View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Open Repository and Bibliography - Luxembourg

Towards the Systematic Analysis of Non-Functional Properties in Model-Based Engineering for Real-Time Embedded Systems

Guillaume Brau[a,b], Jérôme Hugues[b], Nicolas Navet[a]

[a] *University of Luxembourg, CSC Research Unit,*
6 rue R. Coudenhove-Kalergi, L-1359 Luxembourg, Luxembourg.
{guillaume.brau, nicolas.navet}@uni.lu
[b] *Université Fédérale Toulouse Midi-Pyrénées, ISAE-SUPAERO,*
10 avenue E. Belin, 31055 Toulouse, France.
jerome.hugues@isae-supaero.fr

Abstract

The real-time scheduling theory provides analytical methods to assess the temporal predictability of embedded systems. Nevertheless, their use is limited in a Model-Based Systems Engineering approach. In fact, the large number of applicability conditions makes the use of real-time scheduling analysis tedious and error-prone. Key issues are left to the engineers: *when to apply a real-time scheduling analysis? What to do with the analysis results?* This article presents an approach to systematize and then automate the analysis of non-functional properties in Model-Based Systems Engineering. First, *preconditions* and *postconditions* define the applicability of an analysis. In addition, *contracts* specify the analysis interfaces, thereby enabling to reason about the analysis process. We present a proof-of-concept implementation of our approach using a combination of constraint languages (REAL for run-time analysis) and specification languages (Alloy for describing interfaces and reasoning about them). This approach is experimented on architectural models written with the Architecture Analysis and Design Language (AADL).

Keywords: Model-Based Systems Engineering; Non-Functional Properties; Analysis; Contracts; Real-Time Scheduling; Architecture Description Languages

1. Introduction

Context. Embedded systems have become an integral part of our daily life. We can found them in cars, aircrafts, trains, robots, healthcare equipments, mobile phones, consumer electronics, etc. In particular, a major issue related to embedded systems is to fulfill the non-functional requirements dictated by their environment, expressed for example in terms of timing, dependability, security, or other performance criteria. In safety-critical applications for instance (e.g. in an airplane), missing a non-functional requirement can have severe consequences, e.g. loss of life, personal injury, equipment damage, environmental disaster, etc.

Model-Based Systems Engineering (MBSE) makes it possible to deal with critical design constraints, especially real-time constraints. An MBSE approach uses domain-specific models as first-class artifacts to design and then develop the system. The non-functional properties of the system are to be analyzed from these models throughout the design process, e.g. by *model checking*, simulation or with other analytical methods; and the analysis results must be taken into account in the design process.

Thus, an important challenge is to provide: (1) tools that implement both modeling languages and analysis engines; (2) engineering support for developers to use models and analyses.

Problem. Modeling and analysis features are usually provided as part of distinct tools: (1) languages such as AADL [1], EAST-ADL combined with AUTOSAR [2, 3], or SysML and MARTE UML profiles [4, 5] provide standardized notations to model a system;(2) analytical frameworks for Verification & Validation activities including real-time scheduling tools [6, 7], model checkers [8, 9], etc. enable to analyze the diverse non-functional properties of embedded systems.

An approach commonly used to connect the toolsets is to translate a model used for design into a model used for analysis, as represented in Figure 1. In that context, one can either implement a comprehensive model transformation (e.g. metamodeling with the MOF standard in the Eclipse Modeling Framework [10, 11], transformation with a dedicated language such as ATL [12]); or more often relies on an *ad hoc* transformation chain to deal with the design and analysis models within the different tools. Yet, we note several limitations with this approach:

- downstream, it does not address the validity of the transformation: is the analysis applicable on the design model which is considered? If not, is it relevant to apply the transformation?
- upstream, the analysis result is not handled: what is the meaning of the analysis result? How to take the analysis result into account in the design process?

Therefore, a more systematic approach is necessary so as to manage analysis activities at design time. This approach must be supported by MBSE tools alongside modeling languages and analysis engines.

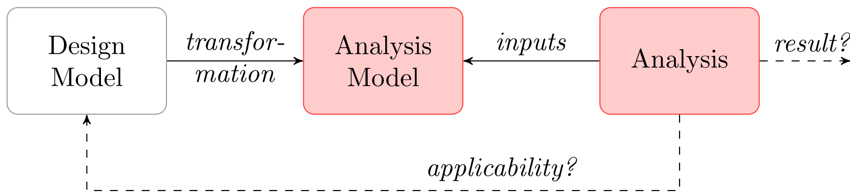

Figure 1: Analysis based on a transformation process. Design and analysis features are part of distinct tools, highlighted in white and red shapes: (1) a model used for design in a first tool is translated into a model used for analysis in a third-party tool; (2) the analysis in the third-party tool is then applied on its own model.

Contribution. In this article, we aim at systematizing and automating the analysis of non-functional properties of embedded systems at design time.

The preconditions are the properties to be true in an input model prior to execute an analysis. The postconditions are the properties guaranteed on the model after the analysis execution. With preconditions and postconditions satisfied, an analysis is complete and sound. Concretely, a full analysis, including preconditions and postconditions, can be implemented by means of constraint languages, e.g. OCL on UML models [13] or REAL with AADL [14].

Then, we use *contracts* to automate the analysis process. A contract completely defines the interfaces of an analysis in terms of processed data and properties. Inputs/Outputs (I/O) describe input and output data. Assumptions/Guarantees (A/G) describe input and output properties. Specific methods can then be used to automatically reason about these interfaces, and answer complex questions about the analysis process such as: which analysis can be applied on a given model? Which are the analyses that meet a given goal? Are there analyses to be combined? Are there interferences between analyses? In practice, contracts can be defined with the help of a specification language such as Alloy [15], and evaluated through associated SAT solvers.

We present and experiment these concepts on an aerospace case study: the Paparazzi Unmanned Aerial Vehicle [16]. The experimental results presented in this paper can be reproduced from our tool prototype and AADL models available online[1].

Work hypotheses. The two following hypotheses fix the limits of our contributions. These hypotheses may be relaxed in future works as discussed at the end of this article:

Design through architectural description languages. We focus on early design phases, especially the architectural design stage supported through Architecture Description Languages [17]. The models mentioned in this paper are written with the Architecture Analysis and Design Language (AADL) [18] and are part of the AADLib project [19], our library of reusable AADL models accessible online.

Real-time properties. We concentrate on real-time properties. We focus on a particular kind of analytical methods called real-time scheduling analyses [20].

Structure of the paper. The paper is organized as follows. Section 2 provides a general overview on architecture descriptions languages and discusses related works. Section 3 deals with the semantics of an analysis. We introduce contracts in section 4. Section 5 discusses potential extensions of our approach. We finally conclude in section 6.

2. Background and related works

Model-Based Systems Engineering is a paradigm that focuses on models in the engineering of complex systems. In a MBSE approach, activities such as specification, design, implementation, integration and verification systematically involve domain-specific models [21].

[1] The tool with the models can be found on the git repository `https://github.com/gbra/maiwen`, december 2017

2.1. Background: modeling and analysis of real-time architectures

In an architecture-centric MBSE approach in particular (for example, see [18]), the design process is based on an architecture description with the help of an Architecture Description Language (ADL). ADLs enable to model and analyze the architecture of a system, *i.e.* the components that comprise a system and their interconnection structure.

Architectural modeling. The Architecture Analysis and Design Language, or AADL for short, is a standardized language to describe the architecture of real-time embedded systems [1, 18]. A model with AADL represents a system architecture made up of a combination of hardware and software components. Components are precisely defined with their non-functional properties (e.g. task execution times, memory footprint) and possible implementations (e.g. tasks with a behavioral specification or the actual source code). Interconnection schemes such as descriptions of data flow or communication protocols are also provided. MARTE (Modeling and Analysis of Real-Time and Embedded Systems) [4] and EAST-ADL (Electronics Architecture and Software Technology - Architecture Description Language) [2] are ADLs providing concepts similar to AADL. MARTE is a UML profile to bring real-time concepts to the designer. EAST-ADL is used for automotive needs and complies with the AUTOSAR architecture [3]. See [22] for a more precise comparison of these languages.

Synchronous models provide an intermediate level of abstraction between programs and the high-level architectural models discussed previously. Unlike high-level ADLs, models with synchronous languages like LUSTRE [23], SIGNAL [24], Esterel [25] or Prelude [26] have a formal execution semantics. The synchronous approach is based on strong mathematical foundations and is suitable for the formal design and verification of reactive systems, e.g. see [27, 28]. Works like [29, 30, 31] show overlaps between high-level ADLs and synchronous ADLs. For instance, the authors in [29] translate a subpart of an AADL model into a LUSTRE program; and evaluate AADL models with tools available for synchronous programs.

Analysis of real-time properties. The real-time properties of a system can be analyzed from an architectural description of the system. For instance, AADL models can be analyzed with a large set of analysis theories and tools[2]: real-time scheduling theory with Cheddar [6] and MAST [7]; real-time process algebra [32]; real-time calculus [33]; network calculus [34]; model checking based on Petri Nets [35, 36] or RT-Maude [37] for behavioral analysis; etc.

The analysis of an architecture model is usually based on a model transformation process that translates the architectural model into a tool-specific model used for analysis: transformations have been implemented to translate AADL models into Cheddar ADL and MAST models with the OCARINA tool suite [38]; transformation chains exist to connect AADL models to UPPAAL [39], TINA [36, 40] or CADP model checkers [41]; etc. A more exhaustive list of analysis tools and transformations applicable to AADL models is available in [42].

[2]An updated list of tools, projects and papers is available at `http://www.aadl.info`.

Work positioning. Architecture Description Languages enable to describe a system architecture, whereas analysis engines (via model transformations) make it possible to analyze the various non-functional properties of a system. However, no support is provided to use these tools and answer the following kind of questions: which analysis can be applied on a given model? Which are the analyses that meet a given goal? Are there analyses to be combined? Are there interferences between the analyses? Hence, dedicated solutions must be provided to systematize the analysis of architectural models.

2.2. Related works: systematizing the analysis of architectural models

We can mention several initiatives that preceded our works. We distinguish between *ad hoc* approaches aiming at automating the selection of real-time scheduling analyses and more general analysis integration approaches based on contracts.

Automatic selection of real-time scheduling analyses. Selecting the analysis that can be applied on an architecture model is a complex task. For instance, Plantec et al. [43] aim at automatically choosing real-time schedulability tests based on architecture models. For this purpose, the authors define the relationships between architectural models and schedulability tests by way of architectural *design patterns* [44]. They also propose their own algorithms to select the schedulability tests in the Cheddar tool [45]. Ouhammou et al. [46] follows a similar path: they formalize the assumptions associated to schedulability tests through the notion of *real-time context*. The decision is made via a set of OCL invariants to be checked on the real-time oriented language MoSaRT. We note that the two approaches are limited to specific ADLs and tools (Cheddar and MoSaRT). It is hence necessary to redefine architectural design patterns (or real-time contexts) for *every* ADL, e.g. Gaudel et al. [47] redefined architectural design patterns for AADL models through AADL *subsets*.

