A Model-Based Testing Framework for Automotive Embedded Systems

Raluca Marinescu*, Mehrdad Saadatmand*†, Alessio Bucaioni*, Cristina Seceleanu*, Paul Pettersson*
*Mälardalen University, Västerås, Sweden, <firstname.lastname>@mdh.se
†Alten AB, Sweden, <firstname.lastname>@alten.se

***Abstract*—Architectural models, such as those described in the EAST-ADL language, represent convenient abstractions to reason about automotive embedded software systems. To enjoy the fully-fledged advantages of reasoning, EAST-ADL models could benefit from a component-aware analysis framework that provides, ideally, both verification and model-based test-case generation capabilities. While different verification techniques have been developed for architectural models, only a few target EAST-ADL. In this paper, we present a methodology for code validation, starting from EAST-ADL artifacts. The methodology relies on: (i) automated model-based test-case generation for functional requirements criteria based on the EAST-ADL model extended with timed automata semantics, and (ii) validation of system implementation by generating Python test scripts based on the abstract test-cases. The scripts represent concrete test-cases that are executable on the system implementation. We apply our methodology to analyze the ABS function implementation of the Brake-by-Wire system prototype.**

***Keywords*-EAST-ADL, model-based testing, UPPAAL PORT, test-case generation, test-case conversion, Python scripts;**

I. INTRODUCTION

The complexity of embedded systems in the automotive domain is continuously increasing, in part due to the replacement of mechanical or hydraulic technologies with electrical/electronic systems that implement complex functions, such as cruise control, automatic braking, etc. Consequently, the system development needs to conform with stringent safety standards. To align with such standards, the development process must provide evidence that requirements are satisfied at each level of system abstraction, from architectural and behavioral models to implementation (e.g., as required by ISO 26262). In this context, there is a real need for advanced and formal methodologies for verification and testing of automotive systems, which can provide industrially relevant artifacts.

Although there is a solid research know-how of generating test-cases from behavioral specification models [1], [2], [3], in principle these methods are not directly applicable to architectural models where the system behavior is defined in terms of function blocks with no formal support to specify and analyze their internal behaviors. The latter is usually described in semi-formal languages such as UML, Simulink etc. The operation of each function block can, for instance, be formalized using notations such as timed automata (TA) [4], which could serve as the semantic representation of the block behavior [5]. Assuming this, the abstract test-case generation can then be carried out using component-aware model-checking algorithms [6]. The resulting abstract test-cases contain internal state information not corresponding to the actual code. Hence, the abstract test-cases need to be transformed into executable scripts that would then be used as concrete test-cases to analyze the system implementation. This need has kindled our motivation to introduce a methodology (see Section IV) for model-based testing against functional requirements of embedded systems, starting from the EAST-ADL architectural models, an emerging standard for automotive industry, already used by Volvo Group Trucks Technologies, Sweden. As part of the methodology, we show how to generate executable test-cases for the system implementation automatically, starting from abstract tests generated by model-checking EAST-ADL high-level artifacts extended with TA behavior. The main goal of this paper is to check the feasibility of the EAST-ADL+TA generated abstract test-cases by actually running the corresponding executable test-cases on the implemented code, in an attempt to obtain a *pass* or *fail* verdict. If the endeavor succeeds, the testing effort of the code could then be reduced by the automatic provision of valid test-cases.

Our contribution assumes three actors in the system development process: the System Designer, the Developer, and the Tester. The methodology presented in this paper describes only two main phases: model-based system implementation and testing. Concretely, we adopt ViTAL [7] as our main modeling and analysis framework, as it integrates component-aware model-checking with EAST-ADL models. In Section V we define an executable semantics of the UPPAAL PORT TA that facilitates code implementation (see Section V) in a semantics-preserving manner. Next, we we show how to generate abstract test-cases for functional requirements, from the TA model of the EAST-ADL system description (see Section VI-A). The functional requirement criterion is formalized as a reachability property in UPPAAL PORT [8], and the result is an abstract test-case defined by an execution trace of the TA models corresponding to the function blocks. In Section VI-B we transform the states and transitions of the test-case into C/C++ code signals, by generating Python test-scripts in Farkle test execution environment [9]. These test-scripts are run against the system under test (SUT) to obtain a pass or fail verdict w.r.t. the testing goal. Our framework adapts already existing model-checking based testing techniques, to obtain a novel inte-

grated approach for testing automotive embedded systems, starting from high-level system artifacts modeled in EAST-ADL, and their requirements specification. To check the applicability of our framework, we illustrate it on a simplified version of Volvo's Brake-by-Wire System prototype, which we describe in Section III. We compare to related work in Section VIII, and summarize our work, together with outlining ideas for future work in Section IX.

II. PRELIMINARIES

In this section we give a brief overview of: (i) ViTAL, used for generating abstract test-cases, and (ii) Farkle, used for transforming the latter into test-scripts.

A. ViTAL

ViTAL, a Verification Tool for EAST-ADL Models using UPPAAL PORT, integrates architectural languages and verification techniques to provide simulation and model-checking of timing and functional behavioral requirements. To achieve this, the tool provides functional and timing behavior for EAST-ADL functional blocks using timed automata semantics, and performs an automatic model transformation to the input language of UPPAAL PORT, which enables the model-checker to handle EAST-ADL models for formal verification.

The tool is an integrated environment based on Eclipse plug-ins and contains an editor for the EAST-ADL model (i.e., Papyrus), an editor for timed automata description of the behavior of EAST-ADL blocks, and a semantic mapping between each EAST-ADL block 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, able to simulate the system model and verify various requirements (e.g., functional, timing), specified in Timed Computation Tree Logic (TCTL). ViTAL integrates the following artifacts:

EAST-ADL: EAST-ADL is an architecture description language dedicated to the development of automotive embedded systems [10]. The definition of an EAST-ADL system model is given at five levels of abstraction representing different stages of the engineering process with complete traceability between them. At each level, the behavioral description relies on the definition of a set of function prototypes (refereed as blocks) f_ps, executed assuming the "read-execute-write" semantics. A block starts executing by reading data, which are constantly replaced by fresh data arriving on ports, performs some calculation and finally outputs data on the output ports.This enables analysis, behavioral composition, and makes the function execution independent of its internal behavior. The functionality of each f_p is defined using different notations and tools, e.g., Simulink or UPPAAL PORT TA in ViTAL.

