July 2008

Analysis of AADL Models Using Real-Time Calculus with Applications to Wireless Architectures

Oleg Sokolsky
University of Pennsylvania, sokolsky@cis.upenn.edu

Alexander Chernoguzov
Honeywell Inc.

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-08-25.

Analysis of AADL Models Using Real-Time Calculus with Applications to Wireless Architectures

Abstract
Architecture Analysis and Design Language (AADL) captures both platform and software architectures of embedded systems in a component oriented fashion. Properties embedded in an AADL model enable several high-level analysis techniques. In this work, we explore how to perform analysis of end-to-end timing characteristics of an AADL model using Real-Time Calculus (RTC). We identify properties of AADL models that are necessary to enable such analysis and develop an algorithm to transform an AADL model into an RTC model.

Comments
University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-08-25.

Analysis of AADL Models Using Real-Time Calculus with Applications to Wireless Architectures*

Oleg Sokolsky
University of Pennsylvania

Alexander Chernoguzov
Honeywell

Abstract

Architecture Analysis and Design Language (AADL) captures both platform and software architectures of embedded systems in a component-oriented fashion. Properties embedded in an AADL model enable several high-level analysis techniques. In this work, we explore how to perform analysis of end-to-end timing characteristics of an AADL model using Real-Time Calculus (RTC). We identify properties of AADL models that are necessary to enable such analysis and develop an algorithm to transform an AADL model into an RTC model.

1 Introduction

Architecture Analysis and Design Language (AADL) [2, 6] is a modeling framework for embedded systems. It captures both platform and software architectures of an embedded system in a component-oriented fashion.

Because systems are specified at a high level, without much behavioral detail, AADL models can be developed relatively early in the development cycle and used for the evaluation of design alternatives. Therefore, there is a great need for analysis techniques that can be applied to AADL models in order to establish global properties of the models, such as schedulability, reliability, latency of data flows through the system, etc.

Many such analysis techniques are available, and analysis models can be extracted from AADL models. For example, fault tree models [3] can be

*Research is supported in part by a grant from Honeywell Inc.

extracted from AADL models equipped with error modeling information. Rate monotonic analysis can be applied to AADL models with periodic tasks in a straightforward ways, and in [7] we presented an approach for the schedulability analysis of more complicated AADL models.

Real-Time Calculus in another high-level analysis technique that allows to compute quantitative estimates of end-to-end timing of stream-processing hard real-time systems. In this work, we show that it is possible to extract a Real-Time Calculus model from an AADL model and perform the analysis of relatively large models.

2 Background

2.1 AADL Overview

Components. The main modeling notion of AADL is a *component*. Components can represent a software application or an execution platform. A component can have a set of externally accessible *features* and an internal implementation that can be changed transparently to the rest of the model as long as the features of the component do not change. Implementation of a component can include interconnected subcomponents. The features of a component include data and event ports and port groups, subroutine call entries, required and provided resources. Data ports represent sampled communication and are unbuffered. Event and event data ports represent message passing. Each input event or event data port has a FIFO buffer associated with it. Interacting components can have their features linked by event, data, and access connections. In addition, application components can be bound to execution platform components to yield a complete system model. Properties, specific to a component type, can be assigned values that describe the system design and can be used to analyze the model. We will discuss properties relevant to the RTC transformation in Section 3. Component types are illustrated in Figure 1. Different component types are shown as different shapes, according to the standard. Solid lines represent connections, while double lines represent bindings.

Execution platform components include *processors*, *buses*, *memory blocks*, and *devices*. Properties of these components describe the execution platform. Processors are abstractions of hardware and the operating system. Properties of processors specify, for example, processing speed and the scheduling policy. Buses can represent physical interconnections or protocol layers. Their properties identify throughput and latency of data transfers, data formats, etc.

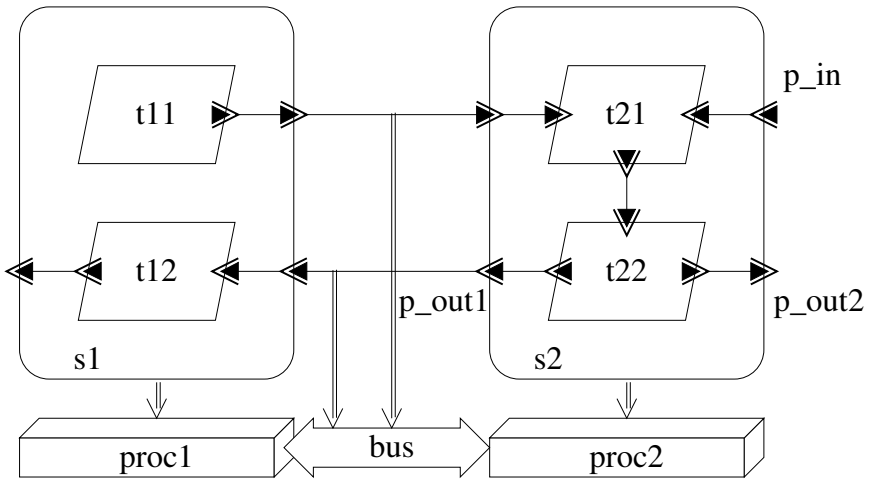


Figure 1: Simple AADL model

Application components include threads and systems. *Threads* are units of execution. Each thread has an associated *semantic automaton* that describes thread states and conditions on transitions between thread states. A thread can be halted, inactive, or active. An active thread can be waiting for a dispatch, computing, or blocked on resource access; etc. A thread can also be recovering from a fault or in the state of non-recoverable error. Properties of the thread specify computation requirements and deadlines in active states of the thread, dispatch policy, etc. Threads are classified into periodic, aperiodic, sporadic, and background threads. They differ in their dispatch policies and their response to external events. A *system* component is a unit of composition. It can contain application components along with platform components, and specifies bindings between them. Systems can be hierarchically organized.

Figure 1 shows a simple AADL model that we will use throughout the paper to illustrate features of the language, the transformation into RTC, and RTC analysis. The system component contains two processors connected by a bus, and two software subsystems. Each of the subsystems is bound to a separate processor. Subsystem *s1* contains one periodic and one aperiodic thread and subsystem *s2* contains two aperiodic threads. The system has one input event data port and one output event data port. Threads communicate via event data ports. Note how features of a component – in this case, in and out event data ports – are mapped by connections to features of its subcomponents.

Connections. AADL connections can connect ports of two components within the same system, or of a system and one of its subcomponents. A sequence of connections, connected by ports at their end points, forms a *semantic connection*. Each semantic connection has an *ultimate source* and *ultimate destination*. Ultimate sources and destinations can be thread or device components. Starting from an ultimate source, a semantic connection follows connections up the component containment hierarchy via the outgoing ports of enclosing components, includes one "sibling" connection between two components, and then follows connection down the component hierarchy until it reaches the ultimate destination. One of the semantic connections in Figure 1 is between threads *t11* in the system *s1* and *t21* in the system `s2`. This connection contains three syntactic connections and is bound to the bus component. When a sporadic or aperiodic thread is the ultimate source of an event connection, it is dispatched by the arrival of an event via that connection. By contrast, periodic threads are dispatched by a timer and use only data parts of events in the queues of their event ports.