Analysis tools integration. Ruchkin et al. [48] deal with the integration of "Cyber-Physical Analyses" in the AADL tool environment OSATE [49]. They acknowledge that properties of AADL models can be computed by tools coming from different scientific domains (e.g. real-time scheduling, power consumption, safety or security). They hence use the contract formalism [50, 51] to represent the interactions between analysis tools and avoid the execution of conflicting tools (*i.e.* to not invalide the properties computed by a tool with another tool). This is made possible with a language to specify contracts (being part of AADL) and a verification engine based on a combination of SMT solving and model checking. They detail an implementation of this approach through the ACTIVE tool, dedicated to AADL, in [52].

Work positioning. The approach presented in this paper builds on these works, but seeks more generality and is implemented differently.

Gaudel et al. [45] and Ouhammou et al. [46] focus on a particular problem which is to automatically select and apply real-time schedulability tests; and each one proposes a different approach (*i.e.* enforcing design patterns or real-time contexts), implemented in a specific analysis tool (*i.e.* Cheddar or MoSaRT), to resolve this problem. On the contrary, in section 3, we emphasize on the definition of the *semantics* of an analysis in general, and show that an analysis can be made equivalent to a Floyd-Hoare triple. Hence, a full analysis, including preconditions and postconditions, can be implemented

via a constraint language, e.g. REAL to express real-time scheduling analyses. In future works, this general formalism will be investigated to express other kinds of analyses (e.g. behavioral, dependability, security, etc.).

Then, in section 4, we use contracts to provide greater decision support. Contracts have been investigated by Ruchkin et al. [48, 52] in a context similar to ours. However, we go further in several aspects. First of all, we note that the contracts in [48] are tied to AADL (contracts are to be defined in terms of AADL types and through an AADL sub-language annex), while on the contrary our contracts are independent of any modeling language and extensible (defined through the Alloy specification language, independent of the AADL type system). Secondly, the ACTIVE tool [52] provides only behavioral verification with *model checking*, whereas we focus on the analysis of real-time properties using the *real-time scheduling theory.* Last but not least, Ruchkin et al. concentrated especially on the run-time verification of assumptions/guarantees formulas against an AADL model through heavyweight and domain-specific verification engines (*i.e.* SMT solving in Z3 or model checking with Spin/Promela). In contrast, we focus on interface reasoning in terms of *both* data (inputs/outputs) and properties (assumptions/guarantees) in Alloy.

3. Defining the analysis execution

This section aims to formalize the analysis execution. As an introductory example, we explain the difficulty to apply real-time scheduling analyses on architectural models representing an unmanned aerial vehicle. We then propose a general formalism to define the semantics of an analysis, and instantiate it to a simple real-time scheduling analysis. We finally present an implementation of this formalism with the help of the REAL constraint language.

3.1. Case study : real-time scheduling analysis of an unmanned aerial vehicle

As a case study, we consider the timing analysis of an unmanned aerial vehicle. We briefly introduce the task model used in the software architecture written in AADL and the real-time scheduling problem. We then present schedulability tests used for timing analysis and the difficulty to apply them on architectural models.

Paparazzi UAV. Paparazzi UAV (Unmanned Aerial Vehicle) is an open-source drone project [16, 53]. The Paparazzi system consists of an airborne system with hardware and autopilot software and a ground control station. The subsystems communicate with each other via a radio link. In the following we focus on the real-time scheduling analysis of the autopilot software architecture.

Real-time task model. Real-time tasks are the basic entities of a real-time system such as the Paparazzi UAV. A task is a logical unit of computation in a processor, that is a set of program instructions that are to be executed by a processor. A task $\tau_i \in \mathcal{T}$ ($card(\mathcal{T}) = n$, $i, n \in \mathbb{N}$) can have several characteristics, e.g. in the context of the seminal works of Liu and Layland [54] the tasks are periodic. A periodic task τ_i consists of an infinite sequence of jobs $\tau_{i,j}$ ($j \in \mathbb{N}$). A task can admit an offset O_i that is the amount of time for the first release of the task. This implies that the jth job of a task is released at time $r_{i,j} = O_i + (j-1) \cdot T_i$ where T_i is the task period. Each job consumes an

amount of processor time C_i called execution time. Finally, a task has a relative deadline D_i, or an absolute deadline expressed on the jth job of the task: $d_{i,j} = r_{i,j} + D_i$. Figure 2 represents a periodic task execution with usual parameters as a Gantt diagram.

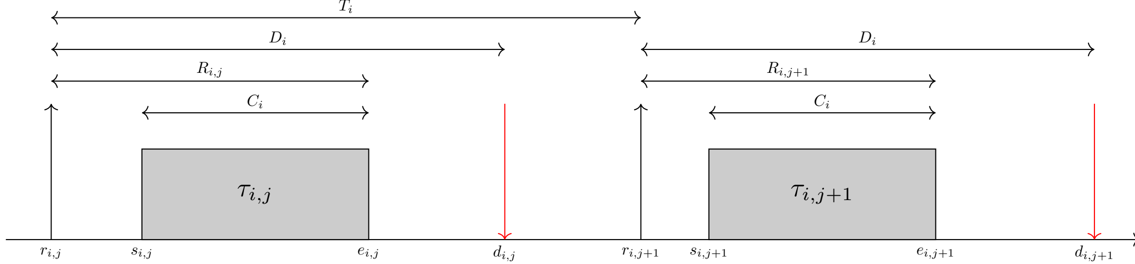

Figure 2: Usual representation of a real-time task with a Gantt diagram. For a task τ_i: T_i the period, C_i the execution time and D_i the relative deadline. $\tau_{i,j}$ denotes the jth job of a task i: $r_{i,j}$ is the release time, $s_{i,j}$ the start time, $e_{i,j}$ the completion time, $d_{i,j}$ the absolute deadline. A system is schedulable if $\forall \tau_i \in \mathcal{T}$, all the response times $R_{i,j}$ respect $R_{i,j} \leq D_i$.

Figure 3 presents an overview of the autopilot software architecture in the AADL graphical syntax (we have updated, corrected and extended the AADL models initially proposed in [55, 56]). The autopilot process consists of 12 tasks (dashed shapes) that use a data (plain rectangle), functioning in three possible modes (hexagons). Listing 1 describes a task implementation in the AADL textual syntax. A task implementation defines important task parameters enumerated above (`Period`, `Compute_Execution_Time`, etc.) and refers to the actual source code (`Source_Text` and `Compute_Entrypoint_Source_Text`).

Real-time scheduling. Real-time scheduling is the problem of coming up with an execution order (*i.e.* a schedule) that meets the timing constraints, usually the deadline constraints. In the case of on-line scheduling, a scheduling algorithm decides the scheduling of a set of tasks $\mathcal{T} = \{\tau_1, \tau_2, \ldots, \tau_n\}$ on a set of processor $\mathcal{X} = \{x_1, x_2, \ldots, x_m\}$ and, possibly, a set of shared resources $\mathcal{U} = \{U_1, U_2, \ldots, U_s\}$. For example, fixed-priority scheduling according to the Rate Monotonic (RM) priority assignment policy is characterized by preemption (*i.e.* it is able to suspend a task execution to execute one or several other tasks, and then resume the execution of the first task), periodic tasks, implicit deadlines ($D_i = T_i$) and fixed-priority scheduling according to the rule "the smaller the period, the higher the priority".

How to apply real-time scheduling analyses on architectural models? Scheduling analysis aims to determine whether the scheduling algorithm will produce a schedule that will meet the timing constraints at run-time. Since the origins of the real-time scheduling theory in the 1970s, the research community has provided a multiplicity of real-time scheduling analyses, targeting many task models or evaluating different performance metrics from these models.

For instance, Liu and Layland [54] proposed a schedulability test that is based on the analysis of the *processor utilization factor* U which is the fraction of processor time used by the tasks. They have shown that a set of n periodic tasks is schedulable with the RM

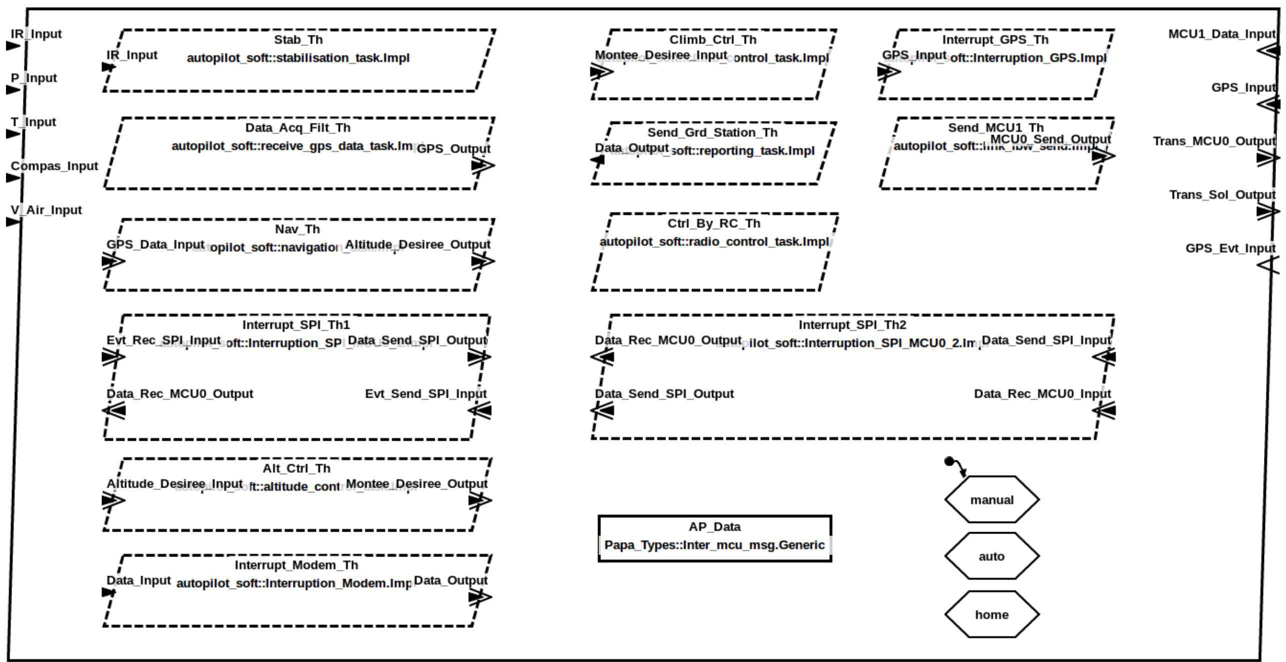

Figure 3: Architecture of the Paparazzi UAV autopilot software in AADL. The overall autopilot process consists of 12 tasks (dashed shapes) that use a data (plain rectangle), functioning in three possible modes (hexagons). One problem is to determine which real-time scheduling analysis can be applied on this model.

```
1  -- Thread implementation instantiated as Alt_Ctrl_Th
2
3  thread implementation altitude_control_task.Impl
4  properties
5     Dispatch_Protocol=> Periodic;
6     Dispatch_Offset => 0 ms;
7     Period => 250 ms;
8     Compute_Execution_Time => 1478 us .. 1660 us;
9     Source_Text => ("autopilot/main.c");
10    Compute_Entrypoint_Source_Text =>"altitude_control_task";
11 end altitude_control_task.Impl;
```

Listing 1: Example of task implementation through an AADL `thread`. A `thread implementation` defines important parameters such as the period or the execution time and refers to the actual source code.

policy if:

$$U \leq n(2^{\frac{1}{n}} - 1) \text{ with } U = \sum_{i=1}^{n} \frac{C_i}{T_i} \quad \text{(LL-test)}$$

The LL-test is a proved *sufficient condition* for the scheduling of a set of periodic tasks under Rate Monotonic and all the tasks have no offsets, the deadlines are equal to the periods and the tasks are independent (that is to say, have no shared resources or precedence constraints).

