AQAT: The Architecture Quality Assurance Tool for Critical Embedded Systems

Andreas Johnsen, Kristina Lundqvist, Kaj Hänninen, Paul Pettersson
School of Innovation, Design and Engineering, Mälardalen University, Västerås, Sweden
andreas.johnsen@mdh.se

***Abstract*—Architectural engineering of embedded systems comprehensively affects both the development processes and the abilities of the systems. Verification of architectural engineering is consequently essential in the development of safety- and mission-critical embedded system to avoid costly and hazardous faults. In this paper, we present the Architecture Quality Assurance Tool (AQAT), an application program developed to provide a holistic, formal, and automatic verification process for architectural engineering of critical embedded systems. AQAT includes architectural model checking, model-based testing, and selective regression verification features to effectively and efficiently detect design faults, implementation faults, and faults created by maintenance modifications. Furthermore, the tool includes a feature that analyzes architectural dependencies, which in addition to providing essential information for impact analyzes of architectural design changes may be used for hazard analysis, such as the identification of potential error propagations, common cause failures, and single point failures. Overviews of both the graphical user interface and the back-end processes of AQAT are presented with a sensor-to-actuator system example.**

Keywords—verification tool; model checking; model-based testing; dependence analysis; regression verification

I. Introduction

Architectural engineering of safety-critical and mission-critical embedded systems is conducted throughout the development process, where faults may be created when the architecture is designed, when the architectural design is implemented, and every time the design is modified due to maintenance. Architectural faults tend to significantly impair the cost and performance of development processes [1] [2] [3] and the dependability of the systems [4] [5]. Rigorous and holistic verification of architectural engineering is consequently essential in the development of safety- and mission-critical embedded systems, from requirements analysis and design to implementation and maintenance. Furthermore, automated verification is essential to reduce the cost of labor and the risk of human error [6]. In this paper, we present the Architecture Quality Assurance Tool (AQAT), an application program developed to provide a holistic, formal, and automated verification process for architectural engineering of critical embedded systems. AQAT corresponds to an implementation of the Architecture Quality Assurance Framework (AQAF) [7], which includes a model checking technique to detect design faults, a model-based testing technique to detect implementation faults, and a selective regression verification technique based on a change impact analysis technique to efficiently detect faults created by maintenance modifications. The verification criteria of the framework enforce assessments of architectural control and data flow paths and their compliance with requirements. The contribution is of industrial importance as contemporary functional safety standards (e.g. ISO 26262) require control and data flow analysis of the architectural design, tests that demonstrate conformance of the implementation with respect to the design, and impact analysis of design changes to identify the necessary reverification measures.

The provided verification techniques are developed upon a common formal foundation constituting a combination of timed automata [8] and architecture flow graphs [7]. Architecture flow graphs identify the prescribed control and data flows of architectural models that must be subjected to verification and also provide control and data dependencies on which change impact analysis automatically can be performed through slicing [9]. The theory of timed automata is used to anchor the semantics of an architectural model in a format appropriate for model checking and model-based test case generation. A common formal underpinning provides traceability between the verification runs, the coverage of the model, and the coverage of the implementation. Regression verification can thereby be efficiently executed by only selecting verification runs of the model and implementation that can be traced from the change impact analysis. AQAT has mainly been developed for verification of architectures with synchronous, fixed-priority preemptive or non-preemptive execution models, as these commonly are used in critical embedded systems. Principles of modularization are implemented to facilitate extensions. Furthermore, AQAT is currently only compatible with architectural models described by the Architecture Analysis and Design Language (AADL) [10], but may be adapted to other languages with a similar expressiveness. Research in this field has developed a number of verification tools for AADL, such as model checking tools (e.g. [11] [12] [13] [14] [15]) and resource scheduling analysis tools (e.g. [16]). However, these tools do not provide verification techniques for architectural engineering that is conducted subsequent to an established design. Moreover, they do not include methods that enforce and measure coverage of the architectural design in the verification process, which is essential to determine the extent to which a design has been verified.

The paper is organized as follows. In section II, we present the front-end of AQAT together with the three main cases in which the tool is intended to be used. In Section III, we present an overview of the back-end processes of AQAT. A list of performance evaluation results is then presented in Section IV, which is followed by related work in Section V, and finally concluding remarks and future work in Section VI.

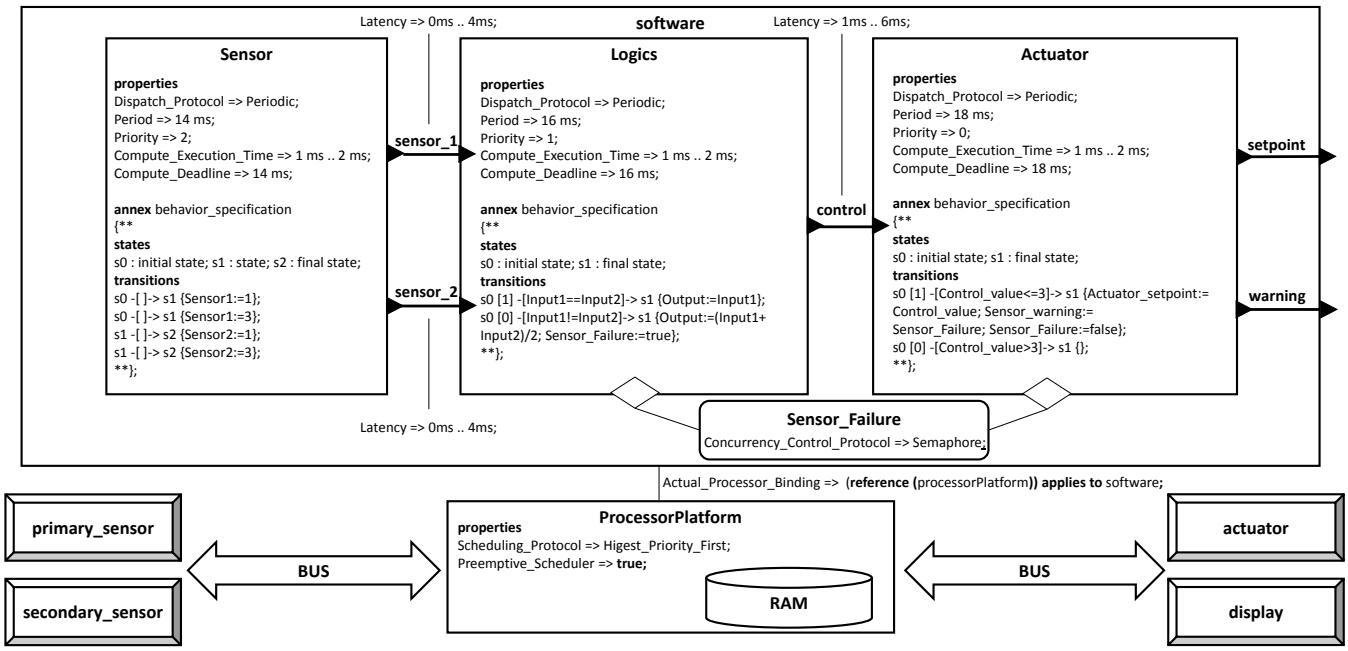


Fig. 1: Running example: a sensors-to-actuator AADL model (the textual and graphical syntax has been simplified).

