Automated Verification of AADL-Specifications Using UPPAAL

Andreas Johnsen, Kristina Lundqvist, Paul Pettersson, Omar Jaradat
School of Innovation, Design and Engineering
Mälardalen University
Västerås, Sweden
forename.surname@mdh.se

***Abstract*—The Architecture Analysis and Design Language (AADL) is used to represent architecture design decisions of safety-critical and real-time embedded systems. Due to the far-reaching effects these decisions have on the development process, an architecture design fault is likely to have a significant deteriorating impact through the complete process. Automated fault avoidance of architecture design decisions therefore has the potential to significantly reduce the cost of the development while increasing the dependability of the end product. To provide means for automated fault avoidance when developing systems specified in AADL, a formal verification technique has been developed to ensure completeness and consistency of an AADL specification as well as its conformity with the end product. The approach requires the semantics of AADL to be formalized and implemented. We use the methodology of *semantic anchoring* to contribute with a formal and implemented semantics of a subset of AADL through a set of transformation rules to timed automata constructs. In addition, the verification technique, including the transformation rules, is validated using a case study of a safety-critical fuel-level system developed by a major vehicle manufacturer.**

I. INTRODUCTION

Safety-critical systems such as real-time embedded systems within the domains of avionics and automotive are developed under stringent requirements on high dependability. Fault avoidance, in addition to fault tolerance, is essential to achieve high dependability levels. Fault avoidance represents means focused on producing a fault-free system, both by proactive methods which prevent faults from being introduced (known as fault prevention), and by reactive methods which remove faults that have been introduced (known as fault removal). Although the development of a fault-free system seldom is practical, and has to be supported by means of retaining dependability even in the presence of faults (known as fault tolerance), automated fault avoidance is necessary to guarantee dependability within a competitive schedule and budget. One of the most critical development phases with respect to fault avoidance is the architecture design phase where the design decisions needed to achieve the required quality attributes, such as dependability and performance, are determined. Due to the fact that these architecture design decisions have the most far-reaching effects on the development, and are the most costly to correct at a later stage, they are the most critical design decisions to evaluate.

This work was partially supported by the Swedish Research Council (VR), and Mälardalen Real-Time Research Centre, Mälardalen University.

The Architecture Analysis and Design Language (AADL) [1] has been developed to provide a formalism from which architectures of real-time embedded systems can be designed and analyzed. In order to provide means for automated fault avoidance when designing systems by AADL, a verification technique based on formal methods has been developed in previous work [2]. Automated fault avoidance is provided through the entire development process by the adaptation of both model checking and model-based testing approaches to an architectural perspective. Assuming a development process where initially a system designed to fulfill the system requirements is specified in AADL, and where the specification is used as a blueprint to guide the subsequent development process, the goals of the technique are:

Goal 1. to ensure completeness and consistency of an AADL specification through model-checking before the development process progresses, and

Goal 2. to ensure conformance of the end product with respect to its AADL specification through model-based testing.

The approach however requires the semantics of AADL to be formalized and implemented. The main contributions of this paper are a formal and implemented semantics of a subset of AADL including features commonly used in real-time systems, and a validation of the verification technique. Formal and implemented semantics is achieved through formally defined transformation rules to constructs in timed automata – the input language to the UPPAAL model checker [3]. The methodology is known as *semantic anchoring*, where the defined transformation rules anchor the semantics of AADL to the – formal and implemented – semantic domain of timed automata in the UPPAAL environment. In addition, a transformation to timed automata allows for the use of the UPPAAL model checker to automatically perform verification of the specification through model-checking, and verification of the implementation through model-based conformance testing. It is important to note that

the transformation rules also enable simulation of AADL specifications, which provides numerous benefits in the development of safety-critical systems, such as improving correctness analysis and understandability. The technique has been applied on a safety-critical fuel-level estimation system developed by a major vehicle manufacturer as an initial validation of its applicability and scalability to industrial systems.

The rest of this paper presents an overview of AADL and the verification technique in Section II. The transformation rules from AADL to timed automata are presented in Section III. Results of the case study are presented in Section IV, followed by related work in Section V and finally concluding remarks in Section VI.

II. AADL and The Verification Technique

AADL was initially released and published as a Society of Automotive Engineers (SAE) Standard AS5506 [1] in 2004, and a second version (AADLv2) was published in 2009. It is a textual and graphical language used to model, specify and analyze software- and hardware-architectures of real-time embedded systems. AADL is based on a component-connector paradigm that hierarchically describes components, component interfaces and the interactions (connections) among components. Hence, the language captures functional properties of the system, such as input and output through component interfaces, as well as structural properties through configurations of components, subcomponents and connectors. Furthermore, means to describe quality attributes, such as timing and reliability, are also provided. AADL defines component abstractions dividable into three groups: application software components (process, thread, thread group, data, subprogram and subprogram group), execution platform components (memory, bus, virtual bus, processor, virtual processor and device) and general composite components (system and abstract). In this paper, we will focus on threads, processes and processors. A thread component represents a schedulable and concurrent unit of (sequential) execution, a process component represents a protected address space (must contain at least one thread subcomponent), and a processor component represents hardware and software responsible for scheduling and executing threads.

AADL specifications have explicit control-flows and data-flows through the architecture as defined in [2]. The flows are dependent on how components transfer control and data through their interfaces which is described in the standard with a precise execution model (dynamic semantics). The possible interactions among components are represented by four different types of connections: *port connections*, *data access connections*, *subprogram calls* and *parameter connections*. *Port connections* represent a transfer of data, control or both, depending on the type of interconnected interfaces (data port, event port or event data port). *Subprogram calls* represent a transfer of control whereas *parameter connections* and *data access connections* represent a transfer of data.

