AQAF: an Architecture Quality Assurance Framework for systems modeled in AADL

Andreas Johnsen, Kristina Lundqvist, Kaj Hänninen, Paul Pettersson
School of Innovation, Design and Engineering
Mälardalen University
Västerås, Sweden
{andreas.johnsen,kristina.lundqvist,kaj.hanninen,paul.pettersson}@mdh.se

Martin Torelm
Bombardier Transportation Sweden AB
Propulsion & Converter Control Standardization
Västerås, Sweden
martin.torelm@se.transport.bombardier.com

***Abstract*—Architecture engineering is essential to achieve dependability of critical embedded systems and affects large parts of the system life cycle. There is consequently little room for faults, which may cause substantial costs and devastating harm. Verification in architecture engineering should therefore be holistically and systematically managed in the development of critical embedded systems, from requirements analysis and design to implementation and maintenance. In this paper, we address this problem by presenting AQAF: an Architecture Quality Assurance Framework for critical embedded systems modeled in the Architecture Analysis and Design Language (AADL). The framework provides a holistic set of verification techniques with a common formalism and semantic domain, architecture flow graphs and timed automata, enabling completely formal and automated verification processes covering virtually the entire life cycle. The effectiveness and efficiency of the framework are validated in a case study comprising a safety-critical train control system.**

I. Introduction

The highly increased utilization of digital technology in critical embedded systems is challenging the ability of dependability-achieving engineering. A problem is that the complexity of the systems is increasing beyond what current engineering is able to manage [1, p. 4]. Quality assurance in the form of verification is essential to achieve dependable systems and often stands for a considerable amount of the total development cost. With an increasing complexity, the ability to assure quality becomes even harder. There is consequently a need for more effective and efficient verification techniques such that acceptable levels of dependability of increasingly complex embedded systems can be maintained. A part of the solution to this problem is advances of verification in architecture engineering as faults emerge in the interactions of components when the complexity increases [1, p. 8]. We denote this type of verification as architecture-based verification. State of the art architecture engineering is based on architecture models expressed in some architecture description language (ADL). An ADL that has been developed for critical embedded systems is the Architecture Analysis and Design Language (AADL) [2] – an overview of AADL can be found in [7]. The use of AADL generates standardized, computer-readable, and semi-formal models of the system architectures. These properties contribute to the assurance of quality by facilitating understandability, communication, and analysis. In addition, they provide the necessary prerequisites for developing computerized verification techniques that are effective and efficient enough in detecting architectural faults that emerge in the development of complex embedded systems. Research in this field has mostly been focused on adapting formal methods to AADL; there exist a number of model checking, formal analysis, and simulation techniques. However, architecture engineering is conducted throughout the entire system life cycle in varying phases, from requirements analysis and design to implementation and maintenance. These conditions promote a holistic approach to architecture-based verification whereas current contributions to this field, to our knowledge, do only address fractions of the system life cycle.

In this paper, we present a possible solution to this problem: the Architecture Quality Assurance Framework (AQAF), illustrated in Fig. 1. AQAF provides a model checking technique to avoid (i) architecture design faults, a model-based testing technique to avoid (ii) architecture implementation faults, and a selective regression verification technique based on change impact analysis through slicing to avoid (iii) faults introduced in response to maintenance.

The primary focus of evaluation at the architectural level is the integration of components, including the structure and the resulting emergent behavior and non-functional properties. General verification objectives are to ensure consistency, completeness, and correctness of component interfaces and the control and data interactions among them. To obtain these objectives, the proposed framework is based on architectural control and data flow verification criteria. This is of industrial importance as some contemporary safety standards (e.g., ISO 26262 [4]) request control and data flow analysis of software architecture designs. Based on the control and data flow verification criteria, model checking and model-based testing techniques are used to automatically and formally avoid (i) and (ii) respectively. The semantic domain of AADL is not based on a mathematical language and cannot be directly explored by a model-checker. The framework therefore includes a mapping from AADL to timed automata such that the verification processes can be executed by the Uppaal model-checker [5]. Uppaal timed automata has been chosen due to its ability to express real-time properties and the maturity of the corresponding model-checker and its model-based testing capabilities. Furthermore, the design and implementation are typically subjected to maintenance modifications. Artifacts must therefore undergo regression verification to verify that no new faults have been introduced in response to a design change. In addition, architectural variants may be designed to

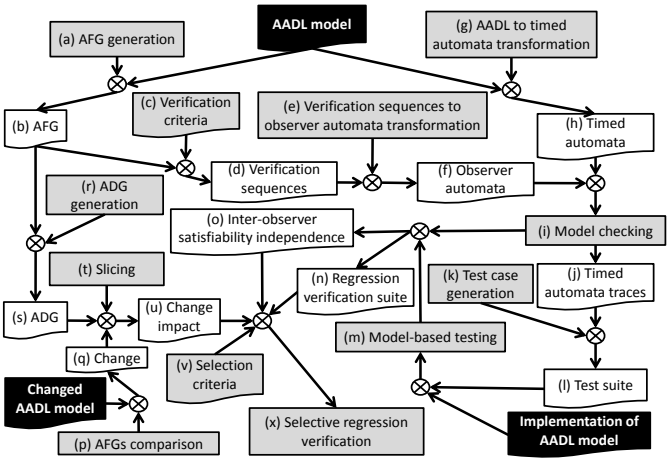

Fig. 1. Flowchart of AQAF. The alphabetical order illustrates a typical application of the framework. A black shape denotes a necessary framework input. A gray shape denotes a formally defined process or rule set (c and v). A white shape denotes an artifact produced by the framework.

develop a product line or to analyze trade-offs. All these scenarios, where different instances of the architecture design or implementation are created, are common and challenged with inefficient regression verification if equivalent parts among the artifacts that are not affected by the changes or variations are unnecessarily re-verified. The framework therefore includes a selective regression verification technique for efficient avoidance of (iii).

These approaches have individually been developed in previous work [3], [6]. However, they have been developed on the idea of a common formalism for architectural control and data flow representation, architecture flow graphs (AFGs), and a common semantic domain, timed automata. Through a common formal underpinning, the techniques can be effectively and efficiently used in an integrated and automated manner with guaranteed semantics preservation. In this paper, we bridge these techniques to achieve this leverage. More specifically, we contribute with a definition of 1. AFGs and how they are generated from AADL models (Section III); 2. how verification sequences are derived from AFGs according to the verification criteria (Section IV); 3. how verification sequences are transformed into observer automata for model checking (Section V and VI); 4. how test cases are generated from observer-generated timed automata traces for model-based testing (Section VII); and 5. how the verification history (n and o in Fig. 1) can be used together with slicing of architecture dependence graphs (ADGs), generated from AFGs, for selective regression verification (Section VIII). In Section IX, we present an empirical validation of AQAF by means of an industrial case study, where the framework is applied to a safety-critical train control system presented in Section II.

II. Line Trip Relay Interface and Supervision

Line trip relay interface and supervision (LTRIS) is a safety-critical train control sub-system embedded in a system of systems developed by Bombardier Transportation AB. An AADL model of LTRIS is partly shown in Table I. The complete model can be found in the technical report of the case study [7]. The core of LTRIS is composed of two periodic tasks: *Controller* and *Tester*. The functionality of *Controller* is to control a safety-critical relay (not shown) according to input, monitor its status, and output feedback data. The feedback data is information on the status of the relay and the status relative to the expected one. *Controller* controls the relay according to input on ports and shared variables from various components. In this manner, *Controller* acts as an interface to the relay. Its input domain is partly determined by *Tester* through *connection1* and *connection2*. The functionality of *Tester* is partly to execute a test sequence, *LtrTsSq*, verifying a correct functioning of the relay. The behavior of *Tester* is specified in a behavioral model, from which *LtrTsSq* is invoked.