II. AQAT Front-end and Usage

A. Use Cases

Architectural design and modeling is conducted in the early phases of the development process of embedded systems to create and represent system structures that generate the required extra-functional properties, such that the requirements of safety, reliability, availability, performance, etc., are achieved [17]. A faulty architectural design may therefore not only cause an erroneous behavior of critical functionality, but also of redundancy and fault tolerance mechanisms that are supposed to maintain dependability in the presence of errors. Moreover, architectural design faults tend to cause extensive rework costs [1]. They are consequently critical to detect and correct prior to any refinement or implementation of the architectural design. This constitutes the chronologically first case in which AQAT may be used in the development process. AQAT provides an architectural model checking feature and an architectural simulation feature by means of the Uppaal environment [8] for the detection and debugging of faults within the design. As a running example, we will use the simplistic sensor-to-actuator AADL model presented in Fig. 1 (the reader is referred to Section III-A for a description of the AADL syntax). The design is composed of three concurrent and periodically dispatched tasks: *Sensor*, *Logics*, and *Actuator*. *Sensor* represents the behavior of a dual modular redundant sensor that periodically outputs two integer values. The output values are transmitted to *Logics* through connections *sensor_1* and *sensor_2*. *Logics* then controls the position of an actuator device based on these values, where the mean of the two sensor values is used if they differ, whereas the value from the primary sensor is used if they are considered as equal. In addition to the computation of the mean if the sensor values differ, the shared data component *Sensor_Failure* is set to the Boolean value *true*. *Actuator* finally acts as an interface to the controlled actuator. If the control signal is below or equal to the threshold of three, *Actuator* positions the device according to the request and displays the state of *Sensor_Failure*. If the signal is higher than the threshold, no values shall be assigned to output interfaces. In addition to the modeled components, interfaces, connections, and scheduling properties, the architecture is modeled with latency properties that constraint the time window in which the interactions should take place to achieve a safe regulation of the actuator device. The tasks are bound to a processing platform with a fixed-priority preemptive scheduler.

Any element of the design in Fig. 1 that causes an incorrect, inconsistent, or incomplete execution of the prescribed control and data flow paths (from the sensors to the actuator), such as incorrect component interactions, inconsistent timing and scheduling properties, excessive response times, missed deadlines, unsatisfiable control expressions, unreachable behavior, and deadlocks, livelocks, and starvations of threads due to misuses of shared resources, can be detected by AQAT [18].

Although a correct, complete, and consistent architectural design is important in the development of critical embedded systems, these verification measures do not imply a corresponding implementation as faults may be created in the process of implementing the design. This constitutes the second case in which AQAT may be used in the development process. AQAT provides a model-based test suite generation feature for the generation of test cases that test the conformance of the implementation with respect to the design. Test cases are generated based on the model checking process, where the input and output behavior of each model-checked control and data flow path is converted to a test case that attempts to observe a corresponding behavior in the implementation.

Finally, the system lifecycle typically includes modifications to the architectural design due to maintenance, product line and variability development, and tradeoff analysis. Modifications require reverification measures since a modification may induce an erroneous behavior to previously functioning ar-

chitectural elements. Reverification of the complete design, i.e. a re-run all approach, is inefficient if the change does not have the corresponding comprehensive effect. This corresponds to the third case in which AQAT may be used in the development process. AQAT provides a selective regression verification feature that only selects those verification sequences that cover a changed or possibly impacted part of the modified design for reverification. The feature assesses the impact of a change through an analysis of dependencies between components of the architecture. The dependencies may, in addition to impact analysis, be used for hazard analysis, where potential fault propagations, common cause failures, single point failures, etc., may be deduced from the information. Furthermore, the information may be used for reusability analysis, parallelization of independent functions, and for the analysis of potential dependencies between critical and non-critical components in mixed criticality systems.

B. User Interface

AQAT is controlled through a graphical user interface (GUI). An AADL model is verified by first creating a new AQAT project through File→New Project in the menu bar, as illustrated in Fig. 5. The user then selects the AADL model to be verified, and possibly a previously saved AQAT project pertaining to the verification of a prior version of the model. The selective regression verification feature is engaged by the latter action.

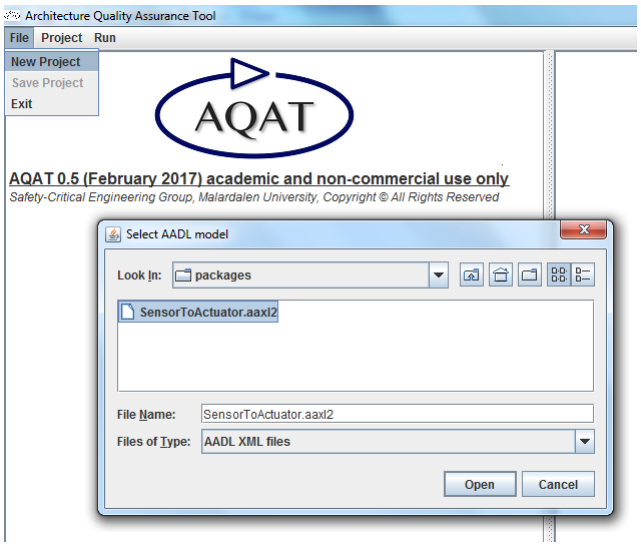


Fig. 2: GUI: creation of a new verification project.

The user is subsequently given the option to configure the verification process according to case-specific needs, as presented in Fig. 3. By default, the tool will not perform verification of, or generate test cases from, control and data flow paths that include component-internal or inter-component loops. If the user needs to include potential loops in the analysis, the user may set the maximum number of loops the tool should consider. These bounds do not have any effect on the verification of the sensor-to-actuator example since it is free from loops. The user may finally request a schedulability analysis, a test suite generation, and/or a record of architectural dependencies to enable a selective regression verification process in a possible future reverification project. A successful verification of control and data flow paths implies that the model is schedulable during the execution of the verified paths. However, in order to ensure that the tasks do not miss their deadlines over one hyperperiod, an explicit schedulability analysis must be conducted. Furthermore, in case of an unsuccessful verification of control and data flow paths, an explicit schedulability analysis facilitates the debugging process by declaring the presence or absence of missed deadlines.

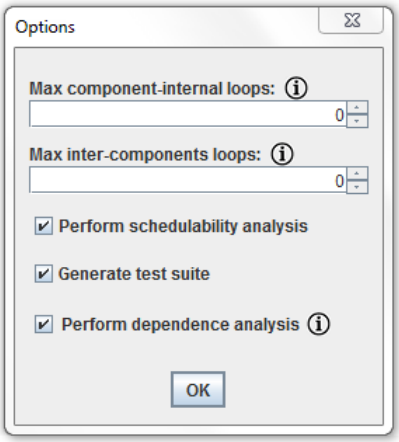


Fig. 3: GUI: selection of verification options.

If a test suite generation is requested, a dialogue box is displayed wherein the user is asked to mark the components of the model that represent the possible behaviors of the system environment, as presented in Fig. 4. The information is necessary for the tool to identify the input interfaces of the model that will constitute the controlled interfaces of the system under test. Test cases shall stimulate the implementation in place of *Sensor* in this case.

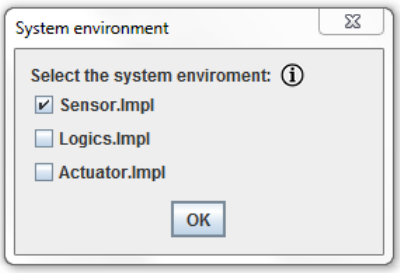


Fig. 4: GUI: selection of system environment.

The tool initiates the verification process when the options have been selected. The progress of the process is displayed in real time through the main window of the GUI, as illustrated in Fig. 5. The left-hand window pane displays the status of major framework processes that are executing and their key results. Verdicts from the model checking process are displayed through green, red, and a yellow symbols, depending on if a path is executable (green circle), unexecutable (red rectangle), or if the executability is inconclusive (yellow circle).

