A Rule-based Approach for Evolution of AADL Models based on Changes in Functional Requirements

Arda Goknil
University of Luxembourg
arda.goknil@uni.lu

Ivan Kurtev
Altran
ivan.kurtev@altran.com

Klaas van den Berg
University of Twente
vdberg.nl@gmail.com

ABSTRACT
The increasing complexity of software systems makes change management costly and time consuming. To ensure the cost-effective system longevity and endurance, it is crucial to apply change management in the early stages of software development. In this paper, we introduce a rule-based approach to make software architecture evolving based on counter examples provided by a model checker for violated, changing functional requirements. The approach works on software architecture in AADL and is based on verifying functional requirements using Maude model checker. Our goal is to provide guidelines to the architect about potential changes. Using an industrial example, we show how our approach helps in determining changes to produce a new version of the architecture.

CCS Concepts
• Software and its engineering → Software evolution;

Keywords
Software Architecture; Change Management; AADL; Maude.

1. INTRODUCTION
The requirements of software systems are subject to changes due to newly emerging business goals and to changes in the systems' operational environment. The increasing system complexity makes change management costly and time consuming. Even a single requirements change may have a significant impact on the system. To reduce the cost of changes, it is crucial to analyze requirements changes as early as possible during software development, and to propagate them to lower-level artifacts.

Considerable research has been devoted to the analysis of requirements and design changes on lower-level artifacts such as source code and test cases. For instance, Briand et al. [2] propose an approach to support test selection from regression test suites based on the change analysis of object-oriented designs. Gallagher et al. [5] employ program slicing to delineate the effects of changes in source code. Mens et al. [16] discuss the need of tracing the architecture to the implementation so that any architecture change is reflected in the implementation. Less attention has been paid to the analysis of requirements changes on high-level design artifacts such as architecture. There are works [15] dealing with architecture evolution, but mainly focus on architecture refactoring. The analysis of requirements changes on architecture are beneficial due to several reasons. Immediate identification of source code changes caused by requirements changes can be difficult since there is an abstraction gap between requirements and code. Architecture is at a higher level of abstraction and can be employed for automated code generation. The system sustainability largely depends on the architecture sustainability. Managing architecture evolution is crucial to ensure the cost-effective architecture longevity and endurance, i.e., the architecture sustainability.

In this paper, we present an approach that proposes changes for software architecture given in Architecture Analysis & Design Language (AADL) when the architecture does not satisfy new/changed functional requirements. The approach is based on verifying functional requirements on the software architecture. The verification output is a counter example if the requirements are not satisfied. The counter example is used together with a classification of architecture changes to identify changes for the architecture. The identified changes are guidelines for the architect during the manual design activity. By considering these changes, the architect manually produces a new version of the architecture that possibly satisfies the requirements.

We use AADL because it is an industry-driven language proposed by companies in different domains and because of the availability of its formal behavioral semantics. We employ behavioral semantics for a part of AADL [19] expressed in rewriting logic supported by the Maude language and tools [3]. Having the formal semantics makes the AADL models executable and enables the formal verification of functional requirements on the architecture, which is also employed in our previous work [7] [10] to generate traces between requirements and architecture. The verification uses the discrete event simulation, which introduces the notion of events, states, and state space. The architecture is executed and a state space is produced. A state describes the loci of data values within the architecture. Two states are connected by a transition encoded by a transition rule, while all states are captured by the state space. A counter example is an ordered set of states which are generated when the requirement is not satisfied in the verification. There is no transition rule applicable (i.e., term rewrite rules and equations in Maude) in the last state of the counter example. A state transition rule is fired if its left-hand side pattern matches the current state. The next state is formed based on the right-hand side of the transition rule. The main idea behind our approach is to make such changes in the architecture that make the application of some transition rules possible in the last state of the counter example. Changes can be applied iteratively until the requirement is eventually satisfied.

The paper is structured as follows. Section 2 gives the background. Section 3 introduces an industrial example used in the following sections. In Section 4, we give the approach overview, while we present the details in Section 5. Section 6 describes the related work, and Section 7 concludes the paper.

2. BACKGROUND

In this section we first give the AADL basics, the main language elements, and the graphical notation (Section 2.1). After Maude is summarized (Section 2.2), we present a brief introduction of the behavioral semantics of AADL in Maude (Section 2.3).

2.1 Preliminaries on AADL

AADL [1] is an industrial standard to describe performance-critical embedded real-time systems in avionics, automotive, medical, and robotics domains.