The complete model is composed of 6 executable software components, one of which is an abstraction of the system (environment) LTRIS is embedded within, 76 interfaces (ports, parameters, and shared variables), 55 connections, and 5 behavioral models (41 local states and 51 local transitions). Non-functional properties of LTRIS have been adjusted in the model according to the abstraction of the environment.

III. Architecture Flow Graphs

An AADL model essentially expresses control and data flows through the architecture that define the architectural behavior. *Control flows* refer to the orders in which software components and their instructions are executed. *Data flows* refer to the orders in which data variables (including interfaces and connections) are assigned by a component and subsequently used, possibly by a different component. In AADL, the possible interactions among components are represented by four different types of connections: *port connections*, *data access connections*, *subprogram calls*, and *parameter connections*. A port connection represents a transfer of unqueued data, queued control, or queued control with associated data (messages), depending on the type of interconnected interfaces (data port, event port, or event data port). A subprogram call represents a transfer of control whereas a subprogram parameter connection represents a transfer of data. A data access connection represents a transfer of data to or from a, possibly shared, data component. In addition, components may be modeled with behavioral models (BMs) describing their logical execution. A BM both yields internal flows of a component, between input and output interfaces, and refines flows to other components as it operates on connected interfaces. These constructs determine the control and data flows of an AADL model in conjunction with scheduling properties and protocols of shared resources. In order to verify consistency, completeness, and correctness of the system architecture, the control and data flows must be analyzed. A common approach to control and data flow analysis is to extract the prescribed flows into a directed graph. Architecture flow graph (AFG) is a type of directed graph for the representation of AADL flows.

An architecture flow graph $AFG(M) = \langle V, A \rangle$ of an AADL model M is a directed graph of a set of vertices $V = \{v \mid v \in EXPR \cup \langle \text{“}ENTRY\text{”}, comp_i \rangle \cup \langle \text{“}REENTRY\text{”}, comp_i \rangle \cup \langle \text{“}EXIT\text{”}, comp_i \rangle\}$ representing AADL expressions and scheduling states of components, and a set of directed arcs $A \subseteq V \times V$ describing how control and data flow through the vertices. v of an arc $\langle v, v' \rangle \in A$ is the tail and v' is the head, denoting that the flow is directed from v to v'. The denotation $\langle v, v' \rangle$ is used interchangeably with $v \rightarrow v'$.

TABLE I. A MINIATURE AADL MODEL OF LTRIS

```
thread Controller
features
B_OpLtr: in data port Base_Types::Boolean;
B_CdLtr: in data port Base_Types::Boolean;
...
end Controller;
...
thread Tester
features
C_LtrTs: in data port Base_Types::Boolean;
B_LtrFl: in data port Base_Types::Boolean;
...
B_OpLtr: out data port Base_Types::Boolean;
B_CdLtr: out data port Base_Types::Boolean;
...
end Tester;

thread implementation Tester.Impl
connections
C_LtrTs_in: parameter C_LtrTs ->LtrTsSq.C_LtrTs;
B_LtrFl_in: parameter B_LtrFl ->LtrTsSq.B_LtrFl;
...
B_OpLtr_out: parameter LtrTsSq.B_OpLtr ->B_OpLtr;
B_CdLtr_out: parameter LtrTsSq.B_CdLtr ->B_CdLtr;
...
annex behavior_specification
{**
variables
...
states
state0 : initial complete final state;
state1 : state;
...
transitions
state0 -[ on dispatch ]->state1 {LtrTsSq(C_LtrTs,B_LtrFl,..., B_OpLtr,
B_CdLtr,...)};
...
**};
end Tester.Impl;
...
subprogram LtrTsSq
features
C_LtrTs: in parameter Base_Types::Boolean;
B_LtrFl: in parameter Base_Types::Boolean;
...
B_OpLtr: out parameter Base_Types::Boolean;
B_CdLtr: out parameter Base_Types::Boolean;
...
end LtrTsSq;
...
process implementation LineTripSoftware.Impl
subcomponents
relayController: thread Controller;
relayTester: thread Tester;
connections
connection1: port relayTester.B_OpLtr ->
relayController.B_OpLtr {Timing =>Immediate;
Latency =>0ms .. 1ms;};
connection2: port relayTester.DHSSMG_B_CdLtr ->
relayController.DHSSMG_B_CdLtr {Timing =>Immediate;
Latency =>0ms .. 1ms;};
...
 end LineTripSoftware.Impl;
...
```

A vertex $v = \langle expr \rangle$ and an arc $\langle v, v' \rangle$ may be attributed with a set of AADL properties: $\langle v, \{prop_1, prop_2, \ldots, prop_n\}\rangle$ and $\langle \langle v, v' \rangle, \{prop_1, prop_2, \ldots, prop_n\}\rangle$.

An arc has one of the following labels to distinguish different types of control and data flows. $\langle v, v' \rangle_c$ represents a component-internal control flow. A vertex v is called a **direct predecessor** of v' and v' a **direct successor** of v iff $\langle v, v' \rangle_c \in A$. Let $outdegree(v)$ be a function mapping the number of direct successors of v and $indegree(v)$ the number of direct predecessors. A vertex can have zero, one, or two direct successors. A vertex v with two direct successors represents a so called **control expression** constituting a Boolean condition. The two outgoing arcs of v are attributed with $\langle v, v_x \rangle_{cT}$ for $true$ and $\langle v, v_y \rangle_{cF}$ for $false$ and correspond to the control flow in response to the condition evaluation. $\langle v, v' \rangle_{c-inter}$ represents an interaction-based control flow due to the activation of a communication protocol. The execution of v' coincides with the execution of v according to the protocol. $\langle v, v' \rangle_{call}$ represents an inter-component control flow due to a raised event or call. $\langle v, v' \rangle_d$ represents a component-internal data flow. $\langle v, v' \rangle_{d-in}$ represents an inter-component data flow due to a data passing by value or by reference (shared data) protocol. The arc indicates data flowing from an output to an input interface. If used together with a function call, the arc indicates the data flowing from an argument to the corresponding subprogram input parameter. $\langle v, v' \rangle_{d-out}$ represents an inter-component data flow due to a data passing by value protocol activated to return from a call. The arc indicates data flowing from an output parameter of a subprogram to the variable assigned by the call.

The $\langle \text{“}ENTRY\text{”}, comp \rangle$ vertex represents the point of the component $comp$ through which control enters and $outdegree(\langle \text{“}ENTRY\text{”}, comp \rangle) = 1$. A $\langle \text{“}REENTRY\text{”}, comp \rangle$ vertex represents a point of the component $comp$ through which control suspends, and reenters when the component has been reactivated/dispatched after the suspension and $outdegree(\langle \text{“}REENTRY\text{”}, comp \rangle) = 1$. A component may have any number of reenter vertices. The $\langle \text{“}EXIT\text{”}, comp \rangle$ vertex represents the point of the component $comp$ through which control exits and $outdegree(\langle \text{“}EXIT\text{”}, comp \rangle) = 0$. A path $P = v_1 \rightarrow_c v_2 \rightarrow_c \cdots \rightarrow_c v_n$ is called a **control path**. A control path is called a **basic block** if $v_1 \neq ENTRY \cup REENTRY$ and, for $i = 1, 2, \ldots, n-1$, $outdegree(v_i) = 1$.