Similarly, semantic access connections describe resources required by a thread that is the ultimate source of an access connection. A resource that serves as the ultimate destination of an access connection is typically a data component. Properties of access connections specify concurrency control protocols for shared resources.

Modes. AADL can represent multi-modal systems, in which active components and connections between them can change during an execution. Mode changes occur in response to events, which can be raised by the environment of the system or internally by one of the system components. For example, a failure in one of the components can cause a switch to a recovery mode, in which the failed component is inactive and its connections are re-routed to other components. The AADL standard prescribes the rules for activation and deactivation of components during a mode switch. The multimodal nature of AADL models, along with the rich semantics for connections between components makes it difficult to apply standard schedulability analysis algorithms that tend to target restricted task models and communication patterns.

In this work, we do not consider multimodal systems. Analysis described here can be applied to each global system mode separately. The OSATE toolset supports this way of analysis by offering a separate single-mode AADL model for each global mode of a multimodal AADL model. Direct support of multiple modes may be achieved by extending RTC model with event sequence automata considered in [9]. Event sequence automata will represent changes to the AADL model during a mode switch. This

approach is left for future work, however.

Language annexes. The mechanism of annexes allows users to extend the core language with additional features. For example, an error modeling annex defines additional properties that describe reliability of the system components and a state machine that specifies error states of the system. The use of this annex enables reliability analysis of an AADL model. A behavioral annex allows us to extend thread components with a state machine that specifies computation performed by a thread in more detail. Such a behavioral description refines the default behavior of a thread and enables more precise analysis of the timing behavior of the system.

2.2 Real-Time Calculus

RTC [8, 1] is a formalism that is based on the network calculus [5]. Our presentation of RTC closely follows that of [9], which give a much more detailed exposition of the approach. RTC is used for system-level performance analysis of stream-processing systems with hard real-time constraints. Modular performance analysis based on RTC [11] represents the embedded system as a collection of abstract processing components, which process incoming events and require a certain amount of resource in order to perform this processing. Such a component can represent either computation or communication in the system. When representing computation, an abstract component can represent, for example, a real-time task. The task is dispatched for execution when an event arrives and requires some amount of time - typically represented as best-case and worst-case execution times - in order to complete the execution. When representing communication, incoming events are messages to be transmitted, and the resource required for processing is the communication link on which the transmission occurs, described by the transmission time. In either case, it is assumed that incoming events are queued as they arrive. Once processing of an event is finished, a new event is generated and sent to other components. This event represents result of the task computation or delivery of the message on the communication link. Availability of resources to perform the processing is affected by other components sharing the resource.

An abstract component, then, has two types of inputs and outputs: event streams and resource supplies. Characteristics of an event stream are represented as a function $e : R^+ \rightarrow N \times N$, where R^+ is the set of non-negative reals and N is the set of non-negative integers. The function $e(\Delta)$ gives upper and lower bounds on the number of events in any interval of time of duration Δ. Similarly, resource supply is represented as a function $r : R^+ \rightarrow R^+ \times R^+$,

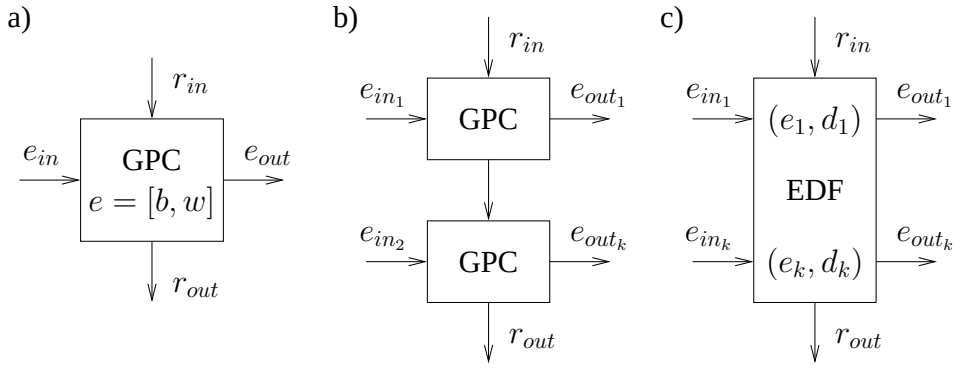


Figure 2: Abstract processing components

giving a lower and an upper bounds on the amount of resource available to the component in any interval of duration Δ. We refer to functions e and r as event arrival and resource curves, respectively. Note that each function contains both the lower and upper bound curves. When we need to refer to one of the two bounds or an event arrival or resource curve, we add the superscript to indicate this: e^l or r^l for lower bound curves and e^u, r^u for upper bound curves.

A commonly used way to specify event streams that are input to the whole system is the (p, j, d) model. (p, j, d)-curves are roughly periodic but at the same time are subject to significant bursts in the short term. Here, p is the period of the stream; j is jitter, which characterizes burstiness of the stream, and d is minimal separation between two consecutive events. For a (p, j, d)-curve e, we have

$$\begin{aligned} e^l(\Delta) &= \left\lfloor \frac{\Delta - j}{p} \right\rfloor \\ e^u(\Delta) &= \min\left(\left\lceil \frac{\Delta - j}{p} \right\rceil, \left\lceil \frac{\Delta}{d} \right\rceil \right) \end{aligned}$$

Each abstract component transforms input event arrival and resource curves into respective output curves. A simple example of a component is the generalized processing component (GPC), shown in Figure 2,a. The component represents, for example, a single preemptible task scheduled under a fixed-priority scheduler. The task is characterized by the execution time e, a tuple of real numbers representing best-case and worst case execution times. Resources that are unused by a task are available to lower-priority tasks, therefore the output resource curve of a GPC becomes the input resource

curve of the component representing the task at the next lower priority level. If all tasks scheduled on the same processor by a fixed priority scheduler have distinct priorities, components representing them can be chained together via their resource curves in the order of decreasing priorities, as shown in Figure 2,b. In more complicated cases, a component represents multiple tasks. For example, if earliest deadline first (EDF) scheduling is used, all tasks have to be analyzed together, represented by a component that has one input and one output resource curves, but multiple pairs of input and output event arrival curves corresponding to different tasks, as illustrated in Figure 2,c. In addition to the execution time, each task specifies a deadline. A similar component is defined for FIFO scheduling - for example, covering the case of fixed-priority threads with equal priorities. The deadline is not specified in this case.