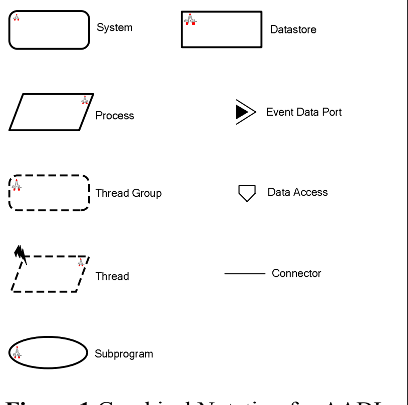

Figure 1 Graphical Notation for AADL

In AADL, a system is given as a hierarchy of hardware and software components. A component contains *interfaces* with input/output ports, *subcomponents* and their interaction, and other type-specific *properties*. At the top-level, there is *system* component, containing other system components as well as of hardware and software components. Hardware components *processor* scheduling and executing threads, *memory* storing code and data, *device* for sensors and actuators, and *bus* connecting processors, memory, and devices. Software components are *thread* representing a unit of concurrent execution, *thread group* used to create a hierarchy among threads, *process* for protected memory accessed by its thread subcomponents, *subprogram* representing a piece of source code, *data* for data types, *data store* storing data, and *data access* for access to the stored data.

There are three types of ports: *data*, *event* and *event-data*. Data ports are for data transmissions without queuing while event ports are for queued communications. *Connectors* between data ports are either immediate or delayed. Event data ports are for message transmission with queuing (see Figure 1 for the AADL notation).

An AADL model specifies how different components are integrated to form a complete system. The AADL standard [1] includes behavioral semantics for exchange and control of data, including *message passing*, *event passing*, *synchronized access to shared components*, *thread scheduling protocols*, *timing requirements*, and *remote procedure calls*. A *mode* abstraction is a configuration of components, connections, and property value associations. Modes represent alternative operational states of a system or component such as a thread or subprogram [1].

A thread is the main unit for modeling the concurrent execution. Its behavior can be given as a state machine. In AADL, thread behavior is typically described using AADL's *behavior annex*, which models programs as transition systems with local state variables. A thread state can be *initial*, *active*, *inactive*, or *completed*. In the annex, a variety of execution properties can be assigned to threads, including timing (e.g., worst case execution times), dispatch protocols (e.g., periodic, aperiodic, etc.), memory size, and processor binding [1].

2.2 Rewriting Logic and Maude

Maude is a language supporting both membership equational logic and rewriting logic [3]. Rewriting logic is about concurrent change based on state and concurrent computations. Membership equational logic is a sub logic of rewriting logic [3] where atomic sentences are equalities $t = t'$ and membership assertions in the form $t: s$, stating that a term t has a sort s.

A Maude module is in the form ($\sum$, E U A, R) where $\sum$ is a signature, E is a set of equations, and A is a set of equational axioms. R is a collection of labeled (conditional) rewrite rules denoting transitions in a state space. Rewrite rules are universally quantified by the variables appearing in the terms given as a signature. In object-oriented Maude, terms are given in the form of classes and objects.

Class $C \mid att_1 : s_1 , \ldots , att_n : s_n$.

$< O : C \mid att_1 : val_1 , \ldots , att_n : val_n >$

where a class C is with the attributes att_1 to att_n of the sorts s_1 to s_n and an object O of the class C with the values val_1 to val_n of the attributes att_1 to att_n. The Maude syntax has three statements: *equations* (the keywords ***eq*** and ***ceq***), *membership* (the keywords ***mb*** and ***cmb***) stating that a term has a sort, and rewrite rules (the keywords ***rl*** and ***crl***). The keywords ***var*** and ***vars*** are used to declare mathematical variables. A conditional rewrite rule is given in the following form.

crl [l] : { t } => { t' } **if** *cond* .

where l is a label for the rewrite rule and *cond* represents the condition. The rewrite rule specifies a transition from a match of t (left-hand) to a match of t' (right-hand).

2.3 Behavioral Semantics of AADL in Maude

The AADL standard is given in English and the behavioral semantics in the standard is ambiguous due to the use of natural language. It is possible to have multiple interpretations of the semantics. To avoid ambiguities and support formal verification in AADL models as part of our approach, we use the behavioral semantics of AADL models given in object-oriented real-time Maude by Ölveczky et al. [19]. Listing 1 gives the representation of an AADL component with an object-oriented style in Maude.

Listing 1 AADL Component Representation in Maude

```
class Component | features : Configuration, subcomponents : Configuration,
                  connections : ConnectionSet, properties : Properties,
                  modes : ModeTransitionSystem, inModes : ModeNameSet .
class System .  subclass System < Component .
```

The semantic definitions are given mainly as equations and rewriting rules in Maude. They can be considered as a formal semantics for AADL and an interpretation of what the informal and ambiguous descriptions in the AADL standard mean [19]. We only give the semantics of message passing along a level-up connection in AADL (Listing 2). The reader is referred to [19] for the details of the AADL behavioral semantics.

The message transmission between ports via a series of connections is implemented as equations [19]. The equation gives the message passing semantics from an out port (*P1*) of a subcomponent (*C1*) to an out port (*P*) of a component (*C*) containing the subcomponent along a connection *(C . C1 . P1 - -> C . P*). By applying the equation, the port *P* now has the value *transfer(ML)*, and the subcomponent's port (*P1*) buffer is empty.

