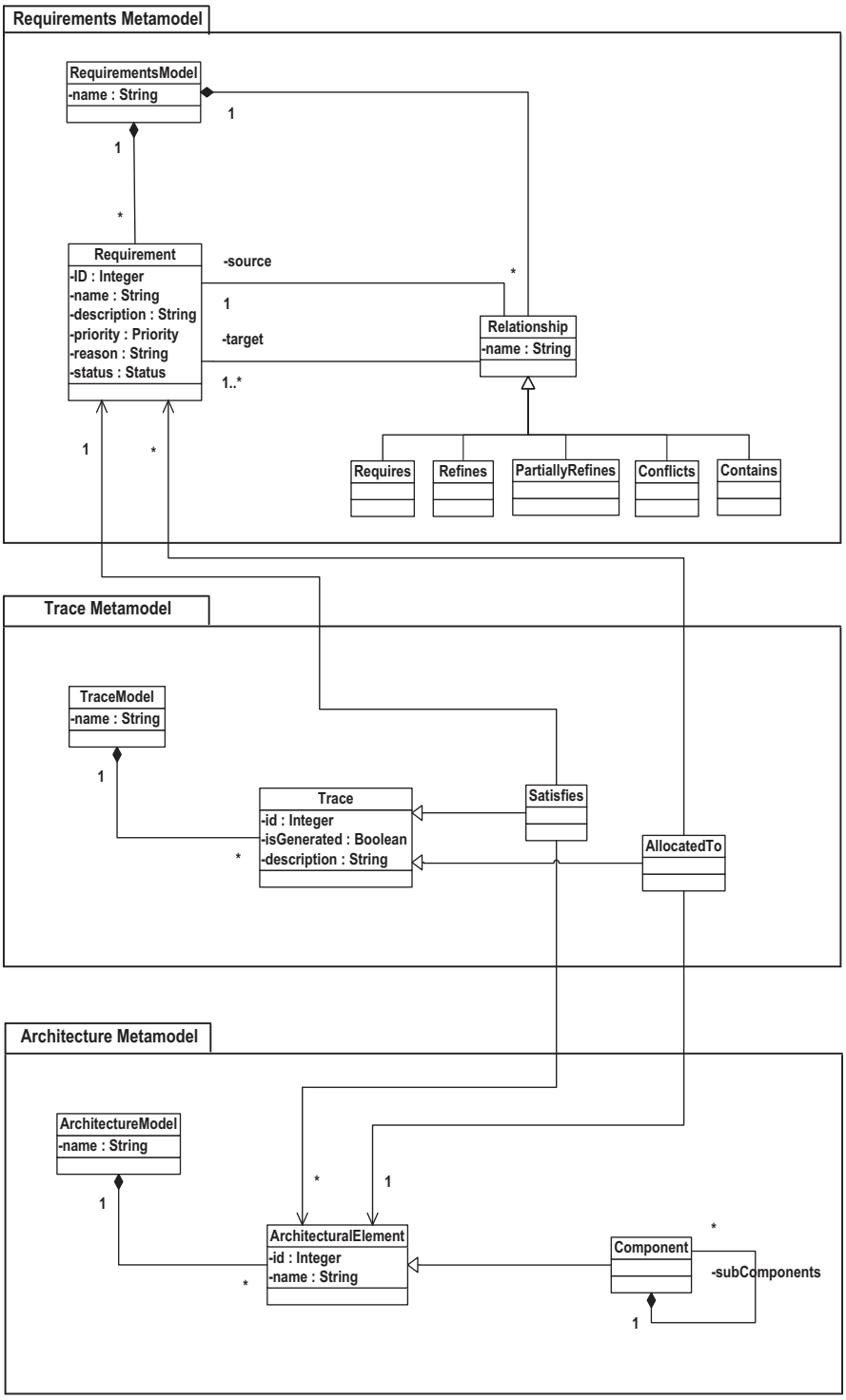


Fig. 4. Trace metamodel for requirements and architecture.

requirements. The output requirements model contains the generated requirements relations. The output error model contains the invalid requirements relations in the input requirements model.

In general, the identified invalid traces and new requirements relations are just suggestions for the architect. They have to be checked by the architect for the final decision.

3. Requirements metamodel

The requirements models in our approach conform to a metamodel that contains elements commonly found in the literature on requirements modeling. The metamodel defines requirement class with its attributes and relations between requirements. We left out other elements such as goals, stakeholders, and test cases. Fig. 3 gives the requirements metamodel.

Please cite this article in press as: Goknil, A., et al., Generation and validation of traces between requirements and architecture based on formal trace semantics. J. Syst. Software (2013), http://dx.doi.org/10.1016/j.jss.2013.10.006

The requirements are captured in a *requirements model*. A requirements model contains *requirements* and their *relations*. Based on (SWEBOOK) we define a requirement as follows:

Definition 1. *Requirement*: A requirement is a description of a system property or properties which need to be fulfilled.

A requirement has a unique identifier (ID), name, textual description, priority, rationale, and status. A system property can be a certain functionality or any quality attribute. In this respect, our requirements relation types and their formalization are applicable to both functional and non-functional requirements.

We identified five types of relations: *requires*, *refines*, *partially refines*, *contains*, and *conflicts*. In the literature, these relations are informally defined as follows.

Definition 2. *Requires relation*: A requirement R_1 *requires* a requirement R_2 if R_1 is fulfilled only when R_2 is fulfilled.

Definition 3. *Refines relation*: A requirement R_1 *refines* a requirement R_2 if R_1 is derived from R_2 by adding more details to its properties.

The refined requirement (R_2) can be seen as an abstraction of the refining requirement (R_1).

Definition 4. *Contains relation*: A requirement R_1 *contains* requirements $R_2 \ldots R_n$ if $R_2 \ldots R_n$ are parts of the whole R_1 (part-whole hierarchy).

This relationship enables a complex requirement to be decomposed into parts. A composite requirement may state that the system shall do *A* and *B* and *C*, which can be decomposed into the requirements that the system shall do *A*, the system shall do *B*, and the system shall do *C*. For this relation, all parts are required in order to fulfill the composing requirement.

Definition 5. *Partially refines relation:* A requirement R_1 *partially refines* a requirement R_2 if R_1 is derived from R_2 by adding more details to properties of R_2 and excluding the unrefined properties of R_2.

Our assumption here is that R_2 can be decomposed into other requirements and that R_1 refines a subset of these decomposed requirements. This relation can be described as a special combination of *contains* and *refines* relations. It is mainly drawn from the decomposition of goals in goal-oriented requirements engineering (van Lamswerdee, 2001).

Definition 6. *Conflicts relation*: A requirement R_1 *conflicts with* a requirement R_2 if the fulfillment of R_1 excludes the fulfillment of R_2 and vice versa.

The *conflicts* relation addresses a contradiction between requirements. We consider *conflicts* as a binary relation.

The definitions given above are informal. The semantics of the relations is formalized in first-order logic (FOL). This allows inferring new requirements relations and checking the consistency in requirements models. In this paper we will use the informal definitions given above. For the detailed description of the formal semantics of the requirements and the relations, the reader is referred to our previous work (Goknil, 2011; Goknil et al., 2011).

4. Trace metamodel

Traces are collected in models that conform to the trace metamodel which contains two trace types *Satisfies* and *AllocatedTo*.

Fig. 4 shows the trace metamodel together with parts of requirements and architecture metamodels. We use AADL to specify architectural models. A fragment of the AADL metamodel is given in the bottom part of Fig. 4.

The *AllocatedTo* and *Satisfies* relations are defined as follows (OMG, 2010; Ramesh and Jarke, 2001; Wasson, 2006):

Definition 7. *AllocatedTo trace:* A requirement *R* is *allocated to* a set of architectural elements *E* if the system properties related to *E* are supposed to fulfill the system properties given in *R*.

The architect can track which component will take care of what requirement by using *AllocatedTo* traces (Ramesh and Jarke, 2001).

Definition 8. *Satisfies trace:* A set of architectural elements *E satisfies* a requirement *R* if the system properties related to *E* fulfill the system properties given in *R*.

A *Satisfies* trace is actually an implication dependency between the system properties given in the requirement and the system properties designed in the architecture. The architecture *satisfies* the requirement if the fulfillment of architecture system properties implies the fulfillment of the system properties given in the requirement.

An *AllocatedTo* trace is assigned when the fulfillment of the requirement is expected. A *Satisfies* trace is assigned or generated when the fulfillment of the requirement is assured.

The literature proposes several types of traces, which are similar to *Satisfies* and *AllocatedTo* but named differently. For example, Khan et al. (2008) propose six types of traces. They differ only in the type of the source requirement. In our approach we abstract the metamodel from this detail thus keeping only the very generic types *Satisfies* and *AllocatedTo*.

5. Formalization of trace types

A software architecture model *AM* is a model conforming to the AADL metamodel. For the formalization of trace types it is sufficient to consider architectural models as sets of model elements. The types of architectural elements in the used subset of AADL are *System*, *Process*, *Thread Group*, *Thread*, *SubProgram*, *Data Store*, *Port*, *Data Access* and *Connector*.

Trace types are subsets of Cartesian products of sets. We define *Satisfies* and *AllocatedTo* trace types as follows:

$$Satisfies \subseteq \wp(SAE) \times SR \text{ and } AllocatedTo \subseteq SR \times \wp(SAE) \quad (1)$$

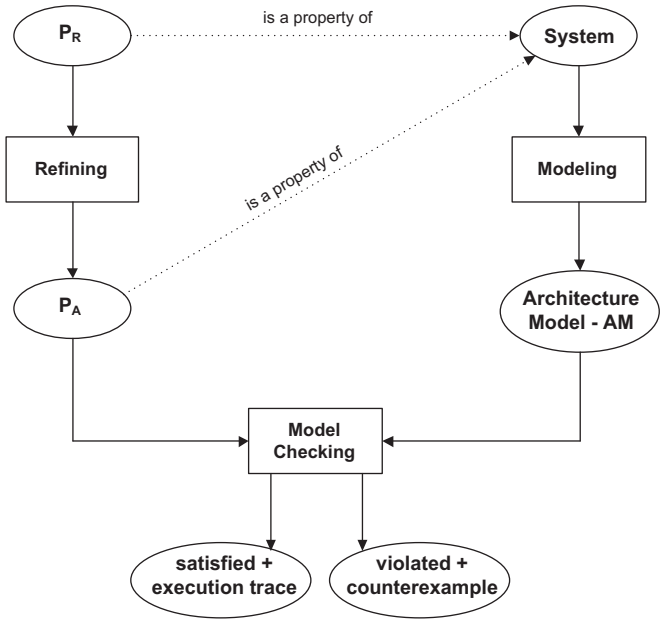


Fig. 5. Schematic view of the relation between P_R and P_A.

Please cite this article in press as: Goknil, A., et al., Generation and validation of traces between requirements and architecture based on formal trace semantics. J. Syst. Software (2013), http://dx.doi.org/10.1016/j.jss.2013.10.006

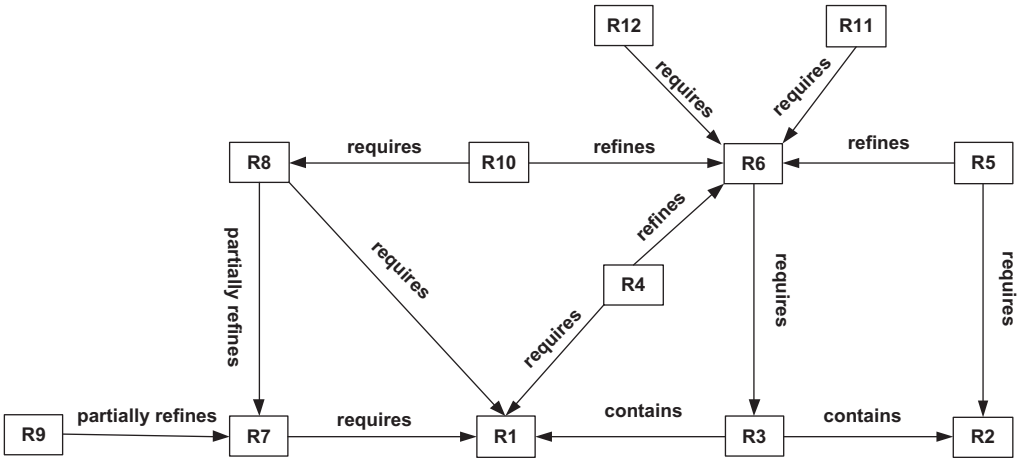


Fig. 6. Part of requirements model for RPM system.

where *SR* is the set of requirements in the requirements model *RM* and *SAE* is the set of architectural elements in the software architecture model *AM*. $\wp(SAE)$ denotes the power set of *SAE*.

The intuition behind the *AllocatedTo* trace type is that a part of software architecture is planned/expected to be an implementation of a set of requirements. For the *Satisfies* traces we have the following additional constraint.

Let R be a requirement and E_A be a set of architectural elements where P_R is a formula that represents R and P_A is a formula whose truth value depends on E_A. We require the following for the *Satisfies* trace type:

> E_A *Satisfies* R iff the following statement holds:
>
> The fulfillment of P_A implies the fulfillment of P_R (2)

For a given property P_A that the architecture has to satisfy, we are interested in identifying the smallest set of architectural elements E_A that are responsible for fulfilling P_A. To identify E_A, P_A is expressed in any suitable logic such as *Linear Temporal Logic* (*LTL*) or *Computation-Tree Logic* (*CTL*). The property P_A thus can be checked over the architecture model *AM* by using architecture verification techniques.

Fig. 5 gives the schematic view of the relation between P_R and P_A. The definition of the *Satisfies* trace type formalizes the intuition that a part of software architecture is an implementation of a set of requirements. P_R is a property that the final developed system must satisfy. A software architecture is a model of such a system that reveals some solution design decisions. Therefore, P_R should be preserved in the architecture. It is reformulated in terms of the architecture, thus obtaining the formula P_A that can be checked over the architecture. P_A is also a property of the final system but expressed in a different level of abstraction compared to P_R. The architectural elements in E_A are those that are in the execution trace[1] of checking P_A. The refinement of P_R to P_A and the modeling of the architecture are performed manually.

The software architecture model usually implements all the architecturally significant requirements. Not all requirements have equal significance with regards to the architecture. According to Malan and Bredemeyer (2002), architecturally significant requirements are those that (1) capture essential functionality of the system, (2) exercise many architectural elements, (3) challenge the architecture, (4) highlight identified issues/risks, (5) exemplify stringent demands on the architecture (e.g. performance requirements), (6) are likely to change, and (7) involve communication and synchronization with external systems. Every architecturally significant requirement should be satisfied and every architectural element should contribute to at least one requirement. We define the *Satisfies* relation between the requirements model *RM* and the architecture model *AM*:

> The Architecture Model *AM satisfies* the Requirements Model *RM* iff the following two statements hold where *R* is a requirement, *SAR* is the set of architecturally significant requirements in the requirements model *RM*, *AE* is an architectural element and *SAE* is the set of architectural elements in the architecture model *AM*:
>
> $$\forall AE(AE \in SAE \rightarrow \exists E_A \exists R(E_A \in \wp(SAE) \wedge AE \in E_A \wedge Satisfies(E_A, R))) \quad (3)$$
>
> $$\forall R((R \in SAR \wedge \neg refined(R)) \rightarrow \exists E_A(E_A \in \wp(SAE) \wedge Satisfies(E_A, R))) \quad (4)$$

refined(R) is true iff *R* is *refined* by one or more requirements in the requirements model. With (4) we ensure that the most concrete requirements are always satisfied by the software architecture.

6. Example: Remote Patient Monitoring system

The approach outlined so far will be illustrated with the Remote Patient Monitoring (RPM) system example. RPM was developed by a company in the Netherlands and had already been implemented and running when we started studying the system. The RPM system has the following stakeholders: *patients*, *doctors*, and *the system administrator*. The main goal is to enable monitoring the patients' conditions such as blood pressure, heart rate, and temperature. For instance, a patient carries a sensor for measuring the body temperature and the values are transmitted to a central system where they are stored.