The runtime configuration of subcomponents and their interactions within a component may change if it is specified with *modes*. For each mode, it is possible to set the active components and connections, mode-specific subprogram calls and mode-specific properties. Furthermore, the logical execution of thread and subprogram components may be modeled by using the Behavioral Annex (BA) [4]. Logical execution is modeled through states, state variables and transitions operating on a component's interfaces which consequently refines the interactions with other components.

The four different types of connections specify the architectural control-flows and data-flows of an AADL specification. These flows may be dependent on mode state machines, refined by the BA and constrained by associated property annotations where conflicts may occur between these constructs.

The objective of the verification criteria defined in the verification technique [2] is, with respect to the semantics (semantic rules) of AADL and Goal 1, to ensure consistency and completeness of and between the AADL flows, their refinements and their constraints through the analysis of **control-flow reachability**, **data-flow reachability** and **concurrency** among flows. Consequently, the verification technique is, with respect to Goal 1, comparable with semantic analysis techniques in compilers.

- **Control-flow reachability** is the property where each architectural element in an execution order can reach the subsequent element to be executed without conflicting constraints of the control-flow.
- **Data-flow reachability** is the property where each data element can reach its target component, where the data is used, from its source component, where the data is defined. The data element should reach the target component without conflicting constraints of the data-flow. Analysis of single flows of data or control is not enough since there are implicit relations between them that may cause deadlocks in the system.
- **Concurrency among flows** is the property where relations between flows should not prevent control-flow reachability or data-flow reachability, and where the system should be free from deadlocks.

Note that the analysis of flows consider constraints from AADL property annotations, such as, the latency of a data flow, or the period, execution time and deadline of threads. Hence, analysis of control-/data-flow reachability and concurrency among flows imply analysis of **additional** aspects, such as timing and schedulability.

To perform the analysis the control- and data-flow diagrams must be extracted from the AADL specification. This is done through defined AADL relations which three of them

we briefly present here but a complete description and the formal definitions can be found in [2]. In the definitions below, N denotes a component, N.Int denotes an interface of a component N, and C denotes a connection. **Component Internal Relation** defines the (possibly constrained) data or control transfer that is generated between two interfaces of a component that are connected through a connection or a BA (Behavior Annex). Relation b in Figure 1 illustrates the definition, where the BA connection *C2* internally connects $N2.Int_1$ to $N2.Int_2$. **Direct Component to Component Relation** defines the (possibly constrained) data or control transfer that is generated between two components that are directly connected through a connection. Both relation a and c illustrate the definition, where $C1$ connects $N1.Int_1$ to $N2.Int_1$, and $C3$ connects $N2.Int_2$ to $N3.Int_1$. **Indirect Component to Component Relation** defines the (possibly constrained) data or control transfer that is generated between two components that are indirectly connected through one or several component(s). Relation d illustrates the definition, where $C1$, $C2$ and $C3$ indirectly connect $N1.Int_1$ to $N3.Int_1$. Based on the relations three types of verification

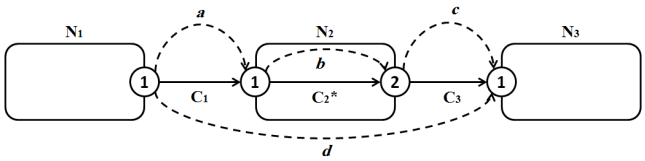

Figure 1. Illustration of relations between three interconnected components. "*" denotes a BA connection.

sequences have been defined [2]. A verification sequence is a sequence of interfaces where each interface has a relation to the subsequent interface in the sequence. They describe the possible internal, direct and indirect paths of execution in control- and data-flow diagrams, that is, the possible interactions among interfaces in an AADL specification. For example, since there exists an indirect relation d from $N_1.Int_1$ to $N_3.Int_1$ as shown in Figure 1, there exists an indirect path of a sequence on the form $Path_d = N_1.Int_1 \rightarrow N_2.Int_1 \rightarrow N_2.Int_2 \rightarrow N_3.Int_1$. A path is constrained if any member in the sequence is associated with a property annotation. For example, assume that relation d exists due to an end-to-end data-flow from $N_1.Int_1$ to $N_3.Int_1$ through the interfaces and that the flow is specified with a *latency* property, path $Path_d$ will be constrained such that the time from when data i sent through $N_1.Int_1$ until it is received through $N_3.Int_1$ must not exceed the value of the property. In addition, assuming that each component in $Path_d$ is a thread specified with *Compute_Execution_Time*, *Compute_Deadline* and *Priority* properties, path $Path_d$ will be constrained such that each thread must execute for a specified amount of time before output is generated, each thread must generate output before a specified deadline, and the sequence of thread execution must not contradict their priorities.

In order to verify the completeness and consistency of an AADL specification, all paths must be verified that they fulfill their constraints. Or to be more precise, each path must be able to be executed according to the semantics (semantic rules) of AADL such that each (active) property-value is valid in each state of the execution. The AADL specification is consistent if each path is free from contradictory behavior, that is, each path does not contradict Control-flow reachability, Data-flow reachability and Concurrency among flows. The AADL specification is complete if each path not yielding an end-to-end flow (typically a sensor-to-actuator flow) is subsumed in another path.