Listing 2 Maude Equation for Message Passing between Ports

```
op transfer : MsgList -> MsgList [ctor] .
vars C C1 C2 : ComponentId .        vars P P1 P2 : PortId .
vars PORTS PORTS2 OTHER-COMPONENTS : Configuration .
vars ML ML' : MsgList .             var CONXS : ConnectionSet .
eq < C : Component | features : < P : OutPort | buffer : nil > PORTS,
   subcomponents :  < C1 : Component |
     features : < P1 : OutPort | buffer : transfer(ML) > PORTS2 >
       OTHER-COMPONENTS,  connections : (C1 . P1 --> P) ; CONXS >
 =  < C : Component | features : < P : OutPort | buffer : transfer(ML) > PORTS,
        subcomponents : < C1 : Component | features : < P1 : OutPort
            | buffer : nil> PORTS2 >  OTHER-COMPONENTS > .
```

3. EXAMPLE: REMOTE PATIENT MONITORING SYSTEM

We use the Remote Patient Monitoring (RPM) system to illustrate our approach. The system was developed by a Dutch company and had already been running when we started studying it. The goal is to enable monitoring patients' conditions such as blood pressure, heart rate, and temperature (see Table 1).

Table 1 Some of the Requirements for the RPM System

R1 The system shall measure temperature from a patient.
R2 The system shall measure blood pressure from a patient.
R3 The system shall measure blood pressure and temperature from a patient.
R4 The system shall store temperature measured by the sensor in the central storage.
R5 The system shall store patient blood pressure measured by the sensor in the central storage.
R6 The system shall store data in the central storage.
R7 The system shall warn the doctor when the temperature threshold is violated.
R8 The system shall generate an alarm if the temperature threshold is violated.
R9 The system shall show the doctor the temperature alarm at the doctors' computers.
R10 The system shall store all generated temperature alarms in a central storage.
R11 The system shall enable the doctor to set the temperature threshold for a patient.
R12 The system shall enable the doctor to retrieve all stored temperature measurements for a patient.
R13 The system shall allow retrieving the stored temperature alarms for a patient.
R14 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.
R15 The system shall store patient Central Venous Pressure (CV Pressure) measured by the sensor in the central storage.

The architecture is derived by reverse engineering the source code of the RPM system (Figure 2). Figure 2 shows only the most abstract components (*system* and *process* in AADL). *SD (Sensor Device)* contains sensors carried by the patient. The sensors perform measurements at a regular interval. *SD* sends the measurements to *HPC (Host Personal Computer)* through *SDC (Sensor Device Coordinator)*. *HPC* contains *SDM (Sensor Device Manager)*, *AS (Alarm Service)* and *WS (Web Server)* process subcomponents. *SDM* stores measurements and alarms in data stores (*Temp_alarms* and *Temp_Meas*) for temperature. *WS* serves as a web-interface for the doctors. *AS* forwards alarms to *CPC (Client Personal Computer)* for the doctors to monitor patients. The *AR (Alarm Receiver)* process in *CPC* receives alarms from *AS* and notifies the doctor. The *WC (Web Client)* process uses *WS* to retrieve measurements and alarms stored by *SDM*.

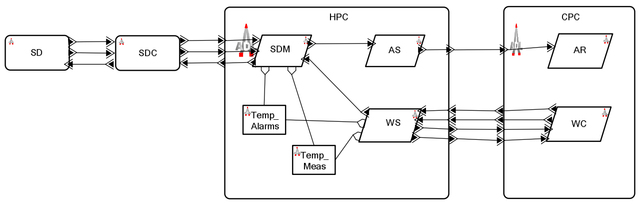

Figure 2 Overview of the RPM Architecture

The RPM architecture has also *thread* and *subprogram* components. The system computation is modeled as subprogram and thread behavior. Subprogram and thread behavior is given with a finite set of states and a set of state variables in the behavioral annex. Listing 3 gives the annex of the behavior of *sdmThr* in *SDM* for storing blood pressure measurements.

Listing 3 AADL Annex for Thread Behavior

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

sdmThr has the event data ports *sdm_blood_edp2* (Line 2) and *sdm_blood_strg* (Line 3) for blood measurements. The annex gives a transition system with state variables where each transition contains a guard (Lines 10-11) for the existence of events/data in the input ports (*sdm_blood_edp2*), and for the value of the data received (*inMessage*). The thread is activated upon receiving the input (Line 4). It has *s0* as the *initial* state and *bloodStored* as the *complete* state (Line 8). If the thread is in the *s0* state and receives the measured data in *sdm_blood_edp2*, then the received data is stored via *sdm_blood_strg* while the *bloodStored* state is reached.