Later developments have improved or proposed new schedulability tests, relaxed the assumptions, or considered new task models. For instance, Sha et al. [57] deal with real-time tasks with shared resources where access to the resources is managed with a concurrency control protocol. They have shown that a set of n periodic tasks using the Priority Ceiling Protocol is schedulable with Rate Monotonic if the following inequation is true:

$$\frac{C_1}{T_1} + \ldots + \frac{C_n}{T_n} + max\left(\frac{B_1}{T_1}, \ldots, \frac{B_{n-1}}{T_{n-1}}\right) \leq n(2^{\frac{1}{n}} - 1) \quad \text{(SRL-test)}$$

In the inequation above, B_i denotes the worst-case blocking time for a task τ_i, that is the time that this task can be blocked by all the lower-priority tasks that can access a shared resource.

Another approach is to calculate the worst-case response time R_i of each task. The set of tasks is schedulable according to a given scheduling algorithm if and only if the worst-case response time of each task is less than, or equal to, its deadline. For example, [58] deals with the response time analysis of a set of tasks scheduled under Rate Monotonic.

There exists plenty of other schedulability tests. To give an idea, 200+ articles are cited by Sha et al. [20] about the advances in real-time modeling and associated analyses. In another survey, Davis et al. [59] examine the schedulability tests available for multiprocessor architectures and list about 120 different works.

Hence, applying the *right* real-time scheduling analysis on the *right* architectural model (for example, the AADL model in Figure 3) is a tedious and error-prone task. The problem for the designer is first to define the conditions under which an analysis can be applied (e.g. assumptions on the system model) and then to state whether the input model complies with these conditions or not (e.g. by analyzing the values of the AADL properties, the connections between the components, etc.). In addition, one must be able to qualify the analysis result: which result is provided? For example, is the analysis result about the processor utilization factor or the worst-case response time? How to conclude about the schedulability status from this result? In the next subsections, we propose solutions to address this problem:

- by fully defining an analysis with pre and postconditions in subsection 3.2,
- by exploring implementation means in subsection 3.3 and subsection 3.4.

3.2. Semantics of an analysis

An elementary model-based analysis process consists of the computation chain shown in Figure 4:

❶ the analysis inputs data from a model,
❷ the analysis program processes the data,
❸ the analysis outputs data about the model (*i.e.* analysis results).

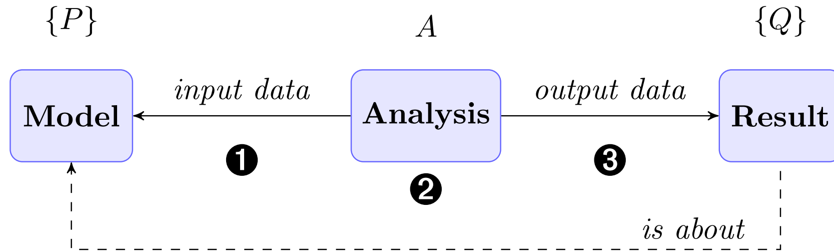

Figure 4: Formalization of the semantics of an analysis with a Hoare triple: $\{P\}\ A\ \{Q\}$. Preconditions P express the properties to hold true on an input model to successfully execute an analysis A. Postconditions Q are the properties guaranteed on the model after execution of the analysis.

Thus, we propose to define the semantics of an analysis with a Hoare-like triple $\{P\}\ A\ \{Q\}$ [60]:

- P is an assertion expressed on input data called the *precondition* of A,
- A is an analysis program to compute output data from input data,
- Q is an assertion expressed on output data called the *postcondition* of A.

Preconditions express the properties that the model must satisfy prior to execute an analysis. Postconditions express the properties that the analysis guarantees in return. For example, assertions can be expressed with first-order logic formulas and checked through a relevant verification engine.

Example: semantics of the test of Liu and Layland. Let us consider a simple input data model that can be used for real-time scheduling analysis, consisting of the tuple $(\mathcal{T},\ G,\ \mathcal{X},\ S)$:

- $\mathcal{T}$ is the set of task, with each $\tau_i \in \mathcal{T}$ is a tuple (T_i, C_i, D_i, O_i) (respectively: the period, the execution time, the deadline and the offset),
- G is the graph (V,E) giving the dependencies between the tasks,
  - V are vertices, each vertex is a task of the model $V \subseteq \mathcal{T}$,
  - $E \in V \times V$ are edges and represent dependencies between tasks,
- $\mathcal{X} = \{x_1, x_2, \ldots, x_m\}$ is the set of processors. Each $x_i \in \mathcal{X}$ can be defined by a boolean value *preempt* $\in \{true, false\}$,
- S is the scheduling algorithm, $S \in \{FP, RM, DM, \ldots\}$ where *FP=Fixed Priority, RM=Rate Monotonic, DM=Deadline Monotonic*, etc.

Liu and Layland defines up to 10 assumptions on the task model to analyze with their schedulability test [54]:

- **mono-processor (p_1)**: there is just one processor,
- **preemption (p_2)**: all the tasks are fully preemptive,
- **periodic tasks (p_3)**: all the tasks are periodic,
- **implicit deadlines (p_4)**: all the tasks have a deadline equal to their period,
- **independent tasks (p_5)**: all the tasks are independent, that is, have no shared resources or precedence constraints,
- **bounded execution times (p_6)**: all the tasks have a fixed execution time, or at least a fixed upper bound on their execution time, which is no greater than their period,
- **no jitter (p_7)**: all the tasks are released exactly at the beginning of periods,
- **no self-suspension (p_8)**: no task may voluntarily suspend itself,
- **no overheads (p_9)**: all overheads (*i.e.* extra delays, in particular the delays due to scheduling and context switching) are assumed to be null,
- **fixed priority (p_{10})**: all the tasks have a priority that is constant over time.

According to the input model defined previously and the assumptions given above, we can define the preconditions with predicates in First-Order Logic:

$$P_{LL-test} = \{p_1 \wedge \ldots \wedge p_{10}\}$$

with

- $p_1 := \{\mathcal{X} \mid card(\mathcal{X}) = 1\}$
- $p_2 := \{\forall x_i \in \mathcal{X} \mid preempt = true\}$
- $p_3 := \{\forall \tau_i \in \mathcal{T} \mid T_i \neq \emptyset\}$
- $p_4 := \{\forall \tau_i \in \mathcal{T} \mid T_i = D_i\}$
- $p_5 := \{G \mid card(V) = 0\}$
- $p_6 := \{\forall \tau_i \in \mathcal{T} \mid C_i \leq T_i\}$
- $p_{10} := \{S \mid S = RM \vee S = FP\}$
- p_7, p_8 and p_9 are axioms, alternatively the data model could be extended with any suitable data structure onto which those predicates could be expressed (for example a graph explaining the task behaviors or a property variable associated with a processor as done for p_2).

Provided the respect of the preconditions, the schedulability test by Liu and Layland computes the processor utilization factor U (see LL-test). According to the LL-test, there is only one postcondition that determines the schedulability of the task set: $Q_{LL-test} = \{q_1\}$ with

- $q_1 := \{U \mid U \leq card(\mathcal{X})(2^{\frac{1}{card(\mathcal{X})}} - 1)\}$

The next subsections presents a practical implementation of this formalism.

3.3. Analysis execution

In the previous section, we showed that an analysis can be made equivalent to a Hoare triple. In particular, the preconditions are the properties to be true in an input model to successfully execute an analysis. The postconditions are the properties guaranteed on the model after the analysis execution. At run-time, we hence evaluate the preconditions prior to execute the analysis, and check the postconditions at the end of the analysis execution.

Figure 5 explains the analysis process in greater detail with a Process Flow Diagram. At the very beginning, we verify the analysis preconditions on the model (1). If the model fulfills the preconditions then we can carry out the analysis (2a). Otherwise, the process terminates (2b). Lastly, we check the analysis postconditions (3). The process ends whether the postconditions are confirmed or not.

In the next section, we show that a complete analysis, including preconditions and postconditions, can be implemented via a constraint language. As an example, we use a constraint language named REAL for the timing analysis of the Paparazzi UAV autopilot software architecture written in AADL.

3.4. An example of implementation with the REAL constraint language

REAL at a glance. In former works, Gilles et al. [14] proposed REAL (Requirements Enforcement and Analysis Language) to express and verify constraints on AADL models. It has been designed as an AADL annex language and works with its own checker.

REAL considers `theorems` as basic execution units. A theorem expresses one or more constraints to be checked on an AADL model based on model queries and analysis capabilities. REAL provides key features for our application:

- it makes it possible to manipulate the elements of an AADL instance model as sets (`thread_set`, `bus_set`, `memory_set`, etc.) with getters for their properties (`get_property_value`),
- it enables arithmetics operations with classical operators (+, −, ×, etc.) and high-level functions (`cardinal`, `min`, `max`, etc.),
- it provides a syntax for predicate calculus with quantifiers (∀, ∃), logical operators (¬, ∧, ∨, etc.) and predicate functions (`is_subcomponent_of`, `is_bound_to`, etc.).

