A Framework for Evaluating Schedulability Analysis Tools

Lijun Shan[1,2(✉)], Susanne Graf[1], Sophie Quinton[2], and Loïc Fejoz[3]

[1] Verimag, Saint-Martin-d'Hères, France
lijun.shan@imag.fr
[2] Inria Grenoble – Rhône-Alpes, Montbonnot-Saint-Martin, France
[3] RealTime-at-Work (RTaW), Villers-lés-Nancy, France

Abstract. There exists a large variety of schedulability analysis tools based on different, often incomparable timing models. This variety makes it difficult to choose the best fit for analyzing a given real-time system. To help the research community to better evaluate analysis tools and their underlying methods, we are developing a framework which consists of (1) a simple language called *RTSpec* for specifying real-time systems, (2) a tool chain which translates a system specification in RTSpec into an input for various analysis tools, and (3) a set of benchmarks. Our goal is to enable users and developers of schedulability analysis tools to compare such tools systematically, automatically and rigorously.

Keywords: Real-time systems · Schedulability analysis · Formal semantics

1 Introduction

Schedulability analysis is an offline approach to evaluating the temporal correctness of real-time (RT) systems in terms of whether all software tasks meet their deadlines at runtime. Numerous timing models and corresponding schedulability tests have been proposed since the 1970s; see [10,24] for surveys. Some of them have been implemented in tools, called *analyzers* in the sequel, e.g. MAST [15], TIMES [5], Cheddar [26], SymTA/S [16], SchedMCore [8], pyCPA [12], etc.

This variety of analyzers makes it difficult to choose the best fit for a given real-time system under study. Indeed, the timing models underlying analyzers are often incomparable, mainly because they make incomparable choices on the precision with which one can express the timing-relevant aspects of an RT system. Such choices mainly concern the models describing (1) the activation of tasks, (2) their resource requirements and (3) the scheduling policies used to arbitrate between them. Also, schedulability is only one possible type of timing requirement: other options include e.g. weakly-hard properties (no more than m deadline misses out of k task executions).

To compare the expressivity of the models used by analyzers as well as the precision of the analysis results that they produce, we need a common set of

L. Aceto et al. (Eds.): Larsen Festschrift, LNCS 10460, pp. 539–559, 2017.
DOI: 10.1007/978-3-319-63121-9_27

test cases provided in a *common input format*. Several formalisms exist (e.g. MARTE [20] and Amalthea [1]) whose goal is to be as expressive as possible. Unfortunately they are not suitable for our purpose as they do not provide a formal semantics. In contrast, timed automata [3] provide a formal model which can be used to represent real-time systems at an arbitrary level of precision, and can thus express the operational semantics of any RT system model. This expressivity comes at a price: there is currently no generic way of specifying real-time systems in a timed automata based tool such as UPPAAL [17].

In this paper we propose *RTSpec*, a formalism for real-time system specification with flexible syntax and rigorous semantics. A modular library of UPPAAL models provides the operational semantics of RTSpec. Based on this library, the timing model of various analyzers can be formalized, and mappings between their respective input formats can be rigorously defined. Our overall target is a framework which comprises the RTSpec formalism, a tool chain for automatically translating RTSpec into the input of various analysis tools, and a set of benchmarks which are synthetic or derived from industrial case studies. Such a framework would provide a systematic, automated and rigorous methodology for evaluating analyzers.

The paper is structured as follows. Section 2 discusses related work. Section 3 sets the background of our work with a brief introduction to RT systems and timed automata. Section 4 overviews the automata library RTLib. Section 5 presents the syntax of RTSpec. Section 6 presents the methodology our framework provides for different types of users. Section 7 concludes the paper and discusses the future work.

2 Related Work

Our contribution relates to the research in three areas: specification languages for real-time systems, formal models for real-time system specification and comparison of real-time analyzers.

2.1 Specification Languages for Real-Time Systems

Let us first note that, in principle, the input format of any existing analyzer could be a candidate for the role of common input format. But these specification languages can only express, quite understandably, the system features that their analyzer can handle. For example, only a few tools such as pyCPA [12] propose an expressive activation model which specifies the maximum number of activations in a given time window. Using an input format which does not encompass such functions would be unfair towards the corresponding family of tools and analysis methods. The same goes for input formats which do not allow specifying offsets, or dependent tasks etc. Note that some simulation tools, e.g. ARTISST [11], provide much more expressive specification languages. In that case however, the semantics of the input format is not formally given and can only be clarified

through simulation. We therefore restrict ourselves for the moment to static analyzers. RTSpec could be a good starting point to providing a common input format for simulation tools as well.