4. OVERVIEW OF THE APPROACH

The approach is based on verifying functional requirements on software architecture. The output of the verification is a counter example if the input requirement to be verified is not satisfied by the architecture. The counter example is used together with a change classification to automatically propose architecture changes. As we already discussed, the main idea behind the approach is to make such changes in the architecture that make the application of some transition rules possible in the last state of the counter example. The counter example analysis has the following limitations and assumptions.

- *Analyzing the counter example in our approach is limited to the behavioral semantics of AADL given in* [19]. The semantics mostly deals with passing & storing data in a data flow, dispatching & executing threads, and switching modes, which are used to identify architecture changes. We need different architecture change types for different architecture description languages with different versions of semantics.
- *Architecture changes in our approach are limited to the possible missing parts of the architecture for mainly data flow and thread execution.* Designing architecture is a creative process. There are infinite designs that satisfy requirements for a given project. Therefore, changes over the architecture are infinite. We do not consider changes such as adding new systems, processes or threads which may cause infinite number of solutions for changed requirements.
- *It is assumed that, with changes in the architecture, it is possible to have a next state from the last state of the counter example.* It is possible that the last state might be the final state and there is no architecture change which makes the application of some transition rules possible in the last state. In this case, the architect should check all the states in the counter example to change the architecture. Even if the last state is not the final state, changing the architecture to enable a next state may not produce an architecture that satisfies the changed requirement. The architect may need iterations for verifying the requirement and changing the architecture.

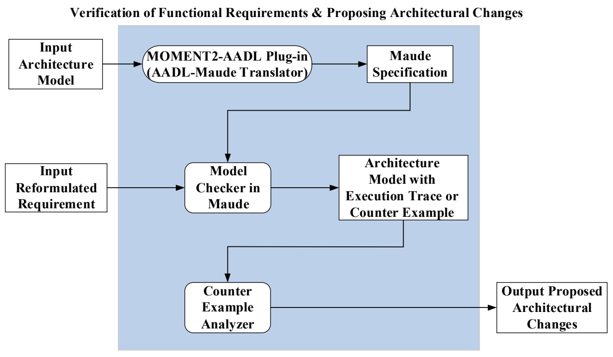
Verification of Functional Requirements & Proposing Architectural Changes
Input Architecture Model
MOMENT2-AADL Plug-in (AADL-Maude Translator)
Maude Specification
Input Reformulated Requirement
Model Checker in Maude
Architecture Model with Execution Trace or Counter Example
Counter Example Analyzer
Output Proposed Architectural Changes

Figure 3 Overview of the Tool Support

The approach contains three components (the rounded boxes in Figure 3). The Open-Source AADL Tool Environment (OSATE) – Topcased (http://www.topcased.org) includes an AADL front-end which provides plug-ins to support architecture analysis capabilities. We employ *MOMENT2-AADL* [19], an OSATE plug-in, to generate the Maude representation of AADL models for verification. It takes the architecture model in AADL as input and produces the Maude specification as an intermediate output (Figure 3). The architecture in AADL is transformed to a Maude term. The generated Maude specification contains AADL behavioral semantics given as rewrite rules and equations [19]. The verification is performed by the *model checker in Maude*. Maude is equipped with an explicit-state linear temporal logic (LTL) model checker analyzing whether all the behaviors from the initial state satisfy an LTL formula. Real-Time Maude extends the Maude model checker for real-time properties such as execution time. Our approach does not consider real-time properties. *Counter Example Analyzer* in Figure 3 parses the counter example and analyzes its last state to identify changes using the change classification and the AADL behavioral semantics.

5. RULE-BASED ARCHITECTURE EVOLUTION

This section details the approach. Section 5.1 explains the verification of functional requirements. In Section 5.2 we present the use of counter example to propose architecture changes.

5.1 Verification of Functional Requirements

The purpose is to check if functional requirements are correctly implemented in the architecture. In our previous work [7] [10], we already give a detailed description of the verification of functional requirements since we also use the verification output (i.e., execution trace and counter example) to generate traces between requirements and architecture. In this section, we revisit the verification employing the Maude model checker (Figure 4).

The verification is represented by the *Satisfies* and *ConformsTo* relations in Figure 4. *ConformsTo* implies that the state space captures the specified properties. We have the following artifacts:

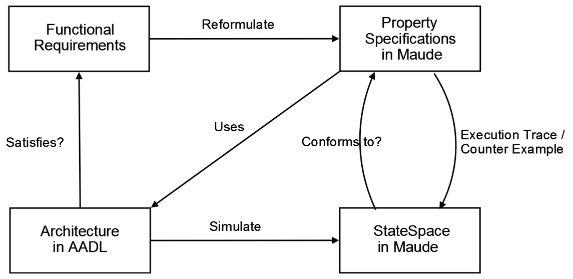

Figure 4 Verification of Functional Requirements

Property Specifications in Maude is the formal description of the required behavior of the architecture. The requirements are reformulated as properties in terms of the solution, which is the architecture (*reformulate* and *uses*). These properties are checked by the model checker. The property specification can use any logic such as *Linear Temporal Logic (LTL)*, *First-Order Logic (FOL)*, or *Computation-Tree Logic (CTL)*. Our approach uses the property specification feature of Maude, which is limited to LTL.

