Experience Report: Evaluating Fault Detection Effectiveness and Resource Efficiency of the Architecture Quality Assurance Framework and Tool

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

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

***Abstract*—The Architecture Quality Assurance Framework (AQAF) is a theory developed to provide a holistic and formal verification process for architectural engineering of critical embedded systems. AQAF encompasses integrated architectural model checking, model-based testing, and selective regression verification techniques to achieve this goal. The Architecture Quality Assurance Tool (AQAT) implements the theory of AQAF and enables automated application of the framework. In this paper, we present an evaluation of AQAT and the underlying AQAF theory by means of an industrial case study, where resource efficiency and fault detection effectiveness are the targeted properties of evaluation. The method of fault injection is utilized to guarantee coverage of fault types and to generate a data sample size adequate for statistical analysis. We discovered important areas of improvement in this study, which required further development of the framework before satisfactory results could be achieved. The final results present a 100% fault detection rate at the design level, a 98.5% fault detection rate at the implementation level, and an average increased efficiency of 6.4% with the aid of the selective regression verification technique.**

Keywords*—*empirical evaluation; case study; fault injection

I. Introduction

Architectural engineering is essential to make critical embedded systems dependable and comprehensively affects both the development process as well as the behavior of the system. System defects caused by erroneous architectural engineering are therefore prone to have major negative impacts on both the dependability of the system and the cost of the development process. Boehm et al. [1] present a study that quantitatively evaluates the return on investment of system engineering based on an analysis of 161 projects. The results suggest that 20% of the defects are responsible for 80% of the rework costs, and that the primary source of these costly defects is inadequate architecture definitions and risk resolutions. Elm et al. [2] concluded that there is a strong positive relation between the qualities of architectural engineering and the performances of the studied projects. In addition to the possibility of introducing costly and hazardous defects during architectural design, defects may be introduced in the process of implementing the design and any time changes later are made due to maintenance. Rigorous and holistic verification of architectural engineering is consequently essential in the development of critical embedded systems. Furthermore, manual verification is labor intensive and error prone, where automation is necessary to cope with the increasing complexity of embedded systems. The Architecture Quality Assurance Framework (AQAF) [3] is a theory that integrates a variety of formal methods with model-driven architectural engineering to provide a holistic, rigorous, and automated approach to verification, from architectural requirements analysis and design to architectural implementation and maintenance. AQAF includes an architectural model checking technique to detect design faults, an architectural model-based test case generation technique to detect implementation faults, and an architectural selective regression verification technique based on a formal change impact analysis technique to efficiently detect faults created by maintenance modifications.

The Architecture Quality Assurance Tool (AQAT) [4] implements the theory of AQAF and enables automated application of the framework. The contribution of this paper is an evaluation of the fault detection effectiveness and the resource efficiency of AQAT and the underlying AQAF theory by means of an industrial case study. The study encompasses an application of AQAT to a safety-critical train control system, where a fault injection methodology is used to ensure coverage of fault types and to create a sufficiently large data sample from which fault detection effectiveness and resource efficiency can be statistically assessed. Effectiveness is assessed based on the ratio of detections with respect to injections. Efficiency, on the other hand, is assessed based on time and memory consumptions and, for the selective regression verification technique in particular, comparisons with a re-run all approach. The presented study is a continuation of a preliminary evaluation of AQAF reported in [3], where framework operations were manually executed against a statistically insignificant number of architectural faults. By means of a comprehensive study of AQAT, we observed important areas of improvement that required further development of the framework before satisfactory results could be achieved.

The paper is structured as follows. In Section II, an overview of the AQAF theory [3] implemented by AQAT [4] is presented. In Section III, the case study design is presented. The results of the case study are then presented in Section IV, which is followed by discussion and lessons learned in Section V, related work in Section VI, and finally conclusion and future work in Section VII.

II. The Architecture Quality Assurance Framework

AQAF has particularly been developed for architectural models expressed by the Architecture Analysis and Design Language (AADL) [5], but may be adapted to any other architecture description language (ADL) with a similar expressiveness. AADL, initially released in 2004 [6], provides hierarchical modeling of abstract system components to concrete application software components (e.g. process, thread, subprogram, and data components) and hardware platform components (e.g. processor, memory, bus, sensor, and actuator components). The behavior of a component is represented by a state transition system and the interaction between components is represented by various types of component interfaces and interface connections. Moreover, each AADL element may be associated with property declarations to impose constraints on its meaning, such as minimum and maximum execution times, priorities, deadlines, and dispatch protocols of threads. These elements together essentially prescribe how data and control should flow through the system, typically from sensors to actuators.

The verification criteria applied by AQAF ensure completeness (no lack or excess), consistency (no contradictions), and correctness (compliance with requirements) of the architectural control and data flows. This is important to industry as contemporary safety standards (e.g. ISO 26262 [7]) request control and data flow analysis of architectural designs. In order to extract the necessary verification data, the framework includes a technique that captures all prescribed control and data flows of an AADL model in a directed graph referred to as the architecture flow graph (AFG) [3], as presented in Fig. 1. The vertices of the AFG represent AADL expressions with execution semantics, component interfaces, and scheduling states of threads. The arcs of the graph represent how control and data should flow through the vertices according to the behavior of components, the component connections, and the semantical rules of AADL.

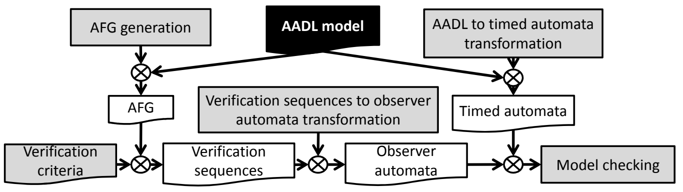

Fig. 1. Flowchart of the model checking process. A black shape represents a necessary framework input. A gray shape represents an algorithm or rule set. A white shape represents an artifact produced by the framework.

An AFG contains different types of paths composed of control and data flows. There are component-internal paths between the entry and exit points of a component and there are inter-component paths between the exit point of a component and the entry point of another if their interfaces are connected. These may together connect indirect paths between two components through one or several intermediate components. The paths are constrained by scheduling properties of threads, scheduling policies of processors, accessibility protocols of shared resources, minimum and maximum latencies of connections, and other expected extra-functional requirements according to the property declarations within the AADL model. A path in conjunction with a set of constraints, and possibly explicitly declared requirements, is referred to as a verification sequence, where AQAF verifies that each path can be executed in compliance with the constraints.

An AADL model is transformed to the formal domain of timed automata such that it can be subjected to the UPPAAL model-checker [8] for the purpose of model checking and/or model-based test case generation. The output of the transformation process is a network of timed automata essentially composed of one automaton for each processor, thread, and subprogram component, where schedulers within the processor automata control dispatches and context switches of threads. Verification sequences are subsequently executed through the generation of observer automata, one for each AFG path (verification sequence). An observer automaton observes a state space search of the transformed AADL model, i.e. the timed automata model, and reaches a final “acceptance” location whenever the corresponding AFG path has been executed in compliance with the requirements and constraints. Satisfiability of all observers imply a complete, consistent, and correct AADL model.

