EAST-ADL Tailored Testing: From System Models to Executable Test Cases

Raluca Marinescu[1], Mehrdad Saadatmand[2], Alessio Bucaioni[1], Cristina Seceleanu[1], and Paul Pettersson[1]

[1] Mälardalen Real-Time Research Centre (MRTC), Mälardalen University, Västerås, Sweden
{raluca.marinescu, cristina.seceleanu, paul.pettersson}@mdh.se, abi11001@student.mdh.se
[2] Xdin AB, Stockholm, Sweden
mehrdad.saadatmand@xdin.com

Abstract. Architectural models, such as those described in the EAST-ADL language, represent convenient abstractions to reason about embedded software systems. To enjoy the fully-fledged advantages of reasoning, EAST-ADL models require a component-aware analysis framework that provide, ideally, both verification and model-based test-case generation capabilities. In this paper, we extend ViTAL, our recently developed tool-supported framework for model-checking EAST-ADL models in UPPAAL PORT, with automated model-based test-case generation for functional requirements criteria. To validate the actual system implementation and exercise the feasibility of the abstract test-cases, we also show how to generate Python test scripts, from the ViTAL generated abstract test-cases. The scripts define the concrete test-cases that are executable on the system implementation, within the Farkle testing environment. Tool interoperability between ViTAL and Farkle is ensured by implementing a corresponding interface, compliant with the Open Services for Lifecycle collaboration (OSLC) standard. We apply our methodology to validate the ABS function implementation of a Brake-by-Wire system prototype.

Keywords: EAST-ADL, model-based testing, test case generation, test case execution.

1 Introduction

The quality of a system's software architecture influences directly the performance and other extra-functional attributes of the implementation, so such descriptions need to be analyzed against functional requirements. For instance, in an automotive system, braking correctly is a stringent requirement that needs to be ensured to avoid fatal consequences. One way to meet this goal is to entwine the formal verification of architectural models, for instance by model-checking, with automatic generation of *abstract test-cases* (ATC) from the same model (also by model-checking), which can be eventually used to validate the system's implementation. In this paper we focus on EAST-ADL [2], an industry-adopted

architecture-description language for developing automotive embedded systems, with support for functional specifications.

Although there is a solid research know-how of generating model-based test cases from behavioral specification models [13, 14, 23], in principle these methods are not directly applicable to structural models that are centered on components and their connections. For such models, the abstract test-case generation must be carried out within a formal framework that preserves the component semantics, most desirably by using component-aware model-checking algorithms. Also, assuming the component behavior formalized using a semantics-preserving notation such as timed automata [1], the generated abstract test-case will contain internal state information not corresponding to the actual code. Hence, the abstract test-cases need to be transformed into scripts, representing *concrete test-cases* that can be executed on the system implementation.

All the mentioned issues have kindled our motivation to introduce a methodology for model-based testing, described in Section 4, tailored to EAST-ADL architectural models, with the ultimate goal of applying the technology to analyze the system implementation, starting from the structural high-level artifact. We adopt ViTAL (see Section 3) as our main model-based framework, since it integrates component-aware model-checking with EAST-ADL. The behavior of EAST-ADL function blocks is formalized in the UPPAAL PORT timed automata (TA). We first show how to generate abstract test-cases for functionality, from the TA model of the EAST-ADL system description. The functional requirement criterion is formalized as a reachability property in UPPAAL PORT, and the result is a trace tailored to function block execution. The main goal of this work is twofold: (i) to validate the actual code that implements the model functionality, and (ii) to exercise the feasibility of the generated abstract test-case, by checking if the corresponding executable test-case ends its run with a *pass* or *fail* result. To meet our ambition, we transform the states and transitions of the ATC into C/C++ code signals, by generating Python test-scripts in Farkle [6].

Concretely, our methodology consists of the following contributions:

- abstract test-case generation from TA behavioral models of EAST-ADL function blocks, in Section 5;
- transformation of TA generated traces into concrete test-cases as Python scripts, in Section 7;
- OSLC adaptor design and implementation for ViTAL-Farkle tool interoperability, in Section 6.

To check the applicability of our framework, we illustrate it on a simplified version of the Brake-by-Wire System prototype, which we describe in Section 2, for which we generate abstract test-cases for the basic ABS functionality, import them via the OSLC adaptor, and transform them into a Pyhton script that we run on the actual code, all presented in Section 8. Finally, we summarize our work and give an overview of the related work in Section 9.

2 Brake-by-Wire Case Study: Function and Structure

In this section, we introduce the Brake-by-Wire (BBW) system, which will be used through the paper as the running example. BBW is a braking system equipped with an ABS function, and without any mechanical connection between the brake pedal and the brake actuators applied to the wheels.

The functionality of the system is divided among sensors, actuators, and computational blocks. The sensor attached to the brake pedal provides a voltage value related to the pedal angle, which is transformed into a percentage of a maximum value of the brake force. Similarly, the sensor attached to each wheel provides the wheel speed in rpm. The input provided by the sensors is used to compute the requested braking force for the entire vehicle, and then a local braking force at each wheel. The ABS controls the wheel braking in order to prevent locking the wheel, based on the slip rate value, and then the wheel actuator applies the braking force provided by the ABS.

The ABS computes the slip value based on the equation:

$$s = (vehicleSpeed - wheelSpeed \times r)/vehicleSpeed,$$

where s is the slip rate, $vehicleSpeed$ is the estimated vehicle speed value, $wheelSpeed$ is the wheel speed sensor value, and r is the wheel radius.