End-to-end paths can also be used to generate test cases for the implementation to test the conformity with its AADL specification. The objective of the verification criteria defined in the verification technique [2] is, with respect to the semantics (semantic rules) of AADL and Goal 2, to ensure conformance of the end product with respect to its AADL specification. From each end-to-end path, it is possible to derive the initial expected state of the system, the input needed to stimulate an execution according to the expected path, the expected output and the expected timing constraints. A step by step description of the complete verification technique can be found in [2].

However, the verification of paths must be executed in accordance to the dynamic semantics (execution model) of the language. To perform this in a formal and automated way the dynamic semantics of AADL must first be formalized and implemented (see Section III). We provide a formal and implemented semantics of AADL through transformation rules (mappings) to timed automata, the input language to the UPPAAL model checker.

III. TRANSFORMATION TO TIMED AUTOMATA

AADL supports specification of real-time and scheduling properties such as dispatch protocol, deadline, execution time, scheduling policy, period, etc. The AADL execution model is described in the AADL standard by terms of thread (automata) states, thread dispatching, and thread scheduling and execution states. It consists of several different aspects of a run-time environment, such as synchronous interactions, asynchronous interactions, nominal execution, recovery execution, etc. The standard defines a default run-time environment with synchronous interactions and pre-emptive scheduling where periodic threads interact through data ports. Data is available at the input ports at the time of a thread's dispatch, which is followed by computation of output data that is transmitted (from out ports) at the time of the sending thread's completion or deadline, to (the in ports of) the recipient thread. Port connections can either be specified as immediate or delayed, the former transmits data at a thread's completion whereas the latter transmits data at a thread's deadline.

Since these features are commonly used in real-time systems, we define formal semantics for a subset of AADL consisting of synchronous interactions with fixed-priority preemptive scheduling (non-preemptive scheduling is subsumed). Formal semantics is defined through transformation rules to the UPPAAL modeling language, an extension of the timed automata theory. The AADL subset and timed automata in UPPAAL are defined as follows, upon which the transformation rules are defined.

Definition 1. An AADL-specification $A = \langle Pr, T, C\rangle$ has a processor component Pr, a set of thread components T and a set of data port connections C of the form *Connection* ::= *Identifier* **: data port** *source_port_reference* **− >** *destination_port_reference***;** | *Identifier* **: data port** *source_port_reference* **− >>** *destination_port_reference***;**.
Let t range over T and c over C.
A thread $t_i = \langle Ident_i, Interf_i, Sch\_Prop_i, MSM_i, BM_i\rangle$ has an identifier $Ident_i$, a set of in and out data port interfaces (features) $Interf_i = In\_Interf_i \cup Out\_Interf_i$ of the form *Feature* :: = *Identifier* **: in data port ;** | *Identifier* **: out data port ;**, scheduling properties $Sch\_Prop_i = \langle Dispatch\_Protocol_i, Period_i, Compute\_Execution\_Time_i, Compute\_Deadline_i, Priority_i\rangle$ of the form *Property* ::= *Identifier* => *Value*, a mode state machine MSM, and a behavioral model BM[1]. We assume that the value of the $Dispatch\_Protocol$ is "periodic".
Let $Connect : Interf \rightarrow C$ be a function which assigns active connections to interfaces. A processor $Pr = \langle Ident, T\_Bind, Scheduling\_Protocol\rangle$ has an identifier $Ident$, a set of threads bound to it $T\_Bind \subseteq T$ and a scheduling protocol property.

Definition 2. A timed automaton $TA = \langle L, \ell_o, X, Var, I, E\rangle$ has a set of locations L, an initial location $\ell_0 \in L$, a set of real-valued variables X called *clocks*, a set of (bounded) integer-typed variables Var, a function assigning invariants to locations $I : L \rightarrow G$ and a set of edges $E \subseteq L \times G \times Act \times U \times L$, where G is a (possibly empty) set of *guards* which are conjunctions of predicates over variables and clock constraints of the form $x\, expr_1\, c$, where $x \in X$, $c \in \mathbb{N}$ and $expr_1 \in \{<, \leq, \geq, >\}$. $Act = I \cup O \cup \{\tau\}$ is a set of input (denoted a?) and output (denoted a!) synchronization actions and the non-synchronization τ. U is a (possibly empty) set of *updates* which are sequences of variable-assignments of the form $v := expr_2$ and/or clock resets of the form $x := 0$, where $v \in Var \cup Var_G$, $x \in X$ and $expr_2$ is an arithmetic expression over integers. We shall use the denotation $\ell \xrightarrow{g,a,u} \ell'$ iff $\langle \ell, g, a, u, \ell'\rangle \in E$.
Semantically, a TA is defined by a timed transition system over states which are pairs of the form $\langle \ell, \phi\rangle$, where $\ell \in L$, $\phi \in \mathbb{R}_+^X$ is a clock valuation and $\phi \models I(\ell)$. Progress is either performed through delay transitions $\langle \ell, \phi\rangle \xrightarrow{d} \langle \ell, \phi \oplus d\rangle$ where $\phi \oplus d$ is the result of adding the delay d to each clock valuation in ϕ, or by discrete transitions $\langle \ell, \phi\rangle \xrightarrow{a} \langle \ell', \phi'\rangle$ where an (instantaneous) edge $\langle \ell, g, a, u, \ell'\rangle$, such that $\phi \models g$, is taken from a location ℓ to another location ℓ'.
A network of timed automata $NTA = \langle \overline{TA}, Var_G, X_G, Ch\rangle$ has a vector of n timed automata $\overline{TA} = \langle TA_0, TA_1, \ldots, TA_{n-1}\rangle$, a set of shared (global) variables Var_G, a set of shared clocks X_G, and a set of synchronization channels Ch. The semantics of a NTA is defined by a timed transition system such that a state is a location vector over all automata and the union of clock valuations from all automata where delay transitions are synchronized and discrete transitions are synchronous over complementary actions (a? complements a!).