In order to apply our approach, we modeled the textual requirements in the RPM requirements document and their relations according to the semantics of the relation types. The requirements model was created in TRIC, our tool for requirements modeling

[1] The concept of execution trace is different from the trace and trace relation. An execution trace is a sequence of states in the verification of an architecture.

and inference (see TRIC). Some of the requirements for RPM can be found in Appendix A. Here, two examples are shown:

Requirement 6 *requires* Requirement 3.

Requirement 3. *The system shall measure blood pressure and temperature from a patient.*

Requirement 6. *The system shall store data measured by sensors in the central storage.*

Fig. 6 shows the relevant part of the requirements model.

The solid arrows indicate the relations given by the requirements engineer. For simplicity, we did not include the relations inferred by TRIC. We constructed the architecture of the system by reverse engineering the source code. Fig. 7 gives the overview of the RPM architecture in AADL visual syntax. The complete explanation of the abbreviations of the components is given in Appendix B.

Fig. 7 shows the most abstract components (*system* and *process* in AADL). These components contain other components which are not represented in Fig. 7. The *SD (Sensor Device)* component contains the sensors carried by the patient. The sensors perform measurements at a regular interval. The *SD* sends the measurements to the *HPC (Host Personal Computer)* component through the *SDC (Sensor Device Coordinator)*. The *SDC* is the ZigBee network coordinator. The details of the coordinating tasks are omitted in the description. The *HPC* consists of the *SDM (Sensor Device Manager)*, *AS (Alarm Service)* and *WS (Web Server)* process subcomponents. The *SDM* stores the measurements and generated alarms in the data stores (*Temp_alarms* and *Temp_Meas* for temperature alarms and measurements). The *WS* serves as a web-interface for the doctors. The *AS* forwards the alarms to the *CPC (Client Personal Computer)* component. The *CPC* is used by the doctors to monitor patients. The *AR (Alarm Receiver)* process in the *CPC* receives the alarms from the *AS* and notifies the doctor about the alarms. The *WC (Web Client)* process uses the *WS* to retrieve the measurements and alarms stored by the *SDM*.

Fig. 7 shows only systems and processes in the RPM architecture. AADL provides also support for *thread* and *subprogram* components. The computation of the system is modeled as subprogram and thread behavior. The current version of the AADL semantics (Ölveczky et al., 2010a; Ölveczky et al., 2010b) in Maude allows modeling subprogram and thread behavior by using AADL's behavioral annex with a finite set of states and a set of state variables. The RPM architecture has behavioral annexes for dynamic behavior of threads in each system component.

The following presents the implementation of the thread in the *SDM* component (*SDM_Thread*) for storing blood pressure measurements. It shows a transition system with state variables where each transition contains a guard (*[sdm_blood_edp2?(inMessage)]* in line 17) on the existence of events/data in the input ports (*sdm_blood_edp2* in line 17), and on the value of the data received (*inMessage* in line 17).

```
1   thread SDM_Thread
2     features
3       sdm_blood_edp2: in event data port Behavior::integer;
4       sdm_blood_strg: out event data port Behavior::integer;
5     properties
6       Dispatch_Protocol => aperiodic;
7   end SDM_Thread;
8
9   thread implementation SDM_Thread.i
10    annex behavior_specification {**
11      states
12        s0: initial complete state;
13        bloodStored: complete state;
14      state variables
15        inMessage: Behavior::integer;
16      transitions
17        s0 -[sdm_blood_edp2?(inMessage)]-> bloodStored { sdm_blood_strg!(inMessage); };
18    **};
19  end SDM_Thread.i;
```

The thread above has event data ports *SDM_BLOOD_EDP2* in line 3 and *SDM_BLOOD_STRG* in line 4 for blood measurements. Since the *Dispatch_Protocol* of the thread is aperiodic (see line 6), this thread is activated upon receiving input. The thread has *s0* as the *initial* state (line 12) and *bloodStored* as the *complete* state (line 13). If the thread is in the *s0* state and receives the measured data at the *SDM_BLOOD_EDP2* event data port, then the received data is stored in the *SDM_BLOOD_STRG* data port and the *bloodStored* state is reached (line 17).

7. Generating and validating traces

After explaining parts of the requirements model and the architecture of RPM we can show an application of our approach for generation and validation of traces. Section 7.1 explains the verification of functional requirements. Section 7.2 gives the details

Please cite this article in press as: Goknil, A., et al., Generation and validation of traces between requirements and architecture based on formal trace semantics. J. Syst. Software (2013), http://dx.doi.org/10.1016/j.jss.2013.10.006

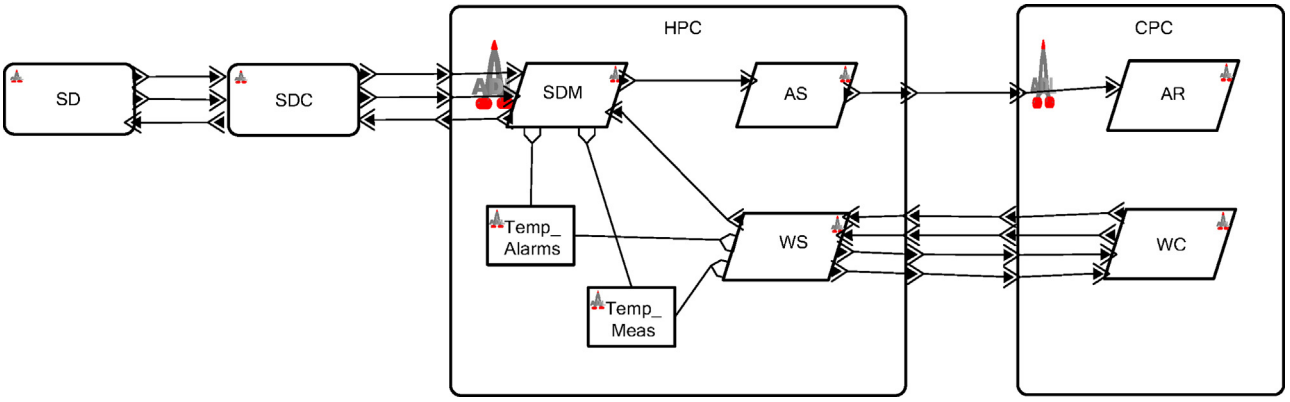


Fig. 7. Overview of the RPM architecture.

about the trace generation by using requirements relations and verification results; Section 7.3 presents the trace validation.

7.1. Verification of functional requirements

The purpose of the verification is to check if requirements are correctly implemented in the architecture. Verification results serve as an input for both trace generation and trace validation activities (Scenarios 1 and 2 in Section 2). Fig. 8 illustrates the verification of functional requirements.

A requirement under verification is reformulated as a formal description of the required behavior of the architecture (*reformulate* and *uses* in Fig. 8). The requirement is first described as a formalized scenario, and then described as a property specification (Boudiaf et al., 2008; Pelliccione et al., 2009; Post et al., 2009). The formalized scenario is a pair of predicates <*pre, post*> encoding the precondition *pre* that is fulfilled before the execution of the scenario and the postcondition *post* that is expected to be fulfilled after the execution of the scenario. The property specification is expressed as an LTL formula that can be evaluated by the Maude model checker.

The availability of a dynamic semantics of AADL makes the architectural models executable. We use the formal semantics of behavioral AADL models in Maude implemented by Ölveczky et al. (2010a,b). The execution is a simulation of the system to be built at the architectural level. The architecture is executed for the formalized scenario and a state space is produced (*simulate* in Fig. 8). In our example a state describes the loci of data values within the architecture. An execution trace is the set of states and state transitions which are generated if the reformulated requirement is satisfied. Counter example is the set of states which are generated where the reformulated requirement is not satisfied. All the used architectural elements in an execution trace are considered collectively responsible for fulfilling a requirement. Therefore, they are the elements for which a *Satisfies* trace will be established.

Example. *Reformulation of Requirements*
Consider the following requirement

Requirement 5 *The system shall store patient blood pressure measured by the sensor in the central storage.*

Fig. 9 is the part of the RPM architecture developed for the system property given in Requirement 5.

Requirement 5 is reformulated as a formalized scenario as follows:

Formalized Scenario: *(contains(SD_BLOOD_EDP1, DI)), (contains(SDM_BLOOD_STRG, DI))*

The scenario states that if the data instance *DI* is contained by the data port *SD_BLOOD_EDP1* of Sensor 2 (*SD* component in Fig. 7), then the data instance *DI* is stored in the data store *SDM_BLOOD_STRG* of the component *SDM* after executing the architecture (see Fig. 7).

The dynamic behavior of a thread is defined in AADL using AADL's behavioral annex with a finite set of states and a set of state variables. In the RPM architecture, the subprogram execution for storing the blood pressure data in the central storage is implemented as a state transition system in the thread *sdmTh* (see Section 6). The sdmTh thread has states *bloodStored*, *temperatureStored*, *highTemperature*, *lowTemperature* and *idle*. When the data instance *DI* is stored in the data store *SDM_BLOOD_STRG* of the component *SDM*, the state of the *sdmTh* thread in the state transition system is set to the *bloodStored* state.

The formalized scenario is the first step to reformulate the requirement in terms of the architecture. The next step is to construct the appropriate LTL formula. The formula is the following:

LTL formula in Maude: *(mc initializeThreads({MAIN system Wholesys. imp}) |=u <> ((MAIN -> hpc -> sdm -> sdmTh) @ bloodStored).)*

The formula states that if the data instance *DI* is contained by the data port *SD_BLOOD_EDP1* of Sensor 2, then eventually in the future the state in the state transition system in the *sdmTh* thread is set to the *bloodStored* state (the data instance *DI* is stored by the data store *SDM_BLOOD_STRG* of the *SDM* component). Please note that the data instance *DI* is created in the initial state by a test thread in the RPM model. Therefore, the LTL formula does not explicitly indicate the data instance *DI* and the data port *SD_BLOOD_EDP1* of Sensor 2. The *initializeThreads({MAIN system Wholesys. imp})* in the formula creates the initial state. *MAIN -> hpc -> sdm -> sdmTh* denotes the full component name of the *sdmTh* thread component. The *@bloodStored* states that the state of the *sdmTh* thread is the *bloodStored*. The '<>' in the LTL formula states '*eventually in the future*'. The LTL formula can be checked on the generated state space in Maude. The formula exemplifies the property P_A in Fig. 5.

7.2. Generating traces

According to the definition of trace types, a *Satisfies* trace is generated between the architectural elements in the execution trace and the requirement that is successfully checked. A counter example means that although the requirement is *allocated to* the architectural elements, i.e., they are involved in the implementation of this requirement, the architecture does not satisfy it. *AllocatedTo* trace can be generated but the *Satisfies* trace is not present.

We modified the transition rules in Maude to be able to record the architectural elements matched and affected by the transition rules. These recorded elements belong to the execution trace and form the set E_A used in the definition of the trace link types. The

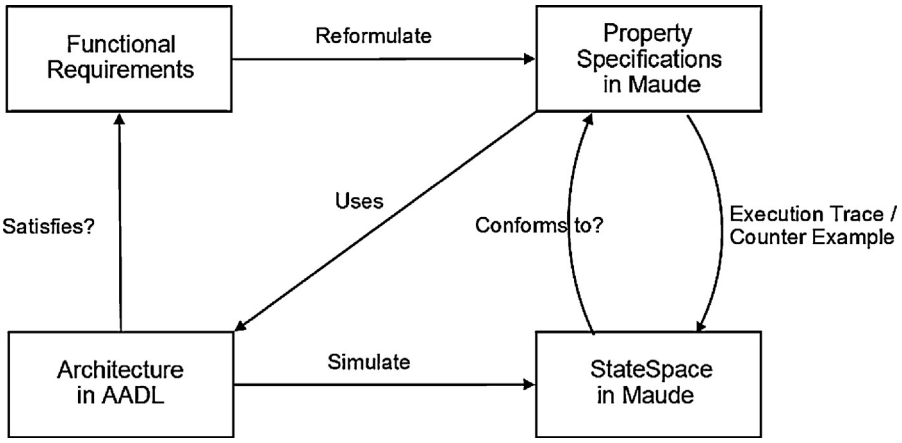


Fig. 8. Verification of functional requirements.

modification adds an attribute called *Used* to the component classes in the AADL metamodel. Each transition rule sets the attribute *Used* of the matched/affected architectural element to *true*.

The *search* command of the Maude model checker returns an execution trace if the requirement is satisfied. There might be multiple execution traces in which the requirement is satisfied. In this case, the *Satisfies* trace is generated for the used architectural elements in each execution trace.

The second way to generate traces is to use the requirements relations (see Scenario 3 in Section 2). We defined several constraints that specify how a relation between two requirements is reflected in the architecture. The constraints are shown in Fig. 10. They are also used to generate requirements relations from traces (see Scenario 4 in Section 2). In Fig. 10, all the constraints are given for the *Satisfies* traces. The same constraints are also valid for the assigned *AllocatedTo* traces.

Fig. 10(a) postulates that the intersection of the traced architectural elements for two requirements is non-empty if one requirement requires another. Fig. 10(b) illustrates that the architectural elements that satisfy a refining requirement also belong to the set of architectural elements that satisfies the refined requirement. Constraints similar to the one in Fig. 10(b) are valid for traces with the *Contains* and *Partially Refines* relations (see Fig. 10(c) and (d)).

In order to generate the *Satisfies* traces for R_1 in Fig. 10(b), (c) and (d), all other requirements ($R_2, R_3, \ldots, R_k$) should be satisfied by the architecture. For instance, if one of the refining requirements (R_2, R_3, …, R_k) in Fig. 10(d) is not satisfied by the architecture, the refined requirement (R_1) is not satisfied either. The partial refinement might not be complete (Fig. 10(d)). In this case, even if all refining requirements are satisfied, the *Satisfies* trace is generated only if it is confirmed that the unrefined properties are also satisfied. The *Satisfies* trace for the unrefined properties in R_1 is established by verification.

Example. *Generation of Traces by Using Verification of Architecture*

In Section 7.1, we gave an example reformulation of a requirement as property specification in Maude. We show how to generate traces for Requirement 5 (see Scenario 1). The LTL formula is:

LTL formula in Maude: *(mc initializeThreads({MAIN system Wholesys. imp}) |=u<>((MAIN -> hpc -> sdm -> sdmTh) @ bloodStored).)*

After running the model checker, the formula is true which means that Requirement 5 is satisfied by the architecture. The *search* command returns an execution trace for Requirement 5:

Search Command in Maude:

```
(utsearch [1]
  initializeThreads({MAIN system Wholesys. imp}) =>* {C:Configuration}
  such that
    ((location of component (MAIN ->hpc -> sdm -> sdmTh) in
C:Configuration) = = bloodStored.)
```

In the *utsearch* command above, the *initializeThreads({MAIN system Wholesys. imp})* creates the initial state where the data instance *DI* is contained by the data port *SD_BLOOD_EDP1* of Sensor 2. The *(location of component (MAIN -> hpc -> sdm -> sdmTh) in C:Configuration)* returns the state in the transition system in the *sdmTh* thread, which should be the *bloodStored* state ('= =*bloodStored*'). '=>*' in the command indicates the form of the rewriting proof from the initial state until the state where the state in the transition system in the *sdmTh* thread is the *bloodStored* state. Then, the *utsearch* command tries to reach that state from the initial state.

The field *used* of the architectural elements matched by the transition rules is set to *true* in the last state of the counter example. Our tool generates the *Satisfies* trace links between Requirement 5 and the architectural elements used in the execution trace. Fig. 11 shows the generated *Satisfies* trace links for Requirement 5.

The verification result, and therefore the traces, depends on the reformulation of the requirement to be checked. Misinterpretation

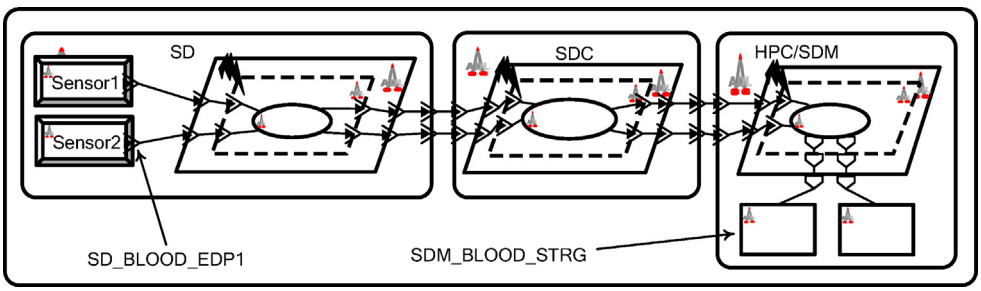


Fig. 9. Part of the RPM architecture.