The presence of the AADL behavioral semantics in Maude makes the AADL models executable. The architecture is executed and a state space is produced (*simulate*). This execution simulates the behavior of the system on the architecture level to see how the system will work. Discrete event simulation, which introduces the notion of events, states, and state space, is used. A state describes the loci of data values within the architecture. Two states are connected by a transition and all states are captured by the state space. The verification result might be a counter example or an execution trace. An execution trace is the ordered set of states which are generated when the reformulated requirement is satisfied. A counter example is the ordered set of states generated when the reformulated requirement is not satisfied.

5.2 Proposing Architecture Changes

We identified a set of *architecture evolution rules* for AADL by using the AADL semantics in Maude. Each rule is a pair of a pattern matching the last state of the counter example and a set of proposed architectural changes. A pattern in the rule is a configuration of architecture elements such as data port, thread, and data. When the verification output is a counter example, our approach searches patterns, given in the rules, in the last state of the counter example, and proposes the corresponding architecture changes for matched patterns if any (*Counter Example Analyzer* in Figure 3). Table 2 gives some architecture evolution rules. We give the complete set of rules in the supplementary material [18].

Table 2 Some of the Architecture Evolution Rules

Patterns	Proposed Architecture Changes
Event/Data *M1* at the buffer of the (event) data-in-port of System *S1*	Add connection to the (event) data-in-port of Subsystem *SS1* of System *S1*
	Add (event) data-in-port to Subsystem *SS1* of System *S1* & Add connection to the added (event) data-in-port of Subsystem *SS1*
	Add connection to the (event) data-in-port of Process *P1* of System *S1*
	Add (event) data-in-port to Process *P1* of System *S1* & Add connection to the added (event) data-in-port of Process *P1*
	Change mode of System *S1*
Event/Data *M1* at the buffer of the (event) data-in-port of Process *P1*	Add connection to the (event) data-in-port of Thread *T1* of Process *P1*
	Add (event) data-in-port to Thread *T1* of Process *P1* & Add connection to the added (event) data-in-port of Thread *T1*
	Change the mode of Process *P1*
Event/Data *M1* at the buffer of the (event) data-in-port of Thread *T1*	Change the behavior of Thread *T1*
	Change the mode of Thread *T1*
Event/Data *M1* at the internalbuffer of the (event) data-in-port of Thread *T1* & Thread *T1* is in the *active* state	Change the mode of Thread *T1*
	Change the behavior of Thread *T1*

The first column represents the patterns. In the second column we give the architecture change alternatives to be proposed when the pattern matches the last state in the counter example. For instance, the architect may change either the behavior or mode of the thread if the last state in the counter example has an event/data at the buffer of the event data-in-port of the thread. We followed two steps to identify the rules in Table 2.

- *Identifying patterns of the architecture evolution rules*. By analyzing the right-hand side patterns of the state transition rules, we identified architecture configurations that need changes to fire further transition rules in the counter example. These configurations are with no more than three architecture elements (e.g., *data* at the *buffer* of a *thread*) and form the patterns in the architecture evolution rules. Table 3 gives the transition rule classification with those patterns.
- *Deriving architecture changes for the patterns.* We compared the left-hand side of the state transition rules with our patterns to determine change alternatives in the architecture evolution rules. For instance, after the thread dispatch, the next transition rule is the execution of a thread. If a thread is dispatched but not executed in the last state of the counter example (*Dispatching Thread T1* in Table 3), we can apply some changes to make satisfied the conditions of the state transition rule for thread execution (the last row of Table 2). To make the dispatched thread executed, either the thread behavior or thread mode should be changed. Table 4 lists the architecture change types.

Table 3 AADL State Transition Rules and Patterns in the Architecture Evolution Rules

State Transition Rules in AADL	Patterns in the Architecture Evolution Rules
Passing Message *M1*	Event/Data *M1* at the buffer of the (event) data-in-port of System *S1*
	Event/Data *M1* at the buffer of the (event) data-in-port of Process *P1*
	Event/Data *M1* at the buffer of the (event) data-in-port of Thread *T1*
	Event/Data *M1* at the buffer of the (event) data-out-port of Device *D1*
	Event/Data *M1* at the buffer of the (event) data-out-port of System *S1*
	Event/Data *M1* at the buffer of the (event) data-out-port of Process *P1*
Dispatching Thread *T1*	Event/Data *M1* at the *internalbuffer* of the (event) data-in-port of Thread *T1* & Thread *T1* is in *active* state
	Thread *T1* is in the *active* state
Executing Thread *T1*	Event/Data *M1* at the buffer of the (event) data-out-port of Thread *T1* & Thread *T1* is in the *completed* state
	Thread *T1* is in the *completed* state
Switching the Mode of Thread *T1*	Thread *T1* is in the *inactive* state
	Thread *T1* is in the *completed* state