In addition, UPPAAL extends the timed automata theory with the possibility to declare locations as urgent or committed and coding of functions (in UCode, a subset of C) callable at transitions. In an urgent location, time is not allowed to progress whereas in a committed location, time is not allowed to progress and the next transition must involve one of its outgoing edges. A timed automata path or trace is defined as a sequence of states such that there exist transitions from each state in the sequence leading to its successor state. Finally, we shall use $Id(c)$, $Id(t_i)$, $Id(Var)$ etc., to denote the *identifier* of the respective element whereas $Val(Var_G)$, $Val(Priority_i)$, etc., is used to denote its *value*.

A. Transformation Rules

In this section, we firstly present the formally defined transformation rules, and secondly, we describe each transformation rule informally while applying them to an AADL-example presented in Example 1. Due to the complexity of the semantics of AADL processor components, the details of the corresponding rules are presented in a separate subsection, Section III-A1.

The transformation rules are defined by means of functions where a transformation is initiated through function $\mathcal{T}_A : A \rightarrow NTA$ which maps an AADL specification A of the form $\langle Pr, T, C\rangle$ to a network of timed automata NTA.

Rule $\mathcal{T}_A$: $\mathcal{T}_A(A) = \langle \overline{TA}, Var_G, \emptyset, Ch\rangle$ such that $\overline{TA}[0] = \mathcal{T}_{Pr}(Pr)$ and for $0 \leq i < |T|$, $\overline{TA}[i+1] = \mathcal{T}_T(t_i)$, $Var_G = \{\mathcal{T}_C(c) \mid c \in C\}$ and Ch is generated as presented in Section III-A1.

Function $T_{Pr} : Pr \rightarrow TA$ maps a processor component Pr of the form $\langle Ident, T\_Bind, Scheduling\_Protocol\rangle$ to a timed automaton TA.

[1]Mode state machines and behavioral models are only briefly discussed due to space limitations

Rule $\mathcal{T}_{\mathbf{Pr}}$: $T_Pr(Pr) = \langle L, \ell_o, X, Var, I, E\rangle$ where L, ℓ_o, X, Var, I and E are as defined in Section III-A1.

Function $\mathcal{T}_T : T \rightarrow TA$ maps a thread component t_i of the form $\langle Ident_i, Interf_i, Sch\_Prop_i, MSM_i, BM_i\rangle$ to a timed automaton TA.

Rule $\mathcal{T}_{\mathbf{T}}$: $\mathcal{T}_T(t_i) = \langle L, \ell_o, X, Var, I, E\rangle$ such that $L = \{awaiting\_dispatch, ready, running\}$, $\ell_0 = awaiting\_dispatch$, $X = \{cl\}$, $Var = \{Period, C\_E\_T, C\_D, Priority\} \cup VarIn \cup VarOut$ where $Val(Period) = Val(Period_i)$,..., $Val(Priority) = Val(Priority_i)$, $VarIn = \{\mathcal{T}_{Interf}(int) \mid int \in In\_Interf_i\}$ and $VarOut = \{\mathcal{T}_{Interf}(int) \mid int \in Out\_Interf_i\}$. Let Vin and $Vout$ range over $VarIn$ and $VarOut$ respectively.
$I(awaiting\_dispatch) = \{cl <= Period\}$ and $E = \{awaiting\_dispatch \xrightarrow{cl>=Period, dispatched[i]!, u_1} ready, ready \xrightarrow{run[i]?} running, running \xrightarrow{preempt[i]?} ready, running \xrightarrow{complete[i]?, u_2} awaiting\_dispatch\}$ where $u_1 = \langle sch\_info[i][0/1/2] := C\_E\_T/C\_D/Priority, Vin_0 = Id(Connect(Vin_0)), Vin_1 = Id(Connect(Vin_1)), \ldots, Vin_n = Id(Connect(Vin_n)), cl := 0\rangle$ and $u_2 = \langle Id(Connect(Vout_0)) := Vout_0, Id(Connect(Vout_1)) := Vout_1, \ldots, Id(Connect(Vout_m)) := Vout_m\rangle$.

Function $\mathcal{T}_C : C \rightarrow Var_G$ maps an interface connection c to a global variable v_G.

Rule $\mathcal{T}_{\mathbf{C}}$: $\mathcal{T}_C(c) = v_G$, such that $Id(c) = Id(v_G)$.

Function $\mathcal{T}_{Interf} : Interf \rightarrow Var$ maps a data port interface int to a (local) variable v.

Rule $\mathcal{T}_{\mathbf{Interf}}$: $\mathcal{T}_{Interf}(int) = v$, such that $Id(int) = Id(v)$.

In the following example we will describe an application of the transformation rules with respect to the (incomplete) AADL specification given in Table I, and under the assumption that there is a scheduler automaton (described in Section III-A1) providing the required thread behavior.

Example 1. Table I comprises typical constructs in the chosen subset of AADL. It includes a process component *process_example.impl* with two thread subcomponents *thread_1* and *thread_2*, which are instances of *thread_example1.impl* and *thread_example2.impl* (not shown in the example) respectively. The *process_example.impl* has two in data ports *Input_1* and *Input_2*, as shown by its process type *process_example*. These ports are connected to the in data ports of *thread_example1.impl*, as shown by the connection declarations of *process_example.impl*. Furthermore, these connection declarations do also show that the out data ports of *thread_example1.impl* are connected to the in data ports of *thread_example2.impl*. The execution platform of the system has been left out in the example due to limited space, however, we assume that the threads are bound to a processing unit with a fixed-priority preemptive scheduler. We refer to *thread_example1(.impl)* when presenting transformation to the thread automaton shown in Figure 2. Note that the following (informal) description of threads' behavior is traceable to the AADL standard unless it is explicitly stated that it is not.