Please cite this article in press as: Goknil, A., et al., Generation and validation of traces between requirements and architecture based on formal trace semantics. J. Syst. Software (2013), http://dx.doi.org/10.1016/j.jss.2013.10.006

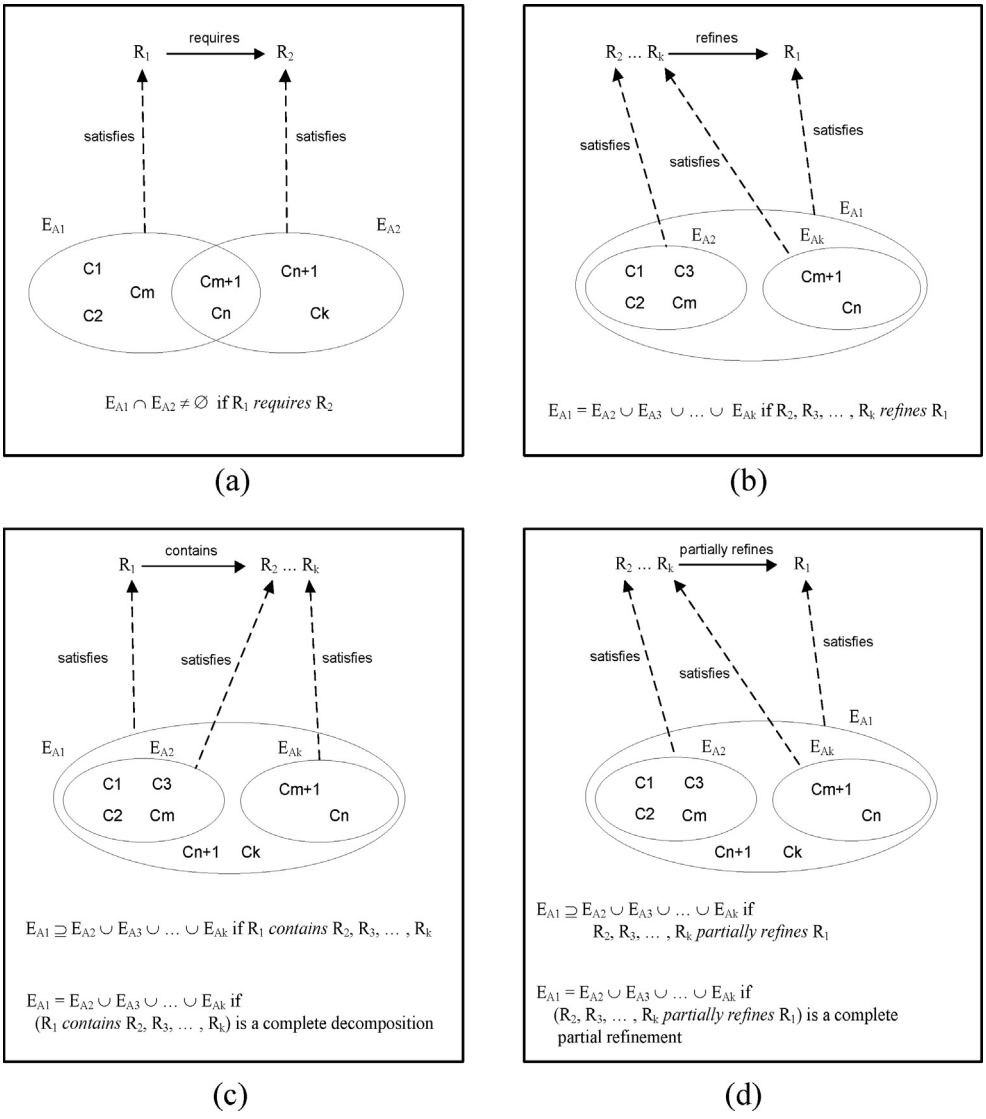


Fig. 10. Constraints based on semantics of traces and requirements relations.

of the requirement and wrong reformulation are causes for potential false positive and missed traces. In case of correct models and correct reformulation, the generated traces are the actual traces.

Traces can also be generated by using verification of architecture and requirements relations (see Scenarios 1 and 3). Consider the following example for the RPM system.

Example. *Generation of Traces by Using Verification of Architecture and Requirements Relations*

We first generate trace links for Requirement 4.

Requirement 4 *The system shall store patient temperature measured by the sensor in the central storage.*

In the model in Fig. 6, Requirement 4 and Requirement 5 refine Requirement 6.

Requirement 6 *The system shall store data measured by sensors in the central storage.*

Based on the constraints in Fig. 10, the set of the generated *Satisfies* traces for Requirement 6 is the union of the trace sets for Requirement 4 and Requirement 5. Fig. 12 shows the generated *Satisfies* traces for Requirement 6 by using the requirements relations.

The following example illustrates the generation of requirements relations by using traces (Scenario 4).

Example. *Generation of Requirements Relations by Using Traces*

Consider the following requirements.

Requirement 4 *The system shall store patient temperature measured by the sensor in the central storage.*

Requirement 12 *The system shall enable the doctor to retrieve all stored temperature measurements for a patient.*

In the previous example, we already stated that Requirement 4 is satisfied by the architecture. It can be shown that the architecture also satisfies Requirement 12. The *Satisfies* trace links are generated for Requirement 12 accordingly.

Fig. 13 shows the non-empty intersection of traces for Requirement 4 and Requirement 12. Based on the constraints in Fig. 10, there might be a *Requires* relation between Requirement 4 and Requirement 12. The presence of such relation should be decided by the requirements engineer. In our example we can conclude that Requirement 12 *requires* Requirement 4 since it is not possible to obtain any data (Requirement 12) if they have not been stored before that (Requirement 4).

Please cite this article in press as: Goknil, A., et al., Generation and validation of traces between requirements and architecture based on formal trace semantics. J. Syst. Software (2013), http://dx.doi.org/10.1016/j.jss.2013.10.006

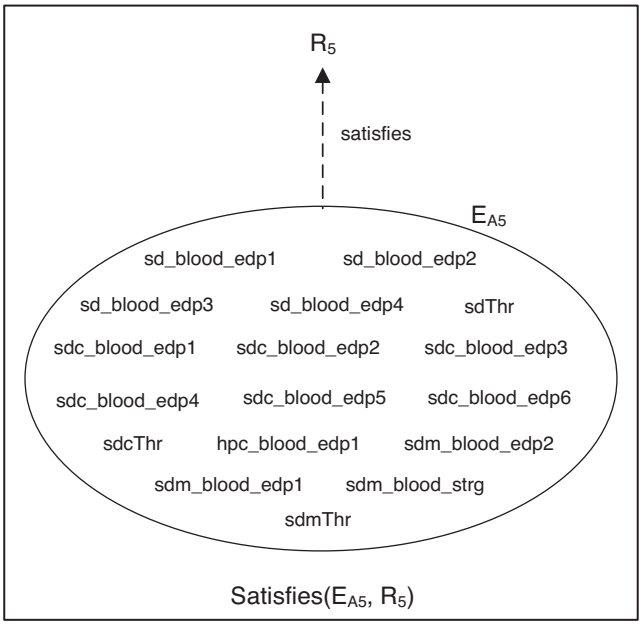


Fig. 11. Generated 'Satisfies' trace for requirement 5 by using verification results.

It should be noted that a non-empty intersection of traces does not always lead to *requires* relation. The fulfillment of two requirements may rely on a common functionality of the system. This commonly used functionality will be then demonstrated by common traced elements in the architecture.

7.3. Validating Traces

Validation of traces aims at identifying the traces which do not obey the trace semantics and the related constraints. This helps to eliminate false positive traces and to detect missing traces. In addition, a violation of the constraints in Fig. 10 may indicate invalid relations among requirements. These relations need to be reconsidered by the requirements engineer and the software architect.

Validation based on requirements relations can be used in two ways (see Scenarios 3 and 4). First, the architect may conclude that an invalid trace is a true positive and then he reconsiders the requirements relations (Scenario 4). Second, the architect may conclude that requirements relations are all valid, and then he/she identifies the invalid traces (Scenario 3).

Our approach also provides validation of traces by using verification results (Scenario 2). This applies to the manually assigned *AllocatedTo* traces. The assigned *AllocatedTo* traces and the generated *Satisfies* traces for a requirement are validated based on the comparison of traces in Fig. 14.

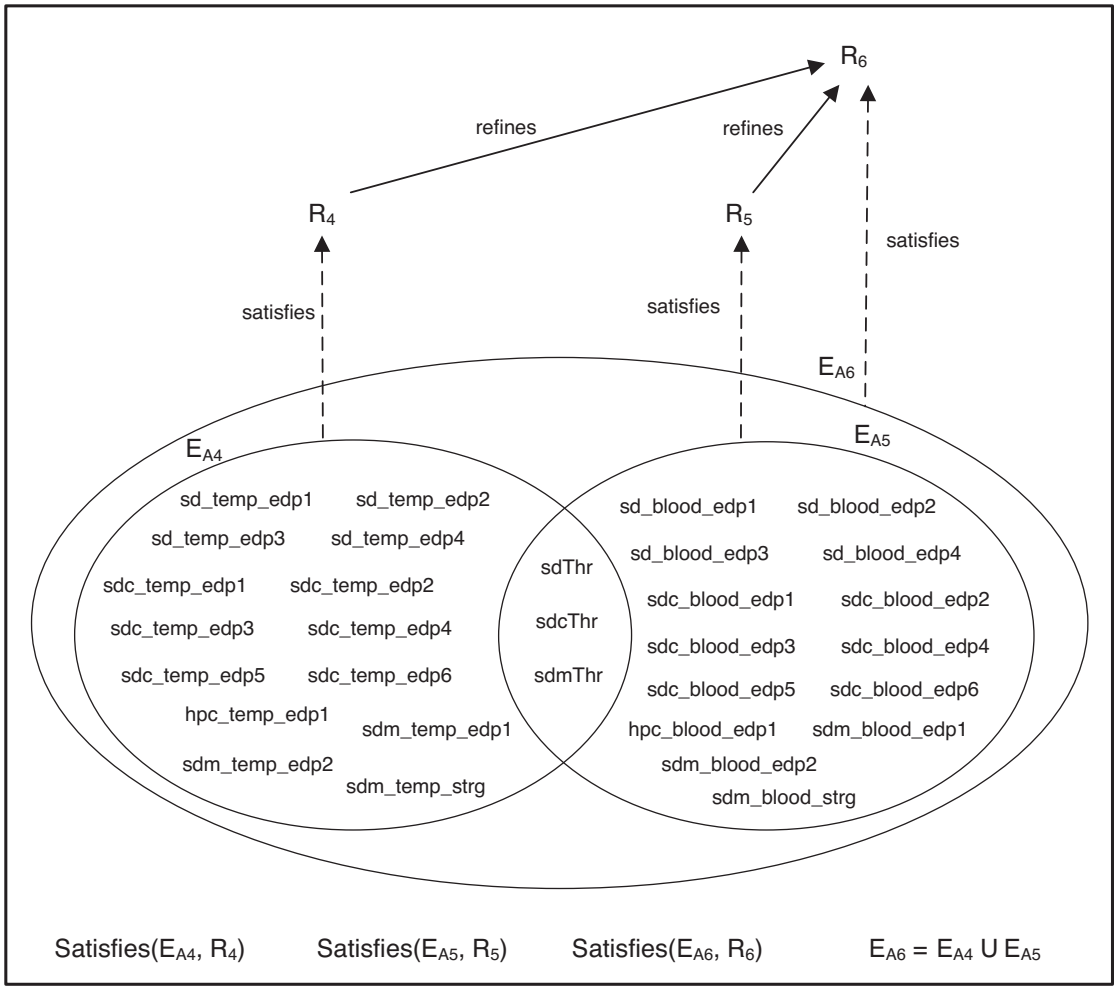


Fig. 12. Generated 'Satisfies' traces for requirement 6 by using requirements relations.

Please cite this article in press as: Goknil, A., et al., Generation and validation of traces between requirements and architecture based on formal trace semantics. J. Syst. Software (2013), http://dx.doi.org/10.1016/j.jss.2013.10.006

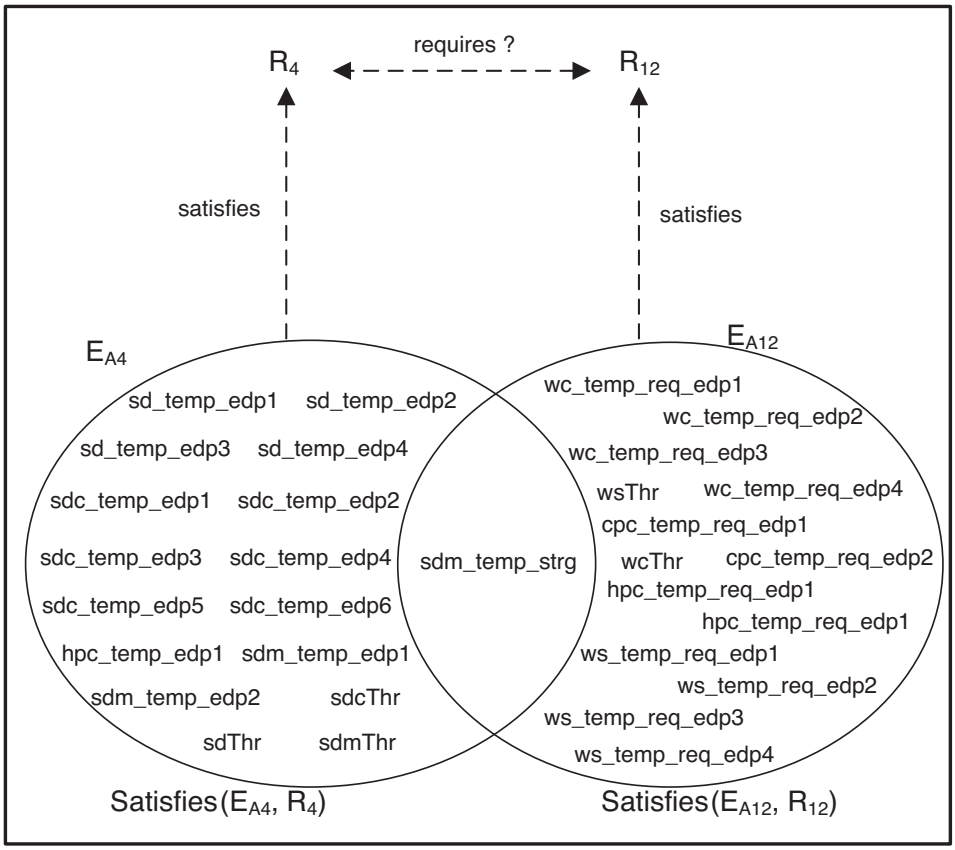


Fig. 13. Generated 'Requires' relation for requirement 4 and requirement 12.

- If (GST\AAT) is non-empty, then either some of the generated *Satisfies* traces (GST\AAT) are false positives or some of the traces are missed while assigning the *AllocatedTo* traces. False positive *Satisfies* traces in (GST\AAT) may be caused by misinterpretation of the requirement and/or wrong reformulation.
- If (AAT\GST) is non-empty, then either some of the assigned *AllocatedTo* traces (AAT\GST) are false positives or some of the *Satisfies* traces are missed while generating the *Satisfies* traces. Again, a misinterpretation of the requirement and wrong reformulation may cause missing *Satisfies* traces.

For the requirements which are not satisfied by the architecture, the *AllocatedTo* traces are generated from the counter example. The assigned and generated *AllocatedTo* traces for a requirement are validated based on the comparison of traces in Fig. 15 which is similar to the comparison table in Fig. 14.

The software architect should check the difference of the sets (*GAT\AAT* and *AAT\GAT*) and conclude about the validity of traces.

The following is an example of validation of traces by using verification of architecture (see Scenario 2).

Example. *Validation of Traces by Using Verification of Architecture*

The example in Section 7.2 shows the generated Satisfies traces for Requirement 5. Fig. 16 shows the generated *Satisfies* and the initially assigned *AllocatedTo* traces for Requirement 5.