There exist a few *high-level* specification languages aiming at generality. For example, MARTE [20] is a UML profile for embedded and real-time aspects of systems that has been defined with the aim of putting together all the concepts used in some existing framework or tool. This generality, however, is mainly meant at the level of vocabulary. No formal semantics is given for the different concepts in the vocabulary, and this is done on purpose, in order to leave room for semantic variations. Note that there exists a semantic framework but it would only allow to define a declarative semantics — defined by a set of constraints on timed event streams. Amalthea [1] is an open source framework for specifying real-time embedded systems maintained by an industrial consortium. It aims to be comprehensive with respect of the real-time features captured by its language. It additionally provides connections to simulation and analysis. Unfortunately, there is no explicit effort to formally define a semantics for the Amalthea language. Instead, the semantics is implicitly defined by the connections with these external tools, and hopefully in a non-contradictory manner.

Another related work is the on-going project Waruna[1], which aims to integrate tools at different development stages so as to automate the analysis of timing properties on design models of real-time systems. Seeing that architecture design models in AADL [13], SysML [28] or MARTE [21] may contain timing properties, Waruna intends to extract timing related information from design models and input them to analysis tools, e.g. Cheddar, MAST, RTaW-Pegase[2], etc. In contrast to RTSpec, the model transformations in Waruna are defined at the metamodel level and lacking of a formal semantics.

We view our RTSpec contribution as complementary to initiatives such as MARTE, Amalthea or Waruna. Indeed, our effort is less focused on having an exhaustive set of timing features. Instead, we provide a well-founded semantic background for those features that can currently be handled by at least one schedulability verification tool.

2.2 Formal Models for Real-Time System Specifications

We aim to provide a unified semantic framework for specifying real-time systems. Let us here review existing formalisms which could provide such a framework and show their limitations.

The UPPAAL [17] model checker can be used for the verification of real-time systems. TIMES [5] is a front-end for UPPAAL dedicated to schedulability analysis. TIMES however deals with a restricted set of concepts (uniprocessor systems with sporadic tasks). Another tool which uses UPPAAL is SchedMCore [8], a multiprocessor schedulability analyzer. But the task model supported by SchedMCore is restricted to periodic tasks.

[1] http://www.waruna-projet.fr/, https://www.polarsys.org/projects/polarsys.time4sys.

[2] http://www.realtimeatwork.com/software/rtaw-pegase/.

UPPAAL has also been used directly for the timing analysis of industrial case studies, e.g. [19,25]. The model proposed in [19] and extended in [25] allows describing uniprocessor systems of periodic tasks with a preemptive fixed-priority scheduler and shared memory. Synchronization protocols for shared memory access are implemented, including priority-inheritance and priority-ceiling.

Even more relevant to us are two UPPAAL-based modeling frameworks: [9] comprises 5 Timed Automata (TA) templates to specify sporadic tasks and partitioned schedulers on multiprocessor systems, and a sub-template for job enqueuing for each scheduling policy. [7] proposes a framework, which consists of templates for specifying sporadic tasks, schedulers and processing units, for hierarchical scheduling systems.

All the above-mentioned approaches are of rather limited expressivity. They do not support, for example, weakly-hard real-time systems. To fit our purpose, they would therefore need to be easily extendable. This is unfortunately not the case because they have not been designed with modularity and reusability in mind. For example, the Task template in all these frameworks captures not only the task activation and task execution pattern, but also its worst-case response-time computation and deadline-miss analysis. As a result, one cannot define independently variants of, e.g., the activation pattern and of the execution pattern. Instead, one would need to define a specific template for all possible combinations of variants of the aspects handled in Task.

In comparison, the primary focus of our formal library is on modularity. Our work builds on top of the TA-based representation of real-time systems — in particular tasks and schedulers — of [19]. Our representation of task activation patterns is inspired by task automata [14], which is a variant of TA for expressing task activation patterns.

2.3 Comparison of Analyzers

To our knowledge, the work presented in [22,23] is the only effort on systematic evaluation of schedulability analyzers. Four tools for performance analysis of distributed embedded systems are evaluated, namely MAST, SymTA/S, Real-Time Calculus [29] and UPPAAL. A formalism in SystemC is proposed for specifying the benchmarks which are then manually translated into an input for each analyzer under study. By observing the output of each tool for the chosen benchmarks, their underlying analysis algorithms are compared in terms of precision and efficiency.