Table I
AN AADL EXAMPLE OF A PROCESS COMPONENT WITH TWO THREAD SUBCOMPONENTS.

```
process process_example
  features
  Input_1: in data port int;
  Input_2: in data port int;
end process_example;

process implementation process_example.impl
  subcomponents
  thread_1: thread thread_example1.impl;
  thread_2: thread thread_example2.impl;
  connections
  Connection_1: data port Input_1 –>
    thread_1.InputPort_1;
  Connection_2: data port Input_2 –>
    thread_1.InputPort_2;
  Connection_3: data port thread_1.OutputPort_1 –>
    thread_2.InputPort_1;
  Connection_4: data port thread_1.OutputPort_2 –>
    thread_2.InputPort_2;
end process_example.impl;

thread thread_example1
  features
  InputPort_1: in data port int;
  InputPort_2: in data port int;
  OutputPort_1: out data port int;
  OutputPort_2: out data port int;
end thread_example1;

thread implementation thread_example1.impl
  properties
  Dispatch_Protocol => Periodic;
  Period => 50ms;
  Compute_Execution_Time => 5ms..10ms;
  Compute_DeadLine => 30ms; – –by default equal to period
  Priority => 1;
end thread_example1.impl;
...
```

In order to transform the given AADL specification **Rule $\mathcal{T}_{\mathbf{A}}$** is first applied. Through the rule, each thread is mapped (**Rule $\mathcal{T}_{\mathbf{T}}$**) to an automaton such as in Figure 2. The AADL execution model (dynamic semantics) is partly specified in the AADL standard as a hybrid automaton describing the different states of an AADL thread from a scheduler's perspective. This hybrid automaton can be reduced to an UPPAAL automaton consisting of three locations — *awaiting*

dispatch, *ready* and *running*. The automaton describes the different states of a thread that are common for all threads, and thus form the basis of the transformation rules since any AADL thread can be mapped to such an automaton. Each thread is initially in the *awaiting_dispatch* location where a transit to the *ready* location depends on the thread's dispatch protocol property. For periodic threads, the occurrence of dispatch is entirely dependent on its period in relation to a clock. A local clock (cl) is used to keep track of dispatches of the particular thread, which is acceptable – from a synchronous perspective – since all UPPAAL clocks progress synchronously. Input on the input ports is frozen and accessible at the time when a thread dispatches and enters the *ready* location. The edge is synchronized with the scheduler to notify the dispatch where the thread's scheduling properties are made available for the scheduler through its Identifier (i). The scheduling properties are mapped to local integer variables *C_E_T*, *C_D* and *Priority*, which with respect to the thread specification are assigned to 10 (worst case execution time considered here), 30 and 1 respectively[2]. Input ports of a thread are mapped (**Rule** $\mathcal{T}_{\mathbf{Interf}}$) to local variables (e.g. *InputPort_1*) which are assigned at dispatch by global variables (e.g. *Connection_1*) mapped (**Rule** $\mathcal{T}_{\mathbf{C}}$) from the connections the input ports are involved with.

Arrival of new input is accessible at the next dispatch or at specified input times (requires modification of the automaton). Threads in the ready location are assigned to be executed, by the processor component they are bound to, according to a scheduling policy property. Assuming a scheduler with fixed priority scheduling policy with preemption, the thread with the highest priority is selected to run on the processor and thus transits, through synchronization with the scheduler, to the *running* location. No more than one thread is allowed to be in a *running* location simultaneously where a running thread can be preempted if a thread with higher priority enters the *ready* location. A thread in the *running* location that completes its execution transits to the *awaiting_dispatch* location to repeat its life cycle. Data on a thread's output ports are transmitted through the connections at a thread's completion, that is, if the connections are declared as immediate. The connections shown in Table I are immediate (represented by "–>") and thus map to the completion edge in the automaton. Output ports and their connections are mapped in the same manner as with the input ports. For delayed connections (represented by "–>>"), the completion edge has to be extended with an intermediate location guarded until the time of the thread's deadline.

1) The Scheduler Automaton: An AADL processor component is mapped (**Rule** $\mathcal{T}_{\mathbf{Pr}}$) to a scheduler automaton such as shown in Figure 4, which provides the required behavior

[2]The mechanism of handling scheduling properties of threads is solely for representation purposes and is not derived from the AADL standard

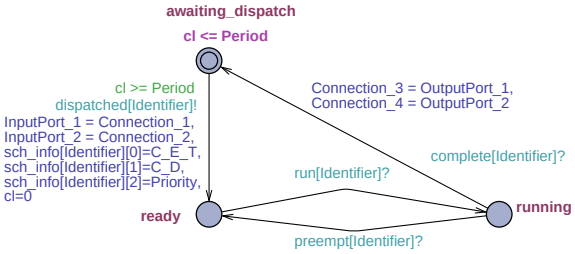

Figure 2. The thread automaton.

of threads bound to a processor. A detailed description of its elements can be found in [5]. The automaton includes two clocks per thread, lists and functions with corresponding variables to handle the thread execution and preemption. The reason for having two clocks per thread (sch_clocks[i][2] is a list of clocks in sets of two, each set referenced by a thread identifier i) is that the UPPAAL language does only allow reset and comparison of clocks, i.e., clocks cannot be read or assigned/set. Because of these constraints, a preempted thread's time of completion can not be obtained solely from its execution time. In order to model thread preemption, a method considering the execution time of the threads causing preemption is used to calculate preempted threads' time of completion.