The state space searches are performed by the UPPAAL model-checker, which also generates traces of these searches. The timed automata trace that satisfies an observer contains information about the initial state of the system and its environment before the path is executed, the input and timing of input that stimulates an execution of the path, and the expected output and timing of output. Each observer trace is therefore used to generate a test case that tests the observed conditions against the architecture implementation when available, to verify its conformance to the architectural design, as illustrated in Fig. 2.

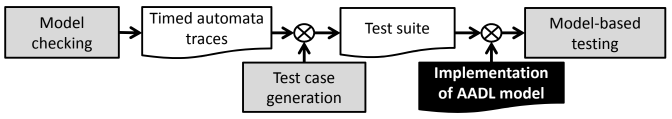

Fig. 2. Flowchart of the model-based testing process.

AQAF also includes a change impact analysis technique for efficient reverification of a modified architecture, where those verification sequences that are unnecessary to execute are excluded in the reverification process, as presented in Fig. 3. The technique identifies the change by comparing the AFGs of the initial and changed model. The remaining parts of the modified architectural design that possibly are impacted by the change are subsequently determined through static forward slicing [9] with respect to the identified change. The forward slice is calculated based on the control and data dependencies within the AFG of the changed model, which are represented

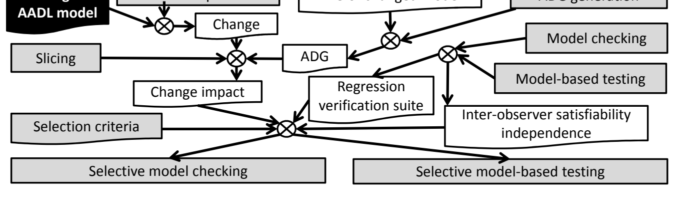

Fig. 3. Flowchart of the selective regression verification process.

in a graph referred to as the architecture dependence graph (ADG). AQAF then selects only those verification sequences that cover vertices in the slice. AQAF further excludes unnecessary verification sequences to re-execute by means of inter-observer satisfiability independence analysis, which adds dynamic dependencies to the selection process.

III. Case Study Design

A. Objectives of Study

The objectives of the study are to quantitatively and with statistical significance evaluate the fault detection effectiveness and the resource efficiency of AQAT in an industrial application. It should be noted that fault detection effectiveness is dependent on the number of true-positives and true-negatives in relation to the number of false-positives and false-negatives, where AQAT is expected to not falsely declare presence or absence of faults. Moreover, the goal of the selective regression verification technique is technically not to detect faults, but rather to select those verification sequences that might. Consequently, the criterion for effectiveness of the selective regression verification technique is that all verification sequences that actually reveal faults are selected in the regression verification process. Resource efficiency of model checking and test suite generation is assessed by measuring the time and memory consumption and by comparing the results with related work. Efficiency of the selective regression verification technique, on the other hand, is assessed by comparing the time and memory consumption with respect to a re-run all approach.

B. Case Selection

The case study is conducted in collaboration with Bombardier Transportation, a large company within the rail vehicle and equipment manufacturing and servicing industry. In this study, we are targeting a representative case subject to produce results that can be generalized, rather than outlier cases, which would yield richer information for limitations evaluation. A system which do not exceed a moderate cyclomatic complexity is therefore targeted. Furthermore, AQAT and AQAF have primarily been developed for architectures with synchronous, fixed-priority preemptive or non-preemptive execution models, as these commonly are used for critical embedded systems. These preferences are used to select a representative case subject. Based on these preferences, the Line Trip Relay Interface and Supervision (LTRIS) system is selected as case subject.

C. The Case Study Subject

LTRIS is a safety-critical train control system embedded in a system of systems developed by Bombardier Transportation. A simplified graphical model of LTRIS is depicted in Fig. 4. LTRIS interacts with numerous systems in the system of systems it is embedded within. A sound architectural representation of LTRIS should include the behavior of the surrounding systems. However, inclusion of all systems is out of scope of the study. Instead, we abstract the behavior of the environment into a single process component, denoted *LineTripEnvironment* in the model, and assume it executes with LTRIS on a common processing platform. Any required input by the LTRIS application software is consequently assumed to be produced by *LineTripEnvironment*. Each of these is referred to as "some_connection" in the model.

The application software of LTRIS is essentially composed of two periodic tasks: *Controller* and *Tester*. The functionality of *Controller* is to control a critical relay, monitor its status, and output feedback data. The feedback data are information on the status of the relay and the status relative to the expected one. *Controller* controls the relay according to data on in ports and on shared variables assigned by components in the environment. In this manner, *Controller* acts as an interface to the relay. The behavior of *Controller* includes two consecutive subprogram calls, first a call to subprogram *LtrInt* followed by a call to subprogram *dcu2_line_trip*. The possible opening and closing requests received at in ports of *Controller*, together with state information of *LineTripEnvironment*, are given as arguments to *LtrInt* when called. *LtrInt* then performs a sequence of operations on the input, as specified by its behavioral model, to determine whether on opening or closing output (return) signal shall be produced. The logic is composed such that an opening request shall be prioritized over a closing request. Output produced by *LtrInt* is subsequently used as argument in the second call to *dcu2_line_trip*, which is responsible of controlling the relay and producing feedback information. The feedback is made available for the environment through out ports of *Controller*.

Controller's input domain is partly set 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. *LtrTsSq* transmits opening and closing requests to *Controller* through *connection1* and *connection2* respectively. The test sequence exercises the relay in the possible ways it can be exercised: starting from an unidentified state of the relay, the test sequence signals an opening request, followed by a closing request, which finally is followed by an opening request. The final request is intentionally delayed in order to make sure that there is enough time for the power supply to close the relay before it finally is opened. Each request is validated through feedback before a subsequent request is sent. The current status of the relay is transmitted to *Tester* through *connection3*. The test process is reset to its default state any time a malfunction signal is received during the execution of the test sequence. Whenever the test sequence successfully has been executed, a corresponding signal is sent to the environment.

Given that the upper bound of feedback loops is set to one (an indirect path may only include a component once), a total of 38 paths (verification sequences) are extracted by AQAT.[1] Without constraints on loops, the number of paths would be infinite as the system is periodic and utilizes feedback.

D. Case Study Method and Expected Results

The faultiness of the AADL model must be controlled in the study to reliably evaluate effectiveness and efficiency. Firstly, an AADL model and an implementation thereof may have different types of faults that AQAT is expected to detect. Secondly, the presence of multiple faults may result in a

[1]The number of extracted paths by AQAT is less than the number of manually extracted paths in the preliminary study due to addressed redundancy of coverage.