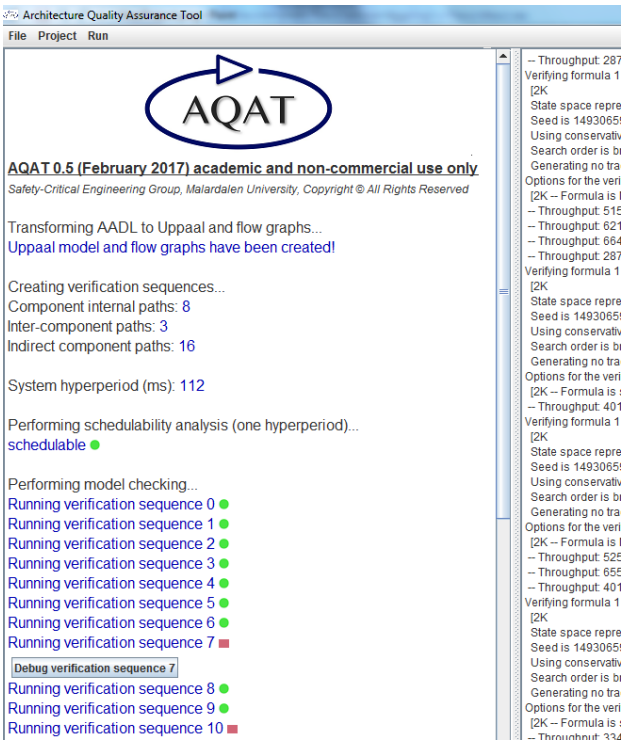
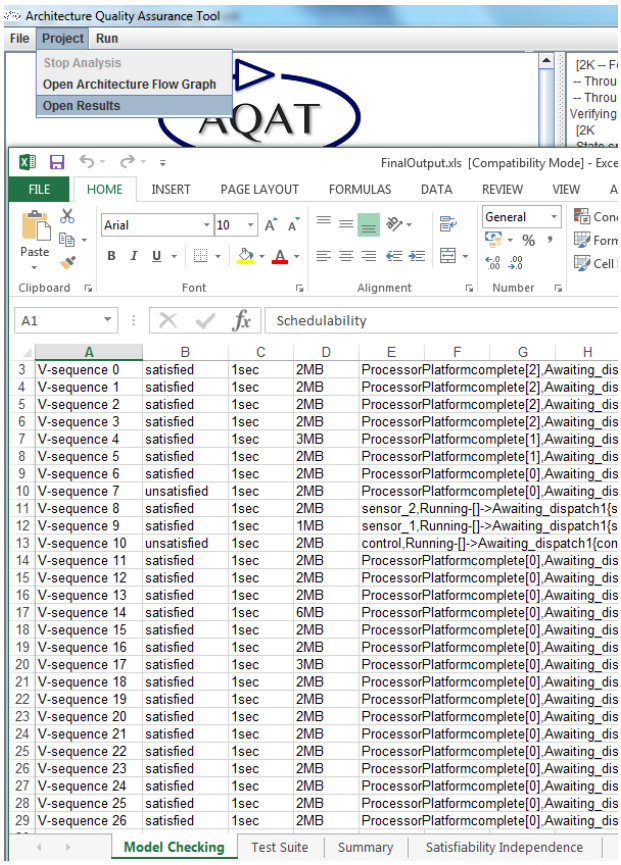


Fig. 5: GUI: main window.



	A	B	C	D	E
3	V-sequence 0	satisfied	1sec	2MB	ProcessorPlatformcomplete[2],Awaiting_dis
4	V-sequence 1	satisfied	1sec	2MB	ProcessorPlatformcomplete[2],Awaiting_dis
5	V-sequence 2	satisfied	1sec	2MB	ProcessorPlatformcomplete[2],Awaiting_dis
6	V-sequence 3	satisfied	1sec	2MB	ProcessorPlatformcomplete[2],Awaiting_dis
7	V-sequence 4	satisfied	1sec	3MB	ProcessorPlatformcomplete[1],Awaiting_dis
8	V-sequence 5	satisfied	1sec	2MB	ProcessorPlatformcomplete[1],Awaiting_dis
9	V-sequence 6	satisfied	1sec	2MB	ProcessorPlatformcomplete[0],Awaiting_dis
10	V-sequence 7	unsatisfied	1sec	2MB	ProcessorPlatformcomplete[0],Awaiting_dis
11	V-sequence 8	satisfied	1sec	2MB	sensor_2,Running-[]->Awaiting_dispatch1{s
12	V-sequence 9	satisfied	1sec	1MB	sensor_1,Running-[]->Awaiting_dispatch1{s
13	V-sequence 10	unsatisfied	1sec	2MB	control,Running-[]->Awaiting_dispatch1{con
14	V-sequence 11	satisfied	1sec	2MB	ProcessorPlatformcomplete[0],Awaiting_dis
15	V-sequence 12	satisfied	1sec	2MB	ProcessorPlatformcomplete[0],Awaiting_dis
16	V-sequence 13	satisfied	1sec	2MB	ProcessorPlatformcomplete[0],Awaiting_dis
17	V-sequence 14	satisfied	1sec	6MB	ProcessorPlatformcomplete[0],Awaiting_dis
18	V-sequence 15	satisfied	1sec	2MB	ProcessorPlatformcomplete[0],Awaiting_dis
19	V-sequence 16	satisfied	1sec	2MB	ProcessorPlatformcomplete[0],Awaiting_dis
20	V-sequence 17	satisfied	1sec	3MB	ProcessorPlatformcomplete[0],Awaiting_dis
21	V-sequence 18	satisfied	1sec	2MB	ProcessorPlatformcomplete[0],Awaiting_dis
22	V-sequence 19	satisfied	1sec	2MB	ProcessorPlatformcomplete[0],Awaiting_dis
23	V-sequence 20	satisfied	1sec	2MB	ProcessorPlatformcomplete[0],Awaiting_dis
24	V-sequence 21	satisfied	1sec	2MB	ProcessorPlatformcomplete[0],Awaiting_dis
25	V-sequence 22	satisfied	1sec	2MB	ProcessorPlatformcomplete[0],Awaiting_dis
26	V-sequence 23	satisfied	1sec	2MB	ProcessorPlatformcomplete[0],Awaiting_dis
27	V-sequence 24	satisfied	1sec	2MB	ProcessorPlatformcomplete[0],Awaiting_dis
28	V-sequence 25	satisfied	1sec	2MB	ProcessorPlatformcomplete[0],Awaiting_dis
29	V-sequence 26	satisfied	1sec	2MB	ProcessorPlatformcomplete[0],Awaiting_dis

Fig. 6: GUI: open results.

An inconclusive verdict indicates that the model-checker ran out of memory in the process of verifying the executability of the path. The right-hand window pane displays details from back-end processes and messages that are sent from the model-checker.

Subsequent to the completion of the model checking process, and the test suite generation and dependence analysis processes if selected, the user may open a comprehensive description of the project and the results through the menu bar, as illustrated in Fig. 6. Detailed examples of results based on the sensor-to-actuator model are presented in Section III. The user may also open the AADL model within the UPPAAL environment as illustrated in Fig. 7, wherein its behavior may be simulated, inspected, and subjected to customized model checking. Customized model checking is especially useful for debugging, where context-specific requirements can be specified and verified to analyze the cause and extent of the erroneous behavior. In case of an unexecutable path verdict, the tool displays a path-specific launch button together with the warning, as shown in Fig. 5, through which the particular faulty behavior can be debugged within the UPPAAL environment.

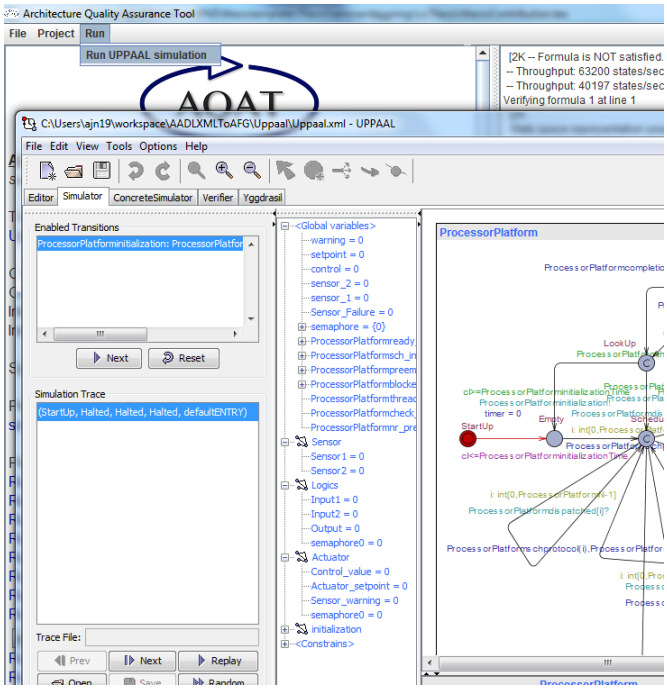


Fig. 7: GUI: open AADL model in UPPAAL.