The method is illustrated in Figure 3. A, B and C are denotations for threads where priority of A $<$ priority of B $<$ priority of C. C_A is the execution time of A and D_A is the deadline for A. c_A (sch_clocks[i][0]) and d_A (sch_clocks[i][1]) are clocks for A, which are used to measure the time of completion and the time of a missed deadline respectively. r_A is a variable used to summarize the time required to complete thread A and all – during the execution of A – dispatched threads with priority higher than A. As shown in the illustration, the time of completion for thread A is when the comparison $c_A = r_A$ evaluates to true. In addition to this comparison, $d_A > D_A$ should not evaluate to true before or while A's completion. The comparison is used for schedulability analysis where an evaluation to true indicates a missed deadline. Note that we are illustrating the method explicitly for thread A though the methodology is applied to each thread. A formal proof of the methodology is presented in [6]. Furthermore, the behavior of the scheduler assumes immediate switching-time of threads. If the processor the threads are bound to is specified with a *Thread_Swap_Execution_Time* property, the scheduler has to be modified with intermediate locations delaying the switching-time according to the specified property.

The scheduler is initially in the *Empty* location, awaiting until the occurrence of a dispatch. When dispatch occurs, the scheduler transits to the *Schedule1* location whereby the corresponding thread is added to the *ready_queue* via the *schprotocol()* function and its deadline clock is reset (corresponds to $d_A = 0$ in Figure 3). The *Schedule1*

location is a committed repetition of the *Empty* location, allowing several threads to be dispatched (through the edge to the same location) simultaneously. Succeeding to all simultaneous dispatches, the scheduler synchronizes with the first thread in the *ready_queue* and transits to the *Running* location through one of two different edges depending on which action should be executed. If the number of preempted threads is zero, or if the number is more than zero and the latest preempted thread is not the first in the *ready_queue*, the execution time clock of the thread to be run is reset (corresponds to $c_A = 0$). If the number of preempted threads are more than zero and the latest preempted thread is the first thread in the *ready_queue*, the scheduler transits to the *Running* location without reseting its execution time clock since it already has been reset (corresponds to the start of execution of A after preemption by B and C). The scheduler remains in the *Running* location until the

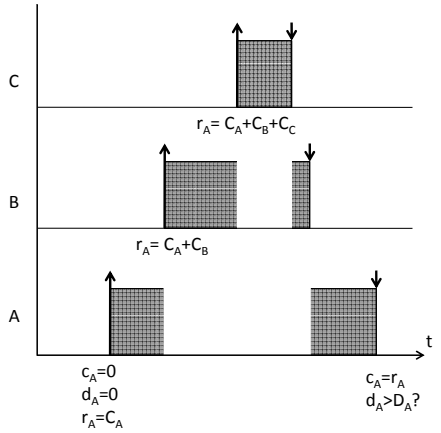

Figure 3. Thread execution schema for threads A, B and C, where ↑ indicates dispatch and ↓ indicates completion.

running thread completes its execution, until another thread is dispatched, or until the running thread misses its deadline. If a running thread completes its execution (corresponds to $c_A = r_A$), the scheduler transits to the "anonymous location" through one of two different edges. Note that the running location is modeled with an invariant in order to force a fire of the completion-edge at the time of completion. The two edges have guards for execution time where additional expressions are used to differentiate between a preempted thread and a thread which has not been preempted. If the thread has not been preempted, the thread is simply removed from the *ready_queue* through the *completion()* function. A preempted thread, on the other hand, is not only removed from the *ready_queue*, but also from the *preempt_stack*. From the "anonymous" location, the scheduler transits to the *Empty* or *Schedule1* location depending on whether there are any dispatched threads left or not.

If a dispatch occurs when the scheduler is in the *Running* location, an edge is fired to the *Schedule2* location, whereupon the thread is added to the *ready_queue* and the corresponding execution time clock is reset. Two different edges are available from the *Schedule2* location depending on if the recently dispatched thread was scheduled as the first thread in the *ready_queue* or not. If scheduled as the first thread in the *ready_queue*, that is, if it preempts the latest running thread, the scheduler transits to the *Preemption* location from where the thread is synchronized for execution. Whereby the edge from the *Schedule2* location to the *Preemption* location, the preempted thread is added to the *preempt_stack* and preempted time is added – to all preempted threads – through the *addTime()* function (corresponds to $r_A = C_A + C_B$ or $r_A = C_A + C_B + C_C$). On the other hand, if the recently dispatched thread does not cause a preemption, no further actions are taken other than adding preempted time – if the thread is scheduled prior to currently preempted threads – to preempted threads, through the *checkTime()* function.

The edge from the *Running* location to the *MissedDeadline* location is modeled for schedulability analysis. Or to be more specific, to provide effective means for checking that control flows do not contradict their scheduling constraints. If the edge is fired (corresponds to $d_A > D_A$), it indicates a missed deadline for the currently running thread. Hence, all direct, internal and indirect paths can be verified to be consistent with the scheduling properties simply by checking if the *MissedDeadline* location is reachable in the corresponding timed automata paths. Note that specified priorities of threads may contradict the modeled interactions among threads (incorrectly specified priorities may prevent interactions to take place although they are modeled), however, the mere ability to exercise the UPPAAL model according to the path imply consistency with the priorities. Furthermore, note that a fire of the edge to the *MissedDeadline* location yields a deadlock in the system due to the design of the model and not due to the dynamic semantics of AADL, which has to be considered at deadlock analysis.