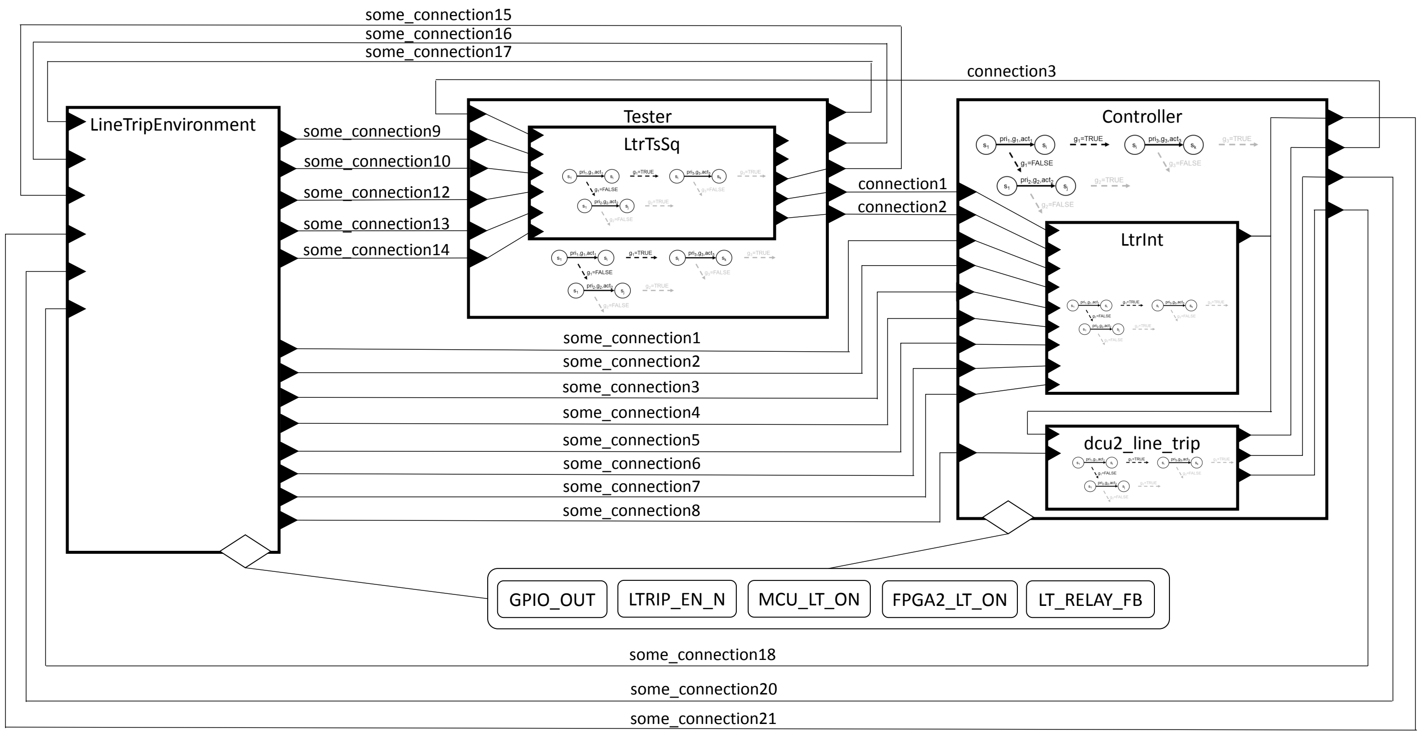

Fig. 4. Simplified model of LTRIS. Due to simplification, the depicted model differs from the graphical syntax of the AADL standard. In addition, the depicted behavior models are only included for illustration and do not represent any actual behavior of LTRIS.

complex combined error behavior that makes it difficult to determine whether AQAT produces any false positives or false negatives. In order to apply the complete framework and simultaneously ensure validity of the results and coverage of fault types, AQAT must be systematically applied to controlled versions of the LTRIS model. A typical application of the complete framework involves the following steps, each of which is executed by AQAT (except for step 7):

1) Generate the architecture flow graph (AFG) of the AADL model.
2) Generate verification sequences by applying the verification criteria to the AFG.
3) Transform the AADL model to a network of timed automata and each verification sequence to an observer automaton.
4) Verify the satisfiability of each observer using the UPPAAL model-checker.
   – The AADL model is complete, consistent, and correct if all observers are satisfiable. The resulting traces may be used to test the conformance of an implementation with respect to the AADL model, i.e., go to step 5.
   – The AADL model is faulty if not all observers are satisfiable. The model should be corrected, where steps 8-11 subsequently may be used to efficiently verify the updated model.
5) Generate a test suite from the produced timed automata traces – one test case for each trace.
6) Execute the test suite on the implementation.
   – The implementation conforms to the AADL model if each test case passes.
7) Modify the AADL model.
8) Generate the AFG and ADG of the modified model.
9) Compare the AFG of the previously verified AADL model with the AFG of the modified model to identify the modification.
10) Slice the ADG of the modified model with respect to the identified modification and reduce the slice with inter-observer satisfiability independences if any. The slice determines which parts of the modified model that may be impacted by the modification.
11) Perform selective regression verification by re-executing (as in step 4 and, if an implementation is available, steps 5-6) only those verification sequences that cover vertices in the slice.

Our approach to a systematic application to controlled versions of the LTRIS AADL model involves two stages. The first stage is to perform steps 1-6 (model checking and model-based testing) based on the original AADL model, which we assume to be free from faults and has been created with such goal. If the steps are valid and the implementation actually conforms to the model, the result of model checking and model-based testing must be satisfied observers and passed test cases. Since the model evidently conforms to the model itself, it, in the form of the generated timed automata model, is treated as the implementation in step 6.

The second stage of the approach is to use the technique of fault injection to create faulty versions of the LTRIS AADL model and expand the steps of stage one such that the application covers the whole framework and ranges over all fault types. By means of the steps taken in the first stage, each fault injection may be treated as a change. If steps 1-4 (the model checking technique) and 8-11 (the selective regression verification technique) are valid, the result of regression verification must be at least one unsatisfied observer, since there now evidently exist a fault. Nevertheless, the selective approach must be contrasted with a re-run all approach to conclude the selection effectiveness and efficiency. If steps 1-4 and 8-11 are valid, the result must be satisfied observers for all non-selected verification sequences, since the impact analysis is expected to select all verification sequences that may be affected. The required overhead expense of performing the selection in addition to the resource consumption of running the selection must either not exceed the cost of a re-run all approach to be resource efficient.