A. Architecture Flow Graph Generation

An AADL model is transformed into an AFG through three operations. The first operation is to generate an individual control flow graph (CFG) for each AADL component representing a, possibly concurrent, unit of sequential execution, i.e., for each thread and subprogam component. This is achieved by analyzing each thread and subprogram component in isolation to find all possible control flows of type $\langle v, v' \rangle_c$. From this perspective, the control flow is entirely determined by the BM of the component. A BM essentially consists of state transitions. A state transition $s \xrightarrow{pri,g,act} s'$, from a state s to a successor state s', has a priority $pri \in \mathbb{N}$, a (possibly empty) set of predicate guards g, and a (possibly empty) sequence of actions act. Each state transition corresponds to a fixed execution order of operations: the guard of the transition is first computed and, if evaluated to the Boolean value $true$, the sequence of actions is executed. Thus, BM guards and actions are the executable operations and yield the vertices of the CFG. The fixed execution order of operations is repeated throughout the BM until a final state is reached, as shown in Fig. 2. If g_1 is evaluated to the Boolean value $true$, act_1 is

executed, resulting in the arrival of a new state s_i whereupon the transition going out from s_i with the highest priority is executed according to the fixed order. On the other hand, if g_1 is evaluated to $false$, another state transition going out from s_1 with the (next) highest priority is executed in the fixed order (in this case, the transition with priority pri_2 is next in line).

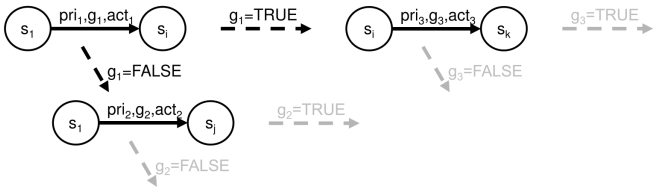

Fig. 2. Illustration of behavioral model semantics. Assume s_1 is the initial state and $pri_1 > pri_2$.

Consequently, each transition $s \xrightarrow{pri,g,act} s'$, where $act = action_1; action_2; ...; action_n$ is a sequence of n actions, maps to a CFG construct of one vertex $v_1 = g$ representing the guard of the state transition, a basic block of n vertices $v_2 = action_1, v_3 = action_2, \ldots, v_{n+1} = action_n$ representing the actions of the state transition, and n arcs $\langle v_1, v_2\rangle_{cT}, \langle v_2, v_3\rangle_c$, $\ldots, \langle v_n, v_{n+1}\rangle_c$ representing the control flow through the executable operations. Note that the arc from the guard to the first action is attributed with a "T". Once a CFG construct has been created for each transition, they are connected according to the order in which states can be reached and the priorities of the state transitions as shown in Fig. 3.

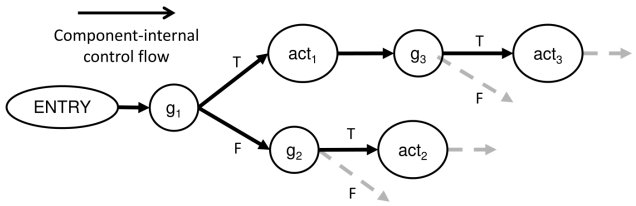

Fig. 3. The control flow graph of the behavioral model example in Fig. 2.

The second operation is to compute the component-internal data flows for each component and annotate them to the CFGs. Such flows can be computed by performing *def-use pairs* analysis of each CFG. Assume that V_{def} is the set of vertices that defines/assigns variable var_i, and V_{use} is the set of vertices that uses/reads var_i. For each pair of vertices $\langle v_x, v_y\rangle \in V_{def} \times V_{use}$ such that there exists a control path $P = v_1 \rightarrow_c v_2 \rightarrow_c \cdots \rightarrow_c v_n$ from v_x to v_y (where $v_1 = v_x$ and $v_n = v_y$) and any other vertex v_z in P does not define/assign var_i, i.e., $v_z \neq V_{def}$ for $z = 2, 3, \ldots, n-1$, there exist a component-internal data flow $\langle v_x, v_y\rangle_d$. If the rule is applied to all variables for each CFG, all the possible component-internal data flows are generated.

The third and final operation is to integrate the CFGs according to the component connections to produce the AFG. Components of AADL models may both transfer data and control through interfaces, where ports and parameters are accessible as variables. Control may be transferred to threads through event ports and event data ports that are included in dispatch conditions, and to subprograms through subprogram calls. In either case, control is transferred to the entry point (including reentry points of threads) of the target component. Data may be transferred through data ports, event data ports, subprogram parameters, and shared data components. Following the default input-compute-output semantics of AADL, each thread dispatch and subprogram invocation includes assignments to in ports and parameters if the component has such connections. In addition, each thread-execution completion and subprogram return includes transmission of output data on out ports and parameters if such connections exist. Consequently, input assignments coincide with entry vertices of subprograms, and with dispatch condition vertices (extensions of entry and reentry vertices to explicitly represent dispatch conditions) of threads. Output assignments, on the other hand, coincide with exit vertices of subprograms whereupon control is returned to the caller. In threads, output assignments coincide with exit and reentry vertices of threads as both represent a completion of the current dispatch when entered.

These inter-component flows are explicitly represented through four distinguished types of vertices (similarly to system dependence graphs defined by Horwitz et al. [8]): (1) *actual-in* vertices on the form $connection = out\_interface$ representing assignments that copy the values of output interfaces to connections; (2) *formal-in* vertices on the form $in\_interface = connection$ representing assignments that copy the values of connections to input interfaces; (3) *formal-out* vertices on the form $connection = out\_parameter$ representing assignments that copy return values of a callee's output parameters to parameter connections; and (4) *actual-out* vertices on the form $in\_interface = connection$ representing assignments that copy return values of parameter connections to destination interfaces of the caller. An actual-in vertex is connected to the corresponding formal-in vertex through a data-in arc. If the connection connects event or even data ports, they are also connected through an event/call arc. A formal-out vertex is connected to an actual out vertex through a data-out arc. To conform to the transmission of output semantics, an interaction-based control flow arc shall be created from each reentry vertex and the exit vertex of the sending thread to the actual-in vertex if it represents a port connection. If it represents a parameter connection, the interaction-based control flow arc to the actual-in vertex is flowing from the call vertex. Similarly, dispatch conditions of entry and reentry vertices of the receiving thread or subprogram must have an interaction-based control flow to the formal-in vertex. With respect to return parameter connections, the exit vertex of the sending subprogram has an interaction-based control flow arc to the formal-out vertex whereas the call vertex has such an arc to the actual-out vertex.

Data access connections to a common data component $data_x$ may represent transfers of data (by reference) if there exist both write-right and read-right access connections. In case this condition holds, and to represent the possible combinations of data flows with respect to concurrency, the data flow between the component $comp_y$ with write-right access and the component $comp_z$ with read-right access is represented through an actual-in vertex $comp_y - comp_z = data_x$, representing the write-right connection, and an inverting formal-in vertex $data_x = comp_y - comp_z$, representing the read-right connection, connected through a data-in arc. Given that a thread or a subprogram gets the data source upon dispatch and releases it upon a completion, each reentry vertex and the exit vertex of the sending thread, or the exit vertex of the sending subprogram, have interaction-based control arcs to the actual-

in vertex. On the other hand, the dispatch conditions of entry and reentry vertices of the receiving thread, or the entry vertex of the receiving subprogram, have interaction-based control arcs to the formal-in vertex.

Once a construct has been created for each connection, each CFG-vertex that operates on a connected interface must subsequently be connected (through component-internal control and data flow arcs) to the corresponding distinguished vertex to finalize the AFG. The result of applying these operations to LTRIS is shown in Fig. 4 (the complete AFG is found in [7]).