The traces in Fig. 16 are validated according to the Venn diagram in Fig. 14. Although Requirement 5 is allocated to the components *AS* and *CPC_AR*, these components are not involved in the

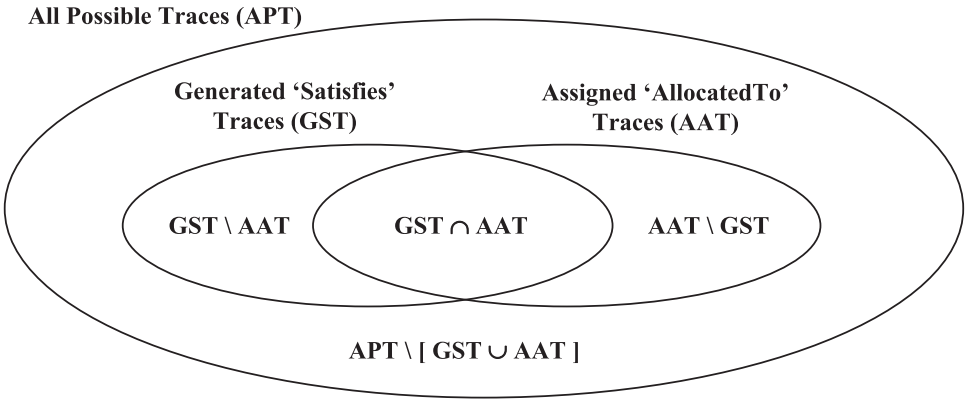


Fig. 14. Venn diagram for generated 'Satisfies' and assigned 'AllocatedTo' traces for a requirement.

Please cite this article in press as: Goknil, A., et al., Generation and validation of traces between requirements and architecture based on formal trace semantics. J. Syst. Software (2013), http://dx.doi.org/10.1016/j.jss.2013.10.006

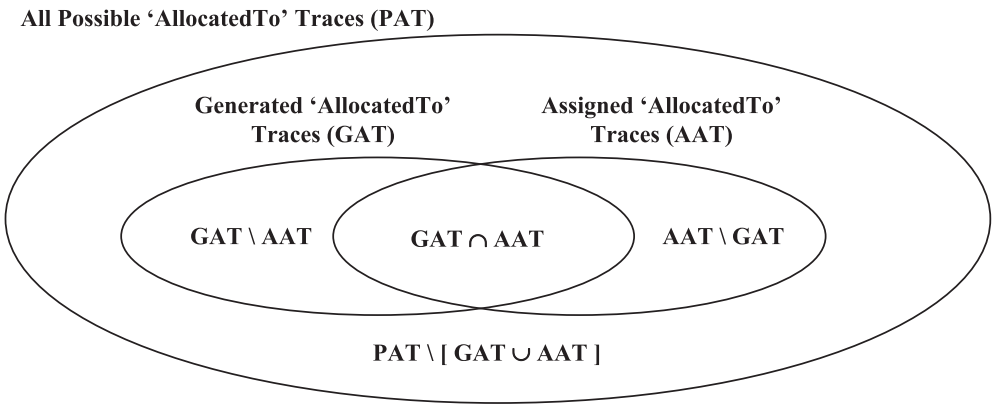


Fig. 15. Venn diagram for generated and assigned 'AllocatedTo' traces for a requirement.

Satisfies traces. We concluded that the two initially assigned *AllocatedTo* traces to the components *AS* and *CPC_AR* are false positives.

The following is the example where the requirements relation is identified invalid based on the constraints in Fig. 10 (see Scenario 4).

Example. *Validation of Requirements Relations by Using Traces*

Fig. 17 shows the assigned *AllocatedTo* traces for Requirement 10 and Requirement 6. In the requirements model, Requirement 10 *refines* Requirement 6 (see Fig. 6).

Requirement 6 *The system shall store data measured by sensors in the central storage.*

Requirement 10 *The system shall store all generated temperature alarms in a central database.*

Based on the constraints and by analyzing requirements, we concluded that the *Refines* relation between Requirement 10 and Requirement 6 is invalid. Indeed, storing a temperature alarm is different than storing a data about the patient condition. Alarms are not considered as measured data.

Removal of the *Refines* relation automatically removes some inferred *requires* relations in the requirements model (see Goknil et al., 2011 for inferring requirements relations). Here, we do not illustrate the update of the requirements model in Fig. 6.

For the validation of traces and requirements relations for cases like we have in Figs. 16 and 17, our tool gives the traces and requirements relations which do not obey the constraints. The architect should decide that either the traces or the requirements relations are invalid.

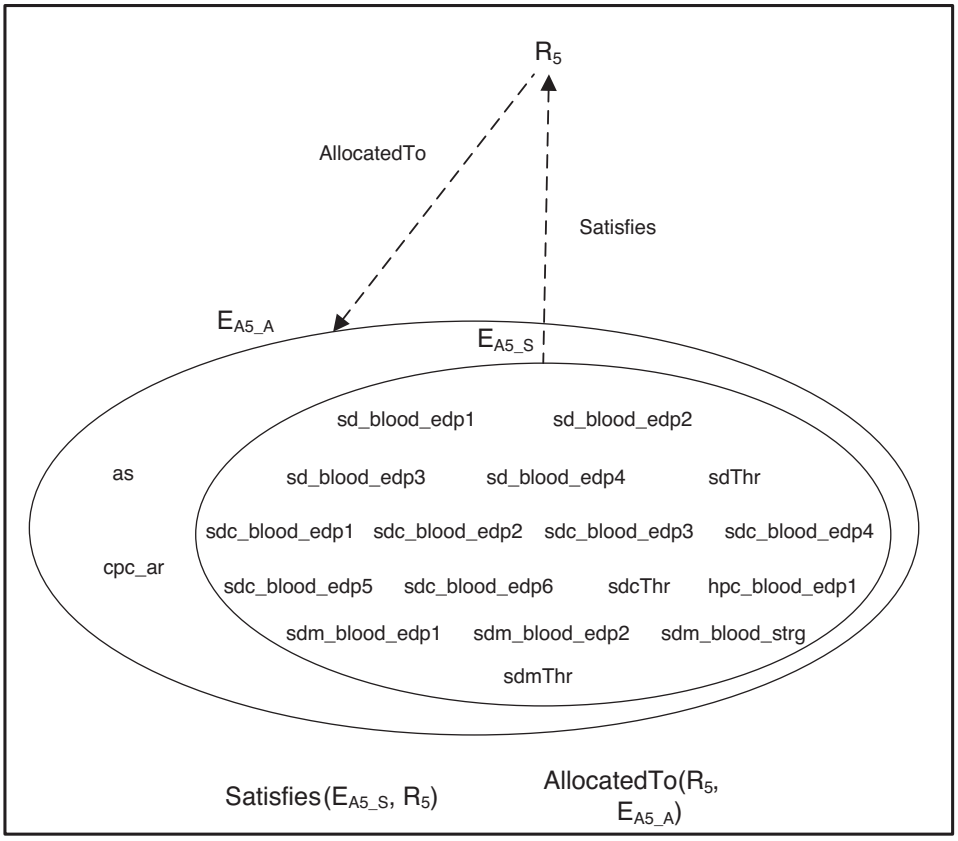


Fig. 16. Generated 'Satisfies' and assigned 'AllocatedTo' traces for requirement 5.

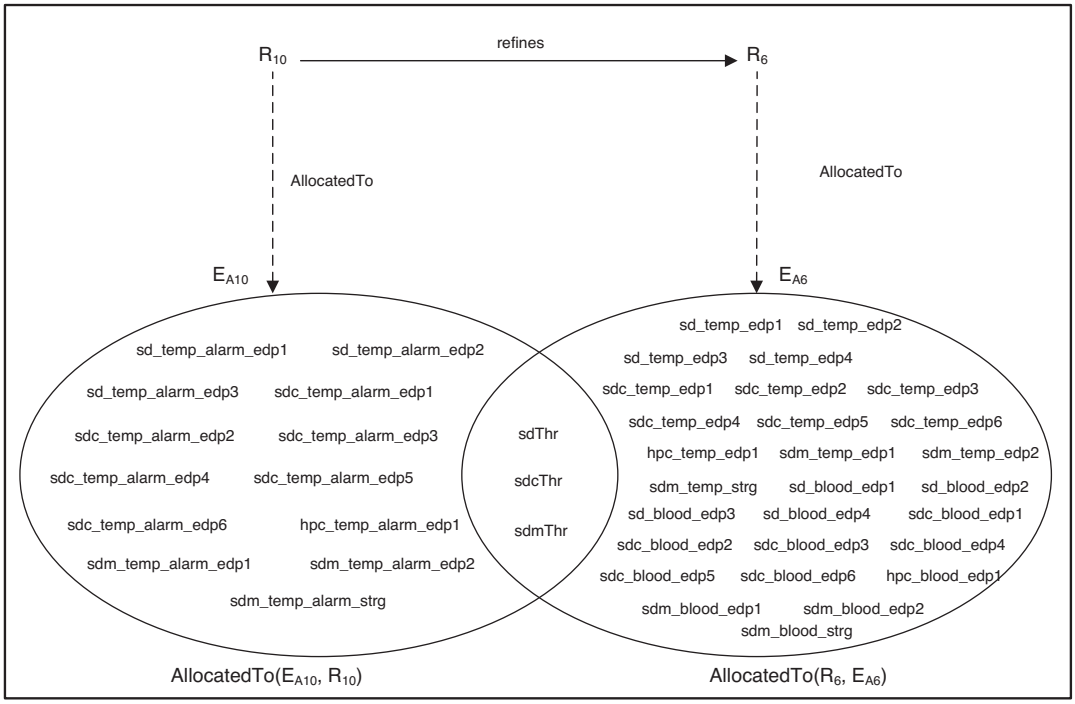


Fig. 17. Assigned 'AllocatedTo' traces with an invalid requirements relation.

8. Tool support

We built a tool that supports our approach. The tool was demonstrated in ECMFA-Traceability Workshop (ECMFA-TW 2010) and is described in (Goknil et al., 2010). This section is mainly based on (Goknil et al., 2010). In Section 8.1, we depict the usage of the tool in the context of a process for modeling requirements and architecture. Section 8.2 gives the architecture of the tool and its main features. Section 8.3 evaluates the tool.

8.1. The modeling process

Fig. 18 gives a UML activity diagram of the process.

The process in Fig. 18 consists of the following activities:

8.1.1. Modeling requirements and designing architecture

This activity takes the requirements document and produces the input requirements model, input architecture model and input trace model. The software architect assigns some initial traces between requirements and architecture. It is a manual activity.

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

8.1.2. Reformulating requirements

The software architect manually reformulates requirements in terms of logical formulas.

8.1.3. Verifying architecture

The activity checks whether the requirements are satisfied by the architecture. It is done automatically in Maude.

8.1.4. Generating trace

It is an automated activity. If only requirements relations and initial traces are used, the activity is performed after the activity *modeling requirements & designing architecture*.

8.1.5. Validating trace

This activity takes the input trace model, requirements model, architecture model and produces an output error model. The activity is automatic. However, the interpretation of the errors in the trace model should be done manually by the software architect.

The process in Fig. 18 is iterative. The requirements engineer and/or the software architect may return to the modeling requirements and designing architecture activity in order to fix requirements relations, traces, and the architecture. If there is no need to update the models, the process is terminated.

8.2. Tool architecture

The tool contains five components (rounded boxes in Fig. 19): (a) Model Checker part of Maude tool set, (b) Trace Generator using Verification Results in ATL, (c) Trace Generator using Requirements Relations in ATL, (d) Trace Validator using ATL, and (e) Requirements Relation Generator using Traces in ATL.

8.2.1. Model checker in Maude

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 a Maude term (Moment2-AADL). The AADL metamodel is encoded as a set of sorts. The dynamic semantics of AADL is given in rewriting rules (Ölveczky et al., 2010a,b).

8.2.2. 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.

8.2.3. Trace generator using requirements relations in ATL

The input of the component is the *Input Architecture Model*, the *Input Trace Model*, and the *Input Requirements Model*. The component is implemented as an ATL transformation. It generates new traces based on the requirements relations and the constraints in Fig. 10.

Please cite this article in press as: Goknil, A., et al., Generation and validation of traces between requirements and architecture based on formal trace semantics. J. Syst. Software (2013), http://dx.doi.org/10.1016/j.jss.2013.10.006

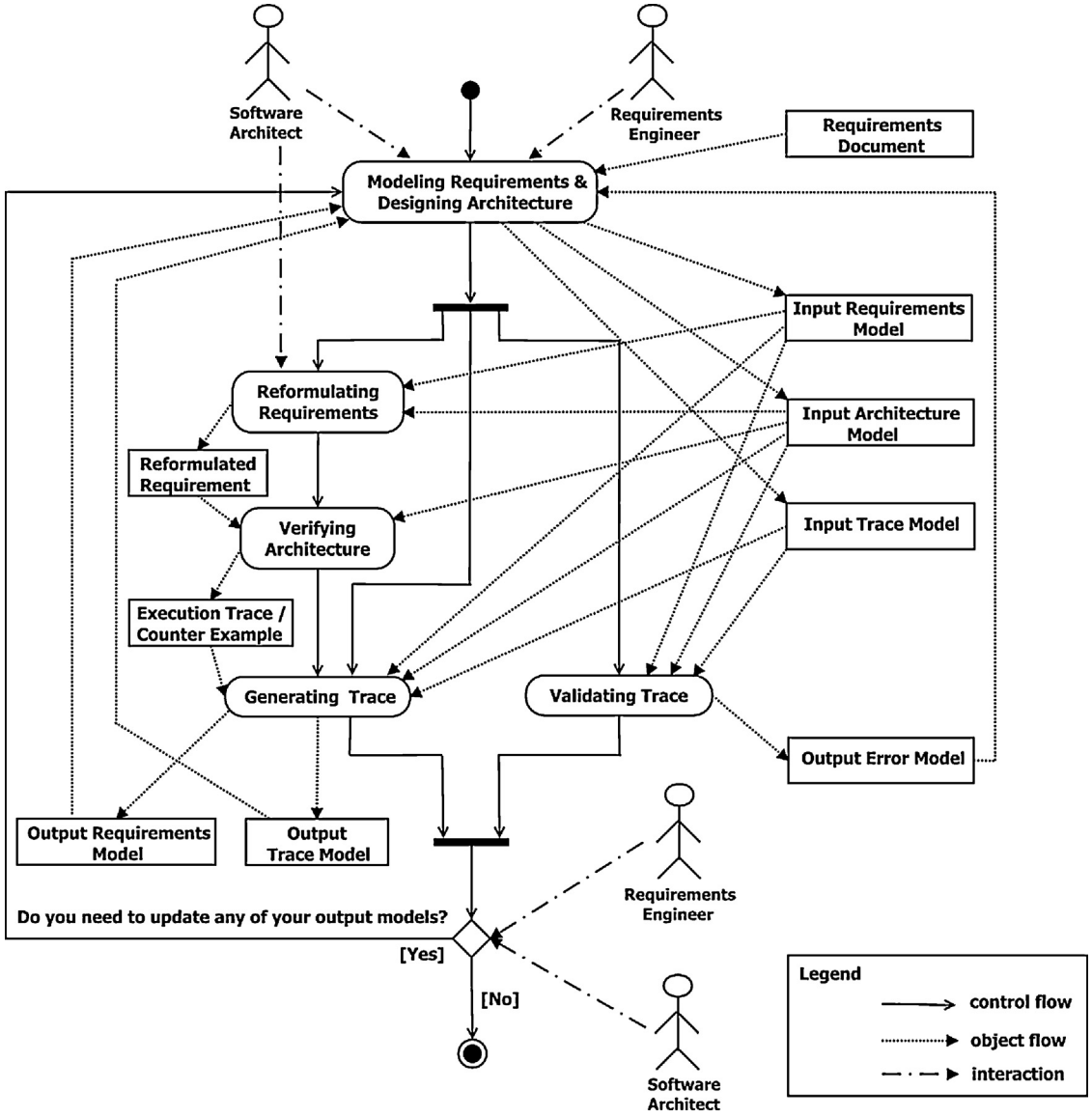


Fig. 18. Modeling process with trace generation and validation.

To represent the 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.

8.2.4. Trace validator in ATL

The input of the component is the *Input Architecture Model*, the *Input Trace Model*, and the *Input Requirements Model*. 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.