Finally, the timed automata models that are generated from the faulty LTRIS AADL versions may be treated as faulty implementations to validate the fault detection effectiveness of the test suite generation conducted in stage one. If steps 1-6 (the model-based test case generation technique) are valid the result must be at least one failed test case for each test suite execution. Consequently, the validity criteria for stage one and stage two are:

CRITERIA STAGE ONE
Number unsatisfied observers = 0
Total time consumption $<\approx$ related work
Total memory consumption $<\approx$ related work
Number failed test cases = 0

CRITERIA STAGE TWO
Number unsatisfied selected observers > 0
Number unsatisfied observers = No. unsatisfied selected observers
Total time consumption of a selective regression verification approach $<$ total time consumption of a re-run all regression verification approach $<\approx$ related work
Total memory consumption of a selective regression verification approach $<$ total memory consumption of a re-run all regression verification approach $<\approx$ related work
Number failed test cases > 0

E. Fault Types and Fault Injections

In this study, we have chosen to consider the following fault types:

FT1: Absent, unachievable, or incorrect control expression (guard)
FT2: Absent or incorrect data assignment, event, or call (action)
FT3: Absent or incorrect port connection
FT4: Absent or incorrect parameter connection
FT5: Absent, incorrect, or incompatible timing property
FT6: Absent, incorrect, or incompatible protocol or use of shared resource (deadlock, livelock, starvation, and priority inversion of threads)
FT7: Absent, incorrect, or incompatible scheduling property (missed deadlines)
FT8: Absent behavior model transition
FT9: Absent or incorrect transition priority

The selection is based on the set of AADL expressions that determine and constrain the control and data flows of an AADL model. By using a common practice confidence level of 95%, a conservative expected standard deviation of 0.5, and a $\pm$ 5% margin of error, the number of fault injections is set to 385 (385 design faults, 385 implementation faults, and 385 modification faults). For the purpose of the case study, AQAT is extended with a module that automatically performs the fault injections. The module essentially parses the AADL model and injects a fault upon the arrival of a selected expression. The distribution of fault injections across fault types is thereby largely determined by the frequency of the corresponding AADL expressions.

F. Case Study Instrumentation

In order to exercise the architectural paths by every class of input, the input domain of LTRIS is divided through equivalence portioning, where concrete values are determined through boundary value analysis. In the LTRIS AADL model, we assume that any required input is generated by the $LineTripEnvironment$ process. The process should be able to stimulate LTRIS with the possible input classes at completions – the environment is assumed to dispatch periodically. Moreover, the values of (input) data objects at the time of LTRIS initialization depends on its environment. The concept of including the complete input range and input behavior of a data object is presented in Fig. 5. In this case, for a Boolean-typed data object. The boundary values of each object are presented in Table I. Note that these values are determined based on an approximation of the environment.

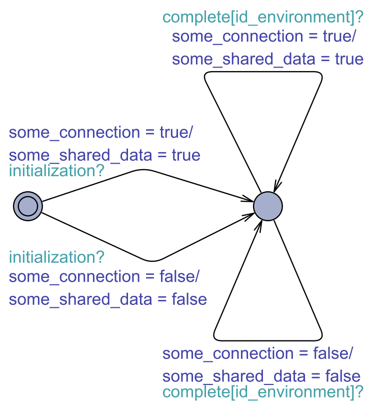

Fig. 5. Template for input generation of a Boolean typed connection/shared data component.

TABLE I. VALUE SETS OF EACH INPUT DATA OBJECT.

Data object	**Initialization**	**Env. completion**
some_connection1-2	{0(false),1(true)}	{0(false),1(true)}
some_connection3	{0,3,4,27,30,31,38}	{0,3,4,27,30,31,38}
some_connection4-14	{0(false),1(true)}	{0(false),1(true)}
GPIO_OUT	{0(false)}	
LTRIP_EN_N	{1(true)}	
MCU_LT_ON	{1(true)}	
FPGA2_LT_ON	{0(false),1(true)}	{0(false),1(true)}
LT_RELAY_FB	{0(false),1(true)}	{0(false),1(true)}

For the purpose of the case study design, each generated test case is scripted in timed automata such that they can be automatically executed. AQAT is extended in the study with a module that automatically performs the encoding. An example of a scripted test case is shown in Fig. 6.

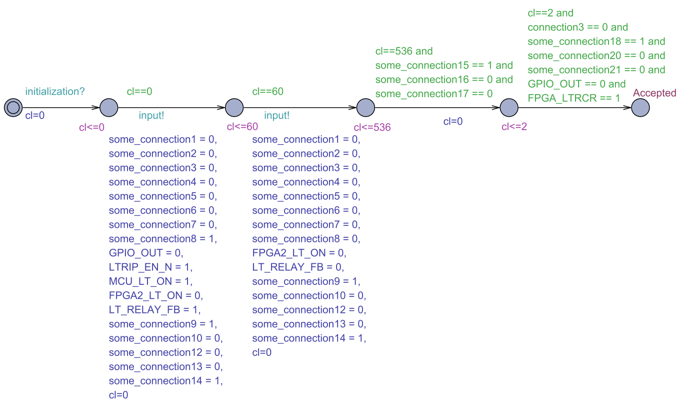

Fig. 6. Example of a test case scripted in timed automata.

According to the test case, a set of (initialization) inputs (values displayed in the figure) at time t = 0 ms, followed by a set of inputs at time t = 60 ms, should result in a set of outputs by *Tester* at time = 596 ms (60+536), followed by a set of outputs by *Controller* at time t = 598 ms. Similarly to observers, if the test script may reach the acceptance location, the test case passes.

IV. Results

The descriptive statistics of the results are presented in Table II, which is supported by six diagram charts in Fig. 7. In Fig. 8, various measurement distributions across the different fault types are presented by box and whisker charts. Table II presents the averages, standard deviations, minimums, maximums, and the totals in five consecutive row sections. The first column of the table presents the fault type ("0" denotes no injected fault). The second column presents the number of extracted verification sequences for full coverage of the AADL model. The third column presents the number of selected verification sequences (for selective regression verification) when the fault injection is treated as a modification of the default model. The fourth column presents the number of unsatisfied selected verification sequences (observers) whereas the fifth column presents the total number of unsatisfied verification sequences (selected and deselected). Columns six to eleven present the time and memory consumption of model checking the selected set of verification sequences versus the complete set of verification sequences. Note that the model-based testing technique uses the results (traces) of model checking to generate the test suite. Hence, the resource consumption of model checking also indicates the resource consumption of test suite generation. The last column presents the number of failed test cases of the test suite generated from the default model.

The distribution of fault injections across the fault types from which the results are generated is presented in chart A of Fig. 7. The results are in compliance with the validity criteria except for three cases, two of which are expected. First, the selective regression verification technique has no effect with respect to FT3, FT6, and FT7. No effect with respect to FT7 is expected since the fault type corresponds to a changed scheduling property, which has no measurable impact on the AFG. Consequently, no slicing nor selection is performed by AQAT in response to such changes. Second, no implementation faults of FT5 were detected by the generated test suite. The invalidity is expected as the fault type is an inconsistent latency property, which in the model does not affect the execution but impose an analysis constraint on it. It is not sound to treat the faulty timed automata model as a faulty implementation in this case since the inconsistent property must be manifested in the execution to emulate an implementation fault. Finally, additional five (the quantity is not deducible from Table II) faults of FT1, FT3 and FT7 were not detected at the implementation level – the minimum number of failed test cases is zero for each type (see Table II).