Table 4 Change Types for AADL

Change Types
Add (event) data-in-port to System
Add (event) data-out-port to System
Add (event) data-in-port to Process
Add (event) data-out-port to Process
Add (event) data-in-port to Thread
Add (event) data-out-port to Thread
Add connection to (event) data-in-port
Add connection to (event) data-out-port
Change the mode of System
Change the mode of Process
Change the mode of Thread
Change the behavior of Thread

There are more change types for AADL such as adding new systems and threads. The number of solutions is infinite for designing architecture. We do not consider changes (e.g., adding new systems, processes or threads) which may lead to infinite number of solutions.

In the following, we use one of the state transition rules in Table 3 (i.e., *Passing Message M1 - Event/Data M1 at the buffer of the (event) data-in-port of Thread T1*) to explain how we identify the architecture evolution rules.

Listing 4 is one of the state transition rules of *Passing Message M1*. It is for transmission of a data along a level-down connection C.P --> C.C1.P1 from *P* data-in-port of *C* component to *P1* data-in-port of *C1* subcomponent of *C* component. As a result of applying the rule to a model where *C1* is a thread, *P1* data-in-port of *C1* thread has the data (Line 10) and the *P* data-in-port's buffer is empty (Line 9).

The possible transition is dispatching the thread when the data is at the buffer of (event) data-in-port of the thread. If the last state of the counter example contains the data at the data-in-port of a thread, an architecture change has to make possible the application of the state transition rule for the thread dispatch.

Listing 4 Maude Equation for Passing Message M1

```
op transfer : MsgList -> MsgList [ctor] .
vars  C  C1 : ComponentId .        vars P  P1 : PortId .
vars PORTS  PORTS2  OTHER-COMPONENTS : Configuration .
vars ML  ML' : MsgList .           var CONXS : ConnectionSet .
eq < C : Component |  features :
   < P : InPort | buffer : transfer(ML) > PORTS, subcomponents :
     < C1 : Component | features : < P1 : InPort | buffer : ML' > PORTS2 >
       OTHER-COMPONENTS,  connections : (P --> C1 . P1) ; CONXS >
    =  < C : Component | features : < P : InPort | buffer : nil > PORTS,
       subcomponents : < C1 : Component | features : < P1 : InPort | buffer : ML' ::
        transfer(ML) > PORTS2 >.  OTHER-COMPONENTS > .
```

There are two state transition rules for thread dispatch: *periodic* and *aperiodic thread dispatch*. Listing 5 gives the conditional rewrite rule [19] in Maude for the aperiodic thread dispatch.

Listing 5 Rewrite Rule for the Aperiodic Thread Dispatch

```
1   var O : ThreadId . var P : PortId. var PROGRAM : ThreadBehaviour .
2   var MTS : ModeTransitionSystem .   var TN : ThreadName .
3   var  IMPL : ImpleName . var  PORTS : Configuration . vars ML  ML' : MsgList .
4     crl [aperiodic-incoming-message] :
5       < O : Thread | properties : aperiodic-dispatch ; TP,  used : U,  modes : MTS,
6         deactivated : false, features : (< P : InEventDataThreadPort | buffer :
7           ML :: transfer(ML') >  PORTS), status : completed, behavior : PROGRAM,
8           threadType : TN, implementationType : IMPL >
9       =  < O : Thread | used : true, features :  dispatchInputPorts( < P :
10            InEventDataThreadPort | buffer : ML :: ML' > PORTS),  status : active >
11      if  someTransEnabled(transitions(TN, IMPL), PROGRAM, dispatchInputPorts
12          (< P : InEventDataThreadPort | buffer : ML :: ML' > PORTS)) .
```

The rule has the left-hand side pattern (Lines 5-8) with the condition part (Lines 11-12). To fire the rule for the *aperiodic-incoming-message*, the following conditions should hold: (1) thread is active (Line 6); (2) thread is in the *complete* state (Line 7); (3) some transitions in the behavioral annex of the thread are enabled (Lines 11-12); (4) there is an incoming data at the buffer of the data-in-port of the thread (Lines 12).