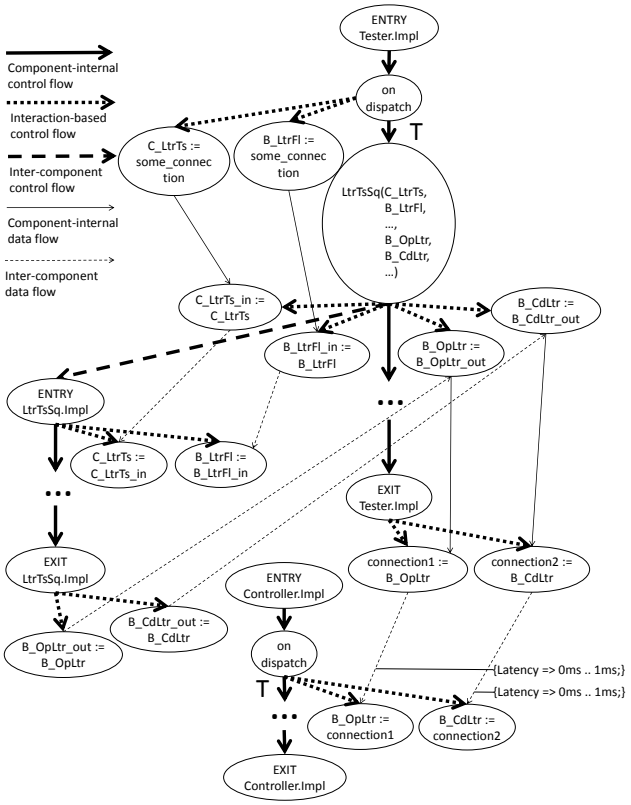

Fig. 4. The AFG of the LTRIS AADL model presented in Table I.

IV. Verification Criteria and Sequences

In order to verify consistency, completeness, and correctness, the architecture flows must be analyzed with respect to requirements and constraints associated with the model. Each control and data flow is composed of a sequence of elements. A flow is constrained if any member in the sequence is associated with a property. A model is consistent if each control and data flow can be fully executed while not contradicting any constraints imposed by properties. In other words, the model must be able to be executed in compliance with the semantic rules such that each flow can be exercised, from the first element to the last according to the order of the sequence, while each (active) property value is valid in each state of the execution. Correctness can only be determined if requirements are associated with the model or if property declarations are considered as requirements. The model is correct if no flow exceeds any requirement declarations while they are executed. The model is complete if all flows can be activated by the specified input classes and a flow will be activated for every class of input. These objectives can be defined in terms of control- and data-flow reachability. Control-flow reachability is the property where each architectural element in an execution order can reach the subsequent element to be executed without conflicting any constraints or requirements. Data-flow reachability is the property where each data element can reach its target component, where the data is used, from its source component, where the data is defined, without conflicting any constraints or requirements. Thus, reachability of each flow imply architecture consistency, correctness, and, if all possible input classes have been covered, completeness. Note that reachability analysis consider properties such as the minimum and maximum latencies of connections, or the period, execution time, and deadline of threads. It therefore implies analysis of aspects such as timing and schedulability. Consequently, flow reachability cannot be achieved if timing constraints are not met or the system is not schedulable.

An AFG contains different structural path types composed of control and data flows. An AFG path in conjunction with the (possibly empty) set of path constraints and requirements is referred to as a *verification sequence*. Three types of paths exist: (1) *Component-Internal Paths* including component-internal flows between interfaces of a component; (2) *Direct Component to Component Paths* including inter-component flows between interfaces of two components; and (3) *Indirect Component to Component Paths* including flows between interfaces of two components through one or several intermediate components. Covering all paths is necessary to ensure completeness, correctness, and consistency. The complete AFG of LTRIS contains 34 component-internal paths, 6 direct paths (three of which are calls with associated parameter data flows), and 17 indirect paths [7].

V. Formal Semantics in Timed Automata

In order to automatically and formally analyze each path, the AADL semantics must be formalized and implemented. Detailed transformation rules from an AADL model to a network of UPPAAL timed automata are presented in [7]. By means of the transformation rules, flow-reachability can be verified using observer automata, the UPPAAL model checker, and reachability formulae in Time Computation Tree Logic (TCTL). In this section, we present the essentials of the transformation rules and define a timed automata syntax and semantics for AADL model checking and model-based testing, as described in Section VI and VII.

A network of timed automata $NTA = \langle \overline{TA}, Var_G, Ch \rangle$ has a vector of n timed automata $\overline{TA} = \langle TA_0, TA_1, \ldots, TA_{n-1} \rangle$, a set of shared (global) variables Var_G, and a set of synchronization channels Ch. A timed automaton $TA = \langle L, \ell_0, X, Var, I, E \rangle$ has a set of locations L, an initial location $\ell_0 \in L$, a set of real-valued variables X called *clocks*, a set of (bounded) integer-typed variables Var, a function assigning invariants to locations $I : L \rightarrow G$, and a set of edges $E \subseteq L \times G \times Act \times U \times L$. G is a set of *guards*, which are conjunctions of predicates over variables and clock constraints of the form $x\, expr_1\, c$, where $x \in X \cup Var \cup Var_G$, $c \in \mathbb{N}$, and $expr_1 \in \{<, \leq, =, \geq, >\}$. $Act = I \cup O \cup \{\tau\}$ is a set of input (denoted a?) and output (denoted a!) synchronization actions and the non-synchronization τ. U is a set of *updates* which are sequences

of variable-assignments of the form $v := expr_2$ and/or clock resets of the form $x := 0$, where $v \in Var \cup Var_G$, $x \in X$, and $expr_2$ is an arithmetic expression over integers. We shall use the denotation $\ell \xrightarrow{g,a,u} \ell'$ iff $\langle \ell, g, a, u, \ell' \rangle \in E$. In addition, locations may be labelled as urgent or committed. In an urgent location, time is not allowed to progress whereas in a committed location, time is not allowed to progress and the next transition must involve one of its outgoing edges.

The semantics of a network of timed automata is defined in terms of a timed transition system over system states. A *system state* is a triple $\langle \overline{\ell}, \overline{\phi}, \overline{\sigma} \rangle$ where $\overline{\ell}$ is a location vector over all automata such that $\overline{\ell^0}, \overline{\ell^1}, \ldots, \overline{\ell^{n-1}}$ denotes the current location of $TA_0, TA_1, \ldots, TA_{n-1}$, $\overline{\phi}$ is a clock valuation vector over all automata such that $\overline{\phi^0}, \overline{\phi^1}, \ldots, \overline{\phi^{n-1}} \in \mathbb{R}_+^X$ and satisfies the invariants of the locations ($\overline{\phi} \models I(\overline{\ell})$), and $\overline{\sigma}$ is a variable valuation vector that maps variables to values and $\overline{\sigma} \models I(\overline{\ell})$. The *initial system state* is a state $\langle \overline{\ell_0}, \overline{\phi_o}, \overline{\sigma_o} \rangle$ where $\overline{\ell_0}$ is the initial location vector, $\overline{\phi_o}$ maps each clock to zero, and $\overline{\sigma_o}$ maps each variable to its default value. Progress is made through *delay transitions* or *discrete transitions*. A *delay transition* is of the form $\langle \overline{\ell}, \overline{\phi}, \overline{\sigma} \rangle \xrightarrow{d} \langle \overline{\ell}, \overline{\phi} \oplus d, \overline{\sigma} \rangle$ where $\overline{\phi} \oplus d$ is the result of synchronously adding the delay d to each clock valuation in $\overline{\phi}$. Let $\overline{\ell}[\ell'_i/\ell_i]$ denote that the ith vector element ℓ_i is replaced by ℓ'_i. A *discrete transition* is of the form $\langle \overline{\ell}, \overline{\phi}, \overline{\sigma} \rangle \xrightarrow{a} \langle \overline{\ell}[\ell'_i/\ell_i, \ell'_j/\ell_j, \ell'_k/\ell_k, \ldots], \overline{\phi'}, \overline{\sigma'} \rangle$ such that there are edges $\ell_{i/j/k\ldots} \xrightarrow{g_{i/j/k\ldots}, a_{i/j/k\ldots}, u_{i/j/k\ldots}} \ell'_{i/j/k\ldots}$ where $\overline{\phi}$ and $\overline{\sigma}$ satisfies $g_i \wedge g_j \wedge g_k \ldots$, the result of updating $\overline{\phi}$ and $\overline{\sigma}$ according to $u_i, u_j, u_k, \ldots$ is $\overline{\phi'}$ and $\overline{\sigma'}$, and the edges are synchronous over complementary actions ($a?$ complements $a!$).