On average, it took 390.5 seconds and 2418.6 MB of memory to model check a faulty version of LTRIS, i.e. to check satisfiability of 36 verification sequences (observers). The default fault-free model, yielding 38 verification sequences, consumed 249 seconds and 1258 MB. By considering the distribution of time consumption of a re-run all approach, presented in chart J of Fig. 8, faults have the ability to both significantly reduce as well as increase the resource consumption of model checking (fault-free reference level is 249 sec). The minimum time consumption of the sample is 15 seconds (Table II) whereas the maximum is approximately 43 minutes (2566 seconds). Since the minimum number of unsatisfied observers over the complete sample of fault injections is larger than zero, 385 out of 385 design faults are detected. Given that the unsound implementation faults of FT5 are disregarding, 338 out of 343 (385-42*FT5) implementation faults are detected. By comparing the distribution of the number of extracted verification sequences by fault types in chart G and the distribution of unsatisfied verification sequences by fault types in chart I of Fig. 8, faulty mechanisms of shared resources (FT6), faulty scheduling properties (FT7), and missing transitions (FT8) cause the most comprehensive negative effect on the architectural design in terms of percentage of unsatisfied verification sequences: on average 82% (31/38), 74% (28.3/38), and 84% (19.5/23.3) respectively. At the implementation level, on the other hand, faults caused by missing connections (FT3) in addition to these fault types on average resulted in the largest portion of failed test cases, as presented in chart L.

In total, as explicitly presented in chart C of Fig. 7, 13844 verification sequences are extracted in the study, 11168 of which are selected for regression verification. 5886 of the selected verification sequences are unsatisfiable compared to 5886 in the re-run all approach. The consistency is presented in scatter chart D, where any data point divergence from the cross section between the selective approach (horizontal axis) and the re-run all approach (vertical axis) would indicate ineffectiveness of the selection process. The distribution of the number of selected verification sequences in contrast to a re-run all approach is presented in chart B of Fig. 7. For the majority of the faulty versions of the LTRIS AADL model (319 out of 385), between 35 to 40 sequences are extract to achieve full coverage for a re-run all approach. For 125 of these, the change impact analysis had little to no effect on the reduction of verification sequences to re-execute. For the other 260 (385-125) faulty versions of the LTRIS AADL model, the selection process on average reduced the necessary number of verification sequences to execute by approximately 19%

TABLE II. CASE STUDY RESULTS

			Model checking & sel. regr. ver. effecti.			Time consump.		Memory consump.		Sel. Efficiency		Testing effectiveness
	Fault type	No. ver. seq.	No. sel.	No. unsat. sel. obs.	No. unsat. obs.	Sel. (sec)	All (sec)	Sel. (MB)	All (MB)	Time (%)	Mem. (%)	No. failed TCs (of 38)
	0	38	n/a	n/a	0	n/a	249	n/a	1258	n/a	n/a	0
AVERAGE	1	37,8	27,7	19,2	19,2	285,3	319,6	1742,9	1983,8	12,3	12,9	19,4
	2	34,1	29,0	11,5	11,5	912,1	925,4	5437,2	5527,9	3,1	3,4	17,0
	3	34,4	34,4	13,1	13,1	595,0	595,0	3855,4	3855,4	0,0	0,0	20,3
	4	33,2	32,4	11,8	11,8	697,4	698,2	4057,9	4061,2	0,4	0,5	19,3
	5	38,0	29,0	6,0	6,0	518,5	530,0	3423,2	3477,7	3,1	2,2	0,0
	6	38,0	38,0	31,0	31,0	38,0	38,0	121,9	121,9	0,0	0,0	31,0
	7	38,0	38,0	28,3	28,3	82,2	82,2	444,0	444,0	0,0	0,0	27,4
	8	23,3	21,1	19,5	19,5	59,1	62,6	285,7	305,7	2,7	2,4	33,2
	9	38,0	24,4	1,0	1,0	306,3	340,9	2125,1	2350,1	11,4	11,1	1,0
	AVG:	**36,0**	**29,0**	**15,3**	**15,3**	**371,7**	**390,5**	**2293,1**	**2418,6**	**6,4**	**6,4**	**16,9**
DEVIATION	1	1,5	12,0	15,0	15,0	403,1	398,4	2682,2	2655,2	20,6	22,3	14,6
	2	7,1	8,4	7,4	7,4	509,1	499,1	2932,3	2856,9	9,3	10,2	8,8
	3	5,9	5,9	12,1	12,1	479,1	479,1	3288,7	3288,7	0,0	0,0	16,7
	4	6,3	6,4	10,3	10,3	525,7	525,6	3029,0	3028,0	0,7	1,2	13,2
	5	0,0	1,4	0,8	0,8	305,7	303,4	1892,4	1872,9	1,7	1,5	0,0
	6	0,0	0,0	0,0	0,0	0,0	0,0	6,4	6,4	0,0	0,0	0,0
	7	0,0	0,0	9,2	9,2	107,2	107,2	619,0	619,0	0,0	0,0	11,3
	8	5,0	4,1	6,7	6,7	80,3	86,2	472,8	511,1	4,7	4,5	7,7
	9	0,0	4,5	0,0	0,0	75,3	54,1	568,9	413,3	8,1	8,6	0,0
	STD:	**5,4**	**9,0**	**13,2**	**13,2**	**436,5**	**433,1**	**2697,4**	**2675,1**	**13,2**	**14,2**	**14,7**
MINIMUM	1	23	1	1	1	32	38	110	113	0,0	0,0	0
	2	23	3	1	1	123	224	789	1096	0,0	0,0	2
	3	22	22	1	1	38	38	119	119	0,0	0,0	0
	4	23	22	1	1	37	38	111	116	0,0	0,0	1
	5	38	27	5	5	271	281	1796	1849	0,7	0,3	0
	6	38	38	31	31	38	38	111	111	0,0	0,0	31
	7	38	38	5	5	38	38	108	108	0,0	0,0	0
	8	15	12	2	2	14	15	30	32	0,0	0,0	8
	9	38	20	1	1	221	273	1483	1863	3,0	1,8	1
	MIN:	**15**	**1**	**1**	**1**	**14**	**15**	**30**	**32**	**0,0**	**0,0**	**0**
MAXIMUM	1	38	38	38	38	2421	2422	16015	16017	66,0	71,4	38
	2	56	45	34	34	2551	2566	15356	15466	59,1	63,6	38
	3	38	38	34	34	1514	1514	10725	10725	0,0	0,0	38
	4	38	38	34	34	1791	1792	11133	11135	2,6	4,3	36
	5	38	30	7	7	1136	1144	7748	7774	5,0	4,3	0
	6	38	38	31	31	38	38	141	141	0,0	0,0	31
	7	38	38	35	35	481	481	2304	2304	0,0	0,0	35
	8	35	33	30	30	273	282	1484	1565	20,0	19,2	37
	9	38	29	1	1	403	416	3249	3308	20,2	20,8	1
	MAX:	**56**	**45**	**38**	**38**	**2551**	**2566**	**16015**	**16017**	**66,0**	**71,4**	**38**
TOTAL	1	4198	3079	2135	2135	31672	35471	193458	220197	10,7	12,1	2154
	2	2014	1713	679	679	53815	54598	320795	326146	1,4	1,6	1001
	3	241	241	92	92	4165	4165	26988	26988	0,0	0,0	142
	4	398	389	142	142	8369	8378	48695	48734	0,1	0,1	231
	5	1596	1218	252	252	21775	22262	143775	146062	2,2	1,6	0
	6	1596	1596	1302	1302	1596	1596	5118	5118	0,0	0,0	1302
	7	836	836	622	622	1809	1809	9769	9769	0,0	0,0	602
	8	723	655	603	603	1831	1942	8857	9476	5,7	6,5	1029
	9	2242	1441	59	59	18071	20115	125381	138653	10,2	9,6	59
	TOT:	**13844**	**11168**	**5886**	**5886**	**143103**	**150336**	**882836**	**931143**	**4,8**	**5,2**	**6520**