There are several auxiliary components that operate on streams. We use three kinds of auxiliary component in our modeling. The first kind, which we visually denote in diagrams as $\oplus$, lets you merge two streams. Given two event arrival curve functions e_1 and e_2, $e_1 \oplus e_2(t) = e_1(t) + e_2(t)$, and similarly for resource curves. The second auxiliary operator allows us to split streams into multiple substreams. Given a event arrival curve function e, we use an operator $\oslash_p$ defined as $\oslash_p e(t) = \lceil p \cdot e(t) \rceil$. Finally, RTC includes a *greedy shaper* component (GSC). A GSC component ensures that its output event stream is bounded from above by a curve e given to it as a parameter. It achieves this by delaying events in the incoming stream. Thus, a GCS component allows us to limit the amount of traffic at the expense of increasing processing delay and buffer requirements. GSC components will be useful for modeling sporadic threads as we discuss below.

Real-time calculus allows us to calculate two important performance measures for a component. One is the maximum delay d_{max}, an upper bound on the latency of processing an event. The other is the maximum buffer space b_{max}, an upper bound on the size of a buffer necessary to avoid losing incoming events. In the case of the GPC, these values are computed as follows [5]:

$$\begin{aligned} d_{max} &\leq \sup_{\lambda \geq 0} \Big\{ \inf\{\tau \geq 0 : e^u(\lambda) \leq r^l(\lambda_\tau)\} \Big\} \\ b_{max} &\leq \sup_{\lambda \geq 0} \{e^u(\lambda) - r^l(\lambda)\} \end{aligned}$$

Tool support for the RTC analysis. Modeling and analysis described above are supported by the RTC toolbox for Matlab [10], implemented by

Lothar Thiele and his collaborators at the Swiss Federal Institute of Technology (ETH) in Zürich. The toolbox provides Matlab functions to create event arrival and service curves, such as `rtcpjd` for a PJD arrival curve, as well as functions that implement abstract components, such as `rtcgpc` for the GPC component. The toolbox is freely available and can be downloaded from the project web site, along with extensive tutorial for its use.

3 Translating AADL to RTC

3.1 Properties Used in the Translation

The AADL model should contain enough information to extract parameters necessary to populate the RTC model. These parameters, primarily, describe duration of individual processing or communication steps as well as input event arrival and resource curves.

Processor components. Every processor that has thread components bound to it should have the `Scheduling_Protocol` property set. Supported scheduling protocols are RMS (rate-monotonic scheduling), EDF (earliest deadline first), and FPS (fixed priority scheduling with explicitly assigned priorities).

Thread components. Every thread component should specify the property `Dispatch_Protocol`. Allowed values of this property are `periodic`, `aperiodic`, and `sporadic`. If the dispatch protocol is periodic of sporadic, the property `Period` needs to be specified. Thread execution time needs to be specified using the property `Compute_Execution_Time`. The property specifies an interval, $[b, w]$, $b \leq w$, where b is the best-case execution time and w is the worst-case execution time.

If the thread is bound to a processor with the FPS scheduling policy, the thread should have the `Priority` property specified. If the thread is bound to a processor with the RMS scheduling policy, the thread should be periodic or sporadic. In this case, threads mapped to the processor have priorities implicitly assigned according to the RMS policy; that is, inversely proportional to the period of the thread. If the tread is bound to a processor with the EDF scheduling policy, it should have the `Deadline` property specified. If the thread is periodic or sporadic, the deadline is, by default, equal to the value given by the `Period` property. A thread cannot have the `Priority` property specified in this case.

Bus components. The time to transmit a message across a bus depends on the bandwidth and propagation delay of the bus, given by the `Bandwidth` and `Propagation_Delay` properties. In addition, the size of the message needs to be obtained from the data type of the connection that is bound to the bus. The data type in AADL is specified by the data component type, which offers the property `Source_Data_Size`.

Ports. Input event and event data ports of the system are the points where flows of messages enter the system. Their properties are used to construct event arrival curves in the RTC model. We use two properties of a data port to capture parameters of the arrival curve. Property `Input_Rate` specifies a range $[s, p]$ of time values. We interpret p as the long-term period of the stream and s as the minimum separation between two events in the stream. If the jitter in the stream needs to be specified, the property `Input_Jitter` is used. By default, the value of jitter is 0.

If a thread has multiple outgoing ports event, by default an event is produced on every output port at the end of every invocation of the thread. This can lead to an overly pessimistic message traffic. More precise information can be specified using `Output_Rate` property. In its simplest form, the property can specify the number between 0 and 1, representing the probability of an event on the port being produced after a thread invocation, assuming a uniform distribution of events. More complex probability distributions can be specified; however, we do not consider them in this work.

Standard vs. non-standard properties. The currently published version 1 of the AADL standard [6] includes a number of component properties and also provides means for defining new properties that can support used-defined modeling and analysis techniques. Most properties mentioned above belong to the standard property set. Exceptions are the properties `Input_Rate`, `Output_Rate`, `Output_Jitter`, `Priority`, and `Bandwidth`. All of these properties will be included into the standard property set in the upcoming version 2 of the standard. In the meantime, to be able to use these properties for RTC-based analysis in the context of AADL version 1, we defined a new property set RTC that includes the above-mentioned properties. The property set also extends the set of allowed values for the `Scheduling_Protocol` property to include FPS.

3.2 Abstract Component Graph Construction

The first stage in the construction of the RTC model is to extract a graph of dependencies between threads and network messages in the AADL model. We then collapse some of the nodes in the graph together to form abstract components.

The graph of dependencies, which we denote $\mathcal{D}$, has its set of nodes partitions into the following sets of nodes: 1) computation nodes n^c; 2) message nodes n_m; 3) event source nodes n^e; and 4) resource nodes n^r. The set n^c contains the node n^c_t for every thread t in the AADL model. The set n^m contains the node n^m_c for every semantic connection c in the AADL model that is bound to a bus. The set n^r contains the node n^r_p (respectively, n^r_b) for each processor p or bus b component in the AADL model. Finally, the set n^e contains the node n^e_{pt} for each input event or event data port pt at the top level of the AADL model and one node n^e_t for each periodic thread t, which represents invocation of the t by the system timer.

Further, $\mathcal{D}$ has the set of directed edges, partitioned into event and resource edges. Event edges represent the flows of events through the system, according to the following rules:

- For each periodic thread t, there is an edge $n^e_t \rightarrow_e n^c_t$;
- For each semantic connection c with the ultimate source t and ultimate destination t', there is an edge $n^c_t \rightarrow_e n^c_{t'}$ is c is not bound to a bus and two edges $n^c_t \rightarrow_e n^m_c$ and $n^m_c \rightarrow_e n^c_{t'}$, otherwise;
- Finally, for each port pt and each thread t that can be reached from pt by traversing a chain of entry connections, there is an edge $n^e_{pt} \rightarrow n^c_t$.