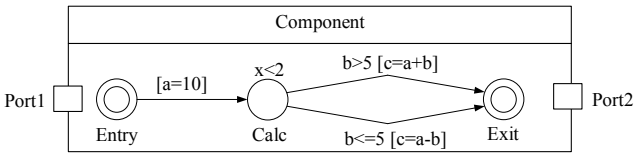

Figure 1: UPPAAL PORT TA Component.

UPPAAL PORT TA Component: As depicted in Figure 1, an UPPAAL PORT [6] component is defined by its interface and its timed behavior. The interface consists of a set of input data ports, a set of output data ports, and a set of trigger ports that define the control flow. The timed behavior is modeled as a tuple:

$$B = (N, l_0, l_f, V_D, V_C, r_0, r_f, Ed, I) \quad (1)$$

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 sets of data and clock variables, respectively, r_0 and r_f are sets of initial and final clock resets, and Ed is a set of edges. The function $I : N \cup \{l_0, l_f\} \rightarrow B(V_C)$, with $B(V_C)$ denoting the set of conjunctive formulas of clock constraints of the form $x_i \sim m$, or $x_i - x_j \sim n$, $x_i, x_j \in V_C$, $\sim \in \{\leq, <, =, \geq, >\}$, $m, n \in N$, assigns each location $l \in N \cup \{l_0, l_f\}$ to its invariant $I(l)$. To describe an edge from location l to l', with guard g, update action e, and clock resets r, we write $l \xrightarrow{g,e,r} l'$, for $(l, g, e, r, l') \in Ed$.

The timed behavior associated to the UPPAAL PORT TA component depicted in Figure 1 has three locations $Entry$, $Calc$, and $Exit$, three data variables a, b, and c, and a clock variable x. When the execution of the TA is triggered, it enters the $Entry$ location, updates variable a to 10 along the first edge to location $Calc$, where it remains as long as the invariant $x < 2$ holds. The update of variable c is determined based on the evaluation to "TRUE" of guards $b > 5$ and $b <= 5$, placed on the edges to location $Exit$.

The semantics of an UPPAAL PORT TA component are defined as a state: (l, u, v), where l is a location, v is a data valuation, and u is a clock valuation. UPPAAL PORT [6] allows the following transitions from one state to another:

- internal transitions: $(l, u, v) \xrightarrow{\tau} (l', u', v')$, along an edge $l \xrightarrow{g,e,r} l'$,
- delay transitions: $(l, u, v) \xrightarrow{\delta} (l, u, v + \delta)$ where $\delta \in \mathbb{R}_{\geq 0}$,
- read transitions: $(idle, u, v) \xrightarrow{read} (l_o, input(u), [r_0 := 0]u)$ if $triggered(v)$,
- write transitions, $(l_f, u, v) \xrightarrow{write} (idle, output(u), [r_f := 0]u)$.

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 of the TA network [8]. This is comple-

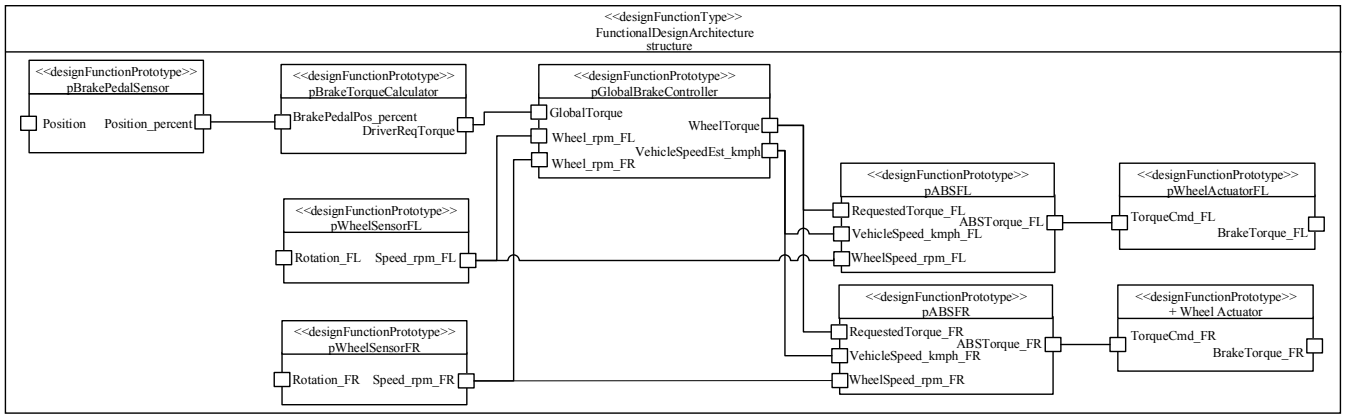

Figure 2: The EAST-ADL model of the BBW system.

mented by the **P**artial **O**rder **R**eduction **T**echnique (PORT) that improves the efficiency of analysis by exploring only a relevant subset of the state-space when model-checking. The tool also uses local time semantics [11] to increase independence, being suited for the analysis of "read-execute-write" component models.

B. Farkle

Farkle is a test execution environment that enables testing an embedded system in its target platform. It uses LINX as the Inter-Process Communication (IPC) protocol to provide direct and asynchronous message passing between tasks. This allows tasks to run on different processors or cores, while utilizing the same message-based communication model as on a single processor, but without using shared memory. The messages that are passed between processes (i.e., tasks) are referred to as *signals*. An example signal definition is shown in Figure 3.

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

Figure 3: Signal example

Using the signal passing mechanisms of LINX, Farkle runs on a host machine and communicates with the target. 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 a test-case are implemented in Python. Moreover, in order for the signal passing mechanism to work between the host and target, the host needs to also have information about signal structures. For this purpose, Farkle also generates signal definitions in Python from the signal definitions of the application source code, which is then imported and used in the Python test-script.