The friction coefficient of the wheel has a nonlinear relationship with the slip rate: when s starts increasing, the friction coefficient also increases, and its value reaches the peak when s is around 0.2. After that, further increase in s reduces the friction coefficient. For this reason, if s is greater than 0.2 the brake actuator is released and no brake is applied, or else the requested brake torque is used. Our goal is to test whether the actual system implementation meets this functional requirement, starting from the BBW's high-level architectural model. In the remainder of the paper, we first overview the background briefly, after which we detail the methodology, and apply it on the BBW example.

3 Preliminaries

3.1 ViTAL

ViTAL [8], a Verification Tool for EAST-ADL Models using UPPAAL PORT, integrates architectural languages and verification techniques, to provide simulation, formal verification, and from now on, also test-case generation to EAST-ADL system models. This section overviews the tool shortly, whereas the test-case generation methodology is further detailed in Section 5.

ViTAL is an integrated environment based on Eclipse plug-ins, consisting of: (i) two standard editors plug-ins, one for the EAST-ADL system model and one for the TA description of the component behavior, (ii) an automated transformation from EAST-ADL models to UPPAAL PORT input model, and (iii) the UPPAAL PORT plug-in for model-checking EAST-ADL models enriched with TA semantics.

The EAST-ADL language describes the abstract functional system model composed of interconnected components, called *function prototypes* (f_ps). As depicted in fig. 3, each EAST-ADL f_p is defined by its interface and a TA behavior. The M2M transformation performs a semantic mapping between each EAST-ADL f_p and a corresponding TA model (e.g., mapping internal TA variables to EAST-ADL external ports). The result of the transformation is compliant to the input language of the UPPAAL PORT model-checker. Further, the model-checker can be used to simulate the system model and verify various requirements (e.g., functional, timing), specified in Timed Computational Tree Logic (TCTL).

Once the EAST-ADL model is validated, in this paper, UPPAAL PORT is used to generate abstract test suites for different coverage criteria.

EAST-ADL. EAST-ADL is an architecture description language dedicated to the development of automotive embedded systems [2]. The definition of an EAST-ADL system model is given at different levels of abstraction representing the stages of the engineering process.

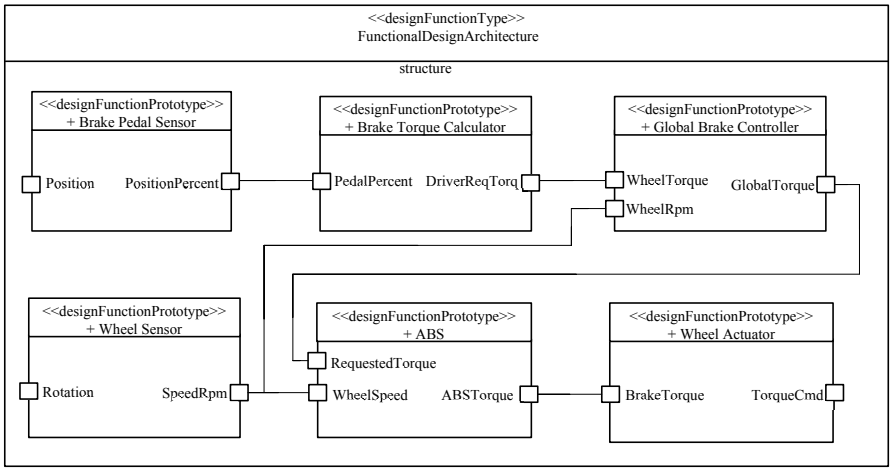

Fig. 1. The EAST-ADL model of the BBW system.

In EAST-ADL, the behavioral description relies on the definition of a set of f_ps, executed assuming the "read-execute-write" semantics. This enables analysis, behavioral composition, and makes the function execution independent of internal behavior. In EAST-ADL the functionality of each f_p is defined using different notations and tools, e.g., the UPPAAL PORT TA in out tool ViTAL.

The interface of an f_p is defined as a set of input and output data ports and triggering elements. Each component is idle until it is triggered, at which point it enters the executing state by reading the data from the input ports, then executes its behavior, and writes the data to the output ports.

Figure 1 presents the EAST-ADL model of the BBW, composed of six f_ps: **Brake Pedal Sensor**, **Brake Torque Calculator**, **Wheel Sensor**, **Global**

Brake Controller, **ABS**, and **Wheel Actuator**. The functionality of the sensors and actuators is straightforward, and acts as the system environment, by providing inputs and receiving output values. The **Brake Torque Calculator** computes the global braking force based on the pedal position received from the **Brake Pedal Sensor**. At the wheel, the **Global Brake Controller** calculates the local braking force by updating the global braking force based on the speed of the wheel. This value is passed to the **ABS**, which calculates the slip rate in order to prevent the locking of the wheel.

Timed Automata of UPPAAL PORT. UPPAAL PORT uses a special type of TA [11], where the syntax is defined by a tuple $B = (N, l_0, l_f, V_D, V_C, r_o, r_f, E, I)$ where N is a finite set of locations, l_0 is the initial location, l_f is the final location, V_D and V_C are a set of data and clock variables, respectively, r_0 and r_f are sets of initial and final reset clocks, E is a set of edges, and I is mapping each location l to its invariant $I(l)$. To describe an edge from location l to l', with guard g, action update e, and reset clocks r, we write $l \xrightarrow{g,e,r} l'$, for $(l, g, e, r, l') \in E$.