A model is transformed into an automata network essentially composed of one scheduler automaton per processor component, one thread automaton per thread component, and one subprogram automaton per subprogram component. The scheduler automata control the transition of thread states, from dispatches to completions, and of preemptions and context switches. A thread automaton, in its most basic form, consists of four locations: $awaiting\_dispatch$, $ready$, $running$, and $awaiting\_resource$. An example of *Tester* in this form is shown in Fig. 5. Each thread is initially in the *awaiting_dispatch* location. An edge to the *ready* location is fired depending on the dispatch protocol. For periodic threads, the time of dispatch is entirely dependent on the clock. Input from connections, represented as global variables, is simultaneously assigned to input ports, represented as local variables. These assignments correspond to actual-in vertices of port connections. Scheduling properties of the thread are simultaneously assigned to the scheduler. Threads in the $ready$ location are executed by the processor component they are bound to according to the scheduling policy property. Assuming a scheduler with fixed priority preemptive scheduling policy, the thread with the highest priority is selected to run on the processor and thus transits to the *running* location. No more than one thread (per processing unit) is allowed to be in a *running* location simultaneously. A running thread is blocked if it is trying to access a shared data component that currently is locked. Shared resources are accessed in critical sections which are entered through $Get\_Resource$ service calls. Only one thread is allowed to be in a critical section of a shared resource at once. The execution of a thread may also be blocked (in response to a $Await\_Result$ service call) to await the return of a remote subprogram call. A running thread is preempted, and thus transits back to the ready location, if another thread with higher priority enters the *ready* location. A thread in the *running* location that completes its execution transits to the *awaiting_dispatch* location. Output is simultaneously assigned to connections. These correspond to formal-in vertices of port connections. If the thread is specified with a BM, the *running* location is replaced with the BM automaton.

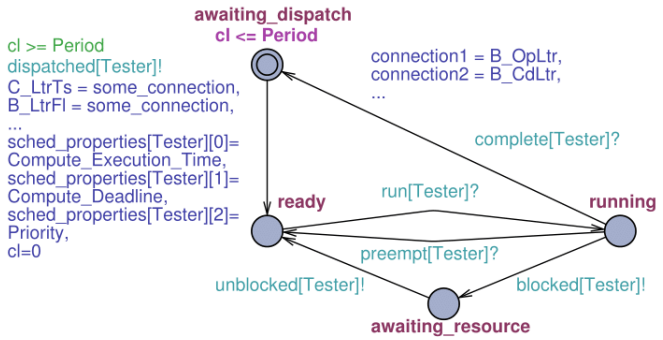

Fig. 5. The thread automaton of *Tester* in Table I. The behavior model of *Tester* has been omitted, which should replace the "running" location.

VI. Observers Generation and Model Checking

Once the UPPAAL model has been generated, the possible paths of the AFG can be verified against their constraints and requirements through reachability analysis. Note that properties that have an effect on the dynamic semantics are transformed into the timed automata model, such as scheduling properties. They are therefore not explicitly included in verification sequences as their validity automatically is verified when the corresponding timed automata paths are executed. Each verification sequence is executed through transformation to an observer automaton [9] and auxiliary variables and clocks (if constrained by timing properties). Observers have been developed to provide a flexible method for specifying coverage criteria for model checking and test case generation. The execution is formulated as a reachability problem, which conforms to our verification criteria. An observer essentially monitors a trace of the timed automata model and reaches an acceptance state whenever the coverage criterion has been met. With respect to verification sequences, reaching an acceptance state denotes flow-reachability of the corresponding AFG path. Thus, reaching all acceptance states imply consistency, correctness, and completeness (assuming all input classes have been covered) of the AADL model. Validity is preserved as observers cannot interfere with the architecture state space.

Formally, an observer $\langle O, o_0, o_{accept}, E_{obs} \rangle$ over a set of auxiliary clocks and variables has a set of observer locations O, an initial observer location $o_0 \in O$, an accepting locations $o_{accept} \in O$, and a set of observer edges E_{obs} on the form $o \xrightarrow{g,a,u} o'$. A coverage criterion is created by dividing it into atomic timed automata items that must be covered and, for each item, generate an observer edge which predicate (g and a) is dependent on that item. An observer edge will thereby be fired when the item has been executed. If the criterion requires the items to be covered in a specific sequence, the edges are structured correspondingly. Moreover, locations may

be labelled with invariants (including urgent and committed) and guards, actions, and clocks may be used to specify additional constraints in which items must be covered. With respect to a verification sequence, the coverage criterion is the corresponding timed automata path. Since each control flow arc in a path corresponds to the firing of one particular edge in the timed automata model, and each data flow arc to a sequence of two edges (one where the variable is defined and one where it is used), the corresponding edges or sequence of edges are the atomic items to be covered. Thus, an observer automaton is created for each verification sequence by creating an observer edge, or a sequence of two edges, for each arc in the path. In addition, each path constraint (property) and requirement is specified through location invariants and transition guards and actions.

Assuming no existence of data flows, a verification sequence of m vertices $\langle v_1 \rightarrow v_2 \rightarrow v_3 \rightarrow \cdots \rightarrow v_m, \{properties\}\rangle$ maps to an observer automaton of $m-1$ observer edges $\langle\{o_1, o_2, o_3, \ldots o_m\}, o_1, o_m, \{o_1 \xrightarrow{g,a,u} o_2, o_2 \xrightarrow{g,a,u} o_3, \ldots, o_{m-1} \xrightarrow{g,a,u} o_m\}\rangle$, where an execution of the edge that corresponds to $v_1 \rightarrow v_2$ is observed by $o_1 \xrightarrow{g,a,u} o_2$, $v_2 \rightarrow v_3$ by $o_2 \xrightarrow{g,a,u} o_3$, etc. If the sequence contains any control flow due to a true or false evaluation of a control expression, it must be complemented with an observer edge that resets the observer to its initial location in case the complementing branch is fired instead. An observer edge primarily observers the coverage item through a broadcast synchronization channel, i.e., given that an execution of $\ell \xrightarrow{g,a_x,u} \ell'$ corresponds to $v_1 \rightarrow v_2$, $\ell \xrightarrow{g,a_x!,u} \ell'$ synchronizes with observer edge $o_1 \xrightarrow{g,a_x?,u} o_2$ through channel a_x. The correspondence between arcs in an AFG and edges in the timed automata model is specified in [7]. Given that there exist an automaton that may stimulate the model with the possible system inputs, the verification sequence is executed by the reachability formula $E <> o_m$ – meaning "there exists one path where o_m eventually holds". The verification sequence passes if the model satisfies the formula.