III. Brake-by-Wire Case Study: Functionality and Structure

Through the paper we use the Brake-by-Wire (BBW) system as a running example. Figure 2 shows the EAST-ADL model of the BBW system at the analysis level. To simplify, we have modeled only two out of the four wheels of the system. The **Brake Pedal Sensor** reads the position of the pedal and the **Brake Torque Calculator** computes the desired braking force based on this value. Similarly, the **Wheel Sensor** reads the rotation speed of the wheel. The **Global Brake Controller** calculates the actual braking force by updating the desired braking force based on the speed of the wheels, and provides it to the **ABS** block, which calculates the slip rate to decide if the braking force can be applied without locking the wheel. Finally, the braking force is applied by the **Wheel Actuator**.

The **ABS** f_p calculates the slip rate s based on the equation:

$$s = (v - w \times R)/v, \tag{2}$$

where w is the rotation speed of the wheel, v is the speed of the car, and R is the radius of the wheel. 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.

IV. From EAST-ADL to Code Validation: Methodology Overview

This section overviews our model-based testing (MBT) framework, which allows test-case generation starting from EAST-ADL models, down to their execution on the system under test (SUT). This framework follows the MBT

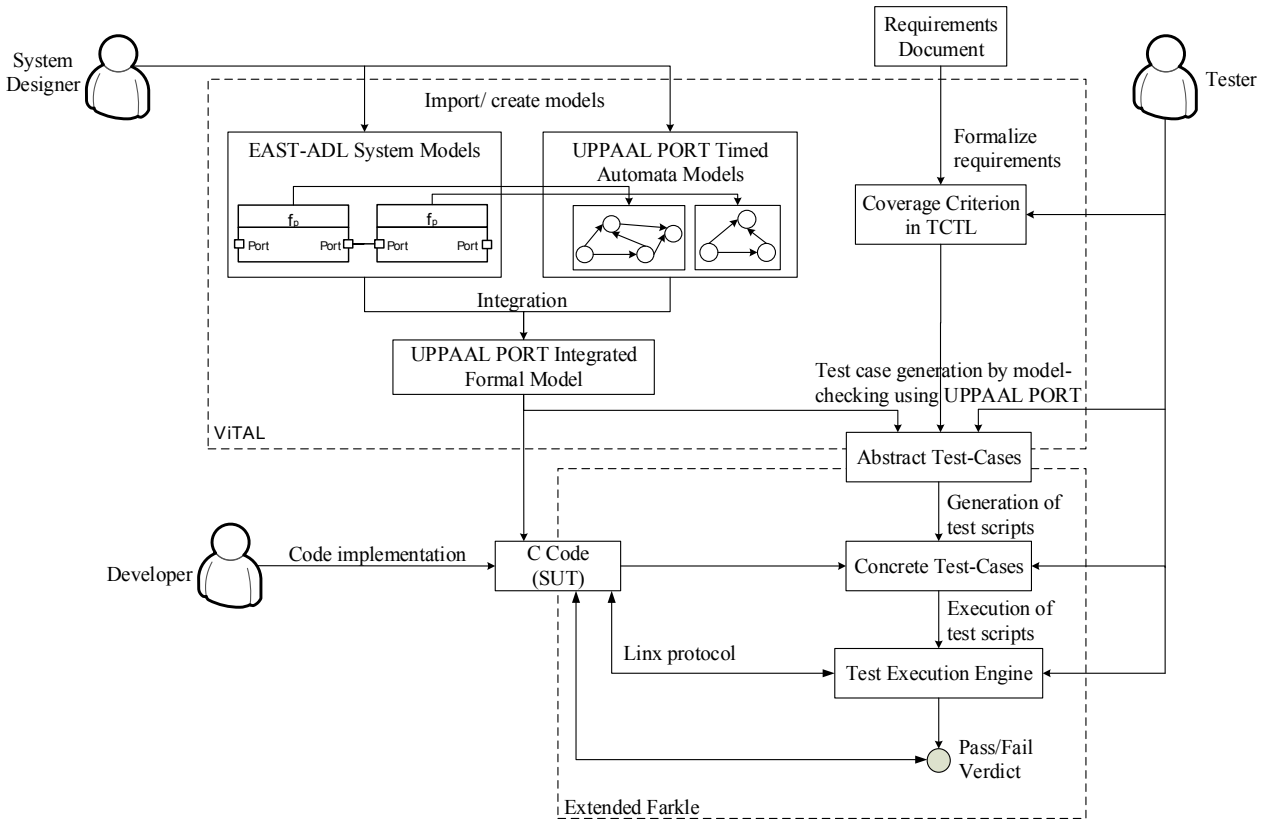

Figure 4: From ViTAL to Farkle: The Methodology

methodology [12], and it is implemented by a tool chain consisting of ViTAL and Farkle, as depicted in Figure 4. We assume three actors in the process: the System Designer, the Developer, and the Tester. Their roles are explained below.

The **System Designer** performs the following actions:

- Imports the EAST-ADL model and creates the associated TA behavior to each EAST-ADL f_p.
- Performs an automatic transformation from the EAST-ADL + TA models to the input language of UPPAAL PORT, in ViTAL. The result is the integrated abstract formal model used for formal verification and test-case generation by means of model-checking.

The **Developer** implements the code (here the SUT) manually, based on the system's integrated abstract formal model (for which we define an executable semantics).

The **Tester** performs the following actions:

- Formalizes the system requirements manually into TCTL properties, the query language of UPPAAL PORT. Each requirement represents a testing goal, whereas their collection is our coverage criterion.
- Generates abstract test-cases with UPPAAL PORT for the integrated formal model, against the above formalized criterion.
- Converts the abstract test-cases into concrete test-cases automatically, by generating Python test-scripts executable by Farkle.
- Executes the concrete test-cases against the SUT, to obtain a *pass* or *fail* verdict, and also code-related information (e.g., variable values).