Resource edges are added to represent resource supply. Let $t_1, t_2, \ldots, t_k$ be a sequence of threads mapped to a processor p, such that the order of the sequence is consistent with the decreasing order of priorities of the threads. That is, if $i < j$, the priority of t_i is no less than the priority of t_j. If p has the EDF scheduling policy, any order is acceptable. Then, $\mathcal{D}$ contains resource edges $n^r_p \rightarrow_r n^c_{t_1}$, $n^c_{t_1} \rightarrow_r n^c_{t_2}$, $n^c_{t_2} \rightarrow_r$, $\ldots$, $\rightarrow_r n^c_{t_k}$. For a bus component and connections bound to it, a chain of resource nodes is constructed in a similar way.

Once $\mathcal{D}$ is constructed, we transform it into a graph of abstract components by adding several auxiliary nodes as described below and by merging the nodes that have to be processed together. Three kinds of auxiliary nodes are introduced:

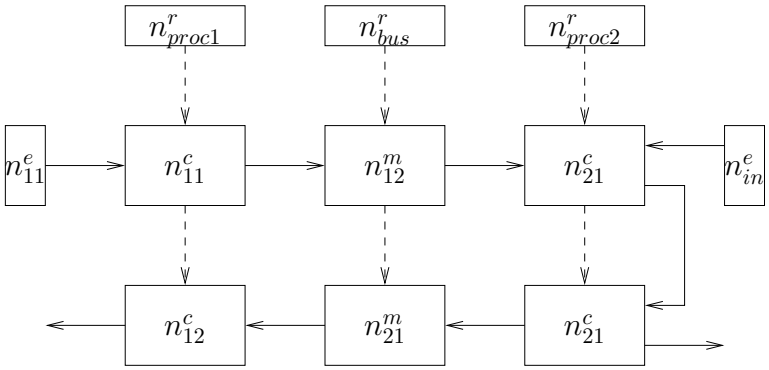


Figure 3: Graph of dependencies between threads and messages

- if a node n has multiple incoming event edges, all of them are redirected to a new merge node n_+ and a new edge $n_+ \rightarrow_e n$ is added;
- if a thread t has an output port with the output rate less than one, a scaling node is added to the respective outgoing edges of the thread node;
- a shaper node is added to the incoming event edge of a node corresponding to a sporadic thread. We discuss this in more detail below.

Let $n^c_{t_i}, n^c_{t_{i+1}}, \ldots, n^c_{t_{i+j}}$ be the nodes corresponding to threads bound to the same bus, which have equal priorities (for EDF, $i = 1$, $j = k - 1$). All of these nodes are merged into a new node and any edge incident to any of these nodes is now incident to the new node. Once the nodes are merged, ones that correspond to EDF policies are turned into the EDF abstract components; fixed-priority nodes with equal priorities are turned into FIFO abstract components; finally, those that were not merged appear as GPC components.

Example. Consider again the example in Figure 1. Assume that *proc1* is using the FPS policy and *proc2* is using EDF. Further, let *t11* have a higher priority than *t12*, and that messages from *s1* have higher priority than messages from *s2*. Finally, assume that the output rate for port *p_out1* is 0.8 and for port *p_out2* it is 0.2. The graph $\mathcal{D}$ for this example is shown in Figure 3. Next, we add auxiliary nodes to the dependency graph. Both messages from the network and externally arriving events cause the dispatch of the thread *t21*, therefore the arrival curves of the two streams are added

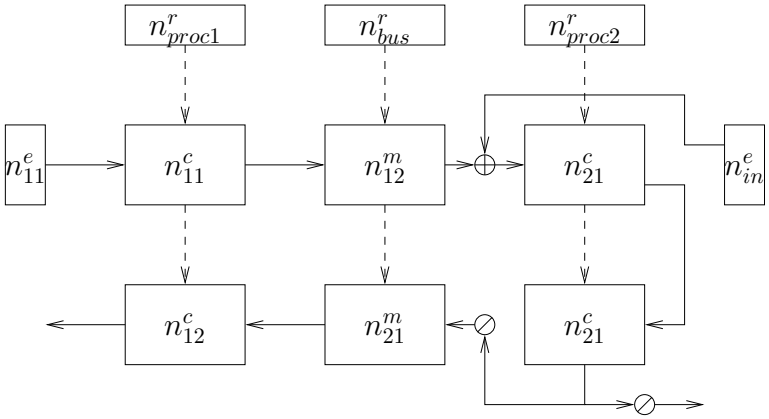


Figure 4: Auxiliary merge and scale nodes in the dependency graph

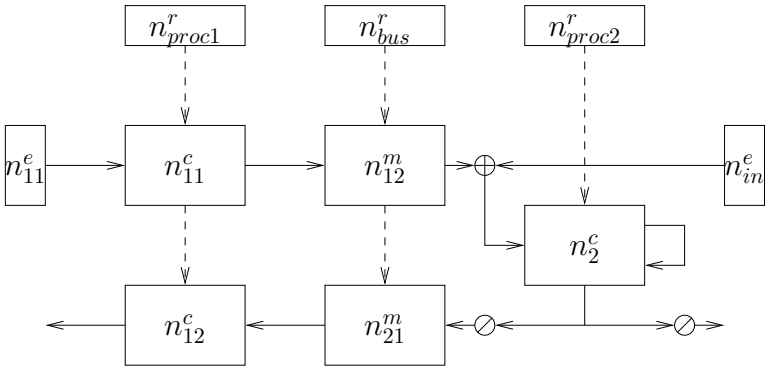


Figure 5: Graph of abstract components

together. Also, the output event stream of thread *t22* is scaled differently according to the `Output_Rate` properties of its output ports. The resulting graph is shown in Figure 4. Finally, since threads on the processor *proc2* are scheduled according to EDF, they need to be put together into the same abstract component. Note that the event edge from *t21* to *t22* becomes a self-loop, which would require us to iterate the analysis in order to compute the fixed point. The graph of abstract components is shown in Figure 5.

Sporadic threads. Sporadic threads have the `Period` property that specified the minimum separation between incoming events that cause the dispatch of the tread. The AADL standard specifies that if events arrive more often, they are queued until minimum separation is achieved. This is exactly the behavior that GSC components offer. If the value of the thread's period is p, we create a PJD curve $(p, 0, 0)^u$ – that is, the worst-case curve for a perfectly periodic arrival of events. This curve is used as the parameter of the GSC component. By placing this GSC component in before the GPC component representing the thread, we achieve the additional buffering that differentiates sporadic from aperiodic threads.

4 Case Study

In order to evaluate the proposed transformation and scalability of the analysis, we conducted a case study of a wireless sensor network architecture based on the application level ISA100 proposal [4].