III. AQAT BACK-END

AQAT executes a framework of back-end processes, illustrated in Fig. 8, to provide a holistic verification process. In this section, we present an overview of these processes and their implementations within the tool. Theoretical details are presented in [7].

A. Architecture Analysis and Design Language

AQAT operates on AADL XML files and utilizes the Java package *javax.xml.parsers* for the preparation of the transformations to architecture flow graphs (AFGs) and UPPAAL timed automata. In this section, we present an overview of AADL in this form and the preparation, whereas the transformation

TABLE I: AADL XML Schema

```
A1:  <SystemType name=" ... "/><SystemImplementation name=" ... ">
A2:    <ownedProcessorSubcomponent name=" ... " processorSubcomponentType="xmi:id of comp. impl."/> ... // Protocol=>FIXED_PRIORITY Preemptive=>true|false
A3:    <ownedProcessSubcomponent name=" ... " processSubcomponentType="xmi:id of comp. impl."/> ...
A4:    <ownedPropertyAssociation property="../../...aadl#Deployment_Properties::Actual_Processor_Binding"> ... </ownedPropertyAssociation > ...
A5:    <ownedAccessConnection name=" ... " ... ><destination ... /><source ... /></ownedAccessConnection> ...
A6:    <owned(Port/Parameter)Connection name=" ... "><ownedPropertyAssociation property="../../.aadl#Communication_Properties::Latency" > ...
A7:    </ownedPropertyAssociation><destination ... /><source ... /></ownedPortConnection> ... </SystemImplementation>
A8:  <ProcessType name=" ... "><owned(Data/Event/EventData)Port name=" ... " direction="in|out|inout" dataFeatureClassifier="reference to data type" /> ...
A9:    <ownedDataAccess name=" ... " kind="requires|provides" dataFeatureClassifier="reference to data type" /> ... </ProcessType> ...
A10: <ProcessImplementation name=" ... "><owned(Thread/Subprogram/Data)Subcomponent ... /> ... <ownedAccessConnection ... > ... ...
A11:   <owned(Port/Parameter)Connection ... > ... </ProcessImplementation> ...
A12: <ThredType name=" ... "><ownedPropertyAssociation property="Dispatch_Protocol" value="Periodic" />
A13:   <ownedPropertyAssociation property="Period" unit="Time_Units.ms/sec/min/..." value=" ... " /><ownedPropertyAssociation property="Priority" value=" ... " />
A14:   <ownedPropertyAssociation property="Compute_Execution_Time" ><minimum unit="Time_Units.ms/sec/min/..." value=" ... "/>
A15:   <maximum unit="Time_Units.ms/sec/min/..." value=" ... "/></ownedPropertyAssociation>
A16:   <ownedPropertyAssociation property="Compute_Deadline" unit="Time_Units.ms/sec/min/..." value=" ... " />
A17:   <owned(Data/Event/EventData)Port ... /> ... <ownedDataAccess ... /> ... </ThreadType> ...
A18: <ThreadImplementation name=" ... "><ownedAnnexSubclause name="behavior_specification" sourceText="variables variable_name : reference to data type; ...
A19:   states textitstate_name : initial|complete|final|state; ... transitions source_state [priority] -[guard]-> destination_state {action1;action2 ... }; ... "/>
A20:   <owned(Subprogram/Data)Subcomponent ... "/> ...
A21:   <ownedAccessConnection ... > ... <owned(Port/Parameter)Connection ... > ... </ThreadImplementation> ...
A22: <SubprogramType name=" ... "><ownedParameter ... /> ... <ownedDataAccess ... /> ... </SubprogramType> ...
A23: <SubprogramImplementation name=" ... "><ownedAnnexSubclause ... /><ownedDataSubcomponent ... /> ... </SubprogramImplementation> ...
```

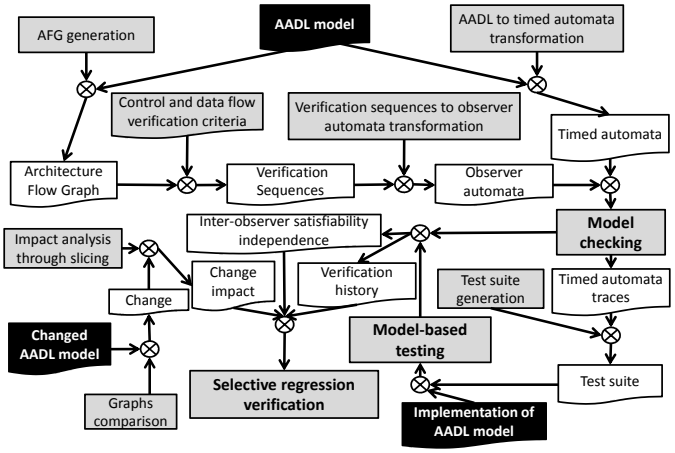


Fig. 8: The Architecture Quality Assurance Framework.

to AFG is presented in Section III-B and the transformation to timed automata is presented in III-D. An illustration of the AADL XML schema that is fundamental to the transformation rules is presented in Table I. AADL provides modeling of component abstractions dividable into three groups: application software components (Process, Thread, Subprogram, and Data); execution platform components (Processor, Virtual Processor, Memory, Bus, Virtual Bus, and Device); and general composite components (System). A component is modeled by a *component type* (e.g. A12-A17) and a *component implementation* (e.g. A18-A21) declaration. A component type defines the externally visible properties (e.g. A12-A16) and interfaces (e.g. A17) of the component. A component implementation declaration defines the component-internal structure, in terms of subcomponents (e.g. A20) and their connections (e.g. A6-A7). These subcomponents can themselves have subcomponents resulting in a hierarchy that eventually describes the whole system. AQAT operates on this type of schema, where the complete architecture is described within a system component (A1-A7) containing at least one process component (A3) composed of at least one thread (A10) that is bound to a processing unit (A3-A4).

A component may essentially be modeled with three types of interfaces: *ports* (e.g. A17), *component accesses* (e.g. A17), and *parameters* (e.g. A22). Ports represent directional interaction points of components for the transmission of data streams, messages, and events. A port can either be declared as a data port, an event port, or an event data port. A data port communicates data without queuing whereas an event data port communicates data with queuing. An event port communicates events with queueing, such as dispatch triggers of threads, triggers for mode switches, and alarms. Parameters represent interaction points of subprograms for the transmission of call (in parameter) and return (out parameter) data. Component access declarations support modeling of shared resources, such as global data components and data buses. Access declarations are named and can be declared with a *provides* or *requires* statement. A provides statement denotes that a component provides access to a data or bus component internal to it. A requires statement denotes that a component requires access to a data or bus component external to it. There are three types of corresponding connections: *port connections* (e.g. A6), *component access connections* (e.g. A5), and *parameter connections* (e.g. A6).

Each AADL element may be associated with a property declaration. A property constraints the expression it is associated with, e.g. in terms of timing (A6-A7) and scheduling (A12-A16). Furthermore, the behavior of a component can be described as a state transition system by using the AADL Behavioral Annex [19] (A18-A19), which is composed of a set of local variables, a set of states (at least one initial and one final state), and a set of state transitions. A state transition $s \xrightarrow{pri,g,act} s'$ has a source state s, a priority $pri \in \mathbb{N}$ (guards are evaluated in a sequence stipulated by the priorities if the source state has several outgoing transitions), a Boolean guard (predicate) g, a sequence of actions act, and a target state s'.

AQAT utilizes the Java package *og.jgrapht* to facilitate the transformation from AADL behavioral models to UPPAAL timed automata, whereby the behavioral model transition systems are virtually created and manipulated such that they conform to the target domain. Manipulation must essentially

be performed to transitions that include timing properties, subprogram calls, or accesses to shared resources to accurately represent timing and potential context switches in timed automata. A jgrapht graph object $G(V, E)$ is composed of a set of vertices V and a set of edges E on the form $e = \langle v_1, v_2 \rangle$, where e connects v_1 to vertex v_2. In order to conform to AADL Behavioral annex transitions, the default implementation of *jgrapht* edges, *DefaultEdge*, is extend with labels according to timed automata edges, as illustrated in Listing 1.