This work sets a good starting point for further investigation in two directions. First, this evaluation of analyzers is restricted to their common functionality. In practice however, given a complex system, various analyzers allow to characterize the system with different abstractions, such that the result obtained using one tool depends both on its timing model and its underlying analysis algorithm. The effect of the composition of these two factors cannot be inferred from the conclusion of [22]. Secondly, a manual construction of the inputs for various tools is impractical for evaluating analyzers on more complex benchmarks than

the ones of [22,23]. We aim to provide a tool chain which automates the translation of a system specification into the input of several tools, so that a system can be specified once in RTSpec and analyzed with multiple tools.

3 Preliminaries

To set the background of our work, this section will first introduce the terminology on real-time systems with some typical examples. Then we will briefly present the timed automata formalism and the UPPAAL model checker.

3.1 Terminology on Real-Time Systems

A real-time system usually comprises three parts: a hardware platform which provides computation and communication resources; a set of software tasks which require access to the resources; a set of schedulers which manage the allocation of resources to tasks.

Platform. The hardware resources of an RT system include processing units, i.e. processors and cores, and possibly shared resources e.g. memory and communication network. For example, a distributed RT system consists of a number of nodes, where each node contains a number of processors. For simplifying schedulability analysis, the communication network which connects the distributed nodes can also be regarded as a node, and messages transmitted over the network can be considered as tasks on this network node.

Task Set. The software in an RT system is usually a task set, where each task is a piece of code. An execution of a task is also called a *job*. One task may have a number of jobs running in parallel.

A task is characterized by a set of parameters on its timing features, including its *arrival pattern* i.e. the time of the task's first arrival and recurrence, its *resource requirement* e.g. the CPU time it needs for execution, and its *timing requirement* e.g. a relative deadline. In their seminal paper [18], Liu and Layland proposed to characterize a task with two parameters: *period* for both the arrival interval and the deadline, and *WCET* for (an upper bound on) the worst-case execution time. This abstraction is called the *periodic task model*. A period task set consists of periodic, synchronous and independent tasks with implicit deadlines. To describe tasks with more diverse features, researchers have extended the Liu and Layland task model with different additional parameters, which lead to variations in the tractability and precision of feasibility and schedulability analysis [27].

Scheduler Set. Scheduling policies can be static (also called offline) or dynamic (also called online). The former group provides a scheduling table for all tasks before the system's execution, hence only applies to periodic task sets. The latter group is applicable to any task sets. Our work focuses on the more challenging

class of dynamic schedulers. Relevant properties of dynamic scheduling policies include preemptiveness (a scheduler can interrupt the execution of a job to allocate the resource to another job), job migration, etc. Additionally a scheduler may take decisions according to fixed priorities assigned to tasks, as for e.g. the Deadline Monotonic (DM) policy, or to dynamic priorities (priorities are assigned at the job level), e.g. Earliest Deadline First (EDF), or without any notion of priority, e.g. First-In-First-Out (FIFO).

Timing Model. A set of parameters which characterize the timing properties of a platform, a task set and a scheduler set of an RT system compose a timing model. A system can be captured by diverse timing models. With a more expressive timing model, the schedulability analysis is more precise at the cost of a higher computational complexity.

Many timing models have been used for specifying and analyzing RT systems. But they lack a common formal background. Even more problematic, different schedulability analysis methods and tools may interpret one parameter differently, which brings difficulty for their users to understand and compare them. In order to clarify the existing terminology and to facilitate rigorous definition of diverse extensions on it, we propose to formally define the semantics of timing models using timed automata.

3.2 Timed Automata

Timed automata [6] has been widely adopted as the language for formal representation and analysis of RT systems, because it allows to specify RT systems at an arbitrary level of precision, meaning that it can express the semantics of any concept proposed by a timing model. Therefore, we chose timed automata as the language for building RTLib, which is the library of TA templates providing a formal semantics for RTSpec.

A timed automaton is a finite automaton consisting of a finite set of nodes denoted *locations* and a finite set of labeled edges denoted *transitions*, extended with real-valued variables [6]. Time progress and time-dependent behavior are expressed using a set of *clocks* that can be started, reset, halted and read. A location can be assigned with an invariant, which is a clock constraint. A transition can be labeled with a *guard* i.e. the condition for enabling the transition, a *channel* with which an automaton synchronizes with another automaton and moves simultaneously, and an *update* which may contain actions and reset of clocks. UPPAAL supports *stopwatch automata* [4], an extension of timed automata where clocks may be stopped occasionally. Syntactically, a stopwatch expression is an invariant in the form of $x' == c$, where x is a clock and c is an integer expression which evaluates to either 0 or 1 [19].