IV. CASE STUDY

In this section we will present an overview of the results from applying the verification technique to a safety-critical fuel level system developed by a major vehicle manufacturer. As in any technique based on a state-space search, the practical applicability and scalability is typically limited due to the state space explosion problem. For this reason, the main goal is to validate that the extensive set of relations that must be exercised does not cause a state space explosion problem. In this sense, only Goal 1 of the verification technique is necessary to pursue in the case study.

The functionality of the Fuel level system is to estimate the fuel level in the vehicle tanks and present this level on the display located in the dashboard. Additionally, the fuel level system must warn the driver if the fuel level is below a predefined level. The functionality is deployed on two ECUs (Electronic Control Unit), one is responsible for estimating the fuel level (called the *Estimator*) and the other

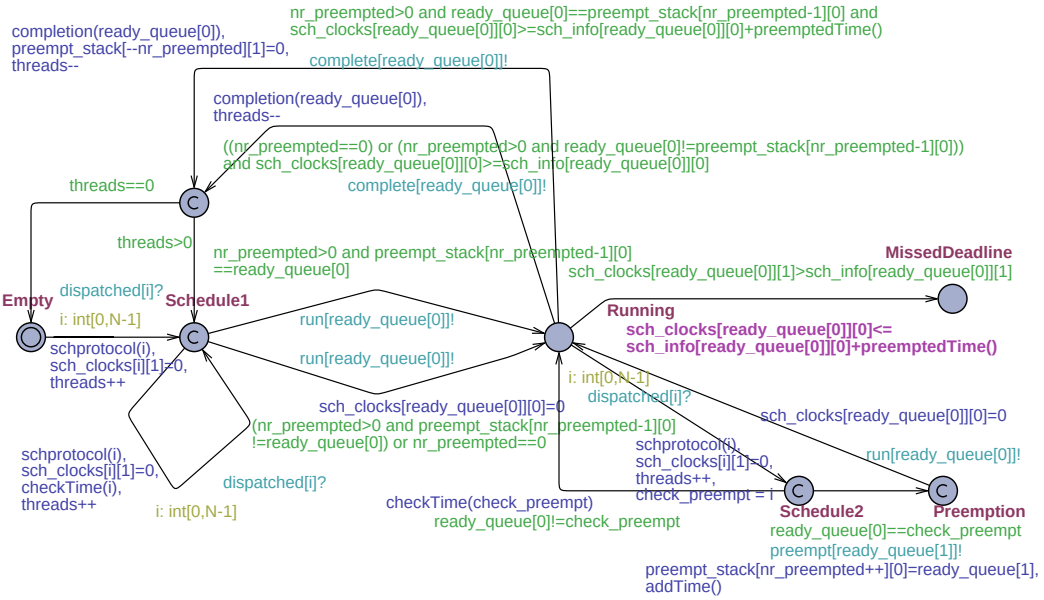

Figure 4. The scheduler automaton.

one is responsible for displaying it (called the *Presenter*). The ECUs are interconnected by a CAN (Controller Area Network) bus. The system has one sensor located in the fuel tank to sense the fuel volume, and two actuators, a fuel level display and a low fuel level lamp, to display the corresponding information to the driver.

As an illustration of the complexity of the system and the verification process we informally describe the control-flow and data-flow related to the *Estimator* ECU, which (graphical) AADL model is shown in Figure 5. The fuel level estimation begins with the fuel level sensor which outputs a voltage signal to the *Estimator* ECU. The received signal is initially processed by an A/D converter, and then transformed into the corresponding fuel volume in percentage (not shown in the Figure). The *Estimator* has a software layer called *Basic Software (B-Software In, B-Software Out)* modeled as a process subcomponent "*BasicSoftware*". The *SoftwareIN* receives the converted value and stores it to the *RTDB* (Real Time Data Base). Subsequently, the fuel level estimation function (FuelEstimation) reads the converted fuel level value from the *RTDB*, and uses it to estimate the fuel level (the complexity of the estimation steps is not shown in the example). The result is written to the *RTDB*. The stored result will be read by 1) *SoftwareOUT* which forwards the result to the second ECU (*Presenter*) via a CAN (Controller Area Network) bus to present the fuel level on the vehicle dashboard and 2) the fuel level warning function (*FuelLevelWarning*) which evaluates if the fuel level is under a predefined value or not, and writes the result to the *RTDB*. The result of the evaluation is read by the *SoftwareOUT* which forwards it to the *presenter* via the CAN bus, to activate the warning lamp if the

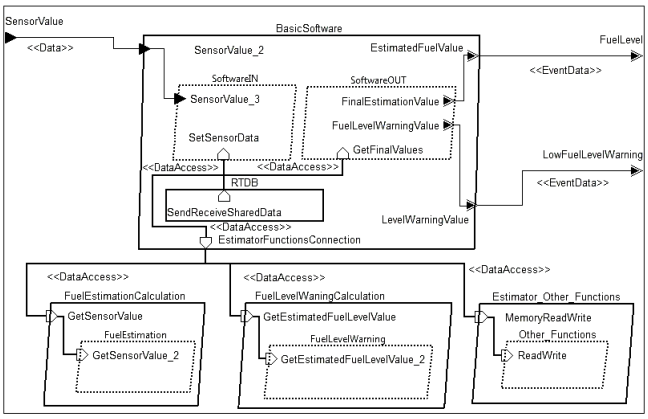

Figure 5. Graphical AADL representation of the Estimator system.

fuel level is too low. The Estimator has numerous other tasks, but they are not involved in the fuel level estimation system and therefor abstracted as a single task in the AADL specification. Each task in the fuel level estimation system has specific properties regarding execution time, deadline and priorities.