8.2.5. Requirements relation generator using traces in ATL

The component generates new requirements relations based on traces in the *Input Trace Model*.

The most important features of the tool are *verifying architecture*, *displaying generated traces*, and *displaying invalid traces*.

8.2.6. Verifying architecture

We use the Open-Source AADL Tool Environment (OSATE) – Topcased which includes an AADL front-end and architecture analysis capabilities as plug-ins. The plug-in (Moment2-AADL) developed by Artur Boronat is used to generate Maude representation of AADL models which can be simulated and verified. Fig. 20 shows the OSATE-Topcased with AADL-Maude plugin.

8.2.7. Displaying generated traces

We use Eclipse model editor (see Fig. 21) to display the *Output Trace Model*.

8.2.8. Displaying invalid traces

The output error model of validating trace activity in Fig. 18 is displayed in the Eclipse model editor (see Fig. 22 for the output trace model).

In Fig. 22 the right-hand side of the window shows the *Output Error Model*. The model contains requirements and requirements relations which cause the invalidity in the trace model. The architectural elements traced from the requirements in the error model can be reached in the trace model in Fig. 21.

Please cite this article in press as: Goknil, A., et al., Generation and validation of traces between requirements and architecture based on formal trace semantics. J. Syst. Software (2013), http://dx.doi.org/10.1016/j.jss.2013.10.006

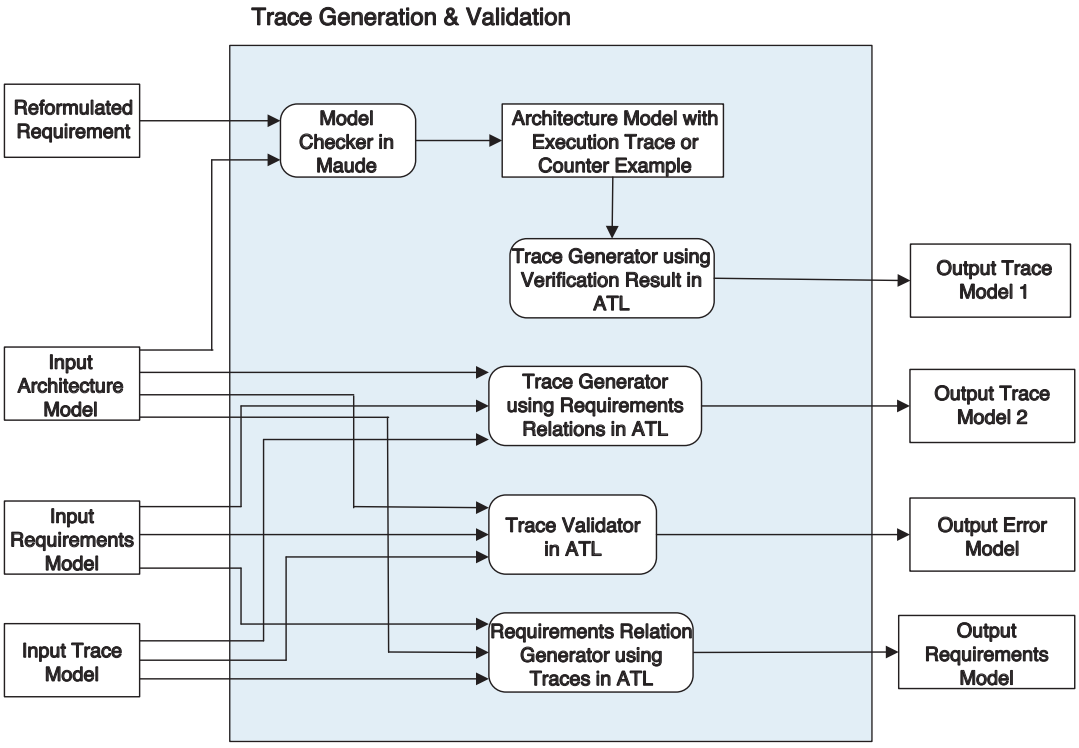


Fig. 19. Overview of the tool architecture.

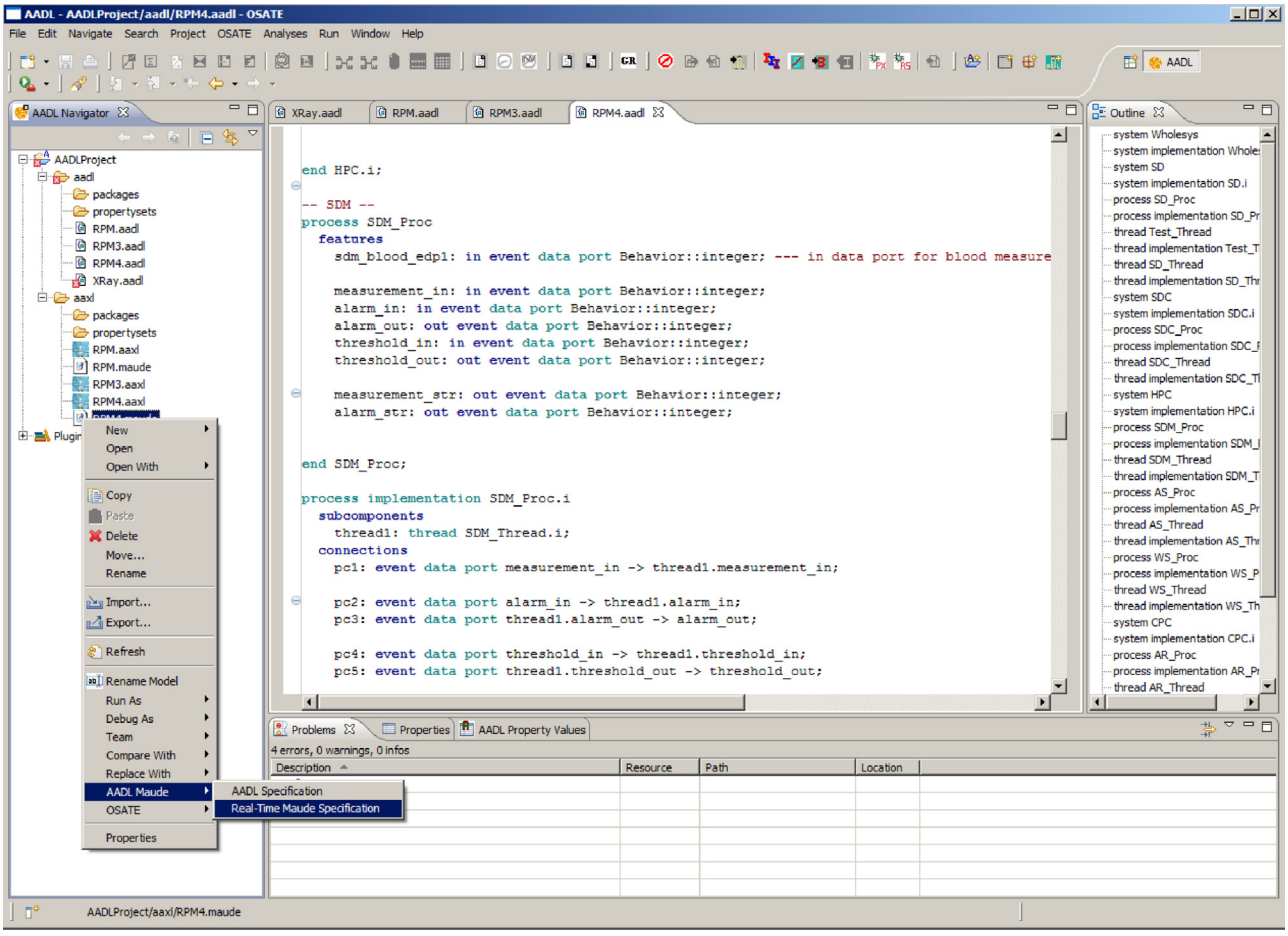


Fig. 20. OSATE with AADL-Maude plugin.

Please cite this article in press as: Goknil, A., et al., Generation and validation of traces between requirements and architecture based on formal trace semantics. J. Syst. Software (2013), http://dx.doi.org/10.1016/j.jss.2013.10.006

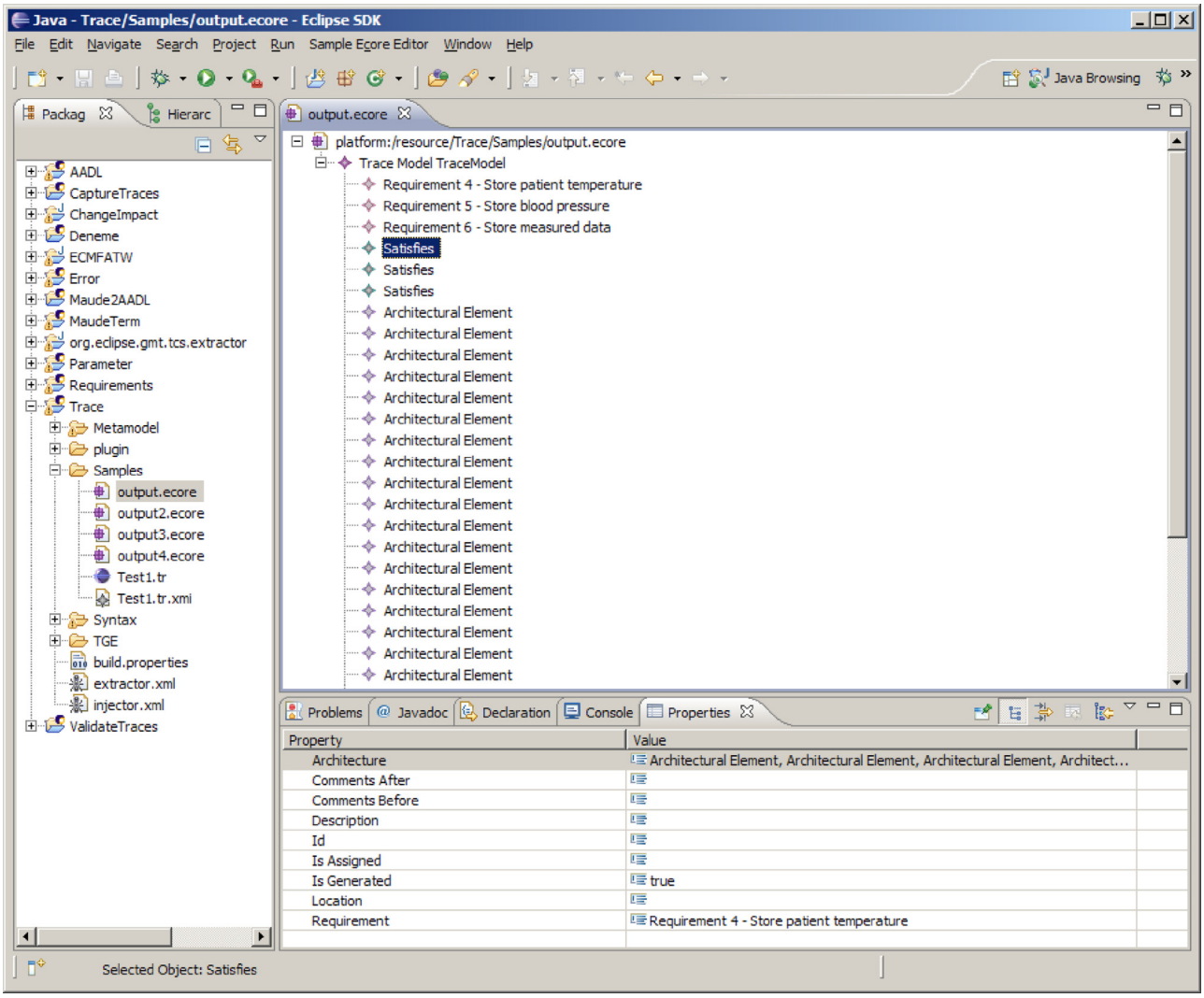


Fig. 21. Output of the generating trace activity.

8.3. Evaluation of the tool

Tools may be evaluated regarding different qualities like *usability*, *performance* and *scalability* among others. The usability of our tool mainly depends on the usability of the Eclipse environment and the available documentation on the approach. For the counter example and execution traces (the output of the component *Architecture Verification in Maude*), no GUI is provided. For a prototype we consider this to be acceptable. This section reports the results of the conducted performance and scalability tests of the tool. The model checking techniques we use may have scalability and performance problems in handling large models and number of states. Therefore, we focus on model checking part of our tool.

Performance testing is conducted to evaluate the compliance of a system or component with specified performance requirements (http://www.aptest.com/glossary.html#S). The requirement in our test is that the tool performs in a reasonable time (say less than one minute) with average number of architectural elements. An estimate for the average number of architectural elements is based on a report by McCormack et al. (2006). 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 (http://www.aptest.com/glossary.html#S). The workload in our performance test is the number of states. Our interpretation of scalability 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 tests is the elapsed time for simulation and verification (in seconds). The independent variable used in the performance tests is the 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. As an example, consider that there are new elements in the architecture for a new system property. New elements may not increase the number of states in the verification of architecture for existing system properties.

Please cite this article in press as: Goknil, A., et al., Generation and validation of traces between requirements and architecture based on formal trace semantics. J. Syst. Software (2013), http://dx.doi.org/10.1016/j.jss.2013.10.006

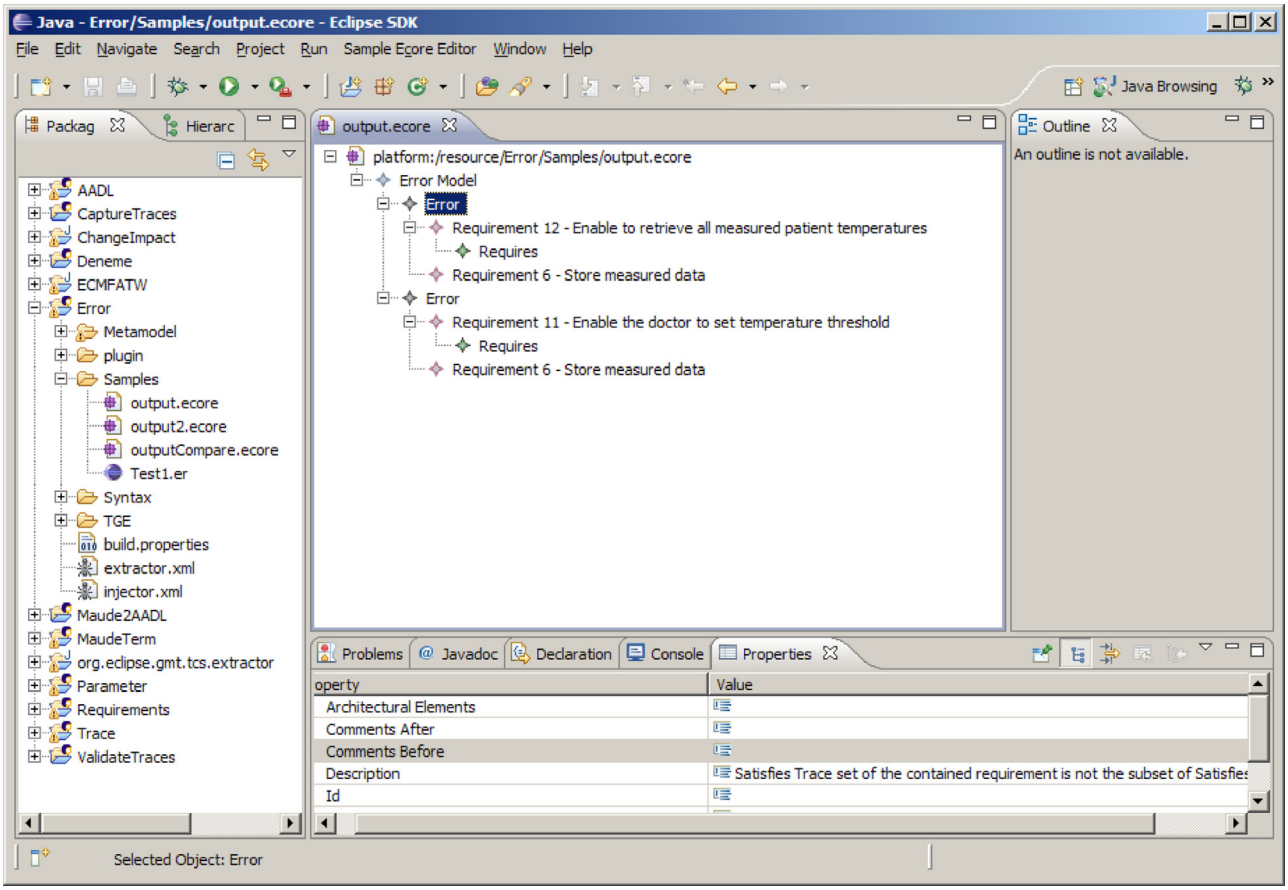


Fig. 22. Output of the validating trace activity.