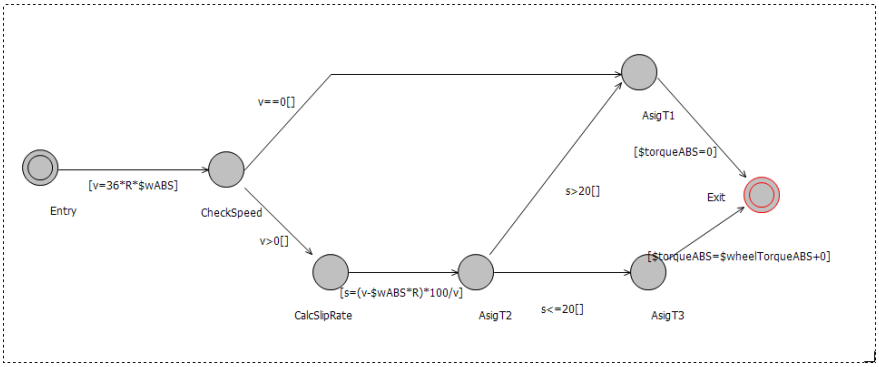

Fig. 2. The TA description of the ABS function block.

Fig. 2 presents the UPPAAL PORT TA behavior of the **ABS** f_p. The TA takes as input the wheel speed $wABS$, and the local braking force $wheelTorqueABS$. First, the TA calculates the vehicle speed v. If the car has no speed, no braking force is applied, even if the pedal is pressed. If the car is moving, the **ABS** calculates the slip rate s, and if $s > 20$, then again no brake is applied, to prevent the locking of the wheel. If $s < 20$, the $wheelTorqueABS$ received from the **Global Brake Controller** is sent to the **Wheel Actuator**.

Semantically, a TA state is a tuple (l, u, v), where l is a location, v is a data valuation, and u is a clock valuation. The transitions from one state to another can be internal transitions, read transitions, write transitions, or delay transitions that do not change the current state.

A TA's execution trace is a sequence of states and transitions, as shown below, for the TA shown in Fig. 2:

(ABS.idle, wABS=0, wheelTorqueABS=0, torqueABS=0, v=0, s=0) →
(ABS.Entry, wABS=10, wheelTorqueABS=0, torqueABS=0, v=0, s=0) $\xrightarrow{v:=36*R*wABS}$
(ABS.CheckSpeed, wABS=10, wheelTorqueABS=0, torqueABS=0, v=360, s=0) $\xrightarrow{v>0}$
(ABS.CalcSlipRate, wABS=10, wheelTorqueABS=0, torqueABS=0, v=360, s=0) $\xrightarrow{...}$...
(ABS.idle, wABS=10, wheelTorqueABS=0, torqueABS=0, v=360, s=97).

UPPAAL PORT Model-checker. UPPAAL PORT is an extension of the UPPAAL tool, which supports simulation and model-checking of component-based systems, without the usual flattening to the model of network TA [10]. This is complemented by the Partial Order Reduction Technique (PORT) that UPPAAL PORT uses, to improve the efficiency of analysis by exploring only a relevant subset of the state-space when model-checking. It uses local time semantics [3] to increase independence, being suited for analyzing "read-execute-write" component models.

3.2 Farkle Environment

Farkle is a test execution environment that enables testing an embedded system in its target environment. It is originally developed for testing embedded systems built using OSE Real-Time Operating System (RTOS) [6] (although it is possible to extend and adapt it to some other OSs such as Linux). OSE is a commercial and industrial RTOS developed specifically for fault-tolerant and distributed systems. OSE provides the concept of direct and asynchronous message passing for communication, so synchronization between tasks and its programming model is based on this concept. This allows tasks to run on different processors or cores, utilizing the same message-based communication model as on a single processor. This programming model has the advantage of not using task shared memory. In OSE, the runnable real-time entity equivalent to a task is called *process*, and the messages that are passed between processes are referred to as *signals* (thus, the terms *process* and *task* in this paper are considered equivalent). Each process can be either in the ready, or running, or waiting (e.g., waiting to receive a signal) state.

An OSE signal is defined as a data structure. Listing 1.1 shows the signal definition for the wheel speed in the BBW system:

```
1 #define WHEEL_SPEED_SIG 1026
2 typedef struct WheelSpeedSignal{
3   SIGSELECT sigNo;
4   float WheelSpeed;
5 } WheelSpeedSignal;
```

Listing 1.1. OSE signal example

Using the signal passing mechanisms of OSE, Farkle runs on a host Linux/Unix machine, and communicates with the target that is implemented using OSE. Hence, Farkle enables testing an embedded system by providing certain inputs to the target in the form of signals and receiving the result as signals containing output values. The test scripts that are used to send and receive signals, and also decide the verdict of test cases are implemented in Python.