A verification sequence that contains a data-flow arc requires observer edges that observe at least two successive coverage items: the edge where the data is defined followed by the edge where it is used. In addition, two auxiliary variables, v_{aux1} and v_{aux2}, are used together with the observer edges to ensure that the use-edge actually uses the data instance defined by the definition-edge. The data $Data_{def}$ that is defined at the definition-edge, and the data $Data_{use}$ that is used at the use-edge, are stored in the auxiliary variables. Once the observer edges have observed a definition (through channel $Chan_{def}$) followed by a use (through channel $Chan_{use}$) of the data component, a guard g composed of predicate $v_{aux1} == v_{aux2}$ of an edge $o \xrightarrow{g,a,u} o'$ ensures data flow reachability before the accepting state o' is reached. Nevertheless, in case of inter-component data flows, threads may be modeled with under-sampled data communication (the receiving thread has a lower dispatch frequency than the sending thread) where a fraction of defined data instances are not supposed to reach the use-edge. In such cases, to prevent false negatives of data flow reachability, an alternative definition-observing edge that may synchronize with new definitions of the data component is added. Consequently, for sequences that contain an inter-component data flow arc $v_1 \rightarrow_{d-in} v_2$ or $v_1 \rightarrow_{d-out} v_2$, the two coverage items (def and use) are observed by two sequential observer edges, possibly one for under-sampled communication, and one for assurance of data-flow reachability: $\langle\{o_1, o_2, o_3, o_4\}, o_1, o_4,$
$\{o_1 \xrightarrow{g,Chan_{def}?,\langle v_{aux1}:=Data_{def}\rangle} o_2,$
$o_2 \xrightarrow{g,Chan_{def}?,\langle v_{aux1}:=Data_{def}\rangle} o_2,$
$o_2 \xrightarrow{g,Chan_{use}?,\langle v_{aux2}:=Data_{use}\rangle} o_3, o_3 \xrightarrow{v_{aux1}==v_{aux2},\tau,u} o_4\}\rangle$
where $Chan_{def}?$ observes the definition edge that corresponds to v_1 and $Chan_{use}?$ observes the use edge that corresponds to v_2. For example, the direct component to component path P = "$connection1$:= B_OpLtr" $\rightarrow_{d-in}$ "B_OpLtr := $connection1$" (not under-sampled) corresponds to a verification sequence $\langle P, \{\langle\langle$"$connection1$:= B_OpLtr", "B_OpLtr := $connection1$"$\rangle_{d-in}$, "$Latency => 0ms..1ms$"$\rangle\}\rangle$. Assuming that the time units in the UPPAAL model are milliseconds, the verification sequence is transformed to an observer automaton $\langle\{o_1, o_2, o_{3(committed)}, o_4\}, o_1, o_4,$
$\{o_1 \xrightarrow{g,Chan_{def}?,\langle v_{aux1}:=Data_{def},cl=0\rangle} o_2,$
$o_2 \xrightarrow{g,Chan_{use}?,\langle v_{aux2}:=Data_{use}\rangle} o_3,$
$o_3 \xrightarrow{cl<=1\ and\ v_{aux1}==v_{aux2},\tau,\langle\rangle} o_4\}\rangle$ where $Chan_{def}$ synchronizes with $running \xrightarrow{g_{tst},a_{tst},u_{tst}} awaiting\_dispatch$ of the tester automaton; $Data_{def}$ is a copy of the value assigned to variable $connection1$ in u_{tst}; $Chan_{use}$ synchronizes with $awaiting\_dispatch \xrightarrow{g_{ctrl},a_{ctrl},u_{ctrl}} ready$ of the controller automaton; and $Data_{use}$ is a copy of the value assigned to B_OpLtr in u_{ctrl}. A clock cl is used to verify the validity of the latency property – the data should be received at most after 1ms. For sequences that contain a component-internal data flow arc, the coverage item is decomposed to and observed as the underlying control path.

VII. Model-based Testing

To verify an implementation, its conformance to the – complete, consistent, and correct – model must be tested. A satisfied observer generates a *trace* $\langle\overline{\ell_o}, \overline{\phi_o}, \overline{\sigma_o}\rangle \xrightarrow{a_1/d_1} \langle\overline{\ell_1}, \overline{\phi_1}, \overline{\sigma_1}\rangle \xrightarrow{a_2/d_2} \cdots \xrightarrow{a_n/d_n} \langle\overline{\ell_n}, \overline{\phi_n}, \overline{\sigma_n}\rangle$ that contains information about the initial state of the system and its environment before the path is executed, the input or the sequence of inputs needed to stimulate an execution of the system according to the expected path, and the expected output or sequence of outputs. In addition, the trace holds information on expected non-functional properties, including timing of input and output. Thus, depending on which automata are accredited as an environment $\overline{E}(p_i) = \langle TA_1, TA_2, TA_3, \ldots\rangle$ for a specific verification sequence with path p_i, an observer trace over its $E(p_i)$ yields a test case. Let $MV(p_i)$ and $MAct(p_i)$ denote the sets of variables and actions (on the form $a!$) in environment $E(p_i)$ that are monitored by the system under test (SUT). Let $CV(p_i)$ and $CAct(p_i)$ denote the sets of variables and actions (on the form $a?$) in $E(p_i)$ that are controlled by SUT. For sensor-to-actuator paths, sensor variables and actions are monitored while actuator variables and actions are controlled. Assuming that the SUT at time $t = 0$ is set according to system state $\langle\overline{\ell_o}, \overline{\phi_o}, \overline{\sigma_o}\rangle$, depending on the used test harness, the tester, test script, or test model is responsible of following the trace

such that: 1. for each encountered *delay transition* $\langle \overline{\ell}, \overline{\phi}, \overline{\sigma} \rangle \xrightarrow{d} \langle \overline{\ell}, \overline{\phi} \oplus d, \overline{\sigma} \rangle$, wait until $t = t + d$; 2. for each encountered *discrete transition* $\langle \overline{\ell}, \overline{\phi}, \overline{\sigma} \rangle \xrightarrow{a} \langle \overline{\ell}[\ell'_i/\ell_i, \ell'_j/\ell_j, \ell'_k/\ell_k, \dots], \overline{\phi'}, \overline{\sigma'} \rangle$ where $\ell_{i/j/k\dots} \xrightarrow{g_{i/j/k\dots}, a_{i/j/k\dots}, u_{i/j/k\dots}} \ell'_{i/j/k\dots}$ are edges of the environment $E(p_i)$ and $a_{i/j/k\dots}$ is a member of $MAct(p_i)$ and/or any assignment $u^x_{i/j/k\dots}$ is on the form $v := expr$ such that $v \in MV(p_i)$, stimulate SUT at time t with actions $a_{i/j/k\dots}$ and data updates $u^x_{i/j/k\dots}$; and 3. for each *discrete transition* where $\ell_{i/j/k\dots} \xrightarrow{g_{i/j/k\dots}, a_{i/j/k\dots}, u_{i/j/k\dots}} \ell'_{i/j/k\dots}$ are not edges of the environment $E(p_i)$ and $a_{i/j/k\dots}$ is a member of $CAct(p_i)$ and/or any assignment $u^x_{i/j/k\dots}$ is on the form $v := expr$ such that $v \in CV(p_i)$, assure at time t that SUT responds with actions $a_{i/j/k\dots}$ and data updates $u^x_{i/j/k\dots}$. The collective set of generated tests creates a test suite that tests the conformance of the implementation with respect to the architecture model.

VIII. Selective Regression Verification

Given a model M, and possibly an implementation of the model $IMPL$, for which a verification suite VS of verification sequences has been generated and executed on M as described in Section VI, and on $IMPL$ as described in Section VII, it is likely that M eventually is modified into another version M', which later may be modified into another version M'', and so forth. A conventional approach to regression verification would be to ensure that a modification has not introduced faults in the model M' and has not violated the conformance with the implementation $IMPL$ by (1) re-executing all "old" but still valid verification sequences $VS'_{old} \subseteq VS$ on M' and $IMPL$, and, if the modification includes an added functionality, behavior, or property, (2) generate a new verification suite VS'_{new} that covers the added part(s) and execute it. A modification corresponds to the set of expressions (vertices) and flows (arcs) that are different among the models, i.e., expressions that exist in one version but not in the other. However, re-execution of all verification sequences is inefficient if the modification does not affect the complete architecture. Nevertheless, determining which ones that still are valid and the new sequences that are necessary to cover new parts is difficult. The problem is that the impact of a modification on the remaining architecture is complex to manually trace. In order to perform regression verification efficiently, we contribute with a technique that selectively re-executes only those verification sequences that are affected by the modification and generates new verification sequences that only cover added parts. The technique uses the concept of slicing through extended system dependence graphs [6], which we hereafter call architecture dependence graphs (ADGs), to exactly identify the parts of a modified AADL model that directly or indirectly are affected by the modification and must be covered by verification sequences in the regression verification process. The concept of slicing is to remove expressions that do not have an effect on and are not affected by the value of a variable at some expression. An ADG provides these dependencies such that causality can be precisely traced and is generated from an AFG through dominance analysis, as defined in [7]. The approach is to apply this idea to variables of the changed or added part such that other parts of the model which behavior now might be faulty are identified for regression verification.