Memory consumption is not measured in the performance tests. The runs for each performance test are executed six times and the time is shown in Tables 1 and 2. The tests are done on Intel(R) Core(TM)2 Quad CPU Q6600 running at 2.40 GHz with 4096 KB cache, and 2 GB of memory, running Kubuntu 10.04. We use Core Maude 2.4 for Linux. The models used in the performance tests are artificially created with certain number of elements and states. We use a version of operational semantics of AADL excluding the real-time semantics. The performance and scalability test results might be different for the real-time semantics encoded in Real-Time Maude.

Table 1
Simulation times in the performance test.

Number of elements	Simulation time (s) for the execution trace		
	Number of states = 500	Number of states = 1000	Number of states = 2000
(a) Simulation with execution trace			
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
Number of elements	Simulation time (s) for the counter example		
	Number of states = 500	Number of states = 1000	Number of states = 2000
(b) Simulation with counter example			
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

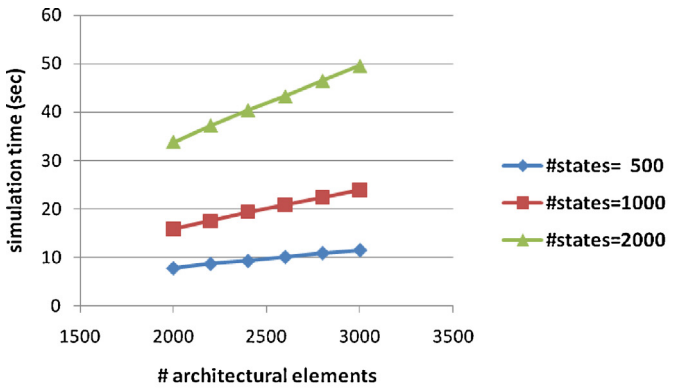
8.3.1. *Performance test*

The test is set up as follows. We increase the number of elements by adding components, data ports and data port 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

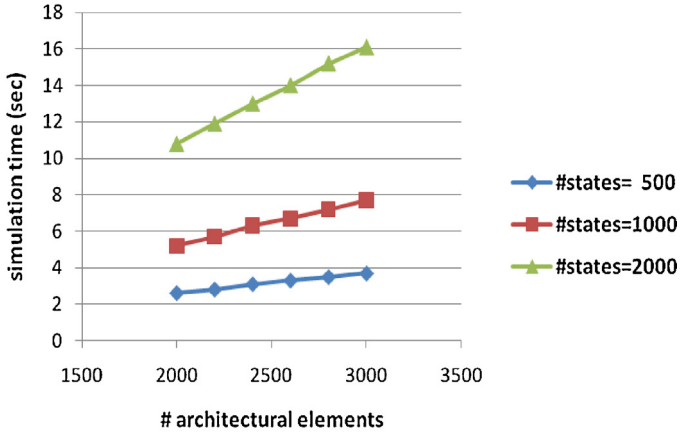
Table 2
Simulation times in the scalability test.

Number of states	Simulation time (s)
(a) Simulation in Maude (number of elements = 10,000)	
10	1.5
100	9.5
1000	82.1
3000	265.4
4500	401.8
5000	–
(b) Simulation in alloy (number of elements = 38)	
20	14.2
40	53.7
60	105.8
80	180.4
100	300.9

Please cite this article in press as: Goknil, A., et al., Generation and validation of traces between requirements and architecture based on formal trace semantics. J. Syst. Software (2013), http://dx.doi.org/10.1016/j.jss.2013.10.006



a) Simulation with Execution Trace



b) Simulation with Counter Example

Fig. 23. Simulation time as the function of the number of architectural elements.

both cases (see Table 1(a) and (b)). The standard deviation of the data is approximately 0.3%.

According to these performance tests, the tool performs below one minute with average number of architectural elements in a real system. The increase in the simulation time is linear and up to 50 s for 2000 states (see Fig. 23).

8.3.2. 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. The 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 (Jackson, 2002). The same execution semantics of AADL in Maude is encoded in Alloy. The first part of the performance test is done in Maude with 10,000 architectural elements (see Table 2(a)). Then, the second part of the performance test is done in Alloy (see Table 2(b)). In (Looman, 2009), we investigated simulation and verification in Alloy. We found that Alloy is not suitable for large number of model elements and states. Therefore, we choose to run the test in Alloy with a smaller number of architectural elements (38 elements) (see Table 2(b)).

According to the scalability test results based on Maude, the simulation time increases linearly when the number of states increases linearly (see Fig. 24). We ran out of memory in Maude when we try simulation for 10,000 architectural elements with 5000 states. In the Alloy experiment, 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 when using Maude rather than using Alloy.

The results depend on the modeling language and its semantics. The results may change with different AADL semantics or with a lower level design language like UML class and activity diagrams.

9. Discussion of the approach

9.1. Maintaining traces

The described fine-grained traceability approach helps in identifying impacted architectural elements if requirements change but after the changes the traces need to be updated as well. The need of trace maintenance occurs when (1) requirements change, and/or (2) architecture changes. With the additional layer of trace links involved in the change management process the requirements engineer and the software architect need (a) to propose changes for requirements and/or architectural elements, (b) to determine impacted elements via traces, (c) to change the impacted requirements/architectural elements for the proposed changes, and (d) finally to maintain traces.

The impact of changes depends on their rationale. For instance, the requirements engineer may delete a property of a requirement because this property is not required any more from the business/stakeholder point of view. The property may be used in other requirements in the model and it also has to be deleted from them. We name these changes and their rationale as *domain changes*. Domain changes modify the overall system properties and require changes in the architecture in order to satisfy the updated/new requirements. Traces between changed requirements and (un-) changed architectural elements might be invalid as well as traces between unchanged requirements and changed architectural elements.

Another type of changes are *semantics-preserving* according to (Buckley et al., 2005) and we consider their rationale as *refactoring* (see Fowler, 1999 for refactoring). For instance, the requirements engineer may delete a property of a requirement to improve the structure of the model without modifying the overall system properties. This property still must hold in the requirements model after the change. Since these changes do not have any impact on the overall system properties, the architecture that satisfies these properties does not need to be changed. Only traces from the changed requirements to the architecture might be invalid. Based on the change rationale and the changed elements, the traces that need to be maintained are determined. This process can be automated and followed by application of the described techniques for trace generation and validation.

9.2. Identifying changes in the code

Our approach can be combined with trace generation methods and tools (Grechanik et al., 2007; Hayes et al., 2006; Murta et al., 2006) for tracing requirements/architecture to source code. In cases where code is automatically generated from the architecture, changes in code can be traced and implemented automatically. Sometimes requirements changes are traced directly to code. It is still beneficial, however, to keep the architecture synchronized with the code and to trace the changes from requirements to the architecture. Architecture, among others, is an aid in understanding and communicating the design of complex systems and should correctly reflect its implementation in code.

Please cite this article in press as: Goknil, A., et al., Generation and validation of traces between requirements and architecture based on formal trace semantics. J. Syst. Software (2013), http://dx.doi.org/10.1016/j.jss.2013.10.006

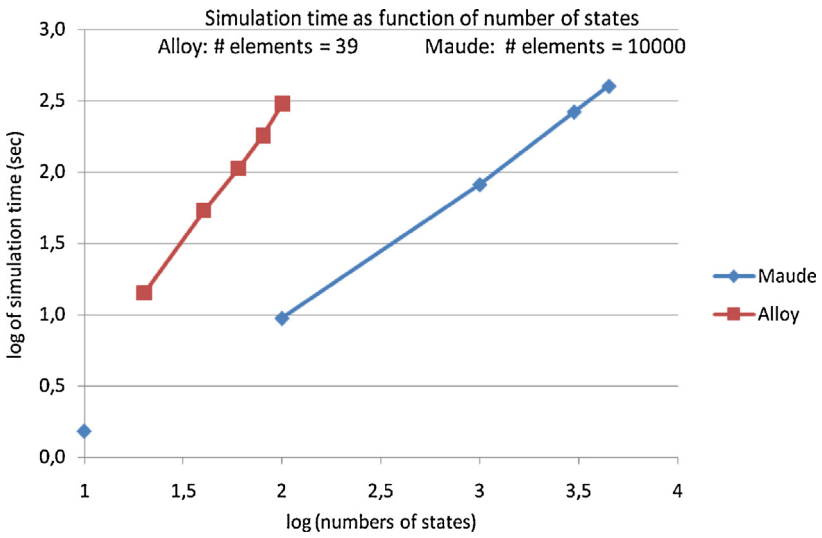


Fig. 24. Simulation time vs. number of states in alloy and Maude.

9.3. Expressing requirements in logic

The approach uses reformulation of requirements in a form suitable for verification. This can be a challenge for the developers in many organizations (Sabaliauskaite et al., 2010). The reformulation is a part of the 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. Therefore, we suggest an iterative semi-automatic process of applying our approach. The software architect can gradually improve the quality of the traces and the requirements.

Case studies conducted with the industry (Ciraci, 2009) show that it is hard to reformulate requirements as LTL/CTL formulas. Domain-specific languages can be used to specify requirements of certain type that allow generation of LTL/CTL formulas (Ciraci, 2009). Starting from natural language, Semantics Business Vocabulary and Rules (SBVR) can support reformulation of requirements in terms of LTL/CTL formulas. Extending our approach with this kind of languages will ease the reformulation of requirements and will improve the usability in terms of the ROI for the traceability.

9.4. Dealing with complex requirements

A requirement may describe multiple system properties or a complex system property amenable to decomposition. In our approach it is not possible to explicitly state which sub-property in a complex requirement fails. The requirements engineer should decompose the requirement into sub-parts (by using the *Contains* relation) until each requirement describes sufficiently simple property. Without such decomposition, the *satisfies* traces for the requirement will be the collection of traces generated from the LTL formulas for each property in the requirement. If one of these LTL formulas fails in the verification, it is concluded that not all properties in the requirement are satisfied. Therefore, satisfies traces cannot be generated for the complex requirement even if other LTL formulas are satisfied. A similar scenario is valid for generating traces by using requirements relations. Based on the constraint in Fig. 10(c) the *satisfies* traces for R_1 is the collection of the satisfies traces for $(R_2, \ldots, R_k)$ where (R_1 contains $R_2, \ldots, R_k$) is a complete decomposition and all contained requirements $(R_2, \ldots, R_k)$ are satisfied. If the complex requirement can be decomposed into other requirements where each requirement specifies a single property represented by an LTL formula, first we can generate traces for the decomposed requirements by verifying the LTL formulas and then we can collectively generate the *satisfies* traces by using the *Contains* relation. Here, we have an underlying assumption that each property in the requirement can be translated into an LTL formula. For the properties that cannot be represented with LTL the software architect has to manually assign traces.

9.5. Manual trace assignment and validation

Our approach supports two kinds of traces between requirements and architecture: *Satisfies* and *AllocatedTo*. *Satisfies* traces are discovered through the formal process of checking requirements against the architecture. For cases where it is not possible to reformulate requirements in a form suitable for verification, the software architect needs to manually assign multiple similar *AllocatedTo* traces at various levels of granularity. For example, traces from a subcomponent to a requirement and from the enclosing component to the same requirement have to be both assigned manually. Obviously, the latter trace can be induced automatically from the former by following the part-whole relations in the architecture. Given the usual high cost and effort of traceability, this deficiency may add an additional overhead. In order to reduce the overhead, the manual trace assignment and validation can be applied for a small set of architecturally significant requirements which cannot be reformulated in the form suitable for verification. On the other hand, the level of automation in the approach can be improved for creating traces at various levels of granularity by exploiting the structural semantics of architecture models.

9.6. Completeness of the architecture

The architecture does not need to be complete in order to check some of its properties. The verification itself enables the architects to determine which properties are not satisfied yet. A counter example means that although the requirement is *allocated to* the architectural elements, i.e., they are involved in the implementation of this requirement, the architecture does not satisfy it and thus, the architecture is still incomplete. On the other hand, some of the properties in the requirements may not be considered by the software architect at all. In this case, it is not useful to translate these properties into LTL formulas to be verified.

Please cite this article in press as: Goknil, A., et al., Generation and validation of traces between requirements and architecture based on formal trace semantics. J. Syst. Software (2013), http://dx.doi.org/10.1016/j.jss.2013.10.006

9.7. Scalability issues

In order to evaluate the approach we focused on model checking part of the tool in the performance and scalability tests since the model checking techniques may have scalability and performance problems in case of large models and number of states. Another scalability challenge is navigating in a huge amount of traces for both generation and validation of traces. This can be addressed by having a risk assessment to detect most volatile requirements. Generating and validating traces for only these requirements may improve the practicality of our approach and to decrease the number of traces. Using queries over traces is another technique for handling large sets of traces (Mader and Cleland-Huang, 2010).

10. Related work

We discuss related work in six categories: *Types and Semantics of Traces*, *Generating and Validating Traces*, *Conformance Assessment*, *Architecture Analysis*, *Analyzing AADL Models*, and *Tool Support*.

10.1. Types and semantics of traces

A number of approaches address types and semantics of traces between R&A. Paige et al. (2011) focus on how to define traces with tool-supported semantics. According to Paige et al. (2011) semantically rich traces possess three characteristics: (1) traces are typed, (2) traces conform to a case-specific trace metamodel, and (3) the case-specific metamodel should be accompanied by a set of case-specific constraints, which cannot be captured by the metamodel. Our trace metamodel includes case-specific trace information such that the trace is generated or assigned. This type of trace information can prevent users and tools from establishing illegitimate traces (Paige et al., 2011). Aizenbud-Reshef et al. (2005) state the need for semantics of traces in general. They present an approach to defining operational semantics for traces in UML. The semantics of a trace is a triplet (*event*, *condition*, and *actions*). This semantics is aimed at change impact analysis. It is hard to use it for generating and validating traces.

Ramesh and Jarke (2001) define *allocated to* and *satisfy* trace types which have definitions similar to ours. Khan et al. (2008) define a dependency model to analyze the impact of evolving requirements dependencies and architecture changes. The dependency model consists of six types of traces: *goal dependency*, *service dependency*, *conditional dependency*, *temporal dependency*, *task dependency* and *infrastructure dependency*. Lago et al. (2009) propose the following trace types between feature models (requirements) and structural models (architecture): *drive*, *modify*, *depend-on*, and *influence*. There is no formal semantics of the trace types in (Khan et al., 2008; Lago et al., 2009; Ramesh and Jarke, 2001). Our trace types are generalizations of these trace types.

10.2. Generating and validating traces

A number of approaches provide generating and validating traces. Egyed et al. (Egyed and Grunbacher, 2002, 2005; Egyed, 2003) provides an automated traceability approach that uses a small number of traces as input. The source code of a system is executed according to some scenarios and then traces are generated between requirements and source code. Dependencies between requirements can be detected based on overlaps among the lines of code implementing those requirements. Our approach uses a similar idea but is applied on executable architecture model.

Schwarz et al. (2010) describe a graph-based traceability approach. 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. Our work assumes that architecture is created manually.

Information retrieval methods are proposed for trace generation. Antoniol et al. (2002) propose an approach for recovering traces between source code and documentation (mainly requirements specifications) using information retrieval methods. Hayes et al. (2006) introduce another approach for trace generation. In both works the main assumption is that programmers use meaningful names for program items so that the analysis of the mnemonics can help to associate high-level concepts with source code.

Grechanik et al. (2007) support generating traces between types and variables in Java programs and elements of use-case diagrams (UCD). The approach combines program analysis, run-time monitoring, and machine learning to generate traces. Relations between program entities are compared with the corresponding relations between elements in UCDs only to validate traces.