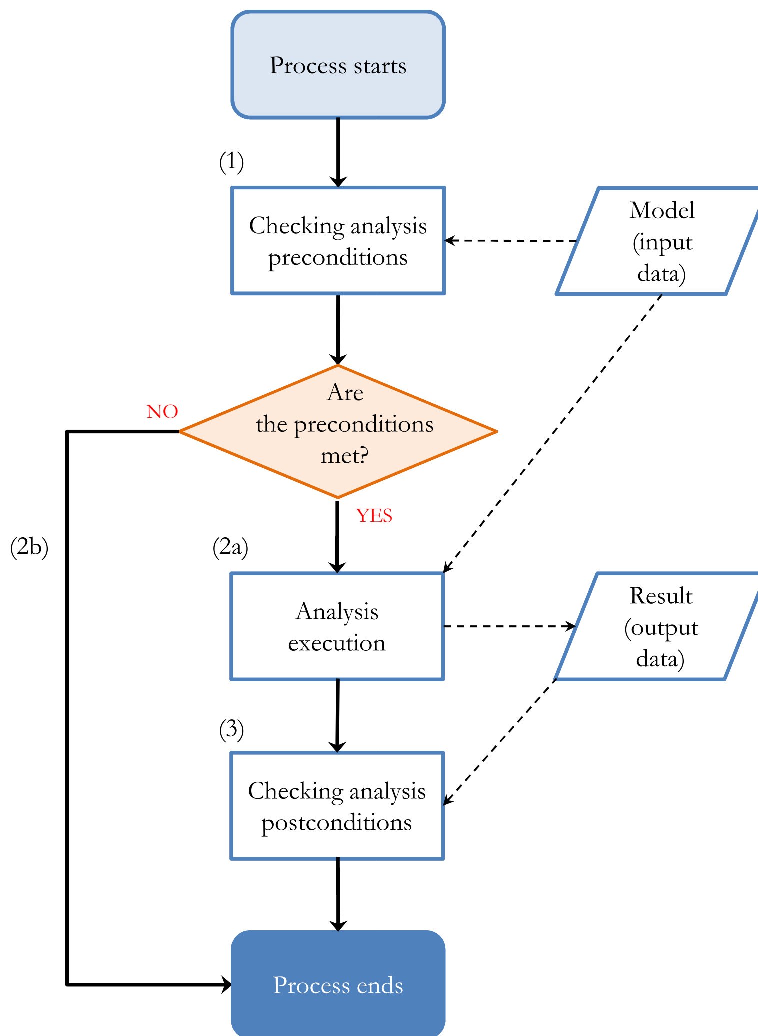

Figure 5: Process Flowchart describing the analysis execution which depends on the verification of analysis preconditions. The analysis result is checked at the end of the analysis execution.

Preconditions of the test of Liu and Layland. In Listing 2, the `periodic_tasks` theorem implements the precondition p_3=“all the tasks are periodic”. The translation of the predicate is straightforward: we check that the `Period` property is provided (`property_exists` predicate function) for each element in the task set (*i.e.* the `thread_set` in the AADL instance model).

The theorems needed to express the **mono-processor** (p_1), **preemption** (p_2), **implicit deadlines** (p_4), **bounded execution times** (p_6), and **fixed priority** (p_{10}) preconditions are of similar complexity.

```
1 -- This theorem checks that all tasks are periodic by checking by a period is
     defined for each task
2 theorem periodic_tasks
3    foreach t in Thread_Set do
4    check (property_exists (t, ‘‘Period’’));
5 end periodic_tasks;
```

Listing 2: An example of REAL theorem. A REAL theorem expresses constraints on a AADL model. The simple theorem here is used to check that the `threads` described in the model are periodic.

Listing 3 provides the theorem for the precondition p_5=“all the tasks are independent”. The `independent_tasks` theorem requires that two sub-theorems are true: `no_task_precedences` and `no_shared_data`.

The first sub-theorem assumes that a `task_precedence` involves a connection between two AADL threads (`Is_Connected_To (t2, t1)` with `t1` and `t2` are elements in the `thread_set`) and checks that the number of precedences is null (`cardinal (task_precedence) = 0`).

In the second sub-theorem, we assume that a shared data situation occurs when at least two AADL threads access a same AADL data (`Is_Accessing_To (t,d)` with `d in Data_Set` and `t in Threads_Set`). We thus check that at most one thread accesses each data (`Cardinal (accessor_threads) <= 1`).

Analysis and postconditions. Listing 4 shows the full implementation of the Liu and Layland’s schedulability test with REAL theorems where the topmost theorem `ll_test` implements the actual schedulability test.

In this theorem, we first evaluate the preconditions under which the analysis is applicable (`requires` keyword at line 6). The preconditions are listed in the `ll_context` sub-theorem (lines 19 to 23) and fully defined in other sub-theorems (e.g. we presented the `periodic_tasks` and `independent_tasks` theorems in the previous paragraphs, see Listings 2 and 3). If the preconditions are met, then the test can be executed (the `requires` command at line 6 aborts the main theorem if any predicate is false).

The analysis then executes (`compute` keyword at line 10). We calculate the processor utilization factor (`var` U, line 10) via the `processor_utilization_factor` sub-theorem (lines 30 to 34). This sub-theorem needs the set of threads, previously retrieved from the AADL model at lines 9.

Lastly, we evaluate the postcondition (`check` keyword at line 12). We check that the processor utilization factor is under the acceptable limit. If the test succeeds, then the task set represented in the AADL model is schedulable.

```
-- independent_tasks : this theorem checks that tasks are mutually independent,
   i.e.
-- (1) tasks have no precedence relationships
-- (2) tasks do not share (access) a same resource and

theorem independent_tasks
  foreach e in Local_Set do      -- the set passed as a theorem argument (none)
    requires(no_tasks_precedences and no_shared_data);
  check (1=1);
end independent_tasks;

-- subtheorem to check task precedences
theorem no_tasks_precedences
  foreach t1 in Thread_Set do
  task_precedence := { t2 in Thread_Set | Is_Connected_To (t2, t1)};
  check (Cardinal (task_precedence) = 0);
end no_tasks_precedences;

-- subtheorem to check shared data
theorem no_shared_data
  foreach d in Data_Set do
  accessor_threads := {t in Thread_Set | Is_Accessing_To (t, d)};
  check (Cardinal (accessor_threads) <= 1);
end no_shared_data;
```

Listing 3: Independent tasks theorem. The theorem on top checks that the `threads` in the AADL model are independent: (1) a task cannot precede another, *i.e.* in AADL a `thread` cannot be connected to another one (second theorem); (2) the `threads` cannot share `data` with each other (third theorem).

3.5. Summary

In this section, we formalized the analysis execution. We showed that an analysis can be made equivalent to a Hoare triple $\{P\}\ A\ \{Q\}$. The preconditions P express the properties that the model must satisfy prior to execute an analysis. The postconditions Q express the properties that the analysis guarantees in return. Hence, a full analysis requires to first validate the preconditions, secondly execute the analysis, and lastly check the postconditions. We presented an implementation of this approach using the REAL constraint language that works with AADL. Let us note that a similar experiment could be performed with OCL on UML-based models. This approach has been applied for the real-time scheduling analysis of a real system, the Paparazzi UAV.

The next section extends this work through the notion of contract to provide greater automation of the analysis process.

4. Managing the analysis process

Preconditions and postconditions discussed in the previous section define the analysis execution. Yet, except ensuring the applicability of an analysis, this does not provide more automation support: e.g. which analysis can be applied on a given model? Which are the analyses that meet a given goal? Are there analyses to be combined? Are there interferences between analyses?

In this section, we firstly explain that analyses are an integral part of Model-Based Systems Engineering approaches, supported for instance via the AADL language. We then present contracts to define the interfaces of an analysis, and contract reasoning to manage the analysis process. We present an implementation of this approach using Alloy.

```
1  -- ll_test : this main theorem implements a schedulability test by Liu and
       Layland
2
3  theorem ll_test
4    foreach e in Processor_Set do
5      -- verification of the analysis preconditions
6      requires ( ll_context );
7      -- analysis computation
8        Proc_Set(e) := {x in Process_Set |  Is_Bound_To (x, e)};
9        Threads := {x in Thread_Set | Is_Subcomponent_Of (x, Proc_Set)};
10     var U := compute processor_utilization_factor (Threads);
11     -- verification of the analysis postcondition
12     check (U <= (Cardinal (Threads) * (2 ** (1 / Cardinal (Threads))) -1));
13 end ll_test;
14
15 -- subtheorem: verification of the test assumptions
16
17 theorem ll_context
18   foreach t in Thread_Set do
19     requires (mono_processor
20         and preemption and periodic_tasks
21         and implicit_deadlines and independent_tasks
22         and bounded_execution_times and fixed_priority);
23     check (1=1);
24 end ll_context;
25
26 -- subtheorem: computation of the processor utilization factor
27
28 theorem processor_utilization_factor
29   foreach e in Local_Set do      -- the set passed as a theorem argument (a
         subset of Thread_Set)
30     var Period := get_property_value (e, ``period'');
31     var WCET := last (get_property_value (e, ``compute_execution_time''));
32     var U := WCET/Period;
33     return (MSum (U));
34 end processor_utilization_factor ;
```

Listing 4: A complete schedulability test implemented in REAL. The analysis starts in the theorem on top. At line 6, the preconditions are verified by calling the second theorem. If all the assumptions associated to the test are true, then the processor utilization factor is calculated by calling the third theorem at line 10. The postconditions are finally checked at line 12.

4.1. Motivating context: analysis in a MBSE process supported by AADL

Architecture Description Languages provide a support for the Model-Based Engineering of real-time embedded systems. Figure 6 depicts an advanced design process based on AADL modeling and systematic analysis of the AADL models:

1. the AADL model is the centerpiece of the process. The AADL model represents the top-level architecture of the system. It describes the static software architecture and the computer platform architecture with behavioral descriptions in a single model,
2. analyses are carried out on the AADL model to provide *feedback* about the system design, e.g. to assess the processor workload or analyze the schedulability of the task set,
3. the system is progressively defined and validated via the successive modeling and analysis steps. The end product files (executable files, configuration files, etc.) can be fully or partially generated from high-level models.

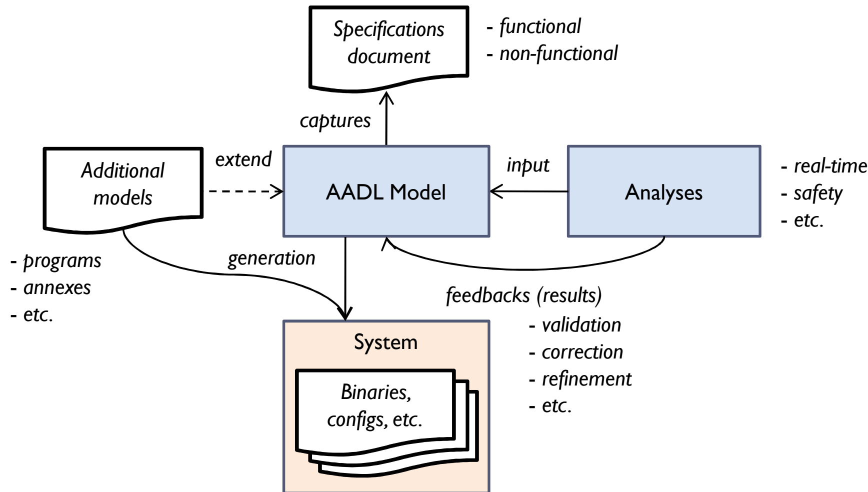

Figure 6: Model-Based Engineering process supported by AADL. The AADL model represents the system architecture with functional and non-functional requirements. Analyses are conducted from AADL analytical representations, e.g. to assess real-time or safety quality attributes. Finally, the end product files such as the runnables can be generated.

How to manage the analysis process? We note that, apart from high-level principles and abstract guidelines, MBSE tools such as OSATE (Open Source AADL Tool Environment) [49] provide little support to carry out the modeling and analysis steps.