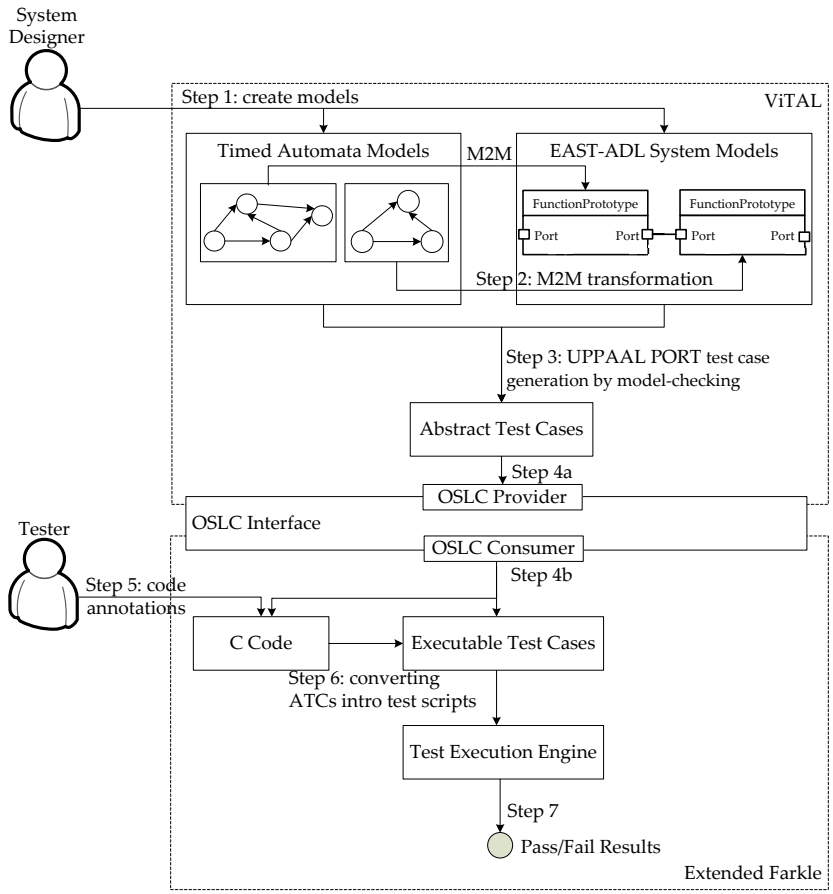

Fig. 3. From ViTAL to Farkle: The Methodology

4 From EAST-ADL Test-Case Generation to Code Validation: Our Methodology

In this section, we present an overview of our complete Model-based Testing (MBT) framework, which allows test-case generation starting from EAST-ADL models, and their eventual execution on the system under test (SUT). This framework follows the MBT methodology [24], and it is implemented by a tool chain consisting of ViTAL and Farkle, as depicted in Fig. 3. The MBT process proceeds as follows:

Step 1: We create/import the EAST-ADL system model, and add TA behavior to each EAST-ADL f_p.

Step 2: ViTAL performs an automatic transformation of the models defined in step 1 into the input language of the UPPAAL PORT. The result is the abstract formal model used for model-checking and test-case generation.

Step 3: We employ UPPAAL PORT to automatically generate abstract test cases. UPPAAL PORT takes as input the formal model of Step 2, together with the coverage criteria formalized as a TCTL property (or a set of properties).

Step 4: To ensure tool interoperability, we have implemented two OSLC adapters: an OSLC provider on top of ViTAL, which allows the export of abstract test cases (Step 4a), and an OSLC consumer on top of Farkle, which allows the import of abstract test cases by another tool (Step 4b).

Step 5: The code, representing the SUT, is annotated in order to enable tracking of state changes at runtime.

Step 6: The ATCs are converted to concrete test cases that are basically test scripts executable by Farkle.

Step 7: The concrete test cases are executed against the SUT, to obtain a *pass* or *fail* result.

Next, we detail our method for the offline generation of ATCs from EAST-ADL models, representing step 3 in fig. 3. We generate tests based on the information from the abstract formal model, and since our tests cannot be executed on the SUT, they are considered ATCs. Also, we call the method *offline generation* because the entire test suite is completely generated prior to the system execution.

5 Timed Automata Generation of Abstract Test Cases for Functional Requirements Criteria

There are two main problems when dealing with offline model-based test-case generation: state explosion and non-determinism. Since EAST-ADL models can be represented at different levels of abstraction, ViTAL can overcome the state explosion problem by generating test cases from a high-level system abstraction. However, non-determinism can still be a problem, especially due to the system's timing behavior. In this paper, we deal only with functional behavior, setting aside the system's timing, hence avoiding time-caused non-determinism.

Embedded systems interact closely with the environment they are deployed in, through different devices, like sensors and actuators. For this reason, most embedded systems are modeled as a SUT receiving input signals from the environment, which is abstracted by a separate model, and producing output signals, correspondingly. However, this is not the case in ViTAL. ViTAL generates the abstract test cases from EAST-ADL models that contain f_ps dedicated to sensors and actuators. Consequently, the formal model used in our test-case generation integrates the environment also. Nevertheless, such sensors can add potential non-determinism to the model, but we have restricted the sensors behavior to some possible inputs only.

The basis of our ATC generation method is the ability of UPPAAL PORT to generate witness traces for reachability properties. To achieve this, ViTAL takes as input: (i) the abstract formal model represented by the UPPAAL PORT compliant EAST-ADL model enriched with TA semantics, and (ii) a functional coverage criterion, formalized as a TCTL reachability property, or a collection of such properties. In TCTL, reachability is encoded as: $E <> p$, and ViTAL checks if a given state formula p may be eventually satisfied. For each reachability property that is satisfied by the model, the UPPAAL PORT model checker generates a witness trace, representing our ATC with respect to the property. All ATCs are collected into an abstract test-suite, and are provided to the Farkle tool through the OSLC interface.

The TCTL properties are given as a command to UPPAAL PORT, and guide the generation of witness traces from an infinite number of possible executions of the system. One of the limitations of this methodology is the fact that UPPAAL PORT cannot generate traces other than for reachability properties. This limits the testing horizon, as the expressiveness of TCTL for encoding other types of coverage criteria cannot be fully exploited.