From the equation for *Passing Message M1*, we already know that there is a data at the buffer of the data-in-port of the thread. Therefore, architecture changes, which make the conditions 1, 2 and 3 hold, should be proposed. For these conditions, there are two architecture changes: *changing the mode of the thread* and *changing the behavior of the thread*. The thread may have different states and transitions in different modes. It might be activated and its state might be set to *complete* by changing the mode of the thread. The thread behavior is coded as states, state transitions, and thread activation in the behavioral annex. Changing the thread behavior (the behavioral annex) may activate the thread and set its state to *complete*. If no transition in the behavioral annex is enabled (the condition 3), either some of the transitions or the mode of the thread should be changed. There might be multiple applicable state transition rules which effect different parts of the architecture. The architect should analyze each proposed change to decide which one to implement.

Example: Proposing Architecture Changes

Suppose that there is a new requirement for the RPM system.

Requirement X *The system shall store patient Pulmonary Artery (PA) Pressure measured by the sensor in the central storage.*

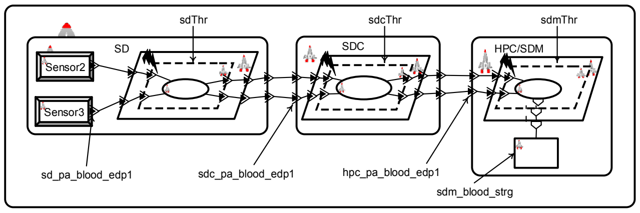

Figure 5 Changed Part of the RPM Architecture

We first decide to change the architecture. We added a new sensor (*Sensor 3*) and new event data ports (e.g., *sd_pa_blood_edp1*, *sdc_pa_blood_edp1*, *hpc_pa_blood_edp1*) to measure and transmit the patient PA pressure (see Figure 5). *Sensor 3* measures and transmits the patient PA pressure via the new event data ports and the threads (i.e., *sdThr*, *sdcThr* and *sdmThr*). The measured PA pressure is stored in the data store *sd_blood_strg*. The following is the LTL formula to verify the new requirement.

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

For the LTL formula, the Maude model checker returns a counter example. Figure 6 gives the last state of the counter example.

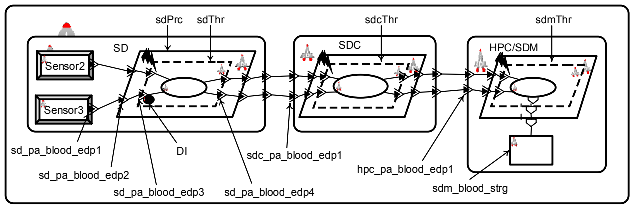

Figure 6 Last State of the Counter Example in the First Check

In Figure 6, the *DI* data instance is at the buffer of the *sd_pa_blood_edp3* data-in-port of the *sdThr* thread. Our approach automatically proposes two changes based on Table 2: *Change the behavior of sdThr* or *Change the mode of sdThr*.

We inspect the requirement, architecture and proposed changes. The *sdThr* thread has no mode. Therefore, we decide to change the thread behavior by introducing new states and state transitions in the annex. We add the following state transition with the new state *cvBloodPassed* in the behavioral annex of the *sdThr* thread:

idle -[sd_pa_blood_edp3?(inMessage)]-> cvBloodPassed { sd_pa_blood_edp4!(inMessage); };

The new state transition is the following: If the *sdThr* thread is in the *idle* state and receives the measured data at the *sd_pa_blood_edp3* event data port, then the received data is passed to the *sd_pa_blood_edp4* event data port. We re-execute the model checker over the changed architecture. The LTL formula is again false and another counter example is returned. After the first check, we have three more iterations that we do not illustrate here because the architecture changes are *changing the behavior of the thread* as well. We add new states and new state transitions in the behavioral annex of the threads *sdcThr* and *sdmThr* after the second and third iterations. In the fourth check, the LTL formula is satisfied with an execution trace (Figure 7).

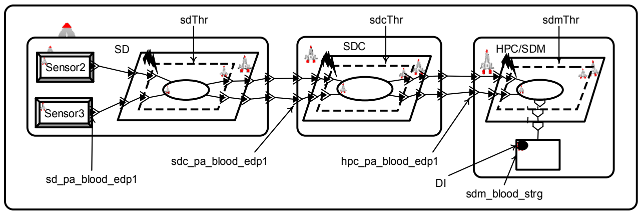

Figure 7 Last State of the Execution Trace

In the last state of the execution trace, the *DI* data instance is stored and the *bloodStored* state is reached. Therefore, the architecture satisfies the new requirement.

6. RELATED WORK

Feng and Maletic [4] address the impact of architecture changes within the same architecture, but do not take into account requirements changes. Tang et al. [17] introduce an approach to capture casual relations between architectural elements and decisions using probabilities. These relations enable architects to identify impacted elements in architecture based on probability theory. However, this approach does not propose any change. Han [13] introduces an approach for change propagation based on dependencies of software artifacts. The approach is applied to identify the impact of design changes in design and source code.