The activities performed by the System Designer are part of our previous work, and for further details we refer the reader to our previous publication [7]. The next sections provide details on semantics preserving code implementation (see Section V), as well as our method for test-case generation from EAST-ADL models with TA semantics, up to test-case execution on the SUT (see Section VI).

V. Implementation Activities

The implementation is an important, labor intensive, and error prone phase in the development process of any software system. To ease the implementation process, we are interested in providing guidelines that could help the developers to implement C code, based on the EAST-ADL system models extended with TA semantics. For this, we introduce an executable semantics for UPPAAL PORT TA that could serve as the basis of future code synthesis.

A. Executable Semantics of UPPAAL PORT TA

In principle, the TA behavior of an EAST-ADL block can be non-deterministic. To obtain an implementation of the EAST-ADL component whose behavior is modeled as UPPAAL PORT TA, we need to define its deterministic

semantics that needs to be obeyed by the code. For this, we adapt the approach proposed by Amnell et. al [13] for task automata code synthesis, to UPPAAL PORT TA.

Similar to ordinary timed automata, the semantics of an UPPAAL PORT timed automaton is given in terms of a labeled transition system. Assume that the set of V_D variable valuations is ranged by v, the set of V_C clock valuations by u, and l stores the automaton's current location, $l \in N \cup \{l_0, l_f\}$. In addition, we say that a transition $trs = (l \xrightarrow{g,e,r} l')$ is enabled in state $s = (l, u, v)$, denoted by Enabled(trs,s), when its guard holds, that is, $u, v \models g$.

The non-determinism of the semantic representation of the UPPAAL PORT TA stems from the internal, read, or write actions, as well as from time-delays. As in previous work on TA, we resolve non-determinism, as follows: (i) Let $Pr : E \mapsto \mathbf{N}$ be a function that assigns unique priorities to each edge in the UPPAAL PORT TA. If several transitions are enabled in a state, the function Pr establishes the order in which the transitions are fired. This resolves the action non-determinism; (ii) Time non-determinism is resolved by implementing the *maximal progress assumption* [14], in which delay transitions are forbidden if an action transition is enabled. The TA should fire all the enabled transitions until no enabled transition exists anymore.

In the following, we write $(l, u, v) \xrightarrow{e,r} (l', u', v')$ for a state-changing *discrete transition* (internal, read, or write) on which update actions in form of assignments e, or clock resets r, occur. In case of a *delay transition* that does not result in a state-change, we write $(l, u, v) \xrightarrow{t} (l, u', v')$, where $u' = u + t$, and $(u + t) \models I(l)$ holds.

Definition 1 (Deterministic Semantics): Let $B = (N, l_0, l_f, V_D, V_C, r_0, r_f, Ed, I)$ be a UPPAAL PORT TA behavior of an EAST-ADL component. Assuming a function Pr that assigns priorities to TA edges, the deterministic semantics of the component's behavior is a labeled transition system defined by the following rules:

- $(l, u, v) \xrightarrow{e,r} (l', u', v')$ if $\mathsf{Enabled}(l \xrightarrow{g,e,r} l', (l, u, v))$, and there is no $edge \in Ed$ such that $Pr(edge) > Pr(l \xrightarrow{g,e,r} l')$ and $\mathsf{Enabled}(edge, (l, u, v))$;
- $(l, u, v) \xrightarrow{t} (l, u + t, v')$ if $(u + t) \models I(l)$, and for all $edge \in Ed$ and $d < t$, $\neg\mathsf{Enabled}(edge, (l, u + d, v))$.

The above definition ensures conformance of the implementation to the high-level behavioral model, since the behavior defined by the deterministic semantics is a subset of the UPPAAL PORT TA behavior. Hence, all the transition sequences possible in the (deterministic) implementation model are also possible in the original (possibly non-deterministic) one, thus guaranteeing preservation of the safety properties of the EAST-ADL behavioral TA model. However, the code generation is not automated yet. Even if the latter were achieved, the code might still need human intervention in implementing primitives or aggregations that would improve its performance. Hence, testing the code itself cannot be avoided.

B. Implementing the System Model

We approach the problem of code implementation as a mapping activity between the EAST-ADL system model extended with TA behavior and C code. We propose a simple 1-to-1 mapping to code elements starting with the elements of the EAST-ADL model focusing on: (i) components, (ii) ports, (iii) connectors and (iv) triggering information. The mapping is defined in Table I.

Table I: Mapping EAST-ADL f_p's interface to C code

EAST-ADL f_p	**C Code**
Component (block)	C function
Port (input and output)	Buffer of size 1
Connector	Connection between buffers
Triggering information	Function calls and time interrupts

This mapping provides guidelines for the structure of the C code, which should also conform to the semantics of Definition 1. For instance, for each f_p in the EAST-ADL model we will create a C function implementing its behavior. Since the input and output data-flow ports can store only the latest value of the corresponding variable, the implemented C function will have a buffer of size 1 for each port. This implies that the connection between a block's output ports and another one's input ports is translated into a link between buffers. The triggering information for each f_p is implemented as a function call or a time interrupt. However, some of the features of the EAST-ADL model are lost at the implementation level. For instance, the C code does not respect the "read-execute-write" semantics of the model.

The C code inside each function is implemented based on the TA behavior. Each element in the TA tuple is mapped to code elements according to Table II.

Table II: Mapping TA behavior to C code

TA model	**C code**
data variables	variables
clock variables	variables of type long
locations	state variables
clock reset (initial and final)	assignment to zero
invariant	while loop
enabled action selection	if/case statement
action update	assignment

Based on these rules and Definition 1, a complete implementation of the system can be obtained. The implementation conforms with the model. This conformance is an important aspect in model-based testing, as the formal model and the SUT need to be in close relation for the abstract test-cases to really aid the generation of executable test-cases, in a meaningful way.