The AADL specification was transformed according to the rules in Section III-A to a timed automata model, whereupon the UPPAAL model checker was used to verify completeness and consistency. Observer automata [7] and auxiliary variables were used to exercise the specific verification sequences and Time Computation Tree Logic (TCTL) queries were used to draw the conclusions. The verification process showed no sign of excessive use of memory or time. As a benchmark, we present the time consumption and memory usage for deadlock analysis in a breath-first

search order. The state exploration was performed on a PC with a Intel(R) Core(TM) i7-2670QM (2.2 GHz, 6MB L3 Cache) processor and a Windows 7 64-bit operating system. The time consumption resulted in 2.4s whereas the memory usage resulted in 2.2MB.

V. Related Work

Several studies have proposed a formal semantics through transformations to constructs in a state-based formalism. A mapping of AADL behavioral semantics into Petri Nets is presented in [8], where the objective is to verify that the system is free from deadlocks and that data communication among threads are correct. The methodology does not preserve any timing constraints of the AADL model due to the lack of timing expressiveness of the target language. Yang et al. in [9] define formal semantics of a synchronous subset of AADL in both TASM [10] and Timed Transition Systems (TTS). A definition of formal semantics in TTS enables proofs of semantic-preservation to be generated during the model transformation from AADL to TASM. The notion of scheduling protocols is not mentioned, and the semantics is restricted to non-preemptive scheduling. In [11], a translation to the BIP (Behavior Interaction Priority) language is defined in a natural language and enables simulation and formal verification. The definition is on a high abstraction level and lacks a precise semantics. In [12], a fixed-priority scheduling semantics is transformed into TTS through an intermediate Fiacre model. The paper does not present any translation rules, or a translation algorithm, and solely non-preemptive scheduling is considered. A single study [13] uses UPPAAL as the formal underpinning for schedulability verification. However, the translation does not consider preemptive scheduling or data communication, thus it cannot be used for our purpose.

VI. Conclusion

Achieving high dependability of real-time embedded systems requires effective and efficient fault avoidance techniques. In this paper, we present an overview of a verification technique providing means for automated fault avoidance when developing systems designed in AADL. Automation of the verification technique is achieved through a formal and implemented semantics of a subset of AADL through semantic anchoring. The formal and implemented semantics, in addition, enables simulation of AADL specifications. Timed automata constructs in UPPAAL are used as the formal underpinning, which semantics is not only formally specified but also implemented. The technique was validated against a safety-critical fuel level estimation system where the case study showed that the verification criteria could be met without any excessive use of memory or time.

References

[1] As-2 Embedded Computing Systems Committee SAE, "Architecture Analysis & Design Language (AADL)," SAE Standards n^o AS5506, November 2004.

[2] A. Johnsen, P. Pettersson, and K. Lundqvist, "An Architecture-Based Verification Technique for AADL Specifications," in *Software Architecture*, ser. Lecture Notes in Computer Science, I. Crnkovic, V. Gruhn, and M. Book, Eds. Springer Berlin / Heidelberg, 2011, vol. 6903, pp. 105–113.

[3] G. Behrmann, R. David, and K. G. Larsen, "A tutorial on UPPAAL." Springer, 2004, pp. 200–236.

[4] 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.

[5] A. Johnsen, "Fixed-Priority Preemptive Scheduling Semantics of AADL in UPPAAL Timed Automata," Mälardalen University, Tech. Rep., July 2012.

[6] E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi, "Schedulability analysis of fixed-priority systems using timed automata," *Theor. Comput. Sci.*, vol. 354, pp. 301–317, March 2006.

[7] J. Blom, A. Hessel, B. Jonsson, and P. Pettersson, "Specifying and Generating Test Cases Using Observer Automata," in *Proc. 4 th International Workshop on Formal Approaches to Testing of Software 2004 (FATES04), volume 3395 of Lecture Notes in Computer Science*. SpringerVerlag, 2005, pp. 125–139.

[8] X. Renault, F. Kordon, and J. Hugues, "From AADL Architectural Models to Petri Nets: Checking Model Viability," in *Proceedings of the 2009 IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, ser. ISORC '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 313–320.

[9] Zhibin Yang and Kai Hu and Jean-Paul Bodeveix and Lei Pi and Dianfu Ma and Jean-Pierre Talpin, "Two Formal Semantics of a Subset of the AADL," *Engineering of Complex Computer Systems, IEEE International Conference on*, vol. 0, pp. 344–349, 2011.

[10] M. Ouimet, "TASM Language Reference Manual," Massachusetts Institute of Technology, USA, Tech. Rep., 2006.

[11] M. Y. Chkouri, A. Robert, M. Bozga, and J. Sifakis, "Models in software engineering," M. R. Chaudron, Ed. Berlin, Heidelberg: Springer-Verlag, 2009, ch. Translating AADL into BIP - Application to the Verification of Real-Time Systems, pp. 5–19.

[12] 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*, ser. Ada-Europe '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 207–221.

[13] Q. Liu, S.-l. Gui, and L. Luo, "Schedulability verification of AADL model based on UPPAAL," *Computer Applications*, vol. 29, no. 7, pp. 1820–1824, 2009.