The generated witness traces can be: *some trace*, the *shortest trace*, or the *fastest trace*, yet trace optimization is out of scope here. Each trace represents an abstract test case, and the collection of all necessary traces forms an abstract test suite for a particular coverage criteria. We instantiate our model-based test-case generation method in Section 8, by applying it to generate ATCs for the BBW functionality.

6 Tool Interoperability

In this section we describe our OSLC adapters, representing steps 4a and 4b in Fig. 3. The tool integration task has been performed by means of Eclipse Lyo[3], which aids the specification of Open Services for Lifecycle Collaboration[4] (OSLC) compliant tools. Our tools, ViTAL and Farkle, are integrated by implementing a provider on top of ViTAL that uploads the generated ATCs, and a consumer on top of Farkle that consumes the latter. In order to ensure standardization, OSLC establishes the exchangeable resources for each integration scenario; in our example, the exchanged artifact is the *abstract test case*.

```
1  @OslcResourceShape(title = "Quality Management Resource Shape",
       describes = Constants.TYPE_TEST_CASE)
2  @OslcNamespace(Constants.MDH_QM_NAMESPACE)
3  public final class TestCase extends AbstractResource {
4    private String  description, identifier;
5    @OslcPropertyDefinition(OslcConstants.DCTERMS_NAMESPACE + "description
         ")
6    @OslcTitle("Description")
7    @OslcValueType(ValueType.XMLLiteral)
8    public String getDescription() {return description;}
9    public void setDescription(final String description) {this.description
         = description;}
10 }
```

Listing 1.2. OSLC Implementation of the TestCase resource

Within Lyo, a consumer is composed of two EMF projects: one that implements the exchanged artifact as an OSLC resource, and another that implements the underlying logic. Listing 1.2 shows an excerpt of the Java implementation of the exchanged artifact, defined as a specialization of the OSLC *AbstractResource*(line 3). The `TestCase` resource definition comprises a set of local variables (line 6-7) along with the getters and setters (lines 8-18). The set of local variables is standard, and so are the Java annotations(lines 1-2, lines 8-10) also; these are responsible to ensure the mapping between the Java object and the OSLC resource properties.

[3] http://www.eclipse.org/lyo/
[4] http://open-services.net/specifications/

```
1  @OslcService(Constants.QUALITY_MANAGEMENT_DOMAIN)
2  @Path("testCases")
3  public class TestCaseService {
4    @GET
5    public TestCase[] getTestCases(@QueryParam("oslc.where") final String
         where) {
6      final List<TestCase> results = new ArrayList<TestCase>();
7      final TestCase[] testCases = Retriever.getTestCases();
8      for (TestCase testCase :testCases){results.add(testCase);}
9      return results.toArray(new TestCase[results.size()]);
10   }
11 }
```

Listing 1.3. OSLC ATC provider

Listing 1.3 presents an excerpt of the implementation logic for the ViTAL provider. Once the function `Retriever.getTestCases()` has set-up the OSLC resources with the needed ViTAL information, the resources can be accessed. The method annotation `@GET` (line 5) specifies the method to be invoked each time the provider receives a consumer HTTP GET.

```
1  final OslcRestClient oslcRestClient = new OslcRestClient(providers,
       queryBase, MEDIA_TYPE, timeout);
2  final TestCase[] enoviaSystemDefinitions = oslcRestClient.
       getOslcResources(TestCase[].class);
```

Listing 1.4. OSLC ATC consumer

Listing 1.4 shows the logic for the Farkle consumer: once the authentication to the given URI (line 1) has been performed, the consumer queries the resulting object (line 2) for retrieving all the test-case resources posted under the respective URI.

7 Creating Executable Test Cases

The final step in our testing chain is the creation of concrete test cases. The abstract test cases generated by ViTAL are provided as input to the Extended Farkle environment through the OSLC layer described previously, which enables the OSLC-based integration of the involved tools. Next, the following steps are taken for generating concrete test cases:

1. The ATCs from ViTAL are retrieved through the OSLC consumer, and are stored locally.
2. Different ATC state machines and their internal states are identified, and based on this information, C/C++ enumerations representing them are generated and stored in a C/C++ file along with a helper function named `set_state(StateMachine, State)`. The file is then included in the implementation code (Step 5 in Fig. 3).
3. The system's implementation code is then annotated using the `set_state()` helper function that maps state machines to code, by marking and adding a call to this function at places in the code where a state change occurs accordingly (Step 5 in Fig. 3).
4. Information about the transitions is retrieved from the abstract test case, and a Python test script executable in Farkle is generated. This means that the values of variables causing each transition are extracted from the abstract test case, and defined as OSE signals in the script, which are then sent to the system under test to invoke and simulate the transitions (Step 6).

5. The testing result is evaluated as *pass* if the (order of) changes in the states at the implementation level matches those at the system model level, and in the abstract test case; otherwise, the test execution ends in *fail*, which is an indication of problems/errors in the model or code, and of inconsistencies between them (Steps 7 in Fig. 3).

The process of generating concrete, executable test cases is semi-automatic only due to step 3 of the above list, which requires some manual intervention. However, if the implementation code is generated from system models while taking into account the information about the state machines of the TA model (for system verification), the mapping of abstract states to code states could then possibly be automated, and performed during code generation. This automation is particularly interesting for the scalability of the approach, considering that there can easily be a large number of states in the system implementation. We will nevertheless investigate this aspect in our future work.

8 Brake-by-Wire Revisited: Applying the Methodology

We illustrate and exercise the applicability of our approach on the BBW system, introduced in Section 2. The BBW EAST-ADL system model is depicted in Fig. 1, while the behavior of the ABS component is shown in Fig. 2.