Listing 1: Extension of org.jgrapht.

```
class BehavioralAnnexEdge<V> extends DefaultEdge{
 private V v1, V v2, String priority,
 String guard, String action;
 public UppaalEdge(V v1, V v2, String priority,
 String guard, String action){this.v1 = v1;
   this.v2 = v2; this.priority = priority;
   this.guard = guard; this.action = action;
  } ... }
```

B. Architecture Flow Graphs

In order to extract the necessary verification data, the framework includes a technique that captures the prescribed control and data flows of an AADL model in a directed graph referred to as the architecture flow graph (AFG) [7]. The vertices of the graph represent operations, interfaces, and scheduling states (such as the standard scheduling states of threads illustrated in Fig 11) of the software components. The arcs represent how control and data flow through the vertices according to the behavioral models, component connections, and the semantical rules of AADL. AFGs are created through three operations. The first operation is to generate a control flow graph (CFG) for each thread and subprogam component of the AADL model. From this perspective, the control flow is determined by the behavioral model of the component. An illustration of the CFG of a behavioral model is presented in Fig. 9.

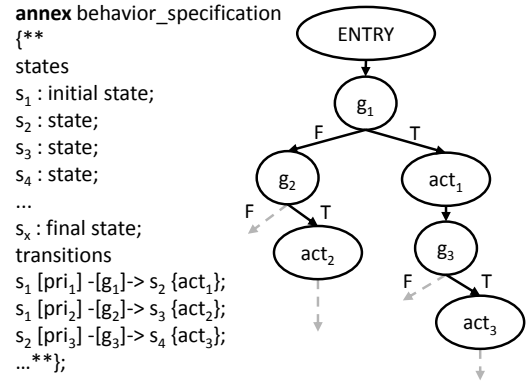


Fig. 9: The CFG (right) of a behavioral model (left). Assume $pri_1 > pri_2$.

The second operation is to compute the component-internal data flows, from input interfaces and to output interfaces, for each component and annotate them to the CFGs. Such flows are computed by AQAT through definition-use pairs analysis of each CFG with respect to each interface of the component, as illustrated with dashed arrows in Fig 10. The flows are necessary for the extraction of all prescribed component interactions through component connections. The third and final operation of creating the AFG is to integrate the individual graphs according to the component connections, as illustrated by bold arrows in Fig 10.

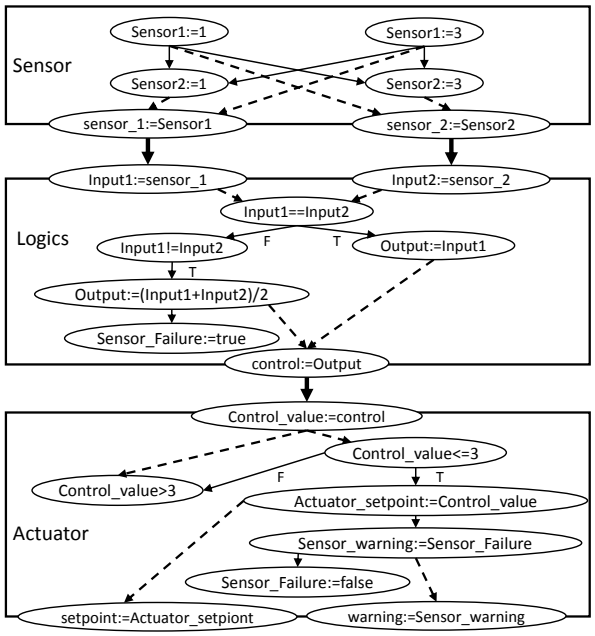


Fig. 10: AFG (excluding scheduling vertices) of the running example: component-internal control flows (solid arrows), component-internal data flows (dashed arrows), and component connections (bold solid arrows).

AQAT uses the *og.jgrapht* package to virtually create and operate on AFGs. In order to conform to the syntax of AFGs, the default implementation of *jgrapht* edges is extended as illustrated in Listing 2. The *type* label is used to denote the type of flow (component-internal or inter-component control or data flow) and the *constraints* label is used to associate the flow with defined AADL properties (e.g. expected minimum and maximum latencies of connections).

Listing 2: Extension of org.jgrapht.

```
class AFGEdge<V> extends DefaultEdge{
 private V v1, V v2, String type, String constraints;
 public AFGEdge(V v1, V v2, String type,
 String constraints){this.v1 = v1; this.v2 = v2;
 this.type = type; this.constraints = constraints;
  } ... }
```

C. Verification Criteria and Verification Sequences

An AFG contains different structural types of paths. There exist component-internal paths from the entry point to the exit point of a component (e.g. from $Input1 := sensor\_1$ to $control := Output$ of *Logics*) and there exist inter-component paths from the exit point of a component to the entry point of another if their interfaces are connected (e.g. from $control := Output$ of *Logics* to $Control\_value := control$ of *Actuator*). These may create indirect paths from one component to another through one or several intermediate

components (e.g. from $Sensor1 := 1$ of *Sensor* to $setpoint :=$ $Actuator\_setpoint$ of *Actuator* through *Logics*). The paths are constrained by property declarations, such as scheduling policies of processors, protocols of shared resources, and minimum and maximum latencies. A path in conjunction with a set of constraints is referred to as a verification sequence. The verification criteria enforced by AQAT ensure that each path is executable and in compliance with the constraints.

AQAT extracts component-internal paths by utilizing the *org.jgrapht.alg* package, in particular by means of the *AllDirectedPaths* class, as illustrated in Listing 3. Inter-component paths are extracted through a tool-specific method, *getAllDirectPaths*, which extracts all AFG arcs labelled as an inter-component flow. The set of indirect component to component paths are finally extracted through an application of the recursive, tool-specific method *getIndirectPaths* to each component-internal path. If the system contains component-internal loops or inter-component feedback loops, the recursion ends according to the upper bounds specified by the tool user. The set of constraints for each path is subsequently accumulated by searching each arc in the path for property associations.

Listing 3: AFG paths extraction methods.

```
AllDirectedPaths<String, AFGEdge> pathGenerator =
new AllDirectedPaths<String, AFGEdge>(CFG);
List<GraphPath<String, AFGEdge>> compInternalPaths =
pathGenerator.getAllPaths(ENTRY, EXIT, false, null);
List<GraphPath<String, AFGEdge>> directCompPaths =
getAllDirectPaths(EXIT, ENTRY, AFG);
List<GraphPath<String, AFGEdge>> indirectCompPaths =
new <GraphPath<String, AFGEdge>>();
for(GraphPath<String, AFGEdge> path: compInternalPaths){
  getIndirectPaths(path, AFG, compInternalPaths,
  directCompPaths, cycles_limit);
}
```

D. Transformation To Timed Automata

The transformation from AADL to the formal domain of UPPAAL timed automata corresponds to a transformation to UPPAAL XML as the utilized model-checker *Verifyta* [8] operates on such files. The outputted XML document consists of three main parts: a declaration, a set of templates (automata), and a system description, as shown in Table II. The declaration part (U1-U7) essentially defines the set of global variables, clocks, and synchronization channels of the system. A template (U8-U12) is composed of a local declaration (U8-U10), a set of locations (U11), and a set of transitions (U11-U12). Finally, the system part (U13) instantiates the templates as automata of the system.