VI. Testing Activities

In model-based testing, the formal model is a faithful yet abstract representation of the intended system, based on requirements and specification, and describes rigorously the intended behavior using formal modeling notations. In our framework, we employ ViTAL to create the formal model starting from the architectural representation of the system in EAST-ADL and describe the behavior of each f_p as a UPPAAL PORT TA model using the specific TA semantics. In this section, we use such a model to generate abstract test-cases with UPPAAL PORT automatically, followed by their automatic conversion to Python scripts, and their execution on the SUT.

A. Generation of Abstract Test-Cases in ViTAL

ViTAL employs UPPAAL PORT to automate the abstract test-case generation and takes as input: (i) the abstract formal model represented by the "UPPAAL PORT compliant" EAST-ADL model with TA semantics, and (ii) a testing goal, i.e., a functional requirement of the system (or a collection of such requirements) formalized as a TCTL reachability property (properties).

UPPAAL PORT is a model-checker designed for the simulation and the formal verification of component-based embedded systems, and is not tailored for test-case generation. However, we exploit its ability to automatically generate witness traces for *reachability* properties specified in TCTL. Such properties are encoded as $E <> q$, where E represents the existential path quantifier, $<>$ is the temporal operator, and q is the goal state. The property can be read as follows: there exists an execution path such that, eventually, the state q is reached. The test goal guides the generation of a witness trace from an infinite number of possible executions of the system.

The witness trace is a sequence of states and transitions, and it represents the abstract test-case (ATC) for the test goal specified by the reachability property:

$$ATC \triangleq (l_0, u_0, v_0) \xrightarrow{a_0} (l_1, u_1, v_1) \xrightarrow{a_1} ... \xrightarrow{a_{n-1}} (l_n, u_n, v_n)$$

In the above, the state of the system is defined by the current locations l_i, data valuations u_i and clock valuations v_i in the TA network that describes the system behavioral model. The transition actions a_j represent either delays, or internal TA transitions, or the special read/write transitions from/to ports, respectively.

For each test goal, the UPPAAL PORT model-checker generates only one trace representing the execution of the system from its initial state to the goal state encoded by the reachability property. Such a trace represents our abstract test-case with respect to a particular system requirement. A collection of such abstract test-cases is provided to Farkle, for further transformations and execution on the code.

B. Generation and Execution of Concrete Test-Cases in Farkle

The abstract test-cases generated by ViTAL are provided as input to the Extended Farkle environment (i.e., Farkle plus Parser, etc.). The abstract test-cases are parsed and the order of states and transitions, along with the values of variables in each state, are identified. Based on this information a test-script is created. The test-script basically creates signals with values of variables at each state extracted by parsing the abstract test-cases, which are then sent to the target system. The initial values of variables that are sent to the target system by the test-script, in form of signals, trigger the SUT to execute and evolve through a set of states and transitions. The information about the actual set of states and transitions taken by the SUT during execution is collected and sent back to the script (again in form of signals). An important contribution here is that we enable tracking of state changes at runtime by implementing the code based on the formal models. In other words, a switch-case structure is used and some variable keeps track of the current system state, and changes accordingly when moving to a different state.

The test-script receives the result, that is, the information on the set of states and transitions, as well as the values of variables at each state. This information, collected during execution and at runtime, is then used to compare the actual order of states and transitions against the expected one originated from the models and specified in the abstract test-case. If any discrepancy between the actual and expected orders of traversed states and transitions (with the option of also checking the expected values of variables) is found, the test result is evaluated to *fail*, otherwise a *pass* verdict is issued. In other words, it is checked that, based on the given inputs, the *exact* same order of states as in the trace and abstract test case appears at runtime, during the execution of the system.

In the above steps, the generation of executable test-scripts is based on the following principles:

- For each variable (in the UPPAAL PORT TA), we create an array containing all of its expected values, respectively, at each state and in the exact same order as it appears in the abstract test-case, as follows: <statemachine_name>_<variable> = [x,..,y].
- The initial values of variables are used in the structure of a signal, which is then sent to the target.
- We define an additional array to preserve the order of states, in the form of: <statemachine_name>_state = [x,..,y]; e.g., line 18 in Figure 8, where the numbers serve as IDs, and each represents a unique state in the automaton, respectively.
- Based on the number of states, we create a loop in the script.
- Inside the loop, we add assertion statements for each variable, e.g., line 28 in Figure 8, to check its expected value at the current state versus its received value at that state,

which is retrieved from the log information sent back from the target (in the form of a signal), respectively.

• An additional assertion statement is used in a similar way, for checking the actual order of visited states in the code, versus the expected ones in the model.

Basically, this allows to verify the internal state of the system, and to determine whether it is behaving as expected (as specified in the models) or not. Moreover, it makes it possible to determine exactly between which states a deviation from the expected behavior has occurred. This mechanism provides a *defect localization* feature as well. In other words, testers can get some insight into the vicinity of a problem in the code, which can ease debugging and fixing that respective problem.

VII. Brake-by-Wire Revisited: Applying the Methodology

We illustrate and exercise the applicability of our approach on the BBW system, introduced in Section III.

A. Creating the formal model

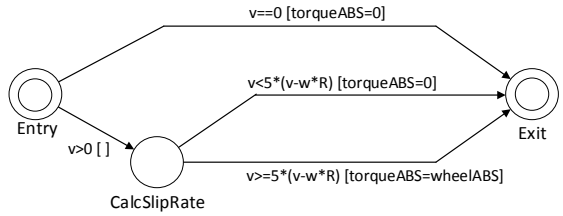

Figure 5: The TA description of the ABS function.

In ViTAL, we have imported the EAST-ADL model described in Section III and created nine TA describing the behavior of each EAST-ADL f_p, respectively. Figure 5 shows the behavior of the **pABSFL** f_p as an UPPAAL PORT TA model.

The functionality of the timed automaton is described as follows. First, the speed of the car is evaluated; if the car has no speed then no brake force is applied which corresponds to transversing the edge annotated with $v == 0[torqueABS = 0]$, otherwise the slip rate is evaluated. If the slip rate exceeds 0.2, no braking force should be applied to not block the wheel. In our TA model, we are evaluating $s > 0.2$ as $v < 5 \times (v - w \times R)$.