4.1 Overview

The case study represents a collection of sensor nodes connected by a multihop backbone to a gateway, which is in turn connected to a wired network that includes operator nodes, alarm handlers, history loggers, etc. The architecture of the system is informally represented in Figurefig:overview. We do not model the wireless network explicitly; however, it affects the wireless subsystem in two ways. On the one hand, the wired network provides a load to the wireless subsystem, which comes in the form of a stream of operator requests. These requests are passed by the gateway to the wireless subsystem. On the other hand, other kinds of load are assumed to directly affect only the gateway. These loads can have widely varying characteristics, from firmware downloads, which are infrequent transmissions of large size on the one end of the spectrum; to frequent bursts of short requests that

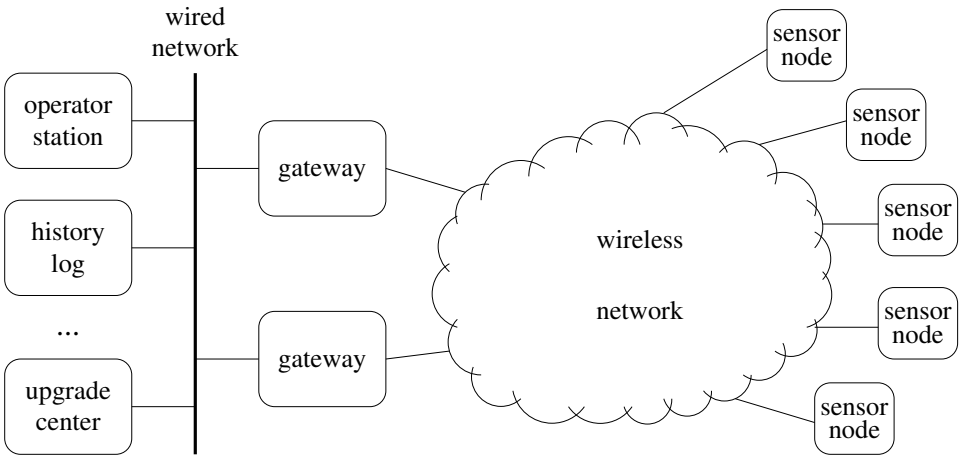


Figure 6: System architecture for the case study

are handled by the gateway - for example, ARP broadcast messages - on the other end of the spectrum. We refer to the latter kind of load on the wired network as network noise. Although these additional loads from the wired network are handled by the gateway, they can affect the wireless subsystem when it comes to handling flows of messages from sensor nodes. Messages from sensor nodes need to be transmitted across the wired network. If the wired network is busy, these messages need to be stored in the gateway, delaying their processing and increasing the buffer space requirements in the gateway.

Data communication with sensor nodes. Sensor nodes support three modes of communication. First, sensor data are periodically published using the TDMA mechanism. Second, sensor nodes can transmit their status information (referred to as parameter values) in response to requests from operators. This communication proceeds in the "client-server" mode using a CSMA protocol, which does not guarantee the absence of collisions. Finally, sensor nodes can spontaneously report alarms that indicate abnormal conditions. Alarm handling is described in detail below. Alarm messages compete with client-server messages for access to the medium.

In order to minimize the number client-server messages traversing the wireless network, the gateway uses a cache. When a request for a particular parameter value arrives, it is checked against the cache and, if found, the value is returned immediately. Otherwise, the request is forwarded to a sensor node across the wireless network. The received response is stored in

the cache and then returned to the operator node that issued the request.

Alarm handling. The gateway receives alarm messages from sensor nodes and forwards them to alarm handlers across the wired network. In order to cope with bursts of alarms, incoming alarm messages are stored in a FIFO queue. Each alarm message is acknowledged upon being queued to the node that raised the alarm. The stream of alarm acknowledgement messages adds to the CSMA traffic on the wireless network. If the alarm queue becomes full, further incoming alarms are dropped without being acknowledged. The alarming sensor node, in that case, eventually times out and retransmits the alarm.

4.2 Architecture Modeling in AADL

Figure 7 shows the overall architecture of the system with one gateway and one sensor node. We model the TDMA and CSMA parts of the wireless medium as two separate networks. Note that we do not model the nodes on the wired network that serve the sources and destinations of message flows through the system. Instead, we model an open system, where sources and sinks of message streams are represented as input and output ports. This modeling device allows us to represent parameters of input streams as properties of the ports and easily vary them in the architecture evaluation. The port labeled *fault* is another modeling device that allows us to represent spontaneous raising of alarms by sensor nodes and capture parameters of event streams.

Figure 8 represents the architecture of a gateway. The assumption was that every kind of incoming message is handled by a separate thread. Ports on the left-hand side of the diagram represent communication on the wired network, while ports on the right-hand side represent wireless communication.

The top portion of Figure 8 represents the alarm stream. The *logger* thread receives alarm messages from the wireless network, puts them into the alarm queue, and sends acknowledgements back. The *handler* thread takes alarm messages from the queue and transmits them across the wired network. Note that the alarm queue is not represented explicitly. Instead, we utilize the queue supplied to us by the AADL semantics. Each in event and in event data port of an AADL thread is equipped with a fixed-size FIFO queue. A queueing policy property specifies the behavior in case of the queue overflow. The `Drop_Newest` perfectly matches the behavior of the alarm queue described above.