Mader et al. (2009) address modification and enhancement of existing traces after changes to artifacts. The approach does not support trace generation. On the contrary, in our approach initial traces can be generated by using architecture verification techniques. A Visual Traceability Modeling Language (VTML) is proposed by Mader and Cleland-Huang (2010). VTML allows users to model trace queries by hiding underlying technical details. The queries created with VTML can be applied on traces generated and validated by our approach.

10.3. Conformance assessment

Conformance assessment is the act of checking whether a requirement is satisfied (Almeida et al., 2007). The assessment can use testing, inspection, model checking or conformation transformation usage (see Almeida et al., 2006 for conformation transformation usage). The usage of architecture verification with requirements relations in our approach can be considered as a conformance assessment of properties in the requirements and architecture.

Almeida et al. (2007) propose a framework that supports management of traces between requirements and design. The framework provides a notion of conformance between application models which reduces the effort for conformance assessment. The conformance between various application models at different levels of abstraction is assessed. In our approach, we focus on conformance assessment between requirements and software architecture.

Paige et al. (2005) give a definition of refinement between models via consistency checking. Formal definitions for model consistency are provided with the definition of refinement in Model Driven Architecture (MDA). The consistency of platform specific and independent models is ensured with cross-model rules that check the preservation of properties between two models. There are other conformance assessment approaches proposed by Egyed (2000), Abi-Antoun and Aldrich (2008), Abi-Antoun and Medvidovic (1999), Moriconi et al. (1995), Heckel and Thone (2005) and Oquendo (2004). Contrary to our approach, most of these works focus on the conformance assessment for architecture and detailed design.

10.4. Architecture analysis

Simulation and model checking of software architecture are parts of our approach for trace generation and validation. Zhang et al. (2009) give a classification and comparison of model checking techniques for software architecture. According to their survey, CHARMY (Pelliccione et al., 2009), using the SPIN model-checker, is

Please cite this article in press as: Goknil, A., et al., Generation and validation of traces between requirements and architecture based on formal trace semantics. J. Syst. Software (2013), http://dx.doi.org/10.1016/j.jss.2013.10.006

one of the most recent architecture analysis approaches. A similar approach that uses SPIN for verifying software architecture is proposed by Bose (1999). These approaches use UML-based notations instead of architecture description languages.

According to Zhang et al. (2009), the Wright language proposed by Allen and Garlan (1997) can be considered as the first work on model checking techniques for software architecture. There are also approaches for analyzing dynamic behavior: Darwin (Magee et al., 1999; Magee, 1999), Chemical Abstract Machine (CHAM) (Compare et al., 1999; Corradini and Inverardi, 1998; Inverardi and Wolf, 1995), dynamic architecture verification using DynAlloy (Bucchiarone and Galeotti, 2008; Bruni et al., 2008b), reconfiguration analysis in service oriented architectures (Baresi et al., 2003), and behavior preservation in dynamic architectures (Heckel and Thone, 2005). An architecture is *dynamic* if it can change during runtime. Typical changes, which are called *re-configurations*, include components joining and leaving the system or changing their connections (Bruni et al., 2008a). Our approach does not support the analysis of dynamic architectures.

ArchJava (Aldrich et al., 2002) is an extension of Java that unifies an architecture with its implementation. It is possible to check if architectural properties are preserved in source code. In our work we study how requirements relations are reflected in the architecture.

10.5. Analyzing AADL models

In the previous subsection, we discussed general architecture analysis techniques. Several authors studied architecture analysis in AADL models. Berthomieu et al. (2008, 2009) perform a formal verification of AADL specifications. A subset of AADL is translated into an extension of Petri nets called *Fiacre* language (Berthomieu et al., 2009). Chkouri et al. (2009) propose another analysis approach using translation of AADL to BIP (Behavior Interaction Priority) language. The analysis technique used by us gives a formal executable semantics to AADL with a behavior annex specification of its thread behavior. On the contrary, the approaches in (Berthomieu et al., 2008, 2009; Chkouri et al., 2009) use translations into imperative languages. Similarly to (Ölveczky et al., 2010a), Yang et al. (2009) propose a formal semantics in Timed Abstract State Machine (TASM) for a limited set of AADL behavior annex (periodic threads and no modes).

Varona-Gómez and Villar (2009) translate AADL models to SystemC models for performance analysis. Bozzano et al. (2009, 2011) present an AADL analysis approach that supports Error Model Annex for modeling faults and repairs. Li et al. (2010) propose the use of Communicating Sequential Processes (CSP) for simulation of AADL models. The works in (Gui et al., 2008) and (Sokolsky et al., 2009) focus on analyzing schedulability with a behavior of a subset of AADL. All these approaches assume that the thread behavior is specified outside AADL.

Apart from simulating and verifying AADL models, de Niz and Feiler (2009) propose the use of AADL models to analyze potentially unintended system behavior. Gilles and Hugues (2008, 2010) present a domain specific language (REAL – Requirement Enforcement Analysis Language) for AADL. Contrary to our approach, the approach in (Gilles and Hugues, 2008, 2010) does not focus on simulation and verification of AADL models.

10.6. Tool support

The INCOSE management tool survey evaluates tools according to several criteria, *traceability analysis* being one of them. According to the survey, current industrial tools provide traceability for requirements to system implementation (including software architecture). However, mechanisms of trace generation and validation for requirements and architecture are not supported in general.

IBM Rational RequisitePro provides only two trace types between requirements, requirements & design, and requirements & implementation: *traceFrom* and *traceTo*. These trace types indicate only the direction of traces. IBM Telelogic Doors provides a mechanism of describing functional decomposition and analysis in UML. The tool supports two types of traces: *internal* and *external*. Internal traces can be created between any two elements in the same model, while external traces can be used to link elements in different models such as traces between R&A. The requirements engineer can also specify his own trace type. Borland Caliber (Borland Caliber Analyst) provides only one trace type. This type can be used for different purposes such as part-whole and refinement relations. A trace can be established between any two artifacts. These artifacts can be of the same type or different types and even external artifacts. The reasoning facilities of IBM Rational RequisitePro, IBM Telelogic Doors, and Borland Caliber are based only on the transitivity property of the traces. These tools do not support automatic generation and validation of traces.

TopTeam Analyst supports four trace types. Three of these traces (*traces into*, *impact*, *used in*) are directed and one of them (*trace*) is undirected. The undirected trace is considered as a generic trace type for other trace types. None of the trace types have formal semantics. The tool does not support trace generation and validation.

11. Final considerations

We presented an approach for generation and validation of traces between requirements and software architecture. The approach assumes that requirements and requirements relations are captured in a model and the software architecture is expressed in AADL, an architectural description standard language driven by major industrial players. We apply model checking for verification and simulation of architecture models based on the available dynamic semantics of AADL. The semantics is expressed as a rewriting logic theory in Maude.

Our approach improves the process of collecting traceability information in two major ways. First, traces can be generated automatically by checking if a requirement given as an LTL formula is satisfied by the architecture. This makes the process of establishing traces faster and less error prone compared to manually assigning traces. Second, traces are validated by using verification techniques and constraints ensuring that requirements relations are reflected in the software architecture. This eliminates false positive traces and helps in identifying missed traces. As an additional result, the requirements model may be improved by detecting invalid requirements relations and discovering new relations. The process is generally semi-automatic and iterative since the software architect has to decide on the outcome of the supporting tools.

Our work can be extended in several ways. In order to apply verification of architecture, requirements need to be reformulated as LTL formulas. Generally this requires certain knowledge in formal methods from the software architects that they do not always master. We plan to investigate automatic generation of LTL formulas from requirements expressed in a restricted natural language, for example, following the SBVR conventions. In our current tool the formulation of the LTL formulas is done manually.

The presented work is independent from the process of obtaining an architecture from requirements. It is possible to use a method that derives an architecture from requirements in a structured way thus obtaining initial traces as a byproduct.

Our approach can be easily generalized to another architectural description language armed with an executable semantics and

Please cite this article in press as: Goknil, A., et al., Generation and validation of traces between requirements and architecture based on formal trace semantics. J. Syst. Software (2013), http://dx.doi.org/10.1016/j.jss.2013.10.006

tools for architectural simulation and verification. For example, the operational semantics of the architecture description language can be formalized in a graph rewriting environment such as GROOVE (GRaphs for Object-Oriented VErification) (Rensink, 2003) and Alloy (Jackson, 2002). Other logics like *Computation Tree Logic* (*CTL*) can also be used.

If another requirements modeling technique is used, we need to interpret the supported requirements relations in terms of relations among the traced architectural elements.

A requirement may describe multiple system properties and/or a complex system property amenable to decomposition. In our approach it is not possible to explicitly state which property in a complex requirement fails. The requirements engineer should decompose the requirement into sub-parts (e.g. by using the *Contains* relation) until each requirement describes only a single property.

The approach aims at preserving the requirements relations in their implementation in the architecture. There might be some cases where extra dependencies not identified in the requirements analysis are manifested in the architecture. The software architect should update the requirements model by introducing new relations among the requirements or should reconsider the architecture.

We mainly focused on the scalability of our tool for generating and validating traces. Model checking techniques may have problems in handling large amounts of model elements and states. Therefore, the scalability of the tool depends on the scalability of the model checking algorithms in Maude. The model checking part of the tool shows a satisfactory performance results for architectures with realistic size. The user interface needs further improvement for usability. The integration of the tool components is currently done manually and we need a proper user interface to control them.

Acknowledgment

The research presented in this paper is part of the QuadREAD project (http://quadread.ewi.utwente.nl).

Appendix A. Part of the RPM requirements document

In this appendix we give an overview of the requirements of the Remote Patient Monitoring (RPM) system as used in this paper.

Requirements (partial)

Requirement 1 *The system shall measure temperature from a patient.*
Requirement 2 *The system shall measure blood pressure from a patient.*
Requirement 3 *The system shall measure blood pressure and temperature from a patient.*
Requirement 4 *The system shall store patient temperature measured by the sensor in the central storage.*
Requirement 5 *The system shall store patient blood pressure measured by the sensor in the central storage.*
Requirement 6 *The system shall store data measured by sensors in the central storage.*
Requirement 7 *The system shall warn the doctor when the temperature threshold is violated.*
Requirement 8 *The system shall generate an alarm if the temperature threshold is violated.*
Requirement 9 *The system shall show the doctor the temperature alarm at the doctors' computers.*
Requirement 10 *The system shall store all generated temperature alarms in a central database.*
Requirement 11 *The system shall enable the doctor to set the temperature threshold for a patient.*
Requirement 12 *The system shall enable the doctor to retrieve all stored temperature measurements for a patient.*
Requirement 13 *The system shall enable the doctor to retrieve all stored temperature alarms for a patient.*
Requirement 14 *The system shall store patient temperature measured by the sensor in the central storage and it shall warn the doctor when the temperature threshold is violated.*
Requirement 15 *The system shall store patient Central Venous Pressure (CV Pressure) measured by the sensor in the central storage.*

Appendix B. Abbreviations of elements in the RPM system

In this appendix we give the explanations of the abbreviations of the architectural elements of the Remote Patient Monitoring (RPM) system used in the paper.

Abbreviation	Explanation
SD	Sensor device
SDC	Sensor device coordinator
SDM	Sensor device manager
AS	Alarm service
AR	Alarm receiver
WS	Web server
WC	Web client
HPC	Host personal computer
CPC	Client personal computer
sd_blood_edp1	Event data port 1 for blood pressure in sensor device
sd_blood_edp2	Event data port 2 for blood pressure in sensor device
sd_blood_edp3	Event data port 3 for blood pressure in sensor device
sd_blood_edp4	Event data port 4 for blood pressure in sensor device
sd_temp_edp1	Event data port 1 for temperature in sensor device
sd_temp_edp2	Event data port 2 for temperature in sensor device
sd_temp_edp3	Event data port 3 for temperature in sensor device
sd_temp_edp4	Event data port 4 for temperature in sensor device
sd_temp_alarm_edp1	Event data port 1 for temperature alarm in sensor device
sd_temp_alarm_edp1	Event data port 1 for temperature alarm in sensor device
sd_temp_alarm_edp3	Event data port 3 for temperature alarm in sensor device
sd_temp_alarm_edp4	Event data port 4 for temperature alarm in sensor device
sdThr	Thread in sensor device
sdc_blood_edp1	Event data port 1 for blood pressure in sensor device controller
sdc_blood_edp2	Event data port 2 for blood pressure in sensor device controller
sdc_blood_edp3	Event data port 3 for blood pressure in sensor device controller
sdc_blood_edp4	Event data port 4 for blood pressure in sensor device controller
sdc_blood_edp5	Event data port 5 for blood pressure in sensor device controller
sdc_blood_edp6	Event data port 6 for blood pressure in sensor device controller
sdc_temp_edp1	Event data port 1 for temperature in sensor device controller
sdc_temp_edp2	Event data port 2 for temperature in sensor device controller
sdc_temp_edp3	Event data port 3 for temperature in sensor device controller
sdc_temp_edp4	Event data port 4 for temperature in sensor device controller
sdc_temp_edp5	Event data port 5 for temperature in sensor device controller
sdc_temp_edp6	Event data port 6 for temperature in sensor device controller
sdc_temp_alarm_edp1	Event data port 1 for temperature alarm in sensor device controller
sdc_temp_alarm_edp2	Event data port 2 for temperature alarm in sensor device controller
sdc_temp_alarm_edp3	Event data port 3 for temperature alarm in sensor device controller
sdc_temp_alarm_edp4	Event data port 4 for temperature alarm in sensor device controller

Please cite this article in press as: Goknil, A., et al., Generation and validation of traces between requirements and architecture based on formal trace semantics. J. Syst. Software (2013), http://dx.doi.org/10.1016/j.jss.2013.10.006

Appendix B (*Continued*)

sdc_temp_alarm_edp5	Event data port 5 for temperature alarm in sensor device controller
sdc_temp_alarm_edp6	Event data port 6 for temperature alarm in sensor device controller
sdcThr	Thread in sensor device controller
sdm_blood_edp1	Event data port 1 for blood pressure in sensor device manager
sdm_blood_edp2	Event data port 2 for blood pressure in sensor device manager
sdm_blood_strg	Storage for blood pressure in sensor device manager
sdm_temp_edp1	Event data port 1 for temperature in sensor device manager
sdm_temp_edp2	Event data port 2 for temperature in sensor device manager
sdm_temp_strg	Storage for temperature in sensor device manager
sdm_temp_alarm_edp1	Event data port 1 for temperature alarm in sensor device manager
sdm_temp_alarm_edp2	Event data port 2 for temperature alarm in sensor device manager
sdm_temp_alarm_strg	Storage for temperature alarm in sensor device manager
sdmThr	Thread in sensor device manager
hpc_blood_edp1	Event data port 1 for blood pressure in host personal computer
hpc_temp_edp1	Event data port 1 for temperature in host personal computer
hpc_temp_req_edp1	Event data port 1 for temperature request in host personal computer
hpc_temp_alarm_edp1	Event data port 1 for temperature alarm in host personal computer
wc_temp_req_edp1	Event data port 1 for temperature request in web client
wc_temp_req_edp2	Event data port 2 for temperature request in web client
wc_temp_req_edp3	Event data port 3 for temperature request in web client
wc_temp_req_edp4	Event data port 4 for temperature request in web client
wcThr	Thread in web client
ws_temp_req_edp1	Event data port 1 for temperature request in web server
ws_temp_req_edp2	Event data port 2 for temperature request in web server
ws_temp_req_edp3	Event data port 3 for temperature request in web server
ws_temp_req_edp4	Event data port 4 for temperature request in web server
wsThr	Thread in web server
cpc_temp_req_edp1	Event data port 1 for temperature request in client personal computer
cpc_temp_req_edp2	Event data port 2 for temperature request in client personal computer
cpc_ar	Alarm receiver in client personal computer

References

Abi-Antoun, M., Aldrich, J., 2008. Static conformance checking of runtime architectural structure. In: Carnegie Mellon University Technical Report, CMU-ISR-08-132.

Abi-Antoun, M., Medvidovic, N., 1999. Enabling the refinement of a software architecture into a design. LNCS 1723, 17–31 (UML'99).

Aizenbud-Reshef, N., Paige, R.F., Rubin, J., Shaham-Gafni, Y., Kolovos, D.S., 2005. Operational semantics for traceability. ECMDA-TW 2005, 7–14.