Table III: Mapping TA variables to EAST-ADL ports

TA variable	EAST-ADL port
w	WheelSpeed_rpm_FL
wheelABS	RequestedTorque_FL
torqueABS	ABSTorque_FL
v	VehicleSpeed_kmph_FL

In ViTAL, the TA local variables need to be mapped to the EAST-ADL ports shown in Figure 2. This mapping is presented in Table III. Next, ViTAL performs an automatic transformation to the input language of UPPAAL PORT. At this point, we can use the UPPAAL PORT model-checker to simulate and formally verify the model against its requirements. Once the correctness of our model is ensured, we start generating test-cases.

B. Code implementation

Based on the guidelines of Section V, we have implemented the code for the BBW system. A section of the code, depicting the functionality of the ABS component is shown in Figure 6.

```
1  void mbatAbs_calc(MbatAbsInput* input, void* hdl)
2  { state = Idle;
3    /* Internal variables of automaton */
4    float s;
5    /* Output variables */
6    U32 TorqueABS=0;
7
8    state = Entry;
9    while(state != Exit) {
10     switch(state) {
11       case Entry: {
12         if(input->v > 0) {state=CalcSlipRate; }
13         else
14           if(input->v == 0) { TorqueABS=0; state=Exit; }
15           else { // Error }
16         break; }
17       case CalcSlipRate: {
18         s = (float)(input->v-input->w*input->R)/input->v;
19         if(s > 0.2) { TorqueABS=0; }
20         else { TorqueABS=input->WheelABS; }
21         state = Exit;
22         break; }
23       case Exit: { break; } }
24     printf("  Tracing ABS calculation state:%d\n",
25         state);
26     mbatAbs_transition(state, input->w, input->WheelABS,
27             input->v, TorqueABS, input->R,
28             (MBAT_TRC*)hdl);
29     state = Idle; } }
```

Figure 6: Implementation of the ABS component

Each location in the TA model depicted in Figure 5 represents a possible value of the variable $state$, which is initially set to $Entry$. While $state$ is different from $Exit$, the code implements all the possible computations of the TA, e.g., if $v == 0$ then $torqueABS$ is set to zero. Note that time is not considered in this implementation, so all transitions are taken instantly.

C. Testing goal

In this paper, we focus on one of the requirements of the ABS function, which states that: "*If the brake pedal is pressed and a wheel has a slip rate > 20%, then the brake torque for that wheel should be set to 0* N/m^2". The requirement is expressed in TCTL as follows:

$E <> (BrakePedalSensor.pos > 0 \; and \; ABS.v < 5 \times (ABS.v - ABS.w \times ABS.R) \; and \; WheelActuator.NoBrake)$

D. Abstract test-case generation

As presented in Section VI-A, we employ UPPAAL PORT to generate abstract test-cases from the abstract model previously constructed. The model-checker takes as input the formal model together with the test goal specified as the TCTL reachability property above.

```
1  State:(ABSFL.idle)
2      ABSFL.w=0 ABSFL.wheelABS=0 ABSFL.torqueABS=-1
3      ABSFL.v=0 ABSFL.R=1/2
4  Transitions: ABSFL.idle->ABSFL.Entry { w:= 8, wheelABS:= 1,
5      v:= 12}
6  State:(ABSFL.Entry)
7      ABSFL.w=8 ABSFL.wheelABS=1 ABSFL.torqueABS=-1
8      ABSFL.v=12 ABSFL.R=1/2
9  Transitions: ABSFL.Entry->ABSFL.CalcSlipRate { v> 0}
10 State:(ABSFL.CalcSlipRate)
11     ABSFL.w=8 ABSFL.wheelABS=1 ABSFL.torqueABS=-1
12     ABSFL.v=12 ABSFL.R=1/2
13 Transitions: ABSFL.CalcSlipRate->ABSFL.Exit
14     { v< 5* (v- w* R), torqueABS:= 0 }
15 State: (ABSFL.Exit)
16     ABSFL.w=8 ABSFL.wheelABS=1 ABSFL.torqueABS=0
17     ABSFL.v=12 ABSFL.R=1/2
18 Transitions: ABSFL.Exit->ABSFL.idle { }
19 State:(ABSFL.idle)
20     ABSFL.w=8 ABSFL.wheelABS=1 ABSFL.torqueABS=0
21     ABSFL.v=12 ABSFL.R=1/2
```

Figure 7: Abstract Test-Case

The UPPAAL PORT model-checker generates the witness trace presented in Figure 7 automatically. The trace represents the execution of the pABSFL f_p. Initially, the TA is in location $idle$ and all variables are zero. The first transition to state $Entry$ is a $read$ transition, where the latest variable values of w, $wheelABS$, and v are read. Since $v > 0$, the TA moves to the $CalcSliprate$ location. On the transition to $Exit$, the $torqueABS$ variable is set to zero, and after the $write$ transition, the TA returns to the $idle$ location.

Model-checking this particular instance has involved exploring 154182 states out of 203384 stored ones, and the abstract test-case generation took 3.27 seconds on a 1.8 GHz Intel Core i5 processor, with 8 GB of RAM memory.