The first step is to determine what expressions, or flows to an expression, that have been removed or changed or added. This is simply done by comparing AFG' of M' with AFG of M to determine the set of removed vertices and arcs $AFG \backslash AFG'$ and the set of added or changed vertices and arcs $AFG' \backslash AFG$. VS'_{old} is thereby easily computed: any $vs \in VS$ that covers a vertex or arc in $(AFG' \backslash AFG) \cup (AFG \backslash AFG')$ is no longer valid. Invalid verification sequences are discarded in the regression verification process if the corresponding architectural paths are removed by the modification. If the paths still exist, the verification sequences are updated according to the modification to become valid. Nevertheless, valid verification sequences that do not cover the modification or affected parts are unnecessary to re-execute on M' or $IMPL$. Affected parts are determined through forward-slicing of the ADG'. An $ADG(M) = \langle V, A \rangle$ of a model is a directed graph of the set of vertices $V \subseteq AFG(M)$ and arcs $A \subseteq V \times V$ representing control and data dependencies. A forward slice $fSlice(Cri)$ with respect to a slicing criterion $Cri = \langle v, var \rangle$, where v is a vertex and $var \in v$ is a variable or data component defined or read at v, consists of all vertices of the model that possibly are dependent on the value of var at v. This corresponds to all vertices that are forward-reachable (through dependence arcs) from v in the ADG of the model. The set of affected vertices $V'_{aff} \subseteq V'$ is thereby determined by, for each $v_x \in AFG \backslash AFG' \cup AFG \backslash AFG'$, and for each defined or read variable var_y in v_x, compute $fSlice(\langle v_x, var_y \rangle)$ of $ADG'(M')$. The regression verification suite VS'_{old} is subsequently efficiently executed by only selecting verification sequences that cover vertices in V'_{aff}.

The set of affected vertices with respect to old verification sequences may be further trimmed by means of the inter-observer coverage data (the trace that satisfies an observer may have satisfied others as well) from the preceding verification. From the data, satisfiability independence between observers may be deduced, which adds a layer of dynamic slicing that regards cyclic dependencies of the system. If an observer obs_1 was satisfied without satisfying another observer obs_2, then obs_1 is satisfiable independently from the path observed by obs_2 even though the path, at a later stage, may generate an erroneous behavior of the path observed by obs_1 (as possibly predicted by static forward slicing). Consequently, a previously satisfied observer which satisfiability is independent to each observer that covers the modification will also be satisfiable in the regression verification process (and therefore unnecessary to re-execute) even if it covers a vertex in the forward slice.

Finally, changed or added vertices and arcs that generates new paths must be covered with new verification sequences. VS'_{new} is generated by applying the verification criteria to the changed and added set $AFG' \backslash AFG$, from which the possible new paths and corresponding set of verification sequences are extracted. If yet another version M'' is developed, the regression verification process is repeated upon the verification history $VS' = VS'_{old} \cup VS'_{new}$, instead of VS.

IX. Validation

By means of LTRIS we conducted a case study with the objective of validating the effectiveness and efficiency of AQAF. To adequately validate these properties, the application of AQAF on LTRIS should cover the different types of

TABLE II. CASE STUDY RESULTS

		Model checking and sel. regr. ver. effectiveness			TOT time cons.(sec)		TOT mem. cons.(MB)		Sel. efficiency		Testing effectiveness
Fault	**No. V-seqs**	**No. sel.**	**No. unsat. sel. Obs.**	**No. unsat. Obs.**	**Sel.**	**All**	**Sel.**	**All**	**Time**	**Mem.**	**No. failed TCs (of 57)**
n/a	57	n/a	n/a	0	n/a	855	n/a	9327	n/a	n/a	0
1	57	4	4	4	196	504	3236	8085	61%	60%	5
2	57	13	7	7	364	527	5768	8728	31%	34%	6
3	49	25	9	9	373	396	5745	7127	6%	19%	25
4	57	30	18	18	1042	1204	15367	17942	13%	14%	18
5	57	24	7	7	235	403	3404	6832	42%	50%	0
6	57	n/a	n/a	20	n/a	492	n/a	7470	0%	0%	7
7	58	50	42	42	51	58	2190	2542	12%	14%	43

faults that may exist in an architecture. The framework must therefore be systematically applied to controlled versions of LTRIS to ensure coverage of fault types and to delimit the probability of false positives and false negatives. Our approach to a systematic application uses the method of fault injection and involves two stages. The first stage is to perform model checking and model-based testing based on a (presumed) fault-free version of the LTRIS model. If the techniques are valid and the architecture implementation truly conforms to the model, the result must necessarily be satisfied observers (No. unsat. Obs. = 0) and passed test cases (No. failed TCs = 0). Since the model certainly conforms to itself, it is treated as the implementation when applying model-based testing. The second stage of the approach is to create mutated versions of the AADL model, each of which containing an injected fault, and extend the steps performed in the first stage such that the application covers the complete framework and ranges over the possible fault types. By means of the artifacts produced in the first stage, each fault injection corresponds to a modification. If the selective regression verification and model checking techniques are valid, the result of regression verification must necessarily be at least one unsatisfied selected observer per modification (No. unsat. sel. Obs. $>$ 0) since there now definitely exist a fault. The selective approach is contrasted with a re-run all approach to assess the selection effectiveness and efficiency. If valid, the result must necessarily be satisfied observers for all non-selected verification sequences (No. unsat. sel. Obs. = No. unsat. Obs.) since the impact analysis is expected to select all verification sequences that possibly are affected by the modification. The required overhead expense of conducting the selection must either not exceed the savings with respect to the cost of a re-run all approach to be efficient (TOT time/mem. cons.(sel.) $<$ TOT time/mem. cons.(all)). Furthermore, each mutated version may be treated as an implementation to validate the effectiveness of the test suite generated in the first stage. If the testing technique is valid, the result must necessarily be at least one failed test case for each tested mutation (No. failed TCs $>$ 0).

Based on this study design, effectiveness of model checking and model-based testing is measured in terms of the ratio of found faults to the number of injected faults. Effectiveness of selective regression verification is measured in terms of the ratio of unsatisfied non-selected observers to the number of selected and non-selected unsatisfied observers. Note that a lower ratio denotes a higher effectiveness in the latter case. A ratio of zero means that the technique did not exclude any verification sequence that reveals a fault in the modified architecture. Efficiency is measured in terms of time and memory consumption. The resource consuming activities of model checking are observers generation (including AFG generation and verification sequences extraction), AADL to timed automata transformation, and satisfiability checking. For model-based testing, the resource consuming activities are identical except that the test data is extracted by searching the resultant traces. In either case, the bottleneck is satisfiability checking of observers due to the state space explosion problem, which is the resource consuming activity of interest in this case study. With respect to efficiency of the selective verification technique, the resource consumption of slicing is included to make sure that the overhead the selective regression verification technique brings does not exceed the savings.