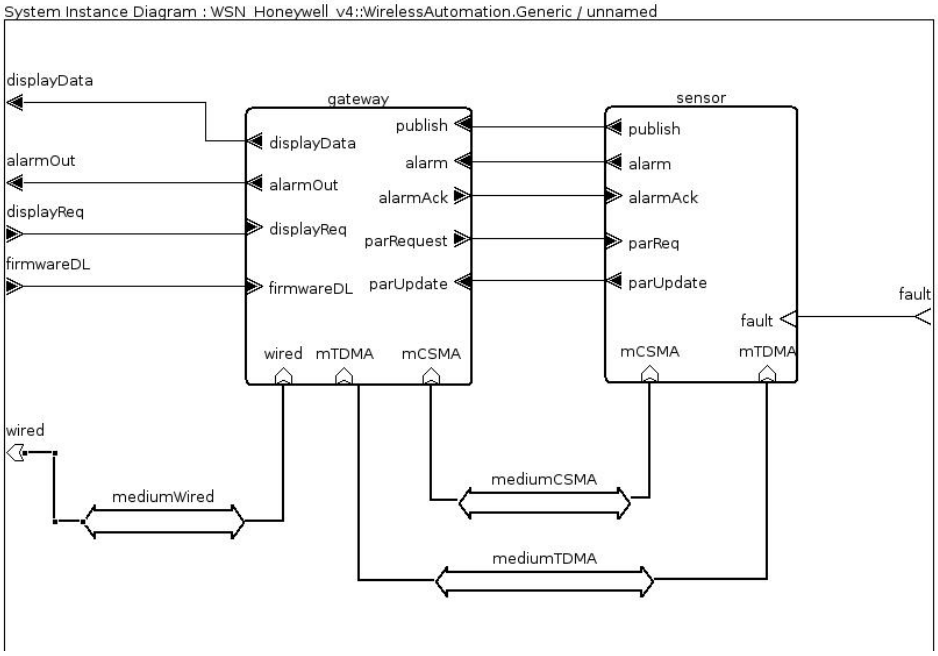


Figure 7: AADL architecture of the case study

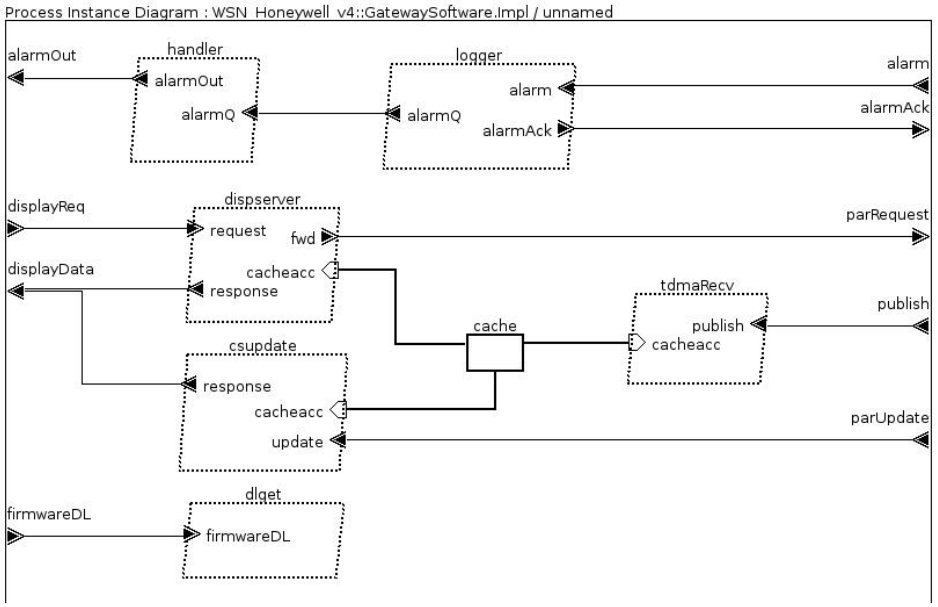


Figure 8: AADL architecture of the gateway

The middle portion of Figure 8 represents the parameter request stream. The *dispserver* threads accepts operator requests from the wired network, consults the cache and either returns the parameter value or forwards the request through the wireless network. The *csupdate* thread receives the response messages from the wireless network, updates the cache, and transmits the updated value to the operator. Note that the sensor values periodically published by sensor nodes are also stored in the cache and served to operators by the mechanism described above.

Finally, the bottom part of the diagram shows the additional load imposed on the gateway by the wired network in the form of firmware updates and network noise as described above. It is represented by the *dlget* thread that serves as the sink for the flow of these messages.

All software on the gateway execute on the same processor under fixed-priority scheduling. The *dlget* thread is given the highest priority to maximize the effect of extraneous loads onto the core of the system. The periodic publish thread has the next highest priority, followed by the alarm logger thread, alarm delivery thread, *dispserver* thread, and, finally, *csupdate* thread. This priority assignment lets the client-server communication suffer the most interference from other aspects of the system. Priority assignments can be easily changes at the AADL level and analysis can be repeated to stress other parts of the system such as alarm handling.

Figure 9 shows the architecture of the sensor node. It contains a periodic thread that publishes sensor data, a fault detection thread that transmits alarms, a thread to service parameter requests, and a thread that collects alarm acknowledgements. Here, we also assume that the processor uses fixed-priority scheduling, with the periodic publish thread having the highest priority, followed by the alarm handling thread, client-server thread, and acknowledgement thread, in that order.

4.3 RTC model for the case study

Figure 10 shows the RTC model of the architecture described above, with one gateway and one sensor node. Note that the client-server messages on the wireless network are processed together in one abstract component. All other resources are assumed to use fixed priorities. The event source node *a_pub* represents the periodic publishing of sensor readings, while the other three event source nodes correspond to the three input ports of the AADL system in Figure 7.

Figure 11 shows the evolution of the RTC model as more sensor nodes are added. The part of the RTC model that describes the wired network