We have modeled the BBW system in ViTAL, as presented in Section 3, and we have simulated and verified our formal model to ensure that it conforms to its specification. For more details on the formal analysis of EAST-ADL in ViTAL, we refer the reader to our recent work [8]. In this section, we will focus on the chain starting with the generation of abstract test cases for a particular functional requirement, from EAST-ADL, and ending by executing the obtained concrete test case on the implementation code, process that terminates with a "pass" or "fail" verdict w.r.t. the considered requirement.

We show the entire testing process for one functional requirement of the BBW system, pertaining to the function of the ABS component, which depends on the value of the slip rate. The goal is to verify that the brake actuator is always released and no brake is applied, if the slip rate s is greater than 0.2. The requirement is expressed in TCTL as follows:

$$A[\,](ABS.s > 0.2 \; imply \; WheelActuator.NoBrake)$$

This property is satisfied by the BBW model, as UPPAAL PORT returns a "passed" verdict at the end of model-checking BBW's TA behavior in ViTAL. Since the UPPAAL PORT model-checker integrated in ViTAL generates witness traces for reachability properties only, for the testing purpose, we need to specify the requirement as below:

$$E <> (ABS.s > 0.2 \; and \; WheelActuator.NoBrake)$$

The above reachability property expressed in TCTL is given as input to UPPAAL PORT, which automatically generates a trace, as a witness. The trace is an execution sequence of states and transitions of the TA formal behavioral model of the BBW, representing an abstract test case for our functional requirement.

```
State: BrakeTorqueSensor.idle, BrakeTorqueCalculator.idle,
    WheelSensor.idle, GlobalBrakeController.idle, ABS.Entry,
    WheelActuator.idle, BrakeTorqueSensor.Pos=0,
    BrakeTorqueCalculator.maxBr=2, BrakeTorqueCalculator.ReqTorque=0,
    BrakeTorqueCalculator.Pos=0, WheelSensor.Rpm=1,
    GlobalBrakeController.Rpm=1, GlobalBrakeController.ReqTorque=0,
    GlobalBrakeController.WheelTorque=0, GlobalBrakeController.W=10,
    ABS.WABS=10, ABS.WheelTorqueABS=0, ABS.TorqueABS=0, ABS.v=0,
    ABS.s=0, WheelActuator.Torque=0, WheelActuator.brake=0.
Transitions: ABS.Entry->ABS.CheckSpeed {v := 36 * R * WABS}
State: BrakeTorqueSensor.idle, BrakeTorqueCalculator.idle
    WheelSensor.idle GlobalBrakeController.idle, ABS.CheckSpeed,
    WheelActuator.idle, BrakeTorqueSensor.Pos=0, ...
Transitions: ABS.CheckSpeed->ABS.CalcSlipRate {v > 0 }
State: BrakeTorqueSensor.idle, BrakeTorqueCalculator.idle,
    WheelSensor.idle, GlobalBrakeController.idle,
    ABS.CalcSlipRate, WheelActuator.idle, BrakeTorqueSensor.Pos=0, ...
Transitions: ABS.CalcSlipRate->ABS.AsigT2{s := (v - WABS * R) * 100/v }
State: BrakeTorqueSensor.idle, BrakeTorqueCalculator.idle,
    WheelSensor.idle, GlobalBrakeController.idle, ABS.AsigT2,
    WheelActuator.idle, BrakeTorqueSensor.Pos=0,
    BrakeTorqueCalculator.maxBr=2, BrakeTorqueCalculator.ReqTorque=0,
    BrakeTorqueCalculator.Pos=0, WheelSensor.Rpm=1,
    GlobalBrakeController.Rpm=1, GlobalBrakeController.ReqTorque=0,
    GlobalBrakeController.WheelTorque=0, GlobalBrakeController.W=10,
    ABS.WABS=10, ABS.WheelTorqueABS=0, ABS.TorqueABS=0,
    ABS.v=360, ABS.s=97, WheelActuator.Torque=0, WheelActuator.brake=0.
```

Listing 1.5. Abstract Test Case

We do not run this abstract test-case against the formal model, instead we transform it into an executable test case, used to test the actual behavior of the system at runtime; this complements the system model verification, which is useful to validate the BBW model. Executable test cases are created by applying the steps of the method described in Section 7. The enumeration types below are generated automatically, and together with the fixed implementation of the `set_state()` helper function, they are "included" and used in the implementation code (as a header file):

```
enum StateMachines {BrakePedalSensor, BrakeTorqueCalculator,
     GlobalBrakeController, WheelActuator, ABS, WheelSensor};
enum States {Entry, CheckSpeed, CalcSlipRate, AsigT1, AsigT2, AsigT3,
     Exit, Brake, NoBrake, Reac, ...};
```

Next, we use `set_state()` to mark and annotate state changes in the code, starting from the states of the TA model of the ABS function block. This results in a mapping between abstract states and code, as shown in fig. 4.

Moreover, from the abstract test case, shown partially in Listing 1.5, the input variable values invoking transitions in the TA model are automatically identified, and a Python test script is generated. When executed by Farkle on the host system, the script sends the signals representing those input values, to the target. An excerpt of the generated script is shown in Listing 1.6.