An AADL model is transformed into an UPPAAL XML document essentially composed of one template for each processor, thread, and subprogram component. Schedulers within the processor templates control the transition of thread scheduling states, from thread dispatches to completions, and of context switches through synchronization channels (U4-U6), as illustrated in Fig 11. Each thread is initially in the *Awaiting_Dispatch* location. The edge to the *Ready* location is subsequently fired according to the period of the thread (A13 $\rightarrow$ U9). Input data from connections are simultaneously assigned to the input ports of the thread, where component interfaces are mapped to local, template variables (U10) whereas connections and shared data objects are mapped to global variables

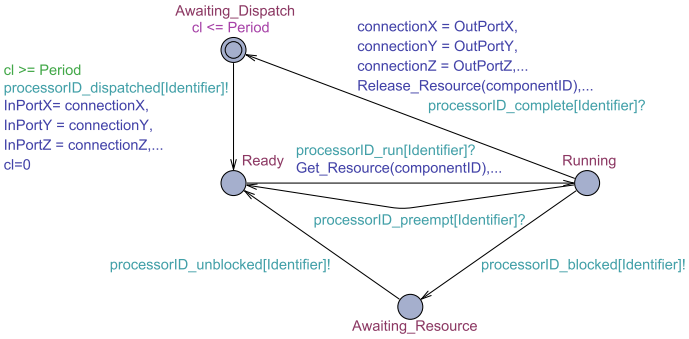


Fig. 11: UPPAAL template of AADL threads. The behavior model of a thread replaces the "Running" location.

(U1). Shared data objects are accessible in critical sections through $Get\_Resource()$ (U3) and $Release\_Resource$ (U3) service calls. If the semaphore (U2) of a resource already is locked at the time of an access attempt, the thread transits to the $Awaiting\_Resource$ location. Threads in the $Ready$ location, i.e., threads in the ready queue (U4), are assigned to be executed by the processor component they are bound to according to the scheduling policy property (U7). The bindings between AADL processes and processing units (A3-A4) are transformed to the timed automata model by distinguished identities of the scheduling synchronization channels (U4-U6). Given a scheduler with fixed priority preemptive scheduling policy, the thread with the highest priority is assigned to the processor and consequently switched to the $Running$ location. The *Running* state of threads constitutes the transition system defined by the AADL behavioral annex (A18-A19), however where AADL transitions that include timing properties, subprogram calls, or accesses to shared resources are complemented in the transformation with constructs for potential context switches. A running thread is preempted and switched to the ready location if another thread with a higher priority enters the *Ready* location. A running thread that completes its execution transits to the *Awaiting_Dispatch* location. Output is simultaneously assigned from output interfaces (local variables) to the corresponding connections (global variables).

E. Model Checking

Path-executability analysis for model checking, and for test suite generation as described in Section III-F, is performed through the generation of observer automata [7]. AQAT generates an observer automaton for each possible AFG path (verification sequence), where each control flow arc of the path is mapped to one observer edge that is dependent on the corresponding transition in the UPPAAL model, and where each data-flow arc of the path is mapped to two consecutive observer edges such that the former is dependent on the corresponding interface/connection definition and the later is dependent on the corresponding interface/connection use. By arranging the observer edges according to the sequence of the AFG path, an arrival to the final observer location implies that the observed path has been successfully executed. Path constraints, such as minimum and maximum latencies of connections, are transformed to constructs of invariants, guards, actions, and clocks of the observer that enforce the conditions through which the flows must be observed. An example of such an observer

TABLE II: UPPAAL XML Schema

U1:	<declaration >*Data_Type Connection_Name*; . . . *Data_Type Data_Component_Name*; . . . */*connections and shared data*/*
U2:	broadcast chan *Subprogram_Name*; . . . */*call channel*/* bool semaphore[*Number_Shared_Data_Components*]; */*for shared data components*/*
U3:	bool Get_Resource(int Component_Identifier){ . . . } void Release_Resource(int Component_Identifier){ . . . } */*system routines to lock and release resources*/*
U4:	int *Processor_name*_ready_queue[*Number_Bound_Threads*]; */*ready queue*/* broadcast chan *Processor_Name*_dispatched[*Number_Bound_Threads*]; */*disp. synch.*/*
U5:	broadcast chan *Processor_Name*_run[*Number_Bound_Threads*]; broadcast chan *Processor_Name*_complete[*Number_Bound_Threads*]; */*context switch synch.*/*
U6:	broadcast chan *Processor_Name*_preempt[*Number_Bound_Threads*]; broadcast chan *Processor_Name*_blocked[*Number_Bound_Threads*]; */interruption synch.*/*
U7:	void *Processor_name*_schprotocol(int Thread_Identifier) . . . *//arranges dispatches to ready queue according to the scheduling protocol* </declaration>
U8:	<template ><name>*Component_Name*</name ><declaration >*/*scheduling properties (only for threads)*/*
U9:	int Period = *value*; int Priority = *value*; int Compute_Execution_Time = *value*; int Compute_Deadline = *value*; int Identifier = Priority;
U10:	*/*interfaces and local variables (only for threads and subprograms)*/ Data_Type Port_Name*; . . . *Data_Type Variable_Name*; . . . </declaration>
U11:	<location id="*Location_Identifier*"/> . . . <transition><source ref="*Location_Identifier*"/><target ref="*Location_Identifier*"/>
U12:	<label kind="guard">*guard*</label><label kind="assignment">$action_1$,$action_2$ . . . </label></transition> . . . </template> . . .
U13:	<system>$Component\_Name_1$,$Component\_Name_2$ . . . ; </system>

is presented in Fig. 12. The automaton observes data flows through connection *sensor_1* of the sensor-to-actuator model. Since the connection is under-sampled, i.e. the sending task (*Sensor*) has a higher dispatch frequency than the receiving thread (*Logics*), the observer allows sending of new data before the previously sent data have been read, to ensure that some data are received but not necessarily all.

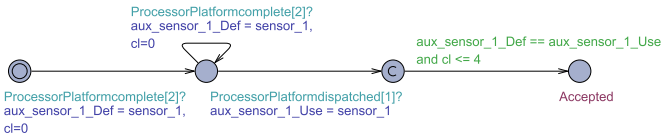


Fig. 12: Observer for connection *sensor_1* in Fig. 1.

Verification sequences are subsequently executed by AQAT through the invocation of *Verifyta* with formulae on the form $E <> ObserverX.Acceptance$, pronounced "there exists one path where $ObserverX.Acceptance$ eventually holds", whereupon the observer monitors a state space search of the model and reaches the final acceptance state whenever the coverage criterion has been satisfied. The tool invokes the model checker through a command on the form: *runtime.exec("verifyta -t0 -f tracefile.xtr Uppaal.xml query.q")*, where the reachability formulae are contained within the *query.q* file. With respect to the sensor-to-actuator model, two faults are detected by the tool. First, the minimum latency property of connection *control* (1 ms) is exceeded in the second dispatch of *Actuator* as the dispatch coincides with the completion of *Logics* (all tasks are simultaneously released at system initialization). The time from when output is produced by *Logics* until it is read by *Actuator* will consequently be below the required minimum. Second, the control signal to *Actuator* cannot be higher than three according to the modeled input range and computations. The corresponding predicate within *Actuator* is consequently unreachable.

F. Test Suite Generation

A satisfied observer produces a trace, written by *Verifyta* to *tracefile.xtr*, 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. The trace also holds information about the timing of input and output. Each observer trace may therefore be used to test the observed path against the architecture implementation when available. AQAT includes a module that converts the generated observer traces into test cases. Examples of test cases generated from the running example is presented in Table III.

TABLE III: Examples of generated test cases.

Test case 11	Test case 12	Test case 13
INPUT 0(time=0): sensor_2 = 0, sensor_1 = 0	INPUT 0(time=0): sensor_2 = 0, sensor_1 = 0	INPUT 0(time=0): sensor_2 = 0, sensor_1 = 0
INPUT 1(time=2): sensor_2 = 1, sensor_1 = 1	INPUT 1(time=2): sensor_2 = 1, sensor_1 = 1	INPUT 1(time=2): sensor_2 = 1, sensor_1 = 1
INPUT 2(time=16): sensor_2 = 1, sensor_1 = 1	INPUT 2(time=16): sensor_2 = 3, sensor_1 = 1	INPUT 2(time=16): sensor_2 = 1, sensor_1 = 1
INPUT 3(time=30): sensor_2 = 3, sensor_1 = 1	INPUT 3(time=30): sensor_2 = 1, sensor_1 = 1	INPUT 3(time=30): sensor_2 = 3, sensor_1 = 3
Ensure OUTPUT(time=38): Sensor_Failure==0 and warning==1 and setpoint==2	Ensure OUTPUT(time=38): Sensor_Failure==0 and warning==0 and setpoint==1	Ensure OUTPUT(time=38): Sensor_Failure==0 and warning==0 and setpoint==3

G. Selective Regression Verification

AQAT also includes a technique for efficient reverification of a modified architecture, where only those verification sequences that may be impacted by the modification are re-executed. The first step of the technique identifies the change by comparing the AFGs of the initial and changed model. The second step identifies the remaining parts of the modified architecture design that possibly are impacted by the change. Impact analysis is performed through static forward slicing of the changed model based on control and data dependencies of its AFG, which are represented in a directed graph referred to as the architecture dependence graph (ADG). The tool then selects only those verification sequences that cover changed vertices or vertices that are forward-reachable from the changed vertices, i.e. vertices that possibly are dependent on the change.