Let us discuss a simple design flow represented with a directed graph in Figure 7. The vertices represent the modeling and analysis activities, whereas the directed edges represent the transitions between the activities:

- the designer starts by modeling the system with AADL (M vertex),

- the designer can apply an analysis (*ll_test* vertex) to determine the schedulability of the task set. To apply this schedulability test, the designer must check that the analysis preconditions are true (*ll_context* vertex) and compute the processor utilization factor used by this analysis (*comp_U* vertex),
- if the schedulability test succeeds, the model is validated (G vertex), if not the model is to be corrected (M' vertex).

From Figure 7, we observe that the analysis process comprises several elements: (1) *models* that must be analyzed (M, M'); (2) *goals* that are the properties to be assessed on those models (G); (3) *analyses* that can be applied on these model to meet the goals (*ll_context*, *comp_U*, *ll_test*). Hence, the problem for the designer is to decide the sound analysis process to apply in presence of multiple models, analyses and goals: *given a model and a set of analyses, which is the analysis process that meets the goals?*

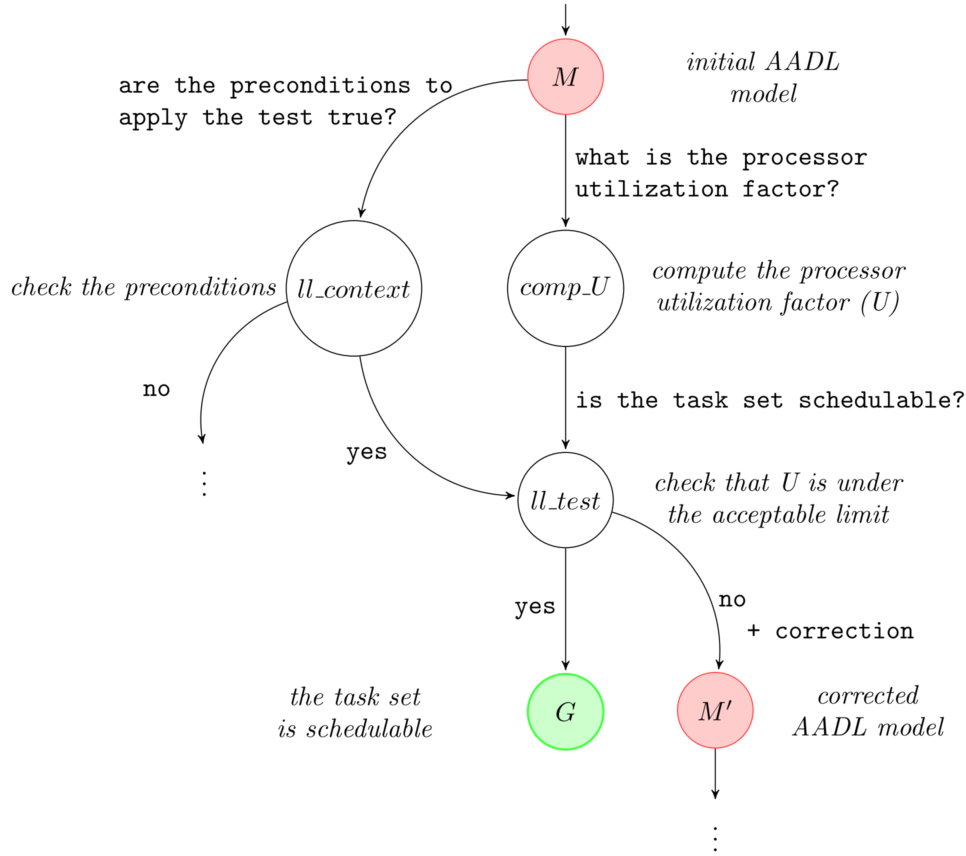

Figure 7: An example of design flow. The design flow involves modeling and analysis steps (represented in red and white shapes respectively) to achieve a goal (green shape). Vertically: one must execute several analyses in a precise order to determine whether the task set represented in the initial AADL model is schedulable or not. If not the model must be corrected.

We must adopt a more systematic view to define and automatically compute analysis

features, e.g. interfaces and properties, during the design process. We present our solutions in the next subsections:

- we first present contracts in subsection 4.2,
- we then explain how contracts can be used to automate the analysis process in subsections 4.3, 4.4 and 4.5.

4.2. Contracts

We first remind the reader of the notion of contract which has been formerly introduced in [61]. A contract $K = (I, O, A, G)$, represented in Figure 8, formally defines the interfaces of a model, an analysis or a goal (hereinafter referred to as the 'element') in terms of data and properties:

- I are inputs: the data required by the element,
- O are outputs: the data provided by the element,
- A are assumptions: the properties required by the element,
- G are guarantees: the properties provided by the element.

Notice that the properties relate to preconditions and postconditions introduced in section 3, whereas the data are the data from which these properties are computed.

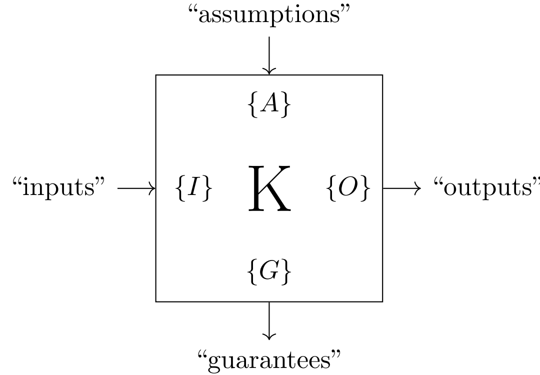

Figure 8: Representation of a contract. A contract formally defines the interfaces of a model, an analysis or a goal in terms of required and provided data and properties. It specifies the data through inputs and outputs, and properties via assumptions and guarantees.

For example, the contract for the Liu and Layland's schedulability test (involving computation of the processor utilization factor) can be defined as follows: $K_{LL-test} = (I, O, A, G)$ with

- $I = \{\mathbf{Per}, \mathbf{Exec}, \mathbf{Sched}, \ldots\}$. The analysis inputs data from the model such as the task periods (**Per**), execution times (**Exec**) or scheduling policy (**Sched**).

- $O = \{\mathbf{U}\}$. The analysis computes (outputs) data about the model: the processor utilization factor (**U**),
- A = "Liu and Layland's assumptions" = $\{\mathbf{perTasks}, \mathbf{boundedExec}, \mathbf{fixedPrio}, \ldots\}$. The analysis requires several properties to be true: tasks must be periodic (**perTasks**), tasks have fixed or upper bounded execution times (**boundedExec**), etc.
- $G = \{\mathbf{isSched}\}$. The analysis provides a guarantee on the model: the set of tasks is schedulable or not (**isSched**).

In the next subsections, we explain how contracts can be used to automatically carry out the analysis process.

4.3. Contract-driven analysis process

We propose the approach represented with a Process Flow Diagram in Figure 9. Our approach relies on contracts to set up the analysis paths to be executed in order to reach goals from an input model. The approach consists of 3 main steps.

At first, contracts must be defined for the design elements (*i.e.* models, analyses and goals). Contracts specify the interfaces of an element with first-order logic formulas from both the data (inputs/outputs) and properties (assumptions/guarantees) points of view. Subsequently, contracts are evaluated, which translates into the problem of the satisfiability of contract formulas. A satisfiable interpretation of the contracts defines an analysis graph compliant with a model and a goal. Lastly, we can execute the analysis graph, *i.e.* we visit the graph and execute the analyses.

In the following, Subsection 4.4 presents an implementation of *contracts definition* and their *evaluation* using Alloy. Subsection 4.5 introduces the *execution* engine.

4.4. An example of implementation in Alloy

We implement *contracts definition* and their *evaluation* in Figure 9 by using the Alloy specification language.

Alloy at a glance. Alloy is a language for expressing complex structural constraints completed with a tool for analyzing them [15]. It provides key advantages for our application:

- Alloy is a formal language with abstract and analytical notations that we use to specify contracts,
- Alloy provides tool support to analyze an Alloy specification. We use the Alloy analyzer to evaluate the contracts.

Contracts definition. Alloy is based on a specification that contains *signatures*. Signatures may have *fields* to define relationships with other signatures. In addition, *facts* express constraints on the signatures and fields. We define contracts with Alloy in two parts:

- a basic *signature* specifies the structure of a contract: fields are not only used to represent the contract interfaces (`inputs`, `outputs`, `assumptions` and `guarantees`) but also dependencies with other contracts (`nextHoriz` and `nextVertical`). Listing 5 provides the contract structure in the Alloy syntax,

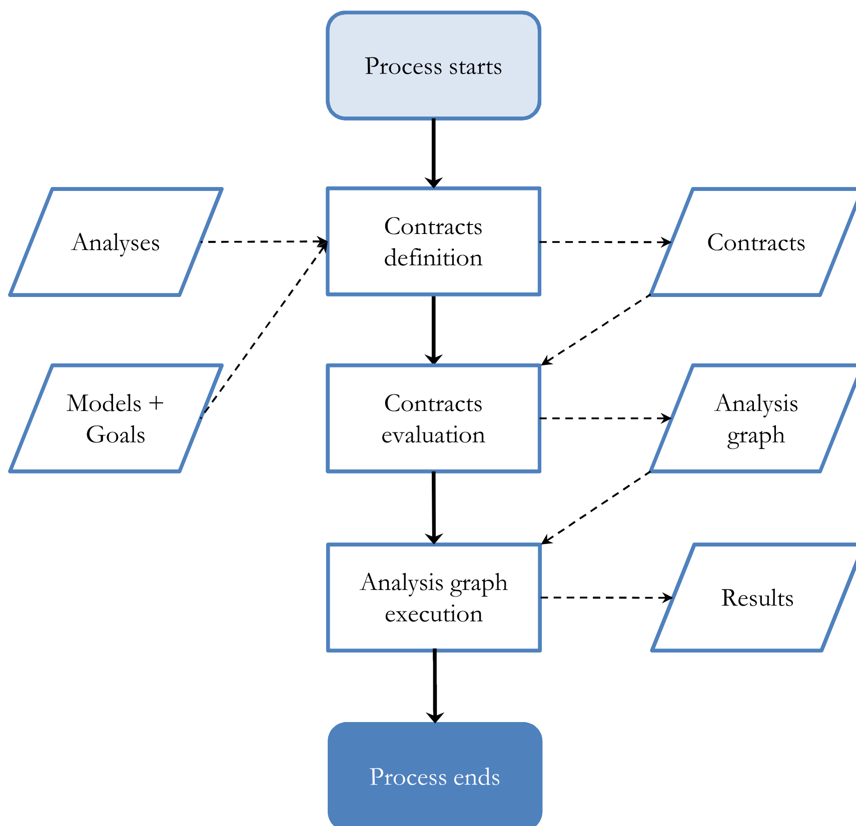

Figure 9: Process Flowchart for contract-driven analysis. The analysis process is executed according to the definition and evaluation of contracts.