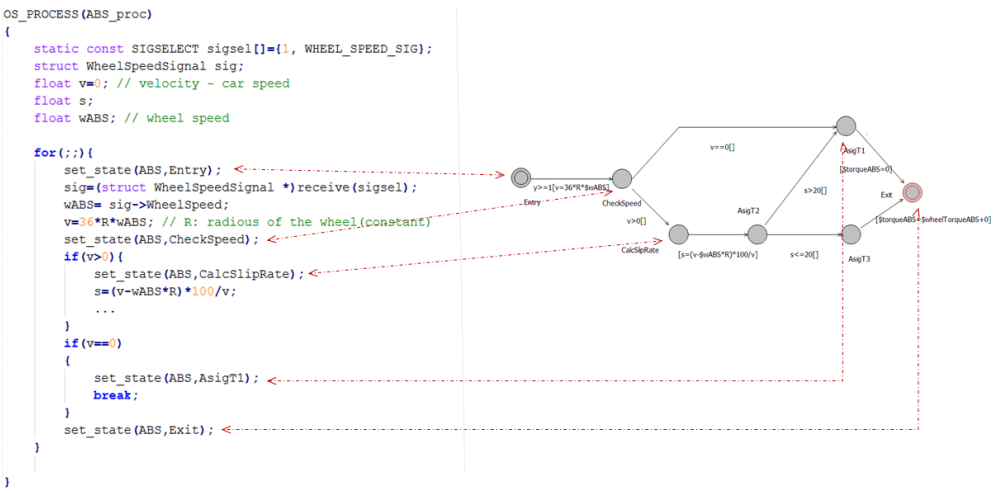

Fig. 4. Mapping of states to the code (ABS function)

```
1  import sys
2  import os
3  from signals import *
4  sys.path.append("/media/E21408B714089129/runtime-EclipseApplication/pp/"
       )
5  class TestABS:
6    def TestCase_WheelSpeed1(self):
7      GW = 'tcp://10.0.0.4:21768'
8      NAME = 'ABS_proc'
9      self.proc = ogre.Process(GW, NAME)
10     p = BBWsignals.WHEEL_SPEED_SIG()
11     p.WheelSpeed=10
12     self.proc.send(p)
13 ...
```

Listing 1.6. Generated Python script sending signal to the ABS process on the target

On the target, the OSE process to which the signal is sent starts running, including the execution of `set_state()` statements that have been added to the code. The function `set_state()` also keeps track and logs state changes of the TA model. If one observes any discrepancy between the order of the logged states (as the result of running the code) and the order produced in the abstract test case (originated from the model), running the Python scripts ends with a *fail* result as the testing verdict. Otherwise, checking state changes and sending signals to invoke transitions continues until all the states and transitions from the abstract test-case are covered, resulting in a *pass* verdict. In our concrete case, the ABS functional requirement passes when tested on the code.

9 Conclusions and Related Work

9.1 Conclusions

In this paper, we have presented a methodology and tool support for testing system implementations, starting from EAST-ADL architectural models. The framework assumes the EAST-ADL artifact as the input model, as well as the timed automata behavioral models of the EAST-ADL function blocks. The first step consists in generating abstract test-cases for functional requirements specified in TCTL as reachability properties, by model-checking the TA enriched EAST-ADL component behavior in UPPAAL PORT. Next, the resulting abstract test-cases are semi-automatically transformed into Python scripts representing the concrete test-cases that are finally run on the actual code. Our work is an attempt of checking the feasibility of test-case generation from EAST-ADL models, which is an adopted structural language in companies such as Volvo Technology. The method has shown encouraging results when applied on a simplified Brake-by-Wire prototype implementation.

As future work, we plan to investigate abstract test-case generation for timing and architectural properties of EAST-ADL artifacts, down to the fully automatic transformation into executable scripts, which could be run on the system under test, to validate the implementation.

9.2 Related Work

Model-based testing by model-checking is a technique introduced almost fifteen years ago [7] as an efficient way of using a model-checker to interpret traces as test cases. More details and references on testing with model-checkers can be found in Fraser et al.'s work [9]. Some approaches to testing with model-checkers are applied on real-time reactive systems. Hessel et al. have proposed test case generation using the UPPAAL model-checker for real-time systems [12, 18] using timed automata specifications. The main difference in our work is that we provide an approach tailored to an architectural description language, and we offer an end-to-end tool chain with support for test case generation and execution.

Over the last few years, researchers in testing communities have been investigating how design components and architecture description languages (ADLs) can be used for testing purposes [15]. This effort concretized in testing techniques for ADLs, giving rise to many different approaches [16, 4, 21]. In comparison, in our work we define the semantics of the architecture description language in a formal notation, that is UPPAAL PORT timed automata, and provide functional test goals to be considered by UPPAAL PORT model checker.

As in our approach, some of the related works targeting test case generation explicitly distinguish between abstract and concrete test cases [20, 19, 17]. For instance, Peleska [19] has proposed the RT-Tester tool-suite along with the corresponding methodology, and discusses the two types of test cases, abstract and executable. The major goal in RT-Tester, however, is to execute test cases against the models of the system. In our work, we have introduced an approach that generates concrete test cases that are actually executable against the running system, and in its target environment. Moreover, it is worth noting that there are different static analysis methods that can be applied to ensure that expected properties in a system hold, thus increasing confidence in its correctness. However, despite the application of such methods, there are still situations/systems where the results of such analysis may be invalidated at runtime due

to different factors [22, 5]. In this work, we tackled this issue by complementing formal verification of the system at model level with the verification of its behavior at runtime. We achieved this by introducing a methodology that helps with bridging the (semantic) gap between abstract test cases generated from formal models and concrete ones, and by a tool-chain implementing the methodology. The use of OSLC as the integration mechanism in our tool-chain provides the advantage of not restricting the method to Farkle for the concrete-test-case generation; other tools can replace Farkle as long as they stick to and implement the OSLC standard, via a corresponding consumer.