The selection process is optimized by means of observer satisfiability independence analysis, which adds dynamic dependencies to the selection process. The analysis is performed between verification projects through satisfiability checking of formulae on the form: *E <> ObserverX.Acceptance and not ObserverY.Acceptance*. From the data, satisfiability independence (and dependence) between observers may be deduced. If an observer obs_x may be satisfied without satisfying another

observer obs_y, then obs_x is satisfiable independently from obs_y. This implies that the path observed by obs_x may be executed without an execution of the path observed by obs_y. Consequently, a previously satisfied observer which satisfiability is independent to each previous observer that covers the modification (changed vertices) will also be satisfiable in the regression verification process, and is therefore unnecessary to re-execute even if it covers a vertex in the forward static slice.

Examples of dynamic independences in the sensor-to-actuator model are presented in Table IV. Only observers for component-internal or inter-component paths are presented in the figure, where observers H and K cover the two faults in the model and cannot be analyzed for independence. Any subsequent reverification process therefore implies re-execution of H and K. Multi-independence to a collection of component-internal paths pertaining to a single component are joined by "+". Given that a component has two paths X and Y, an independence of Z to $X + Y$ implies an unconditional independence to the component whereas an independence of Z to X, Y implies a conditional independence, where X must not execute before Z as long as Y executes before Z, and vice versa.

As presented in Table IV, all paths through *Sensor* (A, B, C, and D) are unconditionally independent to *Logics* (E+F) and *Actuator* (G+H). Moreover, each path through *Sensor* may be executed independently from the other paths through the same component and from the component connections (I, J, and K). Path E through *Logics* is conditionally independent to *Sensor* (A+B+C, A+B+D, A+C+D, B+C+D), i.e., E may execute independently from any path through *Sensor* as long as one of the paths, any of them, precedes E. Note that the chronological dependency is not caused by the connections between *Sensor* and *Logics* (path E may execute independently from the output produced by *Sensor*), but due to the scheduling properties, where *Sensor* is scheduled prior to *Logics*. Path E may also execute independently from the other path through *Logics* (F), from the *Actuator* component (G+H), and from the component connections (I, J, and K). On the other hand, Path F through *Logics* is conditionally independent to *Sensor* such that any path through *Sensor* must not execute before F as long as either A or D precedes F. Moreover, contrary to path E, path F is dynamically dependent on E, on path G through *Actuator*, and on connections I and J. The differences in dependencies with respect to the two paths through *Logics*, E: Input1==Input2 $\rightarrow$ Output:=Input1 and F: Input1==Input2 $\rightarrow$ Input1!=Input2 $\rightarrow$ Output:=(Input1+Input2)/2 $\rightarrow$ Sensor_Failure:=true, are caused by the initialization values. Since the input interfaces of *Logics* have equal values (zero) at the time of system initialization, where all threads are simultaneously released, path *E* (in contrast to path F) will always execute in the initial dispatch of *Logics*, regardless of the output produced by *Sensor*. Output of *Sensor* is read at subsequent dispatches of *Logics*. Path E may therefore execute independently from the output produced by *Sensor* whereas an execution of path F requires an execution of either of the two paths through *Sensor* that produces inconsistent output signals. It should be noted that the AADL model may be specified with initialization subroutines to define the (deterministic or non-deterministic) initialization values of variables. Due to the scheduling priorities and default values of input ports in this example, the execution of path E in the initial dispatch of *Logics* will be followed by an execution of path G: Control_value<=3 $\rightarrow$ Actuator_setpoint:=Control_value $\rightarrow$ Sensor_warning:=Sensor_Failure, in the initial dispatch of *Actuator*. Path G must therefore be preceded by some path through *Sensor* and path E through *Logics*. Finally, transfers of data through connections Sensor_1 (I) and Sensor_2 (J) cannot occur until some path through *Sensor*, path E, and path G have been executed in the initial dispatches of the threads, i.e., until subsequent dispatches of *Logics* occur.

TABLE IV: Observer satisfiability independence.

Observer (Sensor) A	B+C+D	E+F	G+H	I	J	K			
Observer (Sensor) B	A+C+D	E+F	G+H	I	J	K			
Observer (Sensor) C	A+B+D	E+F	G+H	I	J	K			
Observer (Sensor) D	A+B+C	E+F	G+H	I	J	K			
Observer (Logics) E	A+B+C	A+B+D	A+C+D	B+C+D	F	G+H	I	J	K
Observer (Logics) F	A+B+C	B+C+D	H	K					
Observer (Actuator) G	A+B+C	A+B+D	A+C+D	B+C+D	F	H	I	J	K
Observer (Actuator) H									
Observer (Sensor_1) I	A+B+C	A+B+D	A+C+D	B+C+D	F	H	J	K	
Observer (Sensor_2) J	A+B+C	A+B+D	A+C+D	B+C+D	F	H	I	K	
Observer (Control) K									

IV. Evaluation

The fault-detection effectiveness and the resource efficiency of AQAF and AQAT have been evaluated in an industrial case study comprising an application to a safety-critical train control system [18]. 385 design faults and 385 implementation faults were injected in the study to guarantee coverage of fault types and statistical significance. The considered fault types are: absent, unachievable, or incorrect control expression (guard); absent or incorrect data assignment, event, or call (action); absent or incorrect port connection; absent or incorrect parameter connection; absent, incorrect, or incompatible timing property; absent, incorrect, or incompatible protocol or use of shared resource (deadlock, livelock, starvation, and priority inversion of threads); absent, incorrect, or incompatible scheduling property (missed deadline); absent behavior model transition; and absent or incorrect transition priority. Results indicate a 100% fault detection rate at the model level, a 98.5% fault detection rate at the implementation level, and an average reduced resource consumption (time and memory) of regression verification by 6.4% with the use of the selective approach in contrast to a re-run all approach. The resource consumption of individual framework operations are negligible with respect to state space searches by the utilized model-checker and not included in the study. In this section, performance measurements of each module of the tool are presented to provide a complete description of the tool performance. More precisely, the time consumption of transformation from AADL to architecture flow graph (AFGs) and timed automata (TA); generation of verification sequences; change impact analysis through slicing; transformation from verification sequences to observer automata; model checking; and test suite generation.

The results of the study are presented in Table V. The utilized architectural model is composed of a single-core processor with preemptive multitasking, 3 tasks, 3 subprograms, 76 interfaces (ports, parameters, and shared data components),

55 connections, and 5 behavioral models (41 local states and 51 local transitions). All measurements have performed in Windows 7 64-bit edition running an Intel Core i7-3667U 2.0 GHz CPU with 8 GB RAM. On average, 36 verification sequences are necessary to extract in order to achieve full coverage of the system. In terms of resource consumption, the bottleneck is model checking by *Verifyta*, which takes minutes to complete in contrast to milliseconds for most framework operations – 4.3 seconds on average for test suite generation. Faults may both reduce and increase time consumption of model checking (and test case generation). The best and worst case of the samples differ greatly, 14 seconds versus 86 minutes, depending on the type and location of the fault.