- signature *facts* specify the concrete constraints about a contract instance. For example, Listing 6 shows an instance of contract called `ll_test` that specifies the interfaces of the schedulability test proposed by Liu and Layland [54] which is explained briefly in subsection 3.1 (see Eq. LL-test). This contract specifies the `inputs`, `outputs`, `assumptions` and `guarantees` of the analysis. For example, this analysis requires a precise hierarchy of components in input: a system with processors and threads, with periods defined for the threads, execution times, etc.

```
1  /*Basic signatures manipulated in Alloy specification*/
2
3  /*Definition of Data and Properties signatures*/
4  abstract sig Data {}
5  abstract sig Property {}
6
7  /*Definition of the structure of a contract*/
8  abstract sig Contract{
9     //interfaces
10    input: set Data,                    //required-provided data
11    output:set Data,
12    assumption: set Property,           //required-provided properties
13    guarantee: set Property,
14    // relationships with other contracts
15    nextHoriz:set Contract,             // output->input
16    nextVertical:set Contract           // guarantee->assumption
17 }
```

Listing 5: Basic signatures of the Alloy specification. Signatures in Alloy describe the entities to reason about. Here, the contract signature specifies the structure of a contract: fields are not only used to represent the contract interfaces (`input`, `output`, `assumption` and `guarantee`) but also dependencies with other contracts (`nextHoriz` and `nextVertical`).

The Alloy specification is completed in Listing 7 with `VerticalPrecedence` and `HorizontalPrecedence` *facts*. They define the logical conditions under which the `nextHoriz` and `nextVertical` relationships hold between two contracts.

Contracts evaluation. The Alloy analyzer provides full and automatic analysis of an Alloy specification. The Alloy analyzer is a model finder: it searches a model that satisfies the logical formula generated from the Alloy specification. If there is a solution that makes the formula true, Alloy will find it. Alloy offers several SAT solvers for this purpose.

For example, Figure 10 shows a solution as visualized in the Alloy environment for the Paparazzi UAV case study. The Alloy visualizer represents the dependencies between models, analyses and goals as a graph. In this example, the graph exhibits the analysis paths that are to be executed to conclude about the schedulability of the Paparazzi UAV autopilot software modeled in AADL.

We do not present each element of the analysis graph in detail but summarize the information provided to the designer:

1. the Alloy analyzer finds the analyses which are directly applicable on the input AADL model, *i.e.* 6 analyses are connected to the `aadl_model` vertex,
2. it also finds all the dependencies between the analyses, *i.e.* 9 dependencies are represented by edges between analyses,
3. it finally identifies the analyses to reach the goal, *i.e.* 3 analyses are connected to the `is_schedulable` vertex.

```
1  /* A data structure in an AADL model */
2  abstract sig Component extends Data {
3     subcomponents: set Component,
4     type: lone ID,
5     properties: set ID
6  }
7
8  /* An analysis contract that uses the component data structure*/
9  one sig ll_test extends Contract{
10 }{
11    //specification of input data structure
12    input={S:Component |
13                S.type=system and (
14                   some sub:S.subcomponents | sub.type =processor and (
                         scheduling_protocol+preemptive_scheduler) in sub.
                         properties) and (
15                   some sub:S.subcomponents | sub.type=process and
16                         thread in sub.subcomponents.type and
17                         ( let th=sub.subcomponents & thread.~type |
18                               (dispatch_protocol +period +compute_execution_time
                                     +priority+deadline) in th.properties
19                         )
20                   )
21             }
22    //specification of output data structure
23    //assumptions and guarantees
24    [...]
25 }
```

Listing 6: Specification of an analysis contract. `Input/output` fields are defined with respect to the `Component` data structure used for AADL modeling. Here, the analysis expects a precise hierarchy of components which consists of a system with processors and threads, with periods defined for the threads, execution times, etc.

```
1  /* Predicate specifying contract inter-dependencies */
2
3  //between inputs/outputs
4  fact HorizontalPrecedence{
5    all c_current:Contract |
6     c_current.nextHoriz={c_next:Contract|
7        (c_current.output & c_next.input != none) and
8        (all a :c_current.assumption| a in Contract.guarantee) and
9        (all a :c_next.assumption| a in Contract.guarantee)
10 }
11
12 //between assumptions/guarantees
13 fact VerticalPrecedence{
14     all c_current:Contract |
15       c_current.nextVertical={c_next:Contract|
16          (c_current.guarantee & c_next.assumption != none)
17    }
```

Listing 7: Additional constraints on signatures and fields expressed with facts. Here, the inter-dependencies between inputs/outputs and assumptions/guarantees fields of contracts are defined by `HorizontalPrecedence` and `VerticalPrecedence` facts respectively

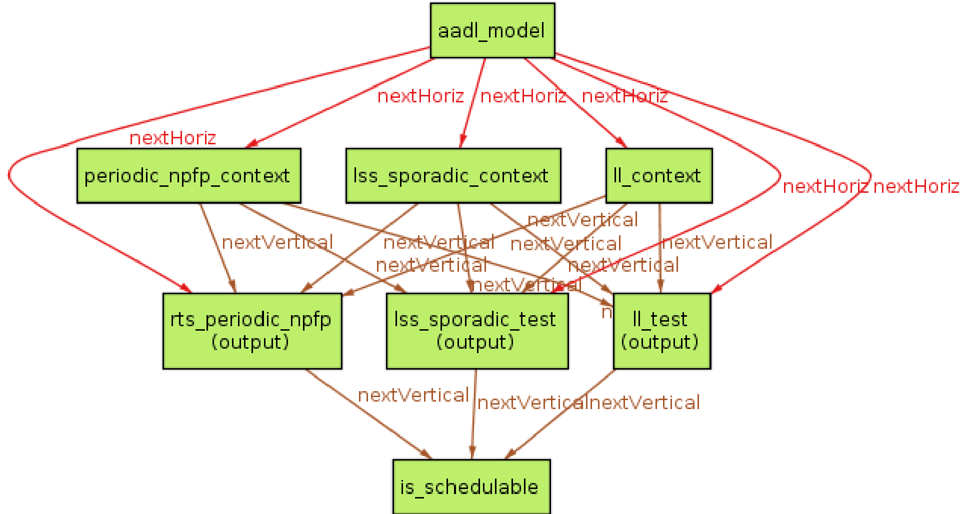

Figure 10: Visualization of a solution found by the Alloy analyzer for contracts specified in Alloy (Paparazzi UAV case study). Here, the structure represents inter-dependencies and precedence order between the models, analyses and goals involved in the analysis process. The solution returned by Alloy can directly be used to execute the analyses.

Next, we can use the graph found by Alloy to execute the analyses. We discuss this process in detail in subsection 4.5.

Performance of Alloy. We evaluated the strengths and shortcomings of an implementation relying on Alloy. We experimented the performance of Alloy on various AADL models (the models are part of the AADLib project accessible online [19]):

- M_1 : a multitasked real-time system implementing the *ravenscar profile*,
- M_2 : a simple *distributed real-time system*,
- M_3 : the *mars pathfinder system*,
- M_4 : a simplified on-board *satellite system*,
- M_5 : a *Flight Management System* (FMS),
- M_6 : the *Paparazzi unmanned aerial vehicle.*

A major benefit of Alloy is that if a solution exists within the user-specified bounds, it will *always* be found. Furthermore, the Alloy analyzer is able to find *all* the solutions in the resolution scope. Concerning experimentations on the case studies, we were able to find solutions for models of realistic complexity in a reasonable time, *i.e.* processing times ranged from a few milliseconds for the simplest model to less than 3 minutes for the most complex case which consists of 5 AADL models, for a total of 125 components and 329 non-functional properties to handle at once. As disadvantages, the use of Alloy requires a minimal expertise to define the contracts and, possibly, adjust manually the resolution scope of the SAT solver. More exhaustive experimental results are presented in [61].

4.5. Analysis graph execution

We can finally use the graph found by Alloy to execute the analyses in a sound order.

Visiting strategy. We could visit the analysis graph in many ways. In particular, the chosen strategy must fulfill two constraints:

1. the graph must be visited in such a way that the data and the properties used by an analysis are computed beforehand,
2. the analyses for which the assumptions are not validated must not be executed; more widely, the analysis paths that include analyses for which the preconditions are no met must be aborted.

We fulfill these requirements in two steps:

1. using a Breadth First Search (BFS) algorithm to enforce the precedences between the analyses,
2. by computing the properties in their priority order, *i.e.* before executing the subsequent analysis. When a precondition is not met, the subsequent analysis path is removed from the execution stack.

According to this policy, the analysis graph found by Alloy for the Paparazzi UAV case study (Figure 10, reproduced in Figure 11) is visited in the following order:

```
aadl_model -> lss_sporadic_context -> ll_context ->
periodic_npfp_context -> lss_sporadic_test -> ll_test ->
rts_periodic_npfp -> isSched.
```

This execution stack enables to compute the data and the properties in a correct order. In addition, if a property (represented with red arrows in Figure 11) is not satisfied the subsequent elements are removed from the execution stack. For instance, if the property computed by the `ll_context` analysis is *false* then the subsequent `ll_test` analysis will not be executed. Notice that discarding a path does not prevent from reaching the goal `isSched` if a correct alternative path exists: for example, one can execute the `periodic_npfp_context -> rts_periodic_npfp` analysis path if the `ll_context -> ll_test` path fails.

Execution of the Paparazzi UAV analysis graph. Let us consider task sets represented in AADL models at different design stages of the Paparazzi UAV autopilot software (the complete models can be found in [19]). These task sets have the following main characteristics:

- **Step 1**: the AADL model describes periodic, non-preemptive tasks and task scheduling is done according to a Fixed Priority scheduling algorithm,
- **Step 2**: we now rather consider preemptive tasks, still periodic and scheduled according to a Fixed Priority algorithm,
- **Step 3**: we model the system more accurately and consider a mixture of periodic and aperiodic tasks, with preemptive and Fixed Priority scheduling. Aperiodic tasks are scheduled through a Sporadic Server [62].

Figure 11 represents the analysis paths executed at each different step. The analysis paths shown with plain-blue arrows comprise the analyses used to verify the schedulability of the task set from the AADL model: *ll_test* is a schedulability test contributed by Liu and Layland [54] (see subsection 3.1), *lss_sporadic_test* is another schedulability test by Lehoczky [63], *rts_periodic_npfp* is a schedulability test based on response time analysis [64]. Sub-paths shown with dashed-red arrows include analyses (*i.e. ll_context*, *lss_sporadic_context* and *periodic_npfp_context* analyses) needed to verify the preconditions of the diverse schedulability tests. Table 1 summarizes the preconditions of the schedulability tests.