Acknowledgment: The research leading to these results has received funding from the ARTEMIS Joint Undertaking under grant agreement number 269335 and from VINNOVA, the Swedish Governmental Agency for Innovation Systems, within the MBAT project, and also partially from the Swedish Knowledge Foundation (KKS) through the ITS-EASY industrial research school.

References

1. Rajeev Alur and David L Dill. A theory of timed automata. *Theoretical computer science*, 126(2):183–235, 1994.
2. The ATESST2 ATESST2 Consortium. EAST-ADL Profile Specification, 2.1 RC3 (Release Candidate). pages 10–75. www.atesst.org, 2010.
3. Johan Bengtsson, Bengt Jonsson, Johan Lilius, and Wang Yi. Partial order reductions for timed systems. In *CONCUR'98 Concurrency Theory*, pages 485–500. Springer, 1998.
4. Antonia Bertolino and Paola Inverardi. Architecture-based software testing. In *Joint proceedings of the second international software architecture workshop (ISAW-2) and international workshop on multiple perspectives in software development (Viewpoints' 96) on SIGSOFT'96 workshops*, pages 62–64. ACM, 1996.
5. S.E. Chodrow, F. Jahanian, and M. Donner. Run-time monitoring of real-time systems. In *Real-Time Systems Symposium, 1991. Proceedings., Twelfth*, pages 74–83, 1991.
6. Enea. The Architectural Advantages of Enea OSE in Telecom Applications. `http://www.enea.com/software/solutions/rtos/`, Last Accessed: May 2013.
7. André Engels, Loe Feijs, and Sjouke Mauw. Test generation for intelligent networks using model checking. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 384–398. Springer, 1997.
8. Eduard Paul Enoiu, Raluca Marinescu, Cristina Seceleanu, and Paul Pettersson. Vital: A verification tool for east-adl models using uppaal port. In *Engineering of Complex Computer Systems (ICECCS), 2012 17th International Conference on*, pages 328–337. IEEE, 2012.
9. Gordon Fraser, Franz Wotawa, and Paul E Ammann. Testing with Model Checkers: a Survey. In *Journal on Software Testing, Verification and Reliability*, volume 19, pages 215–261. Wiley Online Library, 2009.
10. John Håkansson, Jan Carlson, Aurelien Monot, Paul Pettersson, and Davor Slutej. Component-based design and analysis of embedded systems with uppaal port. In *Automated Technology for Verification and Analysis*, pages 252–257. Springer, 2008.
11. John Håkansson and Paul Pettersson. Partial order reduction for verification of real-time components. In *Formal Modeling and Analysis of Timed Systems*, pages 211–226. Springer, 2007.

12. Anders Hessel, Kim Larsen, Brian Nielsen, Paul Pettersson, and Arne Skou. Time-Optimal Real-Time Test Case Generation Using UPPAAL. In *Lecture Notes in Computer Science, Formal Approaches to Software Testing*, pages 114–130. Springer Berlin Heidelberg, 2004.
13. Anders Hessel and Paul Pettersson. Cover-a real-time test case generation tool. In *19th IFIP International Conference on Testing of Communicating Systems and 7th International Workshop on Formal Approaches to Testing of Software*, 2007.
14. Tretmans Jan. Model based testing with labelled transition systems. In *Formal methods and testing*, pages 1–38. Springer, 2008.
15. Henry Muccini. What makes software architecture-based testing distinguishable. In *Software Architecture, 2007. WICSA'07. The Working IEEE/IFIP Conference on*, pages 29–29. IEEE, 2007.
16. Henry Muccini, P Inverardi, and A Bertolino. Using software architecture for code testing. *Software Engineering, IEEE Transactions on*, 30(3):160–171, 2004.
17. C. Nebut, F. Fleurey, Y. Le-Traon, and J.-M. Jezequel. Automatic test generation: a use case driven approach. *Software Engineering, IEEE Transactions on*, 32(3):140–155, 2006.
18. Brian Nielsen and Arne Skou. Automated Test Generation from Timed Automata. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 343–357. Springer, 2001.
19. Jan Peleska. Industrial-strength model-based testing - state of the art and current challenges. In *Proceedings of the Eighth Workshop on Model-Based Testing*, pages 3–28, March 2013.
20. Wolfgang Prenninger, Mohammad El-Ramly, and Marc Horstmann. Chapter 15: Case studies. In Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner, editors, *Model-Based Testing of Reactive Systems: Advanced Lectures (Lecture Notes in Computer Science)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.
21. Hassan Reza and Suhas Lande. Model based testing using software architecture. In *Information Technology: New Generations (ITNG), 2010 Seventh International Conference on*, pages 188–193. IEEE, 2010.
22. Mehrdad Saadatmand, Antonio Cicchetti, and Mikael Sjödin. Design of adaptive security mechanisms for real-time embedded systems. In *Proceedings of the 4th international conference on Engineering Secure Software and Systems (ESSoS)*, pages 121–134, Berlin, Heidelberg, 2012. Springer-Verlag.
23. Manoranjan Satpathy, Michael Leuschel, and Michael Butler. Protest: An automatic test environment for b specifications. *Electronic Notes in Theoretical Computer Science*, 111:113–136, 2005.
24. Mark Utting, Alexander Pretschner, and Bruno Legeard. A taxonomy of model-based testing approaches. *Software Testing, Verification and Reliability*, 22(5):297–312, 2012.