Aldrich, J., Chambers, C., Notkin, D., 2002. Architectural Reasoning in ArchJava. In: ECOOP'02, LNCS, vol. 2374, pp. 334–367.

Allen, R., Garlan, D., 1997. A formal basis for architectural connection. ACM Trans. Softw. Eng. Methodol. 6 (3), 213–249.

Almeida, J.P.A., Dijkman, R., Pires, L.F., Quartel, D., van Sinderen, M., 2006. Model-driven design, refinement and transformation of abstract interactions. Int. J. Coop. Inf. Sys. 15, 599–632.

Almeida, J.P.A., Iacop, M.E., van Eck, P., 2007. Requirements traceability in model-driven development: applying model and transformation conformance. Inf. Syst. Front. 9, 327–342.

Antoniol, G., Canfora, G., Casazza, G., De Lucia, A., Merlo, E., 2002. Recovering traceability links between code and documentation. IEEE Trans. Soft. Eng. 28 (10), 970–983.

Baresi, L., Heckel, R., Thone, S., Varro, D., 2003. Modeling and validation of service-oriented architectures: application vs. style. ACM, 68–77 (ESEC/FSE'03).

Berthomieu, B., Bodeveix, J.P., Farail, P., Filali, M., Garavel, H., Gaufillet, P., et al., 2008. Fiacre: an intermediate language for model verification in the TOPCASED environment. In: 4th European Congress on Embedded Real-Time Software ERTS 2008.

Berthomieu, B., Bodeveix, J.P.C., Dal-Zilio, C., Filali, S., Vernadat, M.F., 2009. Formal verification of AADL specifications in the Topcased environment. In: Ada-Europe'09, LNCS, vol. 5570.

Borland Caliber Analyst. From http://www.borland.com/us/products/caliber/index.html

Bose, P.K., 1999. Automated translation of UML models of architectures for verification and simulation using SPIN. In: ASE'99.

Boudiaf, N., Mokhati, F., Badri, M., 2008. Supporting formal verification of DIMA multi-agents models: towards a framework based on Maude model checking. Int. J. Soft. Eng. Knowl. Eng. 18 (7), 853–875.

Bozzano, M., Cimatti, A., Roveri, M., Katoen, J.P., Nguyen, V.N., Noll, T., 2009. Verification and performance evaluation of AADL models. ACM, 285–286 (ESEC-FSE'09).

Bozzano, M., Cimatti, A., Katoen, J.P., Nguyen, V.N., Noll, T., Roveri, M., 2011. Safety dependability, and performance analysis of extended AADL models. Comput. J. 54 (5), 754–775.

Bruni, R., Bucchiarone, A., Gnesi, S., Hirsch, D., Lafuente, A.L., 2008a. Graph-based design and analysis of dynamic software architectures. In: Concurrency, Graphs and Models LNCS, vol. 5065, pp. 37–56.

Bruni, R., Bucchiarone, A., Gnesi, S., Melgratti, H., 2008b. Modelling dynamic software architectures using typed graph grammars. ENTCS 213 (1), 39–53 (GT-VC 2007).

Bucchiarone, A., Galeotti, J.P., 2008. Dynamic software architectures verification using DynAlloy. In: GT-VMT 2008, Proceedings of the Seventh International Workshop on Graph Transformation and Visual Modeling Techniques.

Buckley, J., Mens, T., Zenger, M., Rashid, A., Kniesel, G., 2005. Towards a taxonomy of software change. J. Softw. Mainten. Evol.: Res. Pract. 17, 309–332.

Chkouri, M.Y., Robert, A., Bozga, M., Sifakis, J., 2009. Translating AADL into BIP – application to the verification of real-time systems. Models in software engineering: workshops and symposia at MODELS 2008. LNCS 5421, 5–19.

Ciraci, S., 2009. Graph based verification of software evolution requirements. University of Twente (PhD thesis).

Clavel, M., Duran, F., Eker, S., Lincoln, P., Marti-Oliet, N., Meseguer, J., et al., 2002. Maude: specification and programming in rewriting logic. Theor. Comput. Sci. 285, 187–243.

Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., et al., 2007. All about Maude – a high-performance logical framework. LNCS, 4350.

Compare, D., Inverardi, P., Wolf, A.L., 1999. Uncovering architectural mismatch in component behavior. Sci. Comput. Prog. 33 (2), 101–131.

Corradini, F., Inverardi, P., 1998. Model checking of CHAM description of software architecture. In: WICSA 1998.

Cuenot, P., Frey, P., Johansson, R., Lönn, H., Papadopoulos, Y., Reiser, M.O., et al., 2011. The EAST-ADL architecture description language for automotive embedded software. In: Model-Based Engineering of Embedded Real-Time Systems – LNCS, vol. 6100, pp. 297–307.

de Niz, D., Feiler, P.H., 2009. Verification of replication architectures in AADL. In: 14th International Conference on Engineering of Complex Computer Systems, IEEE Computer Society, pp. 365–370.

Egyed, A., 2000. Automatically validating model consistency during refinement. In: Technical Report in Center for Software Engineering. University of Southern California.

Egyed, A., 2003. A scenario-driven approach to trace dependency analysis. IEEE Trans. Softw. Eng. 29 (2), 116–132.

Egyed, A., Grunbacher, P., 2002. Automated requirements traceability: beyond the record and replay paradigm. In: 17th IEEE International Conference on Automated Software Engineering (ASE'02), pp. 163–171.

Egyed, A., Grunbacher, P., 2005. Supporting software understanding with automated requirements traceability. Int. J. Soft. Eng. Knowl. Eng. 15 (5), 783–810.

Fowler, M., 1999. Refactoring: Improving the Design of Existing Code. Addison-Wesley, Massachusetts.

Gilles, O., Hugues, J., 2008. Validating requirements at model-level. In: Proceedings of the 4th workshop on Model-Oriented Engineering (IDM'08).

Gilles, O., Hugues, J., 2010. Expressing and enforcing user-defined constraints of AADL models. In: Proceedings of the 5th UML and AADL Workshop (UML and AADL 2010).

Goknil, A., 2011. Traceability of requirements and software architecture for change management. University of Twente, Enschede (PhD thesis).

Goknil, A., Kurtev, I., van den Berg, K., 2010. Tool support for generation and validation of traces between requirements and architecture. In: ECMFA-TW 2010, pp. 39–46.

Goknil, A., Kurtev, I., van den Berg, K., Veldhuis, J.W., 2011. Semantics of trace relations in requirements models for consistency checking and inferencing. Softw. Syst. Model. 10 (1), 31–54.

Grechanik, M., McKinley, K.S., Perry, D.E., 2007. Recovering and using use-case-diagram-to-source-code traceability links. In: ESEC-FSE 2007, pp. 95–104.

Gui, S., Luo, L., Li, Y., Wang, L., 2008. Formal schedulability analysis and simulation for AADL. ICESS 2008, 429–435.

Hayes, J.H., Dekthyar, A., Sundaram, S.K., 2006. Advancing candidate link generation for requirements tracing: the study of methods. IEEE Trans. Softw. Eng. 32 (1), 4–19.

Heckel, R., Thone, S., 2005. Behavior-preserving refinement relations between dynamic software architectures. LNCS 3423, 1–27 (WADT 2004).

IBM Rational RequisitePro. From http://www-01.ibm.com/software/awdtools/reqpro/

Please cite this article in press as: Goknil, A., et al., Generation and validation of traces between requirements and architecture based on formal trace semantics. J. Syst. Software (2013), http://dx.doi.org/10.1016/j.jss.2013.10.006

IBM Telelogic Doors. From http://www.telelogic.com/Products/doors/doors/index.cfm
INCOSE Requirements Management Tool Survey. From http://www.incose.org
Inverardi, P., Wolf, A.L., 1995. Formal specification and analysis of software architectures using the chemical abstract machine model. IEEE Trans. Softw. Eng. 21 (4), 373–386.
Jackson, D., 2002. Alloy: a lightweight object modelling notation. ACM Trans. Softw. Eng. Methodol. 11 (2), 256–290.
Jouault, F., Kurtev, I., 2006. Transforming models with ATL. LNCS 3844 (MoDELS 2005).
Jouault, F., Allilaire, F., Bezivin, J., Kurtev, I., 2008. ATL: a model transformation tool. Sci. Comput. Prog. 72 (1–2), 31–39.
Khan, S.S., Greenwood, P., Garcia, A., Rashid, A., 2008. On the impact of evolving requirements-architecture dependencies: an exploratory study. LNCS 5074, 243–257 (Caise 2008).
Lago, P., Muccini, H., van Vliet, H., 2009. A scoped approach to traceability management. J. Syst. Softw. 82, 168–182.
Li, C., Zhou, X., Dong, Y., 2010. Formal behavior specification for AADL. 2nd International Conference on Industrial and Information Systems IIS, 110–113.
Looman, S.A.M., 2009. Impact analysis of changes in functional requirements in the behavioral view of software architectures. University of Twente (M.Sc. thesis).
Mader, P., Cleland-Huang, J., 2010. A visual traceability modeling language. LNCS 6394, 226–240 (MODELS 2010).
Mader, P., Gotel, O., Philippow, I., 2009. Enabling automated traceability maintenance through the upkeep of traceability relations. LNCS 5562, 174–189 (ECMDA-FA 2009).
Magee, J., 1999. Behavioral analysis of software architectures using LTSA. In: ICSE 1999, pp. 634–637.
Magee, J., Kramer, J., Giannakopoulou, D., 1999. Behaviour analysis of software architectures. In: First Working IFIP Conference on Software Architecture (WICSA). IFIP Conference Proceedings, vol. 140, pp. 35–50.
Malan, R., Bredemeyer, D., 2002. Architectural requirements in the visual architecting process, From http://www.bredemeyer.com/ArchitectingProcess/ArchitecturalRequirements.htm
McCormack, A., Rusnak, J., Baldwin, C.Y., 2006. Exploring the structure of complex software designs: an empirical study of open source and proprietary code. Manage. Sci. 52 (7), 1015–1030.
Moment2-AADL. From http://www.cs.le.ac.uk/people/aboronat/tools/moment2-aadl/
Moriconi, M., Qian, X., Riemenschneider, R.A., 1995. Correct architecture refinement. IEEE Trans. Softw. Eng. 21 (4), 356–372.
Murphy, G.C., Notkin, D., Sullivan, K., 1995. Software reflexion models: bridging the gap between source and high-level models. In: SIGSOFT 1995.
Murta, L.G.P., van der Hoek, A., Werner, C.M.L., 2006. ArchTrace: policy-based support for managing evolving architecture-to-implementation traceability links. In: 21st IEEE International Conference on Automated Software Engineering (ASE'06), pp. 135–144.
Ölveczky, P.C., Boronat, A., Mesequer, J., 2010a. Formal semantics and analysis of behavioral AADL models in real-time Maude. LNCS 6117, 47–62 (FMOODS/FORTE 2010).
Ölveczky, P.C., Boronat, A., Mesequer, J., Pek, E., 2010b. Formal semantics and analysis of behavioral AADL models in real-time Maude. In: Technical Report at UIUC.
OMG, 2010. SysML Specification, Retrieved from http://www.sysml.org/specs.htm (05.01.10).
OMG Semantics of Business Vocabulary and Rules (SBVR). From http://www.omg.org/news/meetings/ThinkTank/past-events/2006/presentations/04-WS1-2_Hall.pdf
Oquendo, F., 2004. π-ARL: an architecture refinement language for formally modelling the stepwise refinement of software architectures. ACM SIGSOFT Softw. Eng. Notes 29 (5), 1–20.
Paige, R.F., Kolovos, D.S., Polack, F.A.C., 2005. Refinement via consistency checking in MDA. ENTCS 137 (2), 151–161.
Paige, R.F., Drivalos, N., Kolovos, D.S., Fernandes, K.J., Power, C., Olsen, G.K., et al., 2011. Rigorous identification and encoding of trace-links in model-driven engineering. Softw. Syst. Model. 10 (4), 469–487.
Pelliccione, P., Inverardi, P., Muccini, H., 2009. CHARMY: a framework for designing and verifying architectural specifications. IEEE Trans. Softw. Eng. 35 (3), 325–346.
Post, H., Sinz, C., Merz, F., Gorges, T., Kropf, T., 2009. Linking functional requirements and software verification. In: RE 2009, pp. 295–302.
Ramesh, B., Jarke, M., 2001. Towards reference models for requirements traceability. IEEE Trans. Softw. Eng. 27 (1), 58–93.
Rensink, A., 2003. The GROOVE simulator: a tool for state space generation. In: Proceedings of Applications of Graph Transformations with Industrial Relevance (AGTIVE) – LNCS 3062, pp. 479–485.
Sabaliauskaite, G., Loconsole, A., Engstrom, E., Unterkalmsteiner, M., Regnell, B., Runeson, P., et al., 2010. Challenges in aligning requirements engineering and verification in a large-scale industrial context. LNCS 6182, 128–142 (REFSQ 2010).
SAE, 2010. Architecture Analysis and Design Language (AADL), Retrieved from http://www.aadl.info (05.01.10).
Schwarz, H., Ebert, J., Winter, A., 2010. Graph-based traceability: a comprehensive approach. Softw. Syst. Model. 9 (4), 473–492.
Sokolsky, O., Lee, I., Clarke, D., 2009. Process-algebraic interpretation of AADL models. Reliable software technologies – Ada Europe. LNCS 5570, 222–236.
SWEBOOK. Guide to Software Engineering Body of Knowledge. IEEE Computer Society.
The Open Source Toolkit for Critical Systems (TOPCASED). From http://www.topcased.org
TopTeam Analyst. From http://www.technosolutions.com/topteam_requirements_management.html
Tool for Requirements Inferencing and Consistency Checking (TRIC). From http://trese.cs.utwente.nl/tric/
van Lamswerdee, A., 2001. Goal-oriented requirements engineering: a roundtrip from research to Practice. Invited Minitutorial, Proceedings RE'01—5th International Symposium Requirements Engineering, 249–263.
van Ommering, R., van der Linden, F., Kramer, J., Magee, J., 2000. The koala component model for consumer electronics software. Computer 33 (3), 78–85.
Varona-Gómez, R., Villar, E., 2009. AADS: AADL simulation and performance analysis in systemC. In: Proceedings of the IEEE/ACM Design, Automation and Test in Europe.
Wasson, C.S., 2006. System, Analysis, Design, and Development: Concepts, Principles and Practices. John Wiley & Sons, New Jersey.
Yang, Z., Hu, K., Ma, D., Pi, L., 2009. Towards a formal semantics for the AADL behavior annex. In: DATE 2009, pp. 1166–1171.
Zhang, P., Muccini, H., Li, B., 2009. A classification and comparison of model checking software architecture techniques. J. Syst. Softw. 83, 723–744.

Arda Goknil is a postdoctoral fellow at AOSTE research team of INRIA in France. He received his PhD from the Software Engineering Group (TRESE) of the University of Twente in the Netherlands in 2011. Goknil received his B.Sc. degree in 2003, M. Sc. degree in 2006, from the Computer Engineering Department of Ege University in Turkey. His research interests include traceability, requirements modeling and change impact analysis.

Ivan Kurtev holds an M. Sc. degree in Computer Science from University of Sofia and a Ph.D. degree in Software Engineering from University of Twente. He is a management consultant at Nspyre in Eindhoven, the Netherlands. His main research interests are in the domain of Model Driven Engineering with a focus on model transformation languages, metamodeling, requirements modeling and traceability.

Klaas van den Berg has been retired from the Software Engineering Group and the University of Twente since July 1st, 2010. He received his B.Sc. and M.Sc. in Electrical Engineering and his PhD degree in Computer Science from the University of Twente. He has been a lecturer for some years at the University of Zambia and the University of Dar es Salaam. He has been a lecturer in the Software Engineering Group of the University of Twente since 1986. His research interests include software measurement and quality metrics, empirical software engineering, software process modelling, software evolvability, traceability and model-driven engineering. As responsible and senior researcher, he was involved in several research projects related to these subjects. He participated in numerous workshops and conferences as organizer and member of the program committee.

Please cite this article in press as: Goknil, A., et al., Generation and validation of traces between requirements and architecture based on formal trace semantics. J. Syst. Software (2013), http://dx.doi.org/10.1016/j.jss.2013.10.006