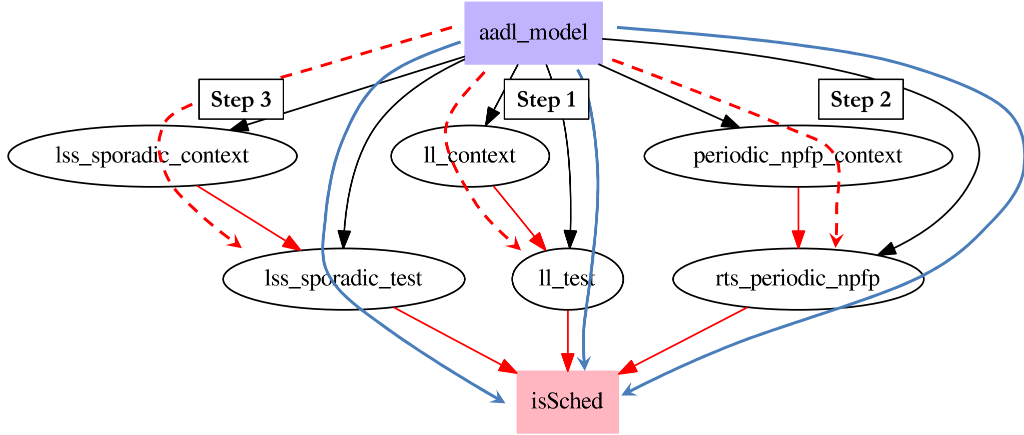

Figure 11: Analysis paths executed at different design stages of the Paparazzi UAV autopilot software. The analysis paths shown with plain-blue arrows comprise the analyses used to verify the schedulability of the task set at each stage. Sub-paths shown with dashed-red arrows include analyses in order to verify the preconditions of the diverse schedulability tests.

For example at step 3, we firstly check the preconditions of the schedulability tests via the *lss_sporadic_context*, *ll_context* and *periodic_npfp_context* analyses. The properties computed by the *ll_context* and *periodic_npfp_context* analyses are *false* because the tasks are not periodic (see Table 1). Therefore, the preconditions of the *ll_test* and the *rts_periodic_npfp* analysis are not fulfilled, meaning that these analyses cannot be executed. Alternatively, we can use the *lss_sporadic_test* as the properties computed by the *lss_sporadic_context* analysis are *true*.

The *lss_sporadic_test* computes the amount of processor time that is used by the set of tasks. In this case, the processor utilization factor encompasses two dimensions: the fraction of processor time consumed by the periodic tasks U_p and the fraction of processor time used by the sporadic server U_s. Lehoczky [63] proved there is a limit not to be exceeded to ensure schedulability:

$$U_p \leq \ln \frac{2}{U_s + 1} \qquad \text{(LSS-test)}$$

According to the result of the *lss_sporadic_test* computed from task parameters described in the AADL model this threshold is respected (Listing 8), meaning that the

Precondition \ Analysis	**ll_test** ([54])	**lss_sporadic_test** ([63])	**rts_periodic_npfp** ([64])
mono-processor	✓	✓	✓
preemption	✓	✓	✗
fixed priority	✓	✓	✓
periodic tasks	✓	✓	✓
aperiodic tasks	✗	✓[1]	✗
jitters	✗	✗	∅
implicit deadlines	✓	✓	∅
bounded execution times	✓	✓	✓
dependent tasks	✗	✗	✗
self-suspension	✗	✗	✗
overhead	✗	✗	✗

[1] aperiodic tasks must be scheduled via a Sporadic Server.

Table 1: Analysis preconditions for the Paparazzi case study. ✓: the predicate must be true. ✗: the predicate must be false. ∅: no restriction.

system is schedulable under a Fixed Priority scheduling algorithm (priority assignment is done according to the Rate Monotonic policy).

```
$ python main.py
[...]
Execute lss_sporadic_test...
lss_sporadic_test is satisfied, U is 0.673264 <= 0.676408064556 -> the tasks set
    is schedulable!
```

Listing 8: Result of the *lss_sporadic_test* computed via our command-line tool for task parameters described in the AADL model.

The analysis process at steps 1 and 2 applies the same strategy as step 3 but it executes different analysis paths, as represented in Figure 11. Indeed, the input model represents different task sets: non-preemptive scheduler and periodic tasks at step 1, preemptive scheduler and periodic tasks at step 2. The complete experimental results for the Paparazzi UAV case study can be found in [65].

4.6. Summary

This section presented a proposal to automate the analysis process. We firstly presented contracts to define the interfaces of an analysis in terms of data (inputs/outputs) and properties (assumptions/guarantees). Then, we used SAT resolution methods to reason about these interfaces. In particular, we were able to find: (1) the analyses that were applicable on a model; (2) the analyses that met a given goal; (3) the data dependencies that exhibited analysis paths. We showed that a specification language such as Alloy could be used to support both the contracts definition and their evaluation with associated SAT solvers. We also presented an execution of the graph found by Alloy for the Paparazzi UAV modeled in AADL.

5. Discussion

In this article, we proposed solutions to systematize the analysis of non-functional properties at design time and discussed our practical experience in applying them on a real system, the Paparazzi UAV. This section discusses some potential extensions of our approach.

Relaxing the initial work hypotheses. We believe that the concepts presented in this paper provide enough generality to address many kinds of models and analyses, not only real-time scheduling analysis of architectural models but also behavioral analysis based on Petri Nets, dependability analysis such as Failure Modes and Effects Analysis (FMEA), and so forth. An extension of our approach will require: (1) new accessors to address various kinds of models (architectural, behavioral, etc.); (2) enriched contract interfaces to express and evaluate new types of analysis interfaces, *i.e.* all types of data and properties that can be computed by analyses. Accessors towards other architectural models have already been implemented for the implementation-oriented language CPAL [66, 65] and we plan to implement them for other languages (e.g. UML-based languages SysML and MARTE, synchronous dataflow languages).

Additional contract strategies. Another extension will be to enrich contracts with quality metrics (e.g. computing time, precision of the result). This will allow to handle the analysis dynamics more precisely: coarse-grained but fast analyses can be used during the early design stages, e.g. for prototyping; in-depth and costly analyses are more relevant at the last stages of the design process (before the implementation phase). We note that the evaluation of the quality metrics adds little algorithmic complexity as it can be performed on a weighted analysis graph, e.g. by looking for the shortest analysis paths. Another advanced contract strategy will be to deal with analysis loops that occur when a data or property required by an analysis is computed by another analysis from a result of the first analysis. We will not need to modify contracts but to extend the visitor to execute such analysis loops.

Providing user feedback. The concepts presented in this paper will help provide information about the analysis process to the designer. We envision three main types of feedback in addition to the raw analysis results:

- analysis solutions: the tool indicates the analyses that are applicable on a model, the analyses that fulfill some of the goals, shows possible analysis combinations, shows all analysis paths or only optimal analysis paths according to quality metrics (e.g. complexity, speed, precision), ...
- advanced analysis results: the tool explains analysis results, suggests corrections to be made on a model, provides automatic integration of results in models, ...
- debugging: the tool points out missing data to apply an analysis, provides a trace of the analysis process, indicates which part of the analysis process is to be re-executed when a model is modified, ...

6. Conclusion

In this article, we presented solutions to systematize and then automate the analysis of non-functional properties of embedded systems at design time.

Our motivation comes from the observation that: (1) Model-Based Systems Engineering can be used to design and develop embedded systems; (2) there are numerous modeling formalisms and analysis techniques to assess the quality of a system (e.g. real-time scheduling theory, model checking, etc.); (3) the modeling and analysis techniques remain poorly coupled together, in theory and in practice.

We thus proposed solutions to bridge modeling and analysis efforts. First, we formalized the analysis execution. We showed that an analysis is basically a program with preconditions and postconditions. The preconditions are the properties to be true in an input model prior to execute an analysis. The postconditions are the properties guaranteed on the model after the analysis execution. With preconditions and postconditions satisfied, an analysis is complete and sound. Secondly, contracts formally define the interfaces of an analysis in terms of processed data and properties, thereby enabling to automatically reason about the analysis process. In particular, we are able to find: (1) the analyses that are applicable on a model; (2) the analyses that meet a given goal; (3) the data dependencies that exhibit analysis paths.

We presented an implementation of our approach using a combination of constraint languages (REAL for run-time analysis) and specification languages (Alloy for describing interfaces and reasoning about them). We validated this approach for the real-time scheduling analysis of an existing embedded system: the Paparazzi UAV designed with the Architecture Analysis and Design Language.

Future works will apply the approach presented in this article to other modeling and analysis domains (e.g. behavioral or dependability), investigate advanced contract strategies and how to provide rich analysis feedback to the designer.

References

[1] SAE International, Architecture Analysis and Design Language (AADL) AS-5506A, 2009.

[2] P. Cuenot, P. Frey, R. Johansson, H. Lonn, Y. Papadopoulos, M.-O. Reiser, A. Sandberg, D. Servat, R. Tavakoli Kolagari, M. Torngren, M. Weber, The EAST-ADL Architecture Description Language for Automotive Embedded Software, in: Model-Based Engineering of Embedded Real-Time Systems, Springer, ISBN 9783642162763, 297–307, 2011.

[3] H. Heinecke, K.-P. Schnelle, H. Fennel, J. Bortolazzi, L. Lundh, J. Leflour, J.-L. Maté, K. Nishikawa, T. Scharnhorst, AUTomotive Open System ARchitecture - An Industry-Wide Initiative to Manage the Complexity of Emerging Automotive E/E-Architectures, Convergence International Congress & Exposition On Transportation Electronics (2004) 325–332.

[4] B. Selic, S. Gerard, Modeling and Analysis of Real-Time and Embedded Systems with UML and MARTE: Developing Cyber-Physical Systems, The MK/OMG Press, Morgan Kaufmann, ISBN 0124166199, 9780124166196, 2013.

[5] T. Weilkiens, Systems Engineering with SysML/UML: Modeling, Analysis, Design, The MK/OMG Press, Morgan Kaufmann, ISBN 0123742749, 9780123742742, 2008.

[6] F. Singhoff, J. Legrand, L. Nana, L. Marcé, Cheddar: a flexible real time scheduling framework, in: ACM SIGAda Ada Letters, vol. 24, ACM, 1–8, software available at `http://beru.univ-brest.fr/~singhoff/cheddar/`, 2004.

[7] M. González Harbour, J. G. García, J. P. Gutiérrez, J. D. Moyano, Mast: Modeling and analysis suite for real time applications, in: 13th Euromicro Conference on Real-Time Systems (ECRTS), IEEE, 125–134, software available at `http://mast.unican.es/`, 2001.

[8] K. G. Larsen, P. Pettersson, W. Yi, UPPAAL in a nutshell, International Journal on Software Tools for Technology Transfer (STTT) 1 (1) (1997) 134–152.