E. Python scripts generation

From the abstract test-case of Figure 7, the input variable values determining transitions in the TA model are identified automatically, 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 Figure 8. Lines 6-10 in the script set the content of the input signal with the initial variable values, whereas line 11 encodes sending the signal to the target. The expected values of variables, as well as the expected order of visited states are defined in lines 13-18, according to the principles described in Section VI-B. The log information sent back to the host, from the target, in the form of a signal is then received (line 20). Again, following the aforementioned principles, a set of assertion statements for checking the returned values versus the expected values are also generated as the body of a loop, depicted in lines 23-39 of the script.

```
1  import sys
2  import signals
3  import xmlrunner
4  ...
5  # Sending input signal to ABSFL
6  sig_send_ABSFL = signals.ABSFL_INPUT_SIG()
7  sig_send_ABSFL.input.ABSFL_w = 8
8  sig_send_ABSFL.input.ABSFL_v = 12
9  sig_send_ABSFL.input.ABSFL_wheelABS = 1
10 sig_send_ABSFL.input.ABSFL_R = 1
11 self.linx.send(sig_send_ABSFL, self.pid_ABSFL)
12 # Expected values
13 ABSFL_w = [8, 8, 8]
14 ABSFL_wheelABS = [1, 1, 1]
15 ABSFL_torqueABS = [-1, -1, 0]
16 ABSFL_v = [12, 12, 12]
17 ABSFL_R = [0.5, 0.5, 0.5]
18 ABSFL_state = [1, 2, 3]
19 # Receive signals from test targets
20 sig_recv_ABSFL = self.linx.receive
21     ([signals.ABSFL_OUTPUT.SIGNO])
22 # Testing of ABSFL
23 for i in range(sig_recv_ABSFL.num_states):
24 print "Transition %d:" %(i+1)
25 self.assertEqual(sig_recv_ABSFL.states[i].state,
26     ABSFL_state[i])
27 print " state = %d" %sig_recv_ABSFL.states[i].state
28 self.assertEqual(sig_recv_ABSFL.states[i].w, ABSFL_w[i])
29 print " w = %d" %sig_recv_ABSFL.states[i].w
30 self.assertEqual(sig_recv_ABSFL.states[i].wABS,
31     ABSFL_wABS[i])
32 print " wheelABS = %d" %sig_recv_ABSFL.states[i].wheelABS
33 self.assertEqual(sig_recv_ABSFL.states[i].tABS,
34     ABSFL_tABS[i])
35 print " torqueABS = %d"
36     %sig_recv_ABSFL.states[i].torqueABS
37 self.assertEqual(sig_recv_ABSFL.states[i].v, ABSFL_v[i])
38 print " v = %d" %sig_recv_ABSFL.states[i].v
39 self.assertEqual(sig_recv_ABSFL.states[i].R, ABSFL_R[i])
40 print " R = %d" %sig_recv_ABSFL.states[i].R
41 ...
```

Figure 8: Generated Python script

When the test-script is executed, an input signal is sent to the target. Upon the receipt of a signal, the process to which the signal is sent starts executing. The different states that a process enters are tracked and logged at runtime and during the execution of the code. This information is then sent back to the script, where it is checked whether the order of the states at runtime and also the value of variables after each state change match the specification in the abstract test-case generated from the TA models. To enable tracking of different states at runtime, the code is implemented in the form of deterministic state-machines, as shown in Figure 6. The Python script of Figure 8 delivers a "pass" verdict on the implementation of Figure 6.

F. Conformance between the abstract test-case and the Python script

Our abstract test-case presented in Figure 7 can be represented operationally by a sequence of states and transitions as follows:

$(idle, w = 0, wheelABS = 0, torqueABS = -1, v = 0, R = 1/2)$
$\xrightarrow{read(v, wheelABS, v)}$
$(Entry, w = 8, wheelABS = 1, torqueABS = -1, v = 12, R = 1/2)$
$\xrightarrow{v>0}$
$(CalcSlipRate, w = 8, wheelABS = 1, torqueABS = -1, v = 12, R = 1/2)$
$\xrightarrow{v<5*(v-w/R), torqueABS=0}$
$(Exit, w = 8, wheelABS = 1, torqueABS = 0, v = 12, R = 1/2)$
$\xrightarrow{write(torqueABS)}$
$(idle, w = 8, wheelABS = 1, torqueABS = -1, v = 12, R = 1/2)$

In a similar manner, the Python scripts give rise to the following:

$(ABSFL\_state = 1, ABSFL\_w = 8, ABSFL\_wheelABS = 1,$
$ABSFL\_torqueABS = -1, ABSFL\_v = 12, ABSFL\_R = 1/2)$
$\xrightarrow{v>0}$
$(ABSFL\_state = 2, ABSFL\_w = 8, ABSFL\_wheelABS = 1,$
$ABSFL\_torqueABS = -1, ABSFL\_v = 12, ABSFL\_R = 1/2)$
$\xrightarrow{v<5*(v-w/R), torqueABS=0}$
$(ABSFL\_state = 3, ABSFL\_w = 8, ABSFL\_wheelABS = 1,$
$ABSFL\_torqueABS = 0, ABSFL\_v = 12, ABSFL\_R = 1/2)$

In the above we have the following: the $Entry$ state in the TA model is $ABSFL\_state = 1$ in the Python script. Given this, we observe that the trace generated from executing the Python script is included in the trace generated by executing the TA model. Thus, it follows that we can actually use the Python script as a concrete test-case on the SUT, so the abstract test-case of Figure 7 has proven to be a feasible abstract test-case candidate.

VIII. Related Work

Model-based testing by model-checking is a technique introduced almost fifteen years ago [15] as an efficient way of using a model-checker to interpret traces as test-cases. 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 [1] using timed automata specifications. In comparison, in our work we provide an approach suited 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 the software testing community have been investigating how design components and architecture description languages (ADLs) can be used for testing purposes. This led several research groups to develop concrete testing techniques for ADLs[16], [17], [18]. Our framework allows the formal specification of both the interface, and the internal behavior of each EAST-ADL block as UPPAAL PORT TA. In addition, in this work we have provided functional test goals to be considered by the UPPAAL PORT model-checker, defined an executable semantics for the UPPAAL PORT TA, and described a method for generating code that preserves the semantics of the TA model.

While a number of groups have made a distinction between abstract and concrete test cases [19], [20], [21], there are also differences in each case. For instance, Peleska [20] has proposed the RT-Tester tool-suite along with the corresponding methodology, and discussed the two types of test-cases, abstract and executable. The major goal in RT-Tester is to execute test-cases against the models of the system. In our work, we have introduced an approach that generates abstract test cases and then concrete ones, which the latter are actually executable against the running system in its target environment. In [22] based on a subset and preliminary version of the approach (which is extended here in this work) and focusing only on the concrete test cases part, we have discussed how having such executable test scripts to test the system behavior also serves as a method to verify architectural consistency. Finally, 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 [23], [24]. In this work, we have tackled this issue by complementing formal verification of the system at the model level with the testing of its behavior at runtime.