(1-11168/13844). Approximately 53% (5886/11168) of these are unsatisfiable (reveal faults). In terms of time and memory consumption, the selective approach on average reduces the necessary resources for regression verification by 6.4% (Table II). Note that the average reductions in percentage are averages of the reduction proportion of each individual data point and therefore do not correspond to the difference between the average consumption of the selective approach and the re-run all approach over all fault types. For example, a reduction of ten seconds of a re-run all execution of twenty seconds (50%) is weighted equal to a reduction of hundred seconds of a re-run all execution of two hundred seconds (50%) although the actual reduction in terms of seconds is ten times higher in the latter case. In total, time and memory are reduced by 4.8% and 5.2% respectively, which approximately correspond to 2 hours and 48 gigabyte of memory. The selective regression verification technique is on average most efficient with respect to faulty transition priorities (11.4% and 11.1% on average). At best, in terms of proportions, a reduction of time and memory by 66.0% and 71.4% is achieved. The selective approach has little to no effect on increased efficiency with respect to FT3, FT4, FT6, and FT7 (expected) in this study, as explicitly presented in chart K of Fig. 8.

The overhead expense of conducting the change impact analyses and the time needed to transform verification sequences to observer automata are negligible in the study and not included in Table II. The sum of all variables that influence the efficiency of the selective approach is presented in chart E of Fig. 7. The impact analysis took in total six minutes to execute. One minute is however saved, in total, by reducing the time needed to transform the necessary number of verification sequences to observers compared to a re-run all approach.

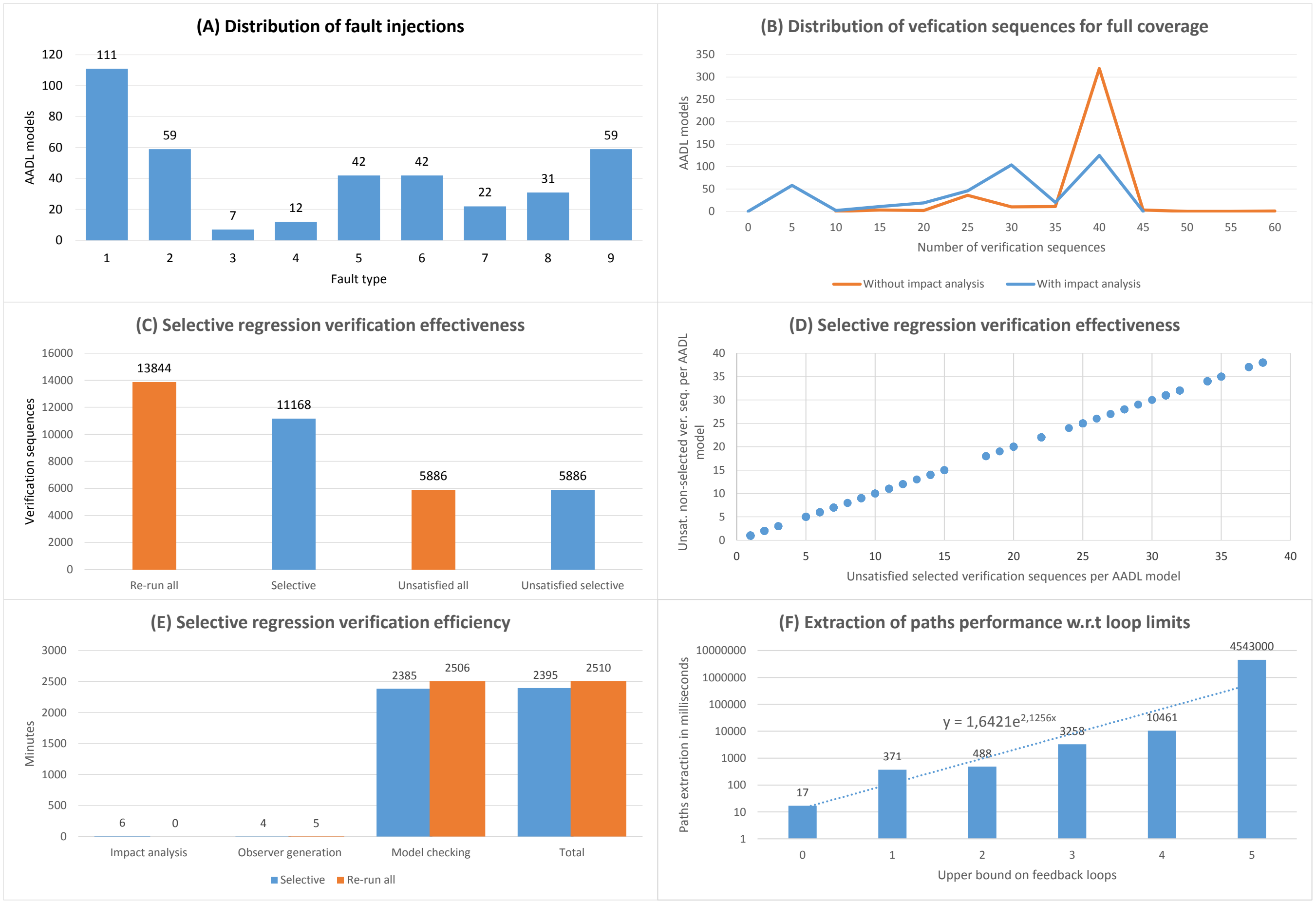

Fig. 7. Results charts A to F

Furthermore, the time consumption of extracting paths is not included in any measurement. With an upper bound on inter-component feedback loops of one (an indirect path may only include a component once), the time needed to extract paths is negligible (371 ms per model), as presented in chart F (an upper bound of zero constitutes no extractions of indirect paths). The measurements of the chart are based on extractions from the default fault-free model. However, the time needed for paths extraction grows exponentially with an increasing limit of feedback loops.