[9] H. Garavel, F. Lang, R. Mateescu, W. Serwe, CADP 2010: A toolbox for the construction and analysis of distributed processes, in: Tools and Algorithms for the Construction and Analysis of Systems, Springer, 372–387, 2011.
[10] Object Management Group (OMG), Meta Object Facility (MOF) Core Specification Version 2.5, 2015.
[11] D. Steinberg, F. Budinsky, E. Merks, M. Paternostro, EMF: eclipse modeling framework, Addison-Wesley, ISBN 0321331885, 9780321331885, 2008.
[12] F. Jouault, I. Kurtev, Transforming models with ATL, in: Workshop on Model Transformations in Practice (MTiP), 2005.
[13] J. B. Warmer, A. G. Kleppe, The object constraint language: getting your models ready for MDA, Addison-Wesley, ISBN 0321179366, 9780321179364, 2003.
[14] O. Gilles, J. Hugues, Expressing and enforcing user-defined constraints of AADL models, in: 5th UML and AADL Workshop (UML and AADL 2010), 2010.
[15] D. Jackson, Software Abstractions: logic, language, and analysis, MIT press, ISBN 0262017156 , 9780262017152, 2012.
[16] P. Brisset, A. Drouin, M. Gorraz, P.-S. Huard, J. Tyler, The paparazzi solution, in: 2nd US-European Competition and Workshop on Micro Air Vehicles (MAV), 2006.
[17] P. C. Clements, A survey of architecture description languages, in: Proceedings of the 8th international workshop on software specification and design, IEEE Computer Society, 16, 1996.
[18] P. H. Feiler, D. P. Gluch, Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language, Addison-Wesley, ISBN 0321888944, 9780321888945, 2012.
[19] Open AADL/AADLib – Library of reusable AADL Models, `http://www.openaadl.org/aadlib.html`, (accessed January, 2017).
[20] L. Sha, T. Abdelzaher, K.-E. Årzén, A. Cervin, T. Baker, A. Burns, G. Buttazzo, M. Caccamo, J. Lehoczky, A. K. Mok, Real time scheduling theory: A historical perspective, Real-Time Systems 28 (2-3) (2004) 101–155.
[21] J. A. Estefan, Survey of Model-Based Systems Engineering (MBSE) Methodologies, Tech. Rep., INCOSE MBSE Initiative, 2007.
[22] J. Hugues, G. Brau, Analysis as a First-Class Citizen: An Application to Architecture Description Languages, in: IEEE 17th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 214–221, 2014.
[23] N. Halbwachs, P. Caspi, P. Raymond, D. Pilaud, The synchronous data flow programming language LUSTRE, in: Proceedings of the IEEE, vol. 79, IEEE, 1305–1320, 1991.
[24] A. Benveniste, P. Le Guernic, C. Jacquemot, Synchronous programming with events and relations: the SIGNAL language and its semantics, Science of Computer Programming 16 (2) (1991) 103–149.
[25] G. Berry, G. Gonthier, The Esterel synchronous programming language: Design, semantics, implementation, Science of Computer Programming 19 (2) (1992) 87–152.
[26] J. Forget, A Synchronous Language for Critical Embedded Systems with Multiple Real-Time Constraints, Ph.D. thesis, Université de Toulouse, 2009.
[27] P. Le Guernic, J.-P. Talpin, J.-C. Le Lann, Polychrony for system design, Journal of Circuits, Systems, and Computers 12 (03) (2003) 261–303.
[28] G. Berry, SCADE: Synchronous design and validation of embedded control software, in: Next Generation Design and Verification Methodologies for Distributed Embedded Control Systems, Springer, 19–33, 2007.
[29] E. Jahier, N. Halbwachs, P. Raymond, X. Nicollin, D. Lesens, Virtual execution of AADL models via a translation into synchronous programs, in: 7th International Conference on Embedded Software (EMSOFT), ACM, 134–143, 2007.
[30] Y. Ma, H. Yu, T. Gautier, P. L. Guernic, J. Talpin, L. Besnard, M. Heitz, Toward polychronous analysis and validation for timed software architectures in AADL, in: Design, Automation and Test in Europe (DATE), 1173–1178, 2013.
[31] Z. Yang, K. Hu, J.-P. Bodeveix, L. Pi, D. Ma, J.-P. Talpin, Two Formal Semantics of a Subset of the AADL, in: 16th International Conference on Engineering of Complex Computer Systems (ICECCS), IEEE, 344–349, 2011.
[32] O. Sokolsky, I. Lee, D. Clarke, Schedulability Analysis of AADL Models, in: 20th International Parallel & Distributed Processing Symposium (IPDPS), IEEE, 2006.
[33] O. Sokolsky, A. Chernoguzov, Analysis of AADL Models Using Real-Time Calculus with Applications to Wireless Architectures, Tech. Rep. No. MS-CIS-08-25., University of Pennsylvania Department of Computer and Information Science, 2008.
[34] M.-Y. Nam, K. Kang, R. Pellizzoni, K.-J. Park, J.-E. Kim, L. Sha, Modeling Towards Incremental

Early Analyzability of Networked Avionics Systems Using Virtual Integration, ACM Transactions on Embedded Computing Systems (TECS) 11 (4) (2013) 81:1–81:23.
[35] X. Renault, F. Kordon, J. Hugues, Adapting models to model checkers, a case study: Analysing AADL using time or colored petri nets, in: 20th International Symposium on Rapid System Prototyping (RSP), IEEE, 26–33, 2009.
[36] B. Berthomieu, J.-P. Bodeveix, C. Chaudet, S. Dal Zilio, M. Filali, F. Vernadat, Formal verification of AADL specifications in the Topcased environment, in: 14th International Conference on Reliable Software Technologies Ada-Europe, Springer, 207–221, 2009.
[37] P. C. Ölveczky, A. Boronat, J. Meseguer, Formal semantics and analysis of behavioral AADL models in Real-Time Maude, in: Formal Techniques for Distributed Systems, Springer, 47–62, 2010.
[38] J. Hugues, B. Zalila, L. Pautet, F. Kordon, From the prototype to the final embedded system using the Ocarina AADL tool suite, ACM Transactions on Embedded Computing Systems (TECS) 7 (4) (2008) 42:1–42:25.
[39] A. Johnsen, K. Lundqvist, P. Pettersson, O. Jaradat, Automated verification of AADL-specifications using UPPAAL, in: 14th International Symposium on High-Assurance Systems Engineering (HASE), IEEE, 130–138, 2012.
[40] J.-P. Bodeveix, M. Filali, M. Garnacho, R. Spadotti, Z. Yang, Towards a verified transformation from AADL to the formal component-based language FIACRE, Science of Computer Programming 106 (2015) 30–53.
[41] H. Mkaouar, B. Zalila, J. Hugues, M. Jmaiel, From AADL Model to LNT Specification, in: 20th International Conference on Reliable Software Technologies Ada-Europe, Springer, 146–161, 2015.
[42] B. Xu, M. Lu, A Survey On Verification And Analysis Of Non-Functional Properties Of AADL Model Based On Model Transformation, in: 5th International Conference on Education, Management, Information and Medicine (EMIM), Atlantis Press, 2015.
[43] A. Plantec, F. Singhoff, P. Dissaux, J. Legrand, Enforcing applicability of real-time scheduling theory feasibility tests with the use of design-patterns, in: Leveraging Applications of Formal Methods, Verification, and Validation, Springer, 4–17, 2010.
[44] P. Dissaux, F. Singhoff, Stood and cheddar: AADL as a pivot language for analysing performances of real time architectures, in: 4th European Congress on Embedded Real Time Software and Systems (ERTS), 21, 2008.
[45] V. Gaudel, F. Singhoff, A. Plantec, S. Rubini, P. Dissaux, J. Legrand, An Ada Design Pattern Recognition Tool for AADL Performance Analysis, in: Annual International Conference on Special Interest Group on the Ada Programming Language (SIGAda), ACM, 61–68, 2011.
[46] Y. Ouhammou, E. Grolleau, P. Richard, M. Richard, Reducing the Gap Between Design and Scheduling, in: 20th International Conference on Real-Time and Network Systems (RTNS), ACM, 21–30, 2012.
[47] V. Gaudel, A. Plantec, F. Singhoff, J. Hugues, P. Dissaux, J. Legrand, Enforcing Software Engineering Tools Interoperability: An Example with AADL Subsets, in: International Symposium on Rapid System Prototyping (RSP), IEEE, 2013.
[48] I. Ruchkin, D. De Niz, S. Chaki, D. Garlan, Contract-based integration of cyber-physical analyses, in: 14th International Conference on Embedded Software (EMSOFT), ACM, 23, 2014.
[49] Software Engineering Institute, OSATE2 : An open-source tool platform for AADLv2, `https://wiki.sei.cmu.edu/aadl/index.php/Osate_2`, 2016.
[50] B. Meyer, Applying "Design by Contract", Computer 25 (10) (1992) 40–51.
[51] A. Benveniste, B. Caillaud, A. Ferrari, L. Mangeruca, R. Passerone, C. Sofronis, Multiple viewpoint contract-based specification and design, in: Formal Methods for Components and Objects, Springer, 200–225, 2007.
[52] I. Ruchkin, D. De Niz, S. Chaki, D. Garlan, ACTIVE: A Tool for Integrating Analysis Contracts, in: 5th Analytic Virtual Integration of Cyber-Physical Systems Workshop (AVICPS), LiU Electronic Press, 2014.
[53] Paparazzi - The Free Autopilot, `http://wiki.paparazziuav.org/wiki/Main_Page`, (accessed January, 2017).
[54] C. L. Liu, J. W. Layland, Scheduling algorithms for multiprogramming in a hard-real-time environment, Journal of the ACM (JACM) 20 (1) (1973) 46–61.
[55] F. Nemer, H. Cassé, P. Sainrat, J.-P. Bahsoun, M. De Michiel, Papabench: a free real-time benchmark, in: 6th International Workshop on Worst-Case Execution Time Analysis (WCET), 2006.
[56] Institut de Recherche en Informatique de Toulouse (TRACES team), PapaBench, `https://www.irit.fr/recherches/ARCHI/MARCH/rubrique.php3?id_rubrique=97`, (accessed January, 2017).
[57] L. Sha, R. Rajkumar, J. P. Lehoczky, Priority inheritance protocols: An approach to real-time

synchronization, IEEE Transactions on Computers (TC) 39 (9) (1990) 1175–1185.
[58] N. Audsley, A. Burns, M. Richardson, K. Tindell, A. J. Wellings, Applying new scheduling theory to static priority pre-emptive scheduling, Software Engineering Journal 8 (5) (1993) 284–292.
[59] R. I. Davis, A. Burns, A survey of hard real-time scheduling for multiprocessor systems, ACM Computing Surveys (CSUR) 43 (4) (2011) 35.
[60] C. A. R. Hoare, An axiomatic basis for computer programming, Communications of the ACM 12 (10) (1969) 576–580.
[61] G. Brau, J. Hugues, N. Navet, A Contract-based approach for Goal-Driven Analysis, in: 18th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), IEEE, 2015.
[62] B. Sprunt, L. Sha, J. Lehoczky, Aperiodic task scheduling for hard-real-time systems, Real-Time Systems 1 (1) (1989) 27–60.
[63] J. P. Lehoczky, Enhanced aperiodic responsiveness in hard real-time environments, in: Proceedings of the IEEE Symposium on Real-Time Systems, 261–270, 1987.
[64] J. Migge, Scheduling of recurrent tasks on one processor : a trajectory based model, Ph.D. thesis, Université de Nice, 1999.
[65] G. Brau, Integration of the Analysis of Non-Functional Properties in Model-Driven Engineering for Embedded Systems, Ph.D. thesis, University of Luxembourg, 2017.
[66] N. Navet, L. Fejoz, L. Havet, S. Altmeyer, Lean Model-Driven Development through Model-Interpretation: the CPAL design flow, in: Embedded Real-Time Software and Systems (ERTS), 2016.