TABLE V: Performance measurements

Process/produced artifact	Average:	Min:	Max:	St. Dev.:
AADL to AFG and TA	188ms	120ms	387ms	41ms
Verification sequences extraction	376ms	10ms	947ms	144ms
Change impact analysis	1sec	1ms	2.6sec	776ms
No. extracted verification seq.	36	15	56	5
Verifi. seq. to observers	622ms	100ms	1.7sec	186ms
Model checking	6min	14sec	86min	8min
No. unsatisfied verification seq.	16	1	38	13
Test suite generation	4.3sec	2.9sec	4.9sec	697ms
No. failed test cases	18	1	38	14

V. Related Work

Research in this field has developed a number of verification tools for AADL. Murugesan et al. [11] present AGREE, a model-checker for functional AADL models. Björnander et al. [12] present the tool ABV, which provides model checking of functional AADL models through a transformation to ML (Meta Language). Singhoff et al. [16] present the tool Cheddar, which provides a schedulability and resource requirements analysis feature for AADL models. Berthomieu et al. [13] present a verification toolchain based on the Topcased environment, where AADL models can be checked by the Tina toolbox through transformations to timed transition systems. Chkouri et al. [15] present the tool AADL to BIP, which provides model checking of event-driven AADL models through a transformation to the BIP (Behavior Interaction Priority) language. Esteve et al. [14] present COMPASS, providing model checking of SLIM models, a variant of AADL, through a transformation to Markov chain.

These contributions do not provide any solutions to the verification of architecture implementations and design modifications. Furthermore, scheduling properties, context switches, concurrency by multitasking and parallel processing, uses of shared resources, and real-time constraints are not jointly considered in the verification of behavior. Since these properties influence each other at runtime, any exclusions of them in the verification implies uncertainty in the results. To our knowledge, AQAT is the only contribution that simultaneously includes all these properties in the verification. Finally, these contributions do not include methods that measure and enforce coverage of the architectural design in the verification process, such as control and data flow path coverage by AQAT, which is essential to determine the extent to which an architecture has been verified. Outside the scope of AADL, Simulink Design Verifier [20] provides a formal verification and analysis framework for Simulink models. Besides the ability to automatically detect design faults and requirements violations through model checking, the tool includes a condition, decision, and modified condition/decision coverage analyser and a slicer for dependency tracing and variability modeling. Inverardi et al. [21] present CHARMY, a tool for UML-based modeling and analysis of software architectures. By means of a transformation to Promela code, the SPIN model-checker [22] is used to verify temporal properties of the architectural model. The authors are planning to extend the tool with dependence analysis and architectural slicing in their future work.

VI. Conclusion and Future Work

We have presented and demonstrated the Architecture Quality Assurance Tool (AQAT), an implementation of the Architecture Quality Assurance Framework (AQAF) [7] that provides a holistic, formal, and automated verification process for architectural engineering of critical embedded systems. The tool addresses architectural design faults, implementation faults, and maintenance faults by means of integrated model checking, model-based testing, and selective regression verification techniques.

Regarding limitations of the technical solution, the algorithms for paths extraction do not take into account potential control dependencies between branching expressions, where paths that should not be able to execute by design nonetheless may be extracted as prescribed execution paths of the architecture. Such dependencies are complex to statically analyze, as the satisfiability of each branching expression must be determined for each possible path and combination of inputs that lead to an execution of the expression. In the current version of the tool, the user may either simply ignore results from verifications of non-designed paths (which will be unsatisfiable and cause the tool to produce alarms) or manually specify such dependencies for the tool (transition pairs that should not appear in the same path), whereupon AQAT disregards the corresponding paths. A similar problem exist with conditional loops, where the paths extraction algorithms are unable to determine the potentially maximum number of iterations a loop should be able to execute by design. The tool user is consequently required to set appropriate upper bounds and keep track of extracted paths that exceed the intentions of the design, such that alarms can be appropriately ignored. In the future work, we are planning to improve the limitation of paths extraction by means of symbolic execution and SAT/SMT solvers, which may be utilized to statically identify the dependencies between branching expressions. Regarding the performance of model checking (including generations of test suites and dynamic dependencies) with respect to different observers (paths), the time consumption of state space explorations may be significantly reduced by distributing the computations to multiple computers. The UPPAAL environment is based on a client-server architecture, where model checking engines may be remotely installed on multiple servers and communicate with clients through TCP/IP. Future work includes a study wherein the possibilities and effects of a model checking parallelization of AQAT is explored, to potentially improve the performance of the tool.

Although AQAT has been evaluated in an industrial case study with satisfactory results, assessments of scalability, usability, and reliability require further studies with a greater

variety of system types and complexities, preferably with authentically created faults instead of deliberately injected. Another area of improvement is compatibility with a wider range of architecture description languages, requirements specifications, and execution, middleware, and hardware models. This essentially entails in extending the set of transformation rules to constructs of timed automata.

ACKNOWLEDGMENT

This research is supported by the Swedish Foundation for Strategic research (SSF) project SYNOPSIS – Safety Analysis for Predictable Software Intensive Systems – and the knowledge foundation (KK-stiftelsen) project DPAC – Dependable Platforms for Autonomous systems and Control.

REFERENCES

[1] B. Boehm, R. Valerdi, and E. Honour, "The roi of systems engineering: Some quantitative results for software-intensive systems," *Syst. Eng.*, vol. 11, no. 3, pp. 221–234, Aug. 2008.

[2] J. Elm, D. Goldenson, K. El Emam, N. Donitelli, A. Neisa, and N. S. E. Committee, "A Survey of Systems Engineering Effectiveness: Initial Results (CMU/SEI-2007-SR-014)," Software Engineering Institute, Carnegie Mellon University, Tech. Rep., 2007.

[3] RTI, "The Economic Impacts of Inadequate Infrastructure for Software Testing," Washington, DC, NIST Planning report 02-3, 2002.

[4] L. Bass, P. Clements, and R. Kazman, *Software architecture in practice*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1998.

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

[6] G. Fraser, F. Wotawa, and P. E. Ammann, "Testing with Model Checkers: A Survey," *Softw. Test. Verif. Reliab.*, vol. 19, no. 3, pp. 215–261, Sep. 2009.

[7] A. Johnsen, K. Lundqvist, K. Hänninen, P. Pettersson, and M. Torelm, "AQAF: An Architecture Quality Assurance Framework for Systems Modeled in AADL," in *2016 12th International ACM SIGSOFT Conference on Quality of Software Architectures (QoSA)*, April 2016.

[8] UP4ALL International AB, "The UPPAAL Model-checking Tool," http://www.uppaal.com, February 2017.

[9] 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.

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

[11] 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.

[12] S. Björnander, P. Graydon, and R. Land, "Towards automatic verification of safety properties in aadl system models," in *Proceedings of the 31st International System Safety Conference (ISSC)*. System Safety Society, August 2013, best paper award winner.

[13] 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.

[14] 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.

[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] F. Singhoff, J. Legrand, L. Nana, and L. Marcé, "Scheduling and Memory Requirements Analysis with AADL," *Ada Lett.*, vol. XXV, no. 4, pp. 1–10, Nov. 2005.

[17] R. Taylor, N. Medvidovic, and E. Dashofy, *Software Architecture: Foundations, Theory, and Practice*. Wiley, 2009.

[18] A. Johnsen, K. Lundqvist, K. Hänninen, P. Pettersson, and M. Torelm, "Experience Report: Evaluating Fault Detection Effectiveness and Resource Efficiency of the Architecture Quality Assurance Framework and Tool," in *Proceedings of the 28th International Symposium on Software Reliability Engineering (ISSRE)*, October 2017.

[19] R. B. Franca, J.-P. Bodeveix, M. Filali, J.-F. Rolland, D. Chemouil, and D. Thomas, "The AADL behaviour annex – experiments and roadmap," in *ICECCS '07: Proceedings of the 12th IEEE International Conference on Engineering Complex Computer Systems*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 377–382.

[20] MathWorks, "Simulink Design Verifier," https://se.mathworks.com/products/sldesignverifier.html, July 2017.

[21] P. Inverardi, H. Muccini, and P. Pelliccione, "Charmy: An extensible tool for architectural analysis," in *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. ESEC/FSE-13. New York, NY, USA: ACM, 2005, pp. 111–114.

[22] G. Holzmann, *Spin Model Checker, the: Primer and Reference Manual*, 1st ed. Addison-Wesley Professional, 2003.