V. Discussion

Improvements of the AQAF theory [3] had to be developed before the presented results were achieved. During the initial instances of this study, we occasionally observed inaccurate impact analyses of architectural design changes that resulted in unsatisfied deselected verification sequences, i.e., some verification sequences that revealed faults were not selected. The cause of the inaccuracy is the lack of considerations to unconditional versus conditional dependencies/independencies in the dynamic slicing technique presented in [3]. We learned from this study that there may exist situations where the executability of an AFG path may be dynamically independent to a set of (other) paths as long as one path of the set, any of them, is executed prior to it, i.e., that the dynamic independencies to the paths of the set are conditional. Conditional multi-dependence/independence information is necessary for dynamical impact analyses of changes that modify several paths; changes to the paths of the set do not impact the executability of the path as long as one path of the set remains unchanged. Some of the injected faults in this study caused changes to such sets, given that each path of the set covered the location of the injected fault. Since conditional dynamic independences are regarded as unconditional by the dynamic slicing technique presented in [3], verification sequences that cover impacted paths were occasionally not selected for regression verification. By improving the dynamic slicing technique with conditional multi-independence analysis, as illustrated in [4], the selection process complied with the expectations in this study.

For 125 of the faulty versions of the LTRIS AADL model, the change impact analysis had little to no effect on the reduction of verification sequences to re-execute. By comparing the distribution of selected sequences and failed tests by fault types, presented in chart H and L of Fig. 8 respectively, FT1, FT3, FT4, FT6, FT7, and FT8 are centered around a moderately high number of selected verification sequences as well as a moderately high number of failed tests. This indicates that the 125 fault injections of which the selective approach has little effect upon do in fact have comprehensive impacts on the architecture. The change impact analysis is consequently not overly pessimistic with respect to these changes. It should be noted that the distribution of FT8 may seem to be clustered around a number of selected sequences that may appear moderately low. However, note

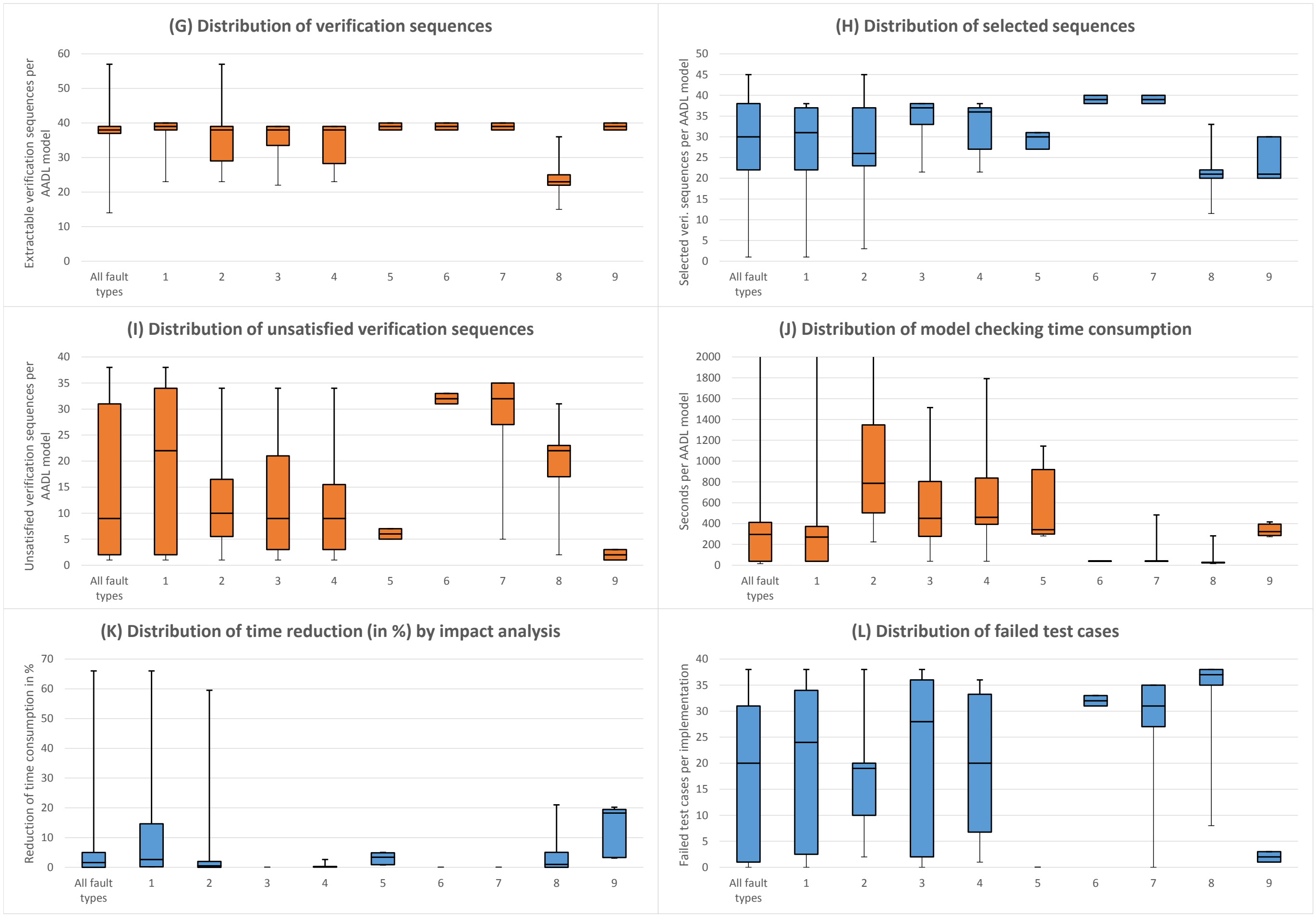

Fig. 8. Results charts G to L

that fault injections of FT8 significantly reduce the number of prescribed paths (23.3 compared to 38 of the default fault-free model), as presented in chart G. Consequently, the distribution of selected verification sequences by FT8 has a concentration of a relatively high number of selected sequences although it may not appear as such in chart H. Furthermore, the selective regression verification technique had little to no effect on increased efficiency for FT3, FT4, FT6, and FT7 (expected) in this study. We argue it is unreasonable to generalize this phenomenon to architectures that are composed of a larger set of functionally independent software components compared to LTRIS. A missing behavioral model transition (FT8) within LTRIS on average significantly reduces the number of prescribed paths, which indicates that the architecture of LTRIS is composed of rather functionally interdependent components. We therefore believe the efficiency of the selective regression verification technique may be significantly higher for architectures that have less functionally interdependent structures.

Five implementation faults that AQAT was expected to detect were not detected in the study. An analysis showed that the faults are not detected because the changed behavior still produced the same output, i.e., the post-conditions of the tests are fulfilled but the behaviors that cause them do not correspond to the architectural design. To detect non-conformance of the implementation with respect to the design under such circumstances, an implementation instrumentation that measures code coverage must be utilized.