The results of the case study with respect to the following considered fault types are presented Table II: (1) absent, unachievable, and incorrect control expressions (guards); (2) absent and incorrect data assignments, events, and calls (actions); (3) absent and incorrect port connections; (4) absent and incorrect parameter connections; (5) absent, incorrect, and incompatible non-functional properties; (6) absent, incorrect, and incompatible protocols or use of shared resources (deadlock, livelock, starvation, and priority inversion of threads); (7) absent, incorrect, and incompatible scheduling properties. The verification was performed in Windows 7 64-bit edition running an Intel Core i7-3667U 2.0 GHz CPU with 8 GB RAM. The results conform to the expectations except in two cases. First, fault No. five was not detected by the test suite. In retrospect, the result is not a surprise as the fault is an inconsistent latency property, which in the model does not affect the execution but impose an analysis constraint on it. Thus, it is not sound to treat the faulty model as a faulty implementation in this case, since the inconsistent property must be manifested in the execution to be a realistic implementation fault. Second, fault No. 6 corresponds to a changed scheduling property which has no relation to the AFG, consequently, no slicing can be performed. On average, it took 555 seconds and 8507 MB to check satisfiability of 56 observers. Seven out of seven design faults were detected and, by disregarding fault five at the implementation-level, six out of six implementation faults. The selective approach, on average, reduced the time and memory consumption of regression verification by 24% and 27% respectively. No verification sequence that reveals a fault in the modified design was unselected.

X. RELATED WORK

A lot of research has been conducted in the area of formal analysis of AADL models and integrations thereof, however, no notable contribution of conformance testing or regression verification techniques for AADL have been recognized. Esteve et al. [11] present the usage of COMPASS in the development of a satellite platform. COMPASS is a toolset for SLIM, a variant of AADL, and provides the ability to model-check functional properties through transformation

to Markov chain and assess dependability through automated generation and analysis of fault trees and FMEA tables. Model checking of both nominal behavior and error behavior is conducted in the case study, where the resultant average time and memory consumptions for verification of 19 properties are 508 sec and 469 MB. It is not evident if, and in that case how, concurrent execution is considered and thereby comparable to model checking capacities of AQAF. Murugesan et al. [12] propose an approach to compositional verification of AADL models against requirements expressed in past-time linear temporal logic (PLTL) by means of AGREE – an AADL model-checker. The approach is compositional in the sense that component-level behavior is described in Simulink and verified by the Simulink Design Verifier. A case study of a medical device resulted in a model checking time consumption of 273 seconds to prove 35 properties. The verification technique is however limited with respect to non-functional properties as concurrent execution, shared resources, scheduling, and timing properties are not considered. Björnander et al. [13] implements a denotational semantics of a subset of AADL in standard ML for the purpose of checking AADL models against CTL properties. The subset does not include execution semantics of threads, where the architecture analysis is restricted to abstract system components. A mapping of AADL to Petri Nets is presented by Renault et al. [14], where the objective is to verify that the system is free from deadlocks and that defined data interactions behave correctly. Based on the presented principles, the approach only supports event-driven architectures without preemption and does not include timing properties in the verification. Chkouri et al. [15] define a translation to BIP (Behavior Interaction Priority) to enable model checking of deadlock-freedom and requirements. As in AQAF, properties are checked by means of observers. It is not evident to which degree concurrency is supported as information on how scheduling properties and preemption are captured in BIP is not given or referenced. Berthomieu et al. [16] presents a transformation of AADL models with fixed-priority scheduling into TTS through an intermediate Fiacre model, which in turn can be checked by the Tina toolbox. The work is limited to non-preemptive scheduling.

XI. Conclusion

In this paper, we presented AQAF: an Architecture Quality Assurance Framework covering virtually the entire life cycle of critical embedded systems. The framework has been developed due to the limited life-cycle coverage of existing verification techniques. AQAF provides a set of techniques for a wide coverage and a joint formalism and semantic domain, architecture flow graphs and timed automata, that allow for an efficient, effective, and fully automatable integration. An industrial safety-critical train control system is used to demonstrate its application in practice and validate effectiveness and efficiency. First, we presented the architecture-based verification criteria and showed how AADL verification data can be represented by AFGs. Second, we presented an overview of how AADL models can be transformed into a network of Uppaal timed automata such that they can be subjected to model checking and model-based testing. Third, we described the process of transforming verification data from AFGs, in the form of verification sequences, to observer automata. Observers drive both the model checking process and the test case generation process through reachability analysis. Satisfied observers indicate a complete, consistent, and correct model whereas passed tests indicate conformance to the model. At last, we presented how selective regression verification can be performed through slicing of architecture dependency graphs generated from AFGs. Results indicate that AQAF is effective in finding the considered types of architectural faults and that the necessary resource consumption is within acceptable limits when contrasted with related work. The results also suggest that the selective approach may significantly reduce the resource consumption of regression verification.

Acknowledgments

This research is supported by the Swedish Foundation for Strategic Research (SSF) project Synopsis.

References

[1] N. G. Leveson, *Engineering a Safer World: Systems Thinking Applied to Safety (Engineering Systems)*, 2012.

[2] As-2 Embedded Computing Systems Committee SAE, "Architecture Analysis & Design Language (AADL)," SAE Standards, 2009.

[3] A. Johnsen, K. Lundqvist, P. Pettersson, and O. Jaradat, "Automated Verification of AADL-Specifications Using UPPAAL," *Ninth IEEE International Symposium on High-Assurance Systems Engineering*, 2012.

[4] I. O. for Standardization, "ISO 26262-1:2011 Road vehicles - Functional safety."

[5] G. Behrmann, A. David, and K. G. Larsen, "A tutorial on Uppaal," 2004.

[6] A. Johnsen, K. Lundqvist, P. Pettersson, and K. Hänninen, "Regression Verification of AADL Models through Slicing of System Dependence Graphs," in *Tenth International ACM Sigsoft Conference on the Quality of Software Architectures*. ACM, June 2014.

[7] A. Johnsen, K. Lundqvist, P. Pettersson, K. Hänninen, and M. Torelm, "Empirical Validation of the Architecture Quality Assurance Framework (AQAF): A Technical Report," Tech. Rep. [Online]. Available: http://www.es.mdh.se/publications/4268-

[8] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs," in *Proceedings of the conference on Programming Language design and Implementation*, 1988.

[9] J. Blom, A. Hessel, B. Jonsson, and P. Pettersson, "Specifying and Generating Test Cases Using Observer Automata," in *Proceedings of the 4th International Workshop on Formal Approaches to Testing of Software*, 2005.

[10] J. Chang and D. J. Richardson, "Static and dynamic specification slicing," in *Proceedings of the Fourth Irvine Software Symposium*, 1994.

[11] M.-A. Esteve, J.-P. Katoen, V. Y. Nguyen, B. Postma, and Y. Yushtein, "Formal Correctness, Safety, Dependability, and Performance Analysis of a Satellite," in *Proceedings of the 34th International Conference on Software Engineering*, 2012.

[12] A. Murugesan, M. W. Whalen, S. Rayadurgam, and M. P. Heimdahl, "Compositional Verification of a Medical Device System," in *Proceedings of the 2013 Conference on High Integrity Language Technology*, 2013.

[13] S. Björnander, C. Seceleanu, K. Lundqvist, and P. Pettersson, "A Formal Analysis Framework for AADL," 2011.

[14] X. Renault, F. Kordon, and J. Hugues, "From AADL Architectural Models to Petri Nets: Checking Model Viability," in *Proceedings of the International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, 2009.

[15] M. Y. Chkouri, A. Robert, M. Bozga, and J. Sifakis, "Models in Software Engineering," 2009, ch. Translating AADL into BIP - Application to the Verification of Real-Time Systems.

[16] B. Berthomieu, J.-P. Bodeveix, C. Chaudet, S. Zilio, M. Filali, and F. Vernadat, "Formal Verification of AADL Specifications in the Topcased Environment," in *Proceedings of the 14th Ada-Europe International Conference on Reliable Software Technologies*, 2009.