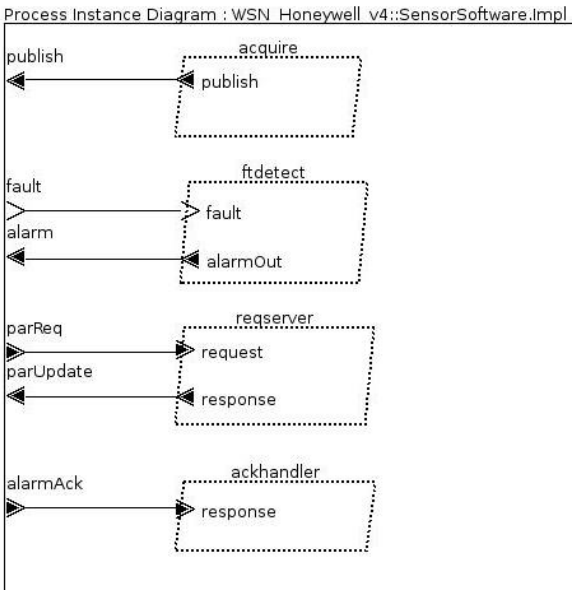


Figure 9: AADL architecture of the sensor node

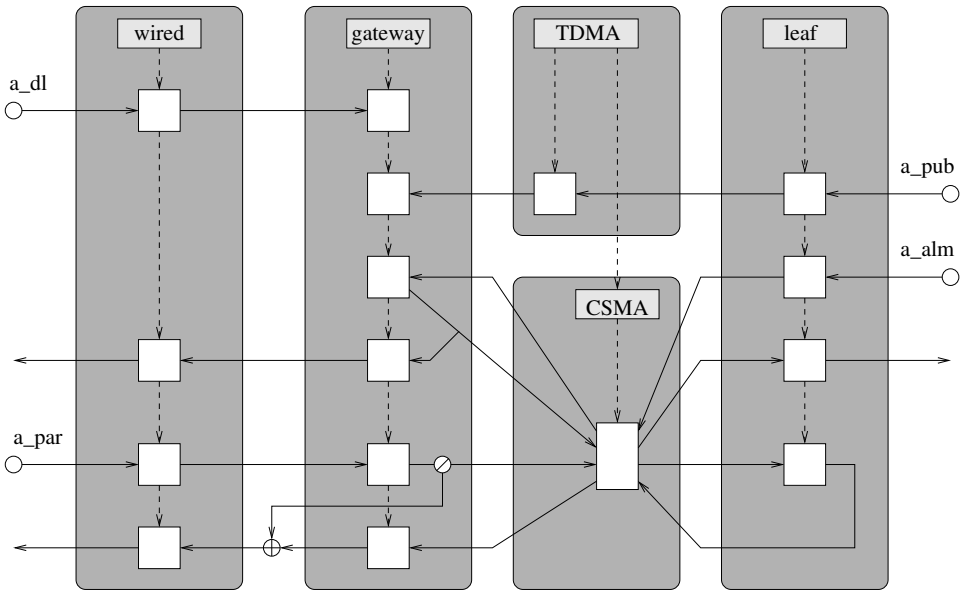


Figure 10: RTC model of the architecture with one sensor node

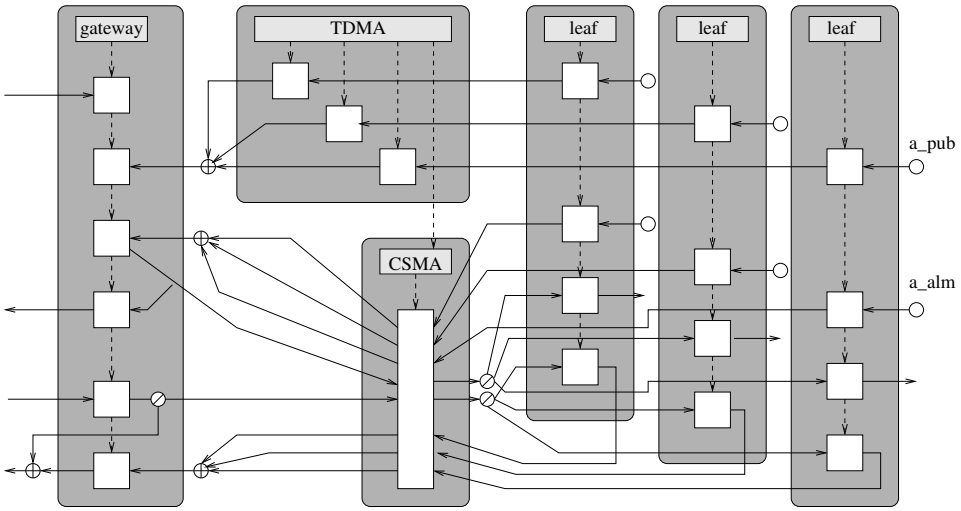


Figure 11: RTC model of the architecture with three sensor nodes

is unchanged compared to Figure 10 and is not shown to avoid cluttering the figure. Event streams from the sensor nodes are merged together before entering the gateway, and event streams from the gateway to the nodes are split and proportionally scaled as they enter the sensor nodes. Each sensor node publishes its readings using a separate TDMA slot, without interference from other nodes. In the RTC model, every TDMA slot is represented as a separate resource. CSMA communication happens in the interval that remains after all TDMA slots have been allocated. This interval is assumed to be contiguous (that is, TDMA slots are allocated next to each other within the service interval. Note that, as more sensor nodes are added, the CSMA interval becomes smaller, affecting performance of client-server communication.

One can notice that the RTC model does not capture the following aspect of the sensor node behavior. If an alarm is not acknowledged by the gateway, the sensor node is supposed to retransmit the alarm. In the initial version of the model, we tried to represent this aspect directly: the acknowledgement traffic was split in some proportion, and that one part travelled as a client server message though the network. The other part was merged, with the retransmission delay, with the alarm traffic from the sensor node. It turns out that this model, which seems more faithful to the real system, has two drawbacks. First, it is not clear, in which proportion should the acknowledgement traffic be split. It becomes another parameter to be pro-

vided by the user, who does not have any principled basis to supply this parameter value. More seriously, the RTC model with such feedback turned out to be hard to analyze: the fixed point computation did not converge in a reasonable number of steps, and the processing time of a step increased dramatically with each next iteration. To avoid this problem, the final version of the model followed a different approach. We assumed that every alarm is acknowledged, so no retransmissions were necessary. We then calculated the buffer requirements for the alarm queue. Once the system satisfies the buffer requirements, no alarms are dropped and the assumption is satisfied.

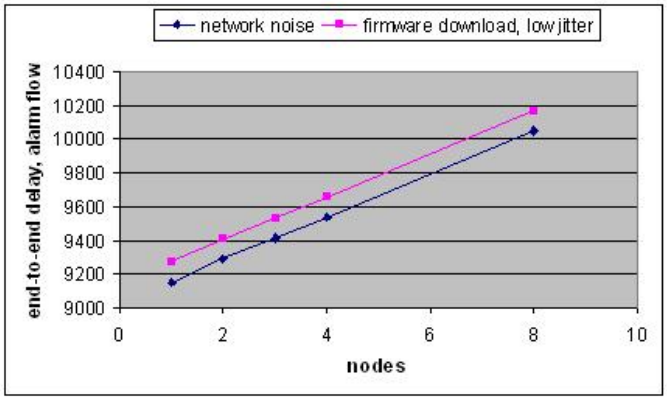
4.4 Analysis results

During the analysis, we considered several configurations of the model. The configurations differed in the parameters of the highest-priority load imposed on the system by the wired network. This turned out to be the most significant factor to affect the running time of the analysis of a single model instance (that is, with the number of nodes fixed). Three configurations were explored. Two configurations describe the "firmware download" kind of wired network load. One had a period of 0.5 hours with very high bursts (jitter equal to 2 hours), and a minimum separation of 1 minute. The other configuration has a reduced jitter parameter value equal to 15 minutes with the other parameters unchanged. The third configuration was the "network noise" load with 1.8 second period, 0.6 seconds minimum separation, and 7.2 seconds jitter.

All experiments described below were performed using Matlab version R2007b and the RTC toolbox Version 1.1 beta 1.03. The platform used was a ThinkPad T61 laptop with a 2.0 GHz dual-core processor and 1GB of main memory, running Ubuntu linux.

End-to-end delay calculations. The first set of results describes end-to-end delays for different event streams in the system. As an example, Figure 12 shows end-to-end delay of the alarm stream, measure from the moment an fault is detected by a sensor node to the moment the alarm is delivered to the destination, an alarm handler node on the wired network. For relatively low network utilization, up to eight sensor nodes, the calculated delay is growing linearly with the number of nodes. However, as Figure 12,b demonstrates, once the network capacity is exceeded, the delay grows up dramatically.

a)



b)