IX. Conclusions and Future Work

In this paper, we have presented a methodology for testing system implementations, starting from EAST-ADL architectural models that are extended by TA behavioral models. The methodology is supported by a tool-chain consisting of ViTAL and Farkle, which can produce and run test-cases automatically. The abstract test-cases for functional requirements result by model-checking the "TA enriched" EAST-ADL models in UPPAAL PORT. Next, such test-cases are transformed into Python scripts representing the executable test-cases that are finally run on the actual code that is implemented based on the (verified) formal model. Our work is an attempt to exercise the feasibility of test-case generation from EAST-ADL models. The method has shown encouraging results when applied on a Brake-by-Wire prototype from Volvo. As future work, we plan to also investigate abstract test-case generation for timing properties of EAST-ADL models, and integrate the results in our methodology. In addition, we envision extending our work towards other ADLs.

Acknowledgment

The authors would like to thank Elaine Weyuker for her valuable comments on this work. This research has received funding from the ARTEMIS JU, 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] A. Hessel, K. Larsen, B. Nielsen, P. Pettersson, and A. Skou, "Time-Optimal Real-Time Test Case Generation Using UPPAAL," in *Lecture Notes in Computer Science, Formal Approaches to Software Testing*. Springer Berlin Heidelberg, 2004, pp. 114–130.

[2] T. Jan, "Model based testing with labelled transition systems," in *Formal methods and testing*. Springer, 2008, pp. 1–38.

[3] M. Satpathy, M. Leuschel, and M. Butler, "Protest: An automatic test environment for b specifications," *Electronic Notes in Theoretical Computer Science*, vol. 111, pp. 113–136, 2005.

[4] R. Alur and D. L. Dill, "A theory of timed automata," *Theoretical computer science*, vol. 126, no. 2, pp. 183–235, 1994.

[5] A. Agrawal, G. Simon, and G. Karsai, "Semantic translation of simulink/stateflow models to hybrid automata using graph transformations," *Electronic Notes in Theoretical Computer Science*, vol. 109, pp. 43–56, 2004.

[6] J. Håkansson and P. Pettersson, "Partial order reduction for verification of real-time components," in *Formal Modeling and Analysis of Timed Systems*. Springer, 2007, pp. 211–226.

[7] E.-Y. Kang, E. P. Enoiu, R. Marinescu, C. Seceleanu, P.-Y. Schobbens, and P. Pettersson, "A methodology for formal analysis and verification of east-adl models," *Reliability Engineering & System Safety*, vol. 120, pp. 127–138, 2013.

[8] J. Håkansson, J. Carlson, A. Monot, P. Pettersson, and D. Slutej, "Component-based design and analysis of embedded systems with uppaal port," in *Automated Technology for Verification and Analysis*. Springer, 2008, pp. 252–257.

[9] Daniel Digerås, "Integration between Optima and Farkle and verification with a use case about file storage stack integration in a quality of service manager in OSE - Master Thesis," http://liu.diva-portal.org/smash/record.jsf?pid=diva2:624122, April 2011.

[10] T. A. ATESST2 Consortium, "EAST-ADL Profile Specification, 2.1 RC3 (Release Candidate)." www.atesst.org, 2010, pp. 10–75. [Online]. Available: www.atesst.org/home/liblocal/docs/

[11] J. Bengtsson, B. Jonsson, J. Lilius, and W. Yi, "Partial order reductions for timed systems," in *CONCUR'98 Concurrency Theory*. Springer, 1998, pp. 485–500.

[12] M. Utting, A. Pretschner, and B. Legeard, "A taxonomy of model-based testing approaches," *Software Testing, Verification and Reliability*, vol. 22, no. 5, pp. 297–312, 2012.

[13] T. Amnell, E. Fersman, P. Pettersson, H. Sun, and W. Yi, "Code synthesis for timed automata," *Nord. J. Comput.*, vol. 9, no. 4, pp. 269–300, 2002.

[14] W. Yi, "Ccs+ time= an interleaving model for real time systems," in *Automata, Languages and Programming*. Springer, 1991, pp. 217–228.

[15] A. Engels, L. Feijs, and S. Mauw, "Test generation for intelligent networks using model checking," in *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 1997, pp. 384–398.

[16] A. Bertolino and P. 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*. ACM, 1996, pp. 62–64.

[17] H. Muccini, P. Inverardi, and A. Bertolino, "Using software architecture for code testing," *Software Engineering, IEEE Transactions on*, vol. 30, no. 3, pp. 160–171, 2004.

[18] H. Reza and S. Lande, "Model based testing using software architecture," in *Information Technology: New Generations (ITNG), 2010 Seventh International Conference on*. IEEE, 2010, pp. 188–193.

[19] C. Nebut, F. Fleurey, Y. Le-Traon, and J.-M. Jezequel, "Automatic test generation: a use case driven approach," *Software Engineering, IEEE Transactions on*, vol. 32, no. 3, pp. 140–155, 2006.

[20] J. Peleska, "Industrial-strength model-based testing - state of the art and current challenges," in *Proceedings of the Eighth Workshop on Model-Based Testing*, March 2013, pp. 3–28.

[21] W. Prenninger, M. El-Ramly, and M. Horstmann, "Chapter 15: Case studies," in *Model-Based Testing of Reactive Systems: Advanced Lectures (Lecture Notes in Computer Science)*, M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner, Eds. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2005.

[22] M. Saadatmand, D. Scholle, C. W. Leung, S. Ullström, and J. F. Larsson, "Runtime verification of state machines and defect localization applying model-based testing," in *Proceedings of the WICSA 2014 Companion Volume*. ACM, 2014.

[23] M. Saadatmand, A. Cicchetti, and M. 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)*. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 121–134.

[24] S. Chodrow, F. Jahanian, and M. Donner, "Run-time monitoring of real-time systems," in *Real-Time Systems Symposium (RTSS), 1991. Proceedings., Twelfth*, 1991, pp. 74–83.