Slicing techniques are employed to understand dependencies using independent slices of the program [5]. Slicing is based on data and control flows and limit change propagation by identifying the scopes of changes. Architectural slicing [20] determines one slice of architecture for proposing changes. It answers the question of 'If a change is made to a component *c*, what other components might be affected by *c*?', while we address the question of 'If a change is made to a requirement *r*, which components might be affected by *r* and what are the proposed changes for impacted components?'.

In our previous work [11] [14], we proposed a change impact analysis approach which propagates requirements changes to other requirements by using requirements relations [6] [8] [9]. We also proposed another approach [12] to propagate requirements changes to architecture but it heavily relies on the types of requirements relations.

7. CONCLUSIONS

In this paper, we presented a rule-based approach that proposes changes for architecture given in AADL. The proposed changes are guidelines for the architect in the manual design activity. The approach is based on verification of functional requirements using Maude, a formal language based on rewriting logic. Our approach has some limitations and assumptions. The architecture change types may not be generalized for other ADLs. We do not consider changes such as adding new systems, processes or threads which may cause infinite number of solutions for changed requirements. Analyzing counter example is limited to the behavioral semantics of AADL in Maude. It is assumed there is a potential next state after the last state of the counter example. It is possible the last state might be the final state where no transition is fired further.

Acknowledgments

Financial support was provided by NWO (www.nwo.nl) and the National Research Fund, Luxembourg (FNR/P10/03).

8. REFERENCES

[1] http://www.sei.cmu.edu/reports/06tn011.pdf.

[2] L. C. Briand, Y. Labiche, and S. He. Automating Regression Test Selection based on UML Designs. *Information and Software Technology*, 51(1):16-30, 2009.

[3] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. All about Maude - A High-Performance Logical Framework. *LNCS*, 4350, 2007.

[4] T. Feng and J. I. Maletic. Applying Dynamic Change Impact Analysis in Component-based Architecture Design. *SNPD'06*, 43-48, 2006.

[5] K. B. Gallagher and J. R. Lyle. Using Program Slicing in Software Maintenance. *IEEE Transactions on Software Engineering*, 17(8):751-761, 1991.

[6] A. Goknil, I. Kurtev, and K. van den Berg. A Metamodeling Approach for Reasoning about Requirements. *ECMDA-FA'08*, 310-325, 2008.

[7] A. Goknil, I. Kurtev, and K. van den Berg. Tool Support for Generation and Validation of Traces between Requirements and Architecture. *ECMFA-TW'10*, 39-46, 2010.

[8] A. Goknil, I. Kurtev, K. van den Berg, and J.-W. Veldhuis. Semantics of Trace Relations in Requirements Models for Consistency Checking and Inferencing. *Software and System Modeling*, 10(1):31-54, 2011.

[9] A. Goknil, I. Kurtev, and J.V. Millo. A Metamodeling Approach for Reasoning on Multiple Requirements Models. *EDOC'13*, 159-166, 2013.

[10] A. Goknil, I. Kurtev, and K. van den Berg. Generation and Validation of Traces between Requirements and Architecture based on Formal Trace Semantics. *Journal of Systems and Software*, 88:112-137, 2014.

[11] A. Goknil, I. Kurtev, and K. van den Berg. Change Impact Analysis for Requirements: A Metamodeling Approach. *Information and Software Technology*, 56(8):950-972, 2014.

[12] A. Goknil, I. Kurtev, and K. van den Berg. A Rule-Based Change Impact Analysis Approach in Software Architecture for Requirements Changes. *arXiv preprint arXiv:1608.02757*, 2016.

[13] J. Han. Supporting Impact Analysis and Change Propagation in Software Engineering Environments. *STEP'97*, 172-182, 1997.

[14] D. ten Hove, A. Goknil, I. Kurtev, K. van den Berg, and K. de Goede. Change Impact Analysis for SysML Requirements Models based on Semantics of Trace Relations. *ECMDA-TW'09*, 17-28, 2009.

[15] T. Mens and T. D'Hondt. Automating Support for Software Evolution in UML. *Automated Software Engineering*, 7:39-59, 2000.

[16] T. Mens, J. Magee, and B. Rumpe. Evolving Software Architecture Descriptions of Critical Systems. *Computer*, 43:42-48, 2010.

[17] A. Tang, Y. Jin, J. Han, and A. Nicholson. Predicting Change Impact in Architecture Design with Bayesian Belief Networks. *WICSA'05*, 67-76, 2005.

[18] http://people.svv.lu/goknil/architecture/SupplementaryMaterialA.pdf

[19] P. C. Ölveczky, A. Boronat, and J. Mesequer. Formal Semantics and Analysis of Behavioral AADL Models in Real-Time Maude. *FMOODS/FORTE'10*, 47-62, 2010.

[20] J. Zhao, H. Yang, L. Xiang, and B. Xu. Change Impact Analysis to Support Architectural Evolution. *Journal of Software Maintenance: Research and Practice*, 14(5):317-333, 2002.