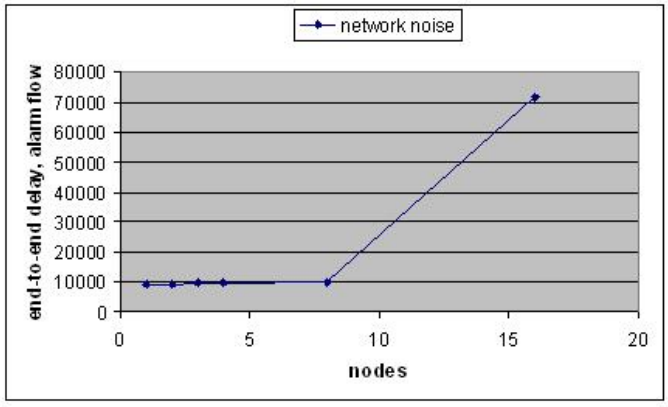


Figure 12: End-to-end delay of the alarm stream

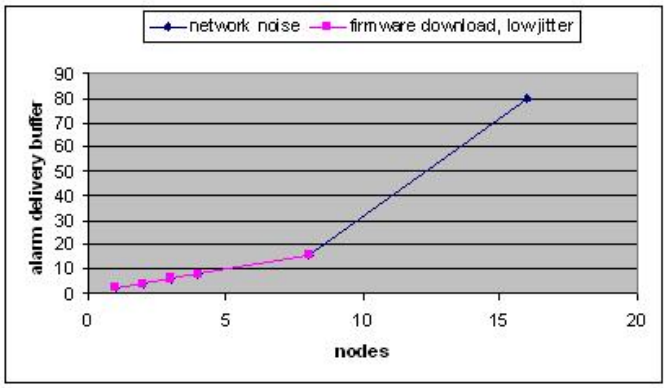


Figure 13: Required alarm queue size

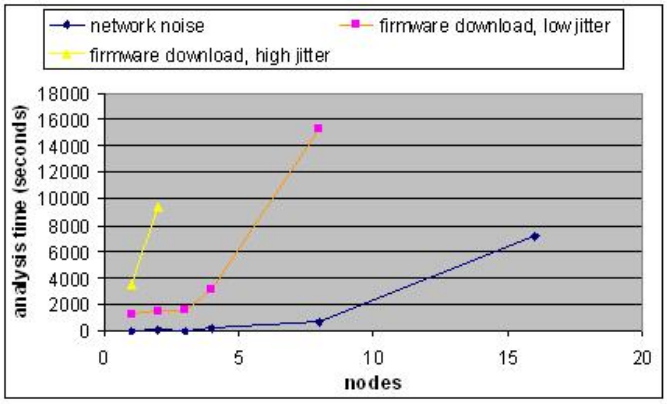


Figure 14: Total running time of experiments

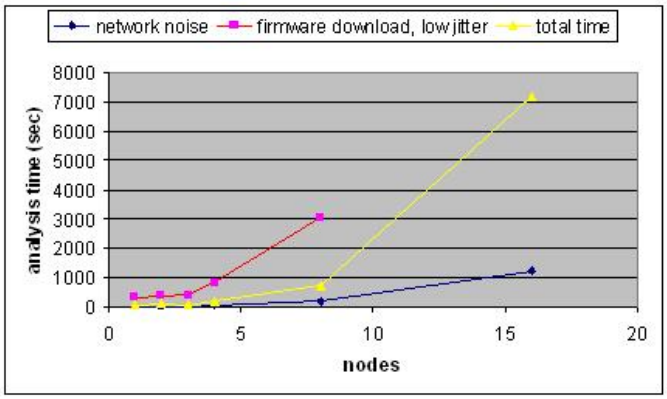


Figure 15: Running time per iteration

Alarm queue. Figure 13 shows the buffer requirements for the alarm queue as a function of the number of nodes. In this case, also, we can see that buffer requirements initially increase linearly with the number of sensor nodes and then, upon reaching a threshold suffer a sharp increase that indicates that the network capacity needs to be enhanced.

Scalability. Figure 14 gives the total running times of the experiments as reported by the RTC toolbox. Clearly, the running time is superlinear with respect to the number of nodes. Note that not only the total running time of each experiment increases with the range of timing constants in different configurations, but also the rate of increase (slope of the curve) depends on

the range as well. Note, however, that larger numbers of nodes required more fixed point iterations to complete the analysis: from four iterations for up to four nodes to six iterations for sixteen nodes. To account for the increased number of iterations, we also calculated analysis time per one iteration, which is shown in Figure 15. For comparison, the total time for the network noise configuration is also shown in Figure 15. Time per iteration can be seen to grow much slower than the total time.

The obvious conclusion from the data is that RTC-based analysis is sensitive to the range of time constants. In all cases, the smallest time constant was on the range of 1 ms. The bursty firmware download configuration was by far the most time-consuming configuration to analyze. It had the jitter parameter value as the largest time constant in the model, and reducing just this value in the second firmware download configuration improved analysis time dramatically. Further reducing timing parameters of that event stream in the network noise configuration improved analysis time further.

5 Conclusions and Future Work

We consider analysis of timing and performance properties of systems expressed in the architecture description language AADL. We presented an algorithm to extract from such an architectural model an analytical model based on Real-Time Calculus, and discussed properties that can be determined using this model. We applied this analysis technique to a case study based on a wireless sensor network architecture. The case study included modeling of a typical architecture and analysis of several variants of the model different number of network nodes and workload parameters and comparative analysis of these configurations.

The case study identified two areas, where this modeling and analysis approach requires improvement before it can be applied to real industrial-scale systems. One deficiency is scalability. Current tools allow analysis of relatively small-scale systems. On the one hand, existing tools can be substantially improved with more efficient implementation and new data structures for event curve representation. On the other hand, research is needed into improved algorithms that would reduce the dependency of running time on the range of timing constants. On the other hand, the proposed technique can suffer from excessive conservatism in analysis results. Precision can be improved by incorporating existing techniques for considering workload variability and event correlations, for example, based on event sequence automata.

References

[1] S. Chakraborty, S. Künzli, and L. Thiele. A general framework for analysing system properties in platform-based embedded system designs. In *IEEE Design Automation and Test in Europe (DATE)*, Mar. 2003.

[2] P. Feiler, B. Lewis, and S. Vestal. The SAE AADL standard: A basis for model-based architecture-driven embedded systems engineering. In *Workshop on Model-Driven Embedded Systems*, May 2003.

[3] D. Hassl. Advanced concepts in fault tree analysis. In *Proceedings of System Safety Symposium*, June 1965.

[4] ISA100 Wireless Working Group. Draft standard ISA100.11a. Internal working draft, May 2008.

[5] J. Y. Le Boudec and P. Thiran. *Network Calculus - A Theory of Deterministic Queuing Systems for the Internet*, volume 2050 of *Lecture Notes in Computer Science*. Springer, 2001.

[6] SAE International. *Architecture Analysis and Design Language (AADL), AS 5506*, Nov. 2004.

[7] O. Sokolsky, I. Lee, and D. Clarke. Schedulability analysis of AADL models. In *Workshop on Parallel and Distributed Real-Time Systems*, Apr. 2006.

[8] L. Thiele, S. Chakraborty, and M. Naedele. Real-time calculus for scheduling hard real-time systems. In *IEEE International Symposium on Circuits and Systems (ISCAS)*, volume 4, pages 101–104, 2000.

[9] E. Wandeler. *Modular Performance Analysis and Interface-Based Design for Embedded Real-Time Systems*. PhD thesis, Swiss Federal Insitute of Technology, 2006.

[10] E. Wandeler and L. Thiele. Real-Time Calculus (RTC) Toolbox. `http://www.mpa.ethz.ch/Rtctoolbox`, 2006.

[11] E. Wandeler, L. Thiele, M. Verhoef, and P. Lieverse. System architecture evaluation using modular performance analysis: a case study. *Software Tools for Technology Transfer*, 8(6):649–667, Nov. 2006.