The resource consumption of model checking varies significantly in the study. There are mainly two parameters that determine the outcome. First, a fault injection may add or remove states of the AADL model, which may exponentially reduce or increase the state space. Thus, possibly lengthens or shortens the state space search by a significant amount. Fault injections of FT6 produced the lowest average time consumption (38 seconds). The cause is a significantly reduced state space due to starvations, missed deadlines, and deadlocks if the change forces *Tester* to enter a critical region that is shared with *Controller*. Fault injections of FT2 produced the highest average time consumption (925.4 seconds). This is however not caused by a significantly increased state space. The significant increase of necessary resources is caused by multiple unsatisfiable verification sequences in relation to relatively large state spaces. Unsatisfiability of verification sequences can only be concluded by searching the complete state space with all possible combinations of system inputs, which tend to be relatively time consuming. Furthermore, the number of unsatisfied verification sequences varies significantly between faults of the same type. Consequently, the impact of a fault on the system architecture and the verification process is largely determined by the combination of its type and its location.

According to the results in chart F of Fig. 8, the time consumption of paths extractions increase exponentially with an increasing loop bound. Nevertheless, depending on the structure and behavior of the architecture, the necessary number of indirect paths to extract may decrease with an increasing feedback loop bound due to a larger coverage by each extracted path. For example, *Tester* of LTRIS requires multiple consecutive feedbacks from *Controller* in response to opening and closing orders to achieve a successful progress of the relay test process. With an upper feedback loop bound of three, a single indirect path may cover several different types of interactions between the components such that a complete relay test behavior is included in a single path, which otherwise are covered by a set of paths. Consequently, an increasing time consumption of paths extraction by increasing feedback loop bounds may, to a certain extent, advantageously reduce the time needed to perform satisfiability checking by reducing the number of verification sequences. An analysis of this threshold is consequently an interesting topic for future research for the purpose of performance optimization.

VI. Related work

Research in this field has developed a number of techniques for AADL, such as model checking techniques (e.g. [10] [11] [12]) and formal analysis techniques (e.g. [13] [14] [15] [16]). However, contrary to AQAT and AQAF, these contributions are lacking verification of architectural engineering that is conducted subsequent to an established design, such as generation of tests that demonstrate the compliance of an implementation with respect to its architectural design and impact analysis of architectural design changes for the identification of the necessary reverification measures. They are also lacking methods that measure and enforce coverage of the architectural design in the verification process, such as control and data flow coverage criteria applied by AQAT, which is essential to determine the extent to which an architecture has been verified. Furthermore, vital properties of critical embedded systems such as real-time constraints, scheduling properties, uses of shared resources, and concurrency by multitasking and parallel processing are not jointly considered in the verification.

Simulink Design Verifier [17] provides a formal verification and analysis framework similar to AQAT, however 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.

To our knowledge, there are only two cases of published related work that present quantitative evaluations of resource consumption. Murugesan et al. [18] present a study where the AADL model checker AGREE verifies a medical device against requirements expressed in past-time linear temporal logic (PLTL). The study resulted in a model checking time consumption of 273 seconds to prove 35 properties. The model checking approach presented by Murugesan et al. is however not comparable with AQAF in the sense that concurrent execution, shared resources, scheduling, and timing properties are not considered. Esteve et al. [19] present a study of COMPASS, a tool-set for SLIM (a variant of AADL), that provides the ability to model-check functional properties through transformation to Markov chain. The study constitutes an application to a satellite platform where model checking of both nominal behavior and error behavior is conducted. The resultant average time and memory consumptions of verification are 508 sec and 469 MB for checking of 19 properties. No inclusion of concurrent execution is presented.

VII. Conclusion and Future Work

We have presented an industrial evaluation of the Architecture Quality Assurance Tool (AQAT) and the underlying Architecture Quality Assurance Framework (AQAF), developed to provide a holistic, formal, and automated approach to verification of architectural engineering. AQAT provides architectural model checking, model-based testing, and selective regression verification techniques for the detection of design faults, implementation faults, and faults created by maintenance modifications. The fault detection effectiveness and the resource efficiency of AQAT are assessed by means of a case study comprising an application to a safety-critical train control system. The method of fault injection is used to ensure coverage of fault types and to produce a data sample size adequate for statistical analysis. The initial results did not comply with the expectations and indicated areas of improvement, in particular within the selective verification technique of AQAF. By improving the impact analysis of architectural design changes with an analysis of conditional and unconditional dependencies, satisfactory results were achieved. Final results suggest a 100% fault detection rate at the model level, a 98.5% fault detection rate at the implementation level, and an average increased efficiency of regression verification by 6.4% compared to a re-run all approach. We learned that an increased fault detection rate at the implementation level requires a monitoring instrumentation that measures code coverage during the execution of the tests. Due to results that indicate a rather interdependent structure of the case subject, we believe the efficiency of the selective regression verification technique may increase significantly for systems with less interdependent structures.

The main area of evaluation improvement is an application to a diverse set of systems. Without results from applications to extreme cases, e.g. a system with a large set of concurrent tasks, the scalability limitations of AQAT and AQAF remain inconclusive. Furthermore, the method of fault injection is utilized in this study to emulate erroneous architectural engineering. Although creations of these faults are plausible in development processes, an assessment of the usability of AQAT and AQAF requires studies with more authentic scenarios to be reliable.

Acknowledgments

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] 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, pp. 31–40.

[4] A. Johnsen, K. Lundqvist, K. Hänninen, and P. Pettersson, "AQAT: The Architecture Quality Assurance Tool for Critical Embedded Systems," in *Proceedings of the 28th International Symposium on Software Reliability Engineering (ISSRE)*, October 2017.

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

[6] ——, "Architecture Analysis & Design Language (AADL)," SAE Standards n^o AS5506, November 2004.

[7] "ISO/DIS 26262-1 - Road vehicles – Functional safety," Geneva, Switzerland, Tech. Rep., Jul. 2009.

[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] 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.

[11] 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.

[12] 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.

[13] G. Chen, L. Luo, R. Gong, and S. Gui, "Dependability Analysis for AADL Models by PVS," in *Proceedings of the 2009 Eighth IEEE International Conference on Dependable, Autonomic and Secure Computing*, ser. DASC '09. IEEE Computer Society, 2009, pp. 19–24.

[14] S. Gui, L. Luo, Y. Li, and L. Wang, "Formal Schedulability Analysis and Simulation for AADL," in *Embedded Software and Systems, 2008. ICESS '08. International Conference on*, July 2008, pp. 429–435.

[15] S. Björnander, L. Grunske, and K. Lundqvist, "Timed Simulation of Extended AADL-Based Architecture Specifications with Timed Abstract State Machines," in *QoSA '09: Proceedings of the 31st International Workshop on Software Quality*, June 2009, pp. 101–115.

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

[18] 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.

[19] 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.