As an example, Fig. 1 shows a timed automaton representing a task execution process in a preemptive environment, assuming that a task has at most one live job at any moment. A clock *resTime* tracks the task's response time, i.e., the time span between its activation and finish. Another clock *exeTime* tracks its execution time. On the location *Scheduled*, the invariant $exeTime <= WCET$

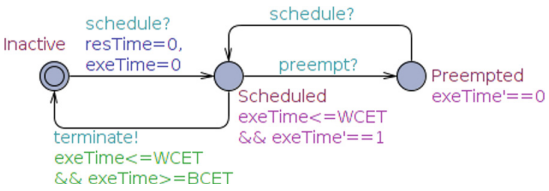

Fig. 1. An example of timed automata: task's execution process

ensures that the duration of the task's execution is at most $WCET$. The invariant $exeTime' == 1$ ensures that the clock $exeTime$ progresses when the task is executing on a processor. The invariant $exeTime' == 0$ on the location *Preempted* stops the clock $exeTime$ when the task is preempted. A job finishes execution when the CPU time it consumes is between BCET (best case execution time) and WCET. This automaton exhibits the operational semantics of the two task parameters BCET and WCET.

3.3 UPPAAL

UPPAAL is the standard tool for editing, executing and analyzing timed automata. UPPAAL supports the specification of timed automata as well as *automata templates*, i.e., parameterized timed automata. In a valid UPPAAL model, automata templates have to be instantiated into automata. As a result, UPPAAL models consist of three parts (the keywords given by UPPAAL are denoted in **bold** font):

- **Declarations.** Elements used by the model templates, which can be:
  - *Generic*: user-defined data types (e.g. bounded integers, structs, arrays); (parameterized) synchronization channels; global variables; functions used within the templates.
  - *Specific* to a system: global variables instantiated with actual parameters.
- **Templates.** Parameterized timed automata.
- **System declarations.** Elements which are specific to an instance system:
  - *Instantiation statements*: statements which instantiate templates into automata by assigning values to the template parameters.
  - **System**: declaration of a system as a set of automata.

Logically, such a UPPAAL input file is composed of two levels:

1. a *system type* made of the *generic* part in **Declarations** and the **Templates**;
2. an *instantiation* into a specific system which consists of the *specific* part in **Declarations** and the **System declarations**.

Since UPPAAL was designed for formalizing specific systems rather than system types, it does not provide sufficient support for defining system types on a higher abstraction level. For example, data type definitions, which are logically a part of *system type*, may depend on instance systems, as we explain now.

UPPAAL provides two basic data types: bool and (bounded) integer — to reduce the state space of a system, all integer variables must be bounded. In particular, this means that all user-defined integer data types must be bounded. As an example, defining a data type for WCET of tasks requires to provide a bound for them, which can only be done for specific system instances. In other words, the system type definition and instance definition are interleaved and mutually dependent, which brings difficulty for users to understand and extend a system type. Note that the limitations of UPPAAL with respect to system type definitions have pushed us to develop a front-end tool called *RTLibEx* to help with the formalization of timing models.

4 The RTLib Library

In this section, we introduce RTLib, a library of UPPAAL templates formalizing concepts which are commonly utilized in real-time system analysis. RTLib provides the basis for a formal semantics of RTSpec, our formalism for specifying real-time systems. The key advantages of RTLib are: (1) its formal basis, (2) its expressivity, which can be easily increased if needed; (3) its modularity, which makes it much easier to compare different models by allowing the user to focus on the concepts that differ. The RTLib library is structured around a core of *basic* concepts that exist in most frameworks. Thanks to its modular structure, one can easily enrich RTLib with *extensions*, i.e., variants of one or more templates of the basic library. Two extensions that can be meaningfully combined at the conceptual level can be combined directly at the library level.

RTLib can be used for specifying concrete real-time systems on which the UPPAAL model checker can conduct exact or statistical schedulability analysis. This can help evaluating the correctness and accuracy of other analyzers on small systems. The main objective of RTLib is however at a higher abstraction level: RTLib is meant to provide a common, formal basis to describe the semantics of the timing models used by analyzers. This will help proposing rigorous transformations between the timing models.

RTLib defines a set of *system types* for real-time systems, i.e. timing models. As discussed in Sect. 3, a *system type* in UPPAAL consists of several parts, among which the most complicated are data types and automata templates. This section overviews these two parts in RTLib before describing how RTLib can be extended.

4.1 Automata Templates

As discussed in Sect. 3, a timing model defines a set of parameters which characterizes the three parts of a real-time system: platform, task set and scheduler. Following this compositional view, we organize RTLib as a hierarchy of templates. The UML class diagram in Fig. 2 shows the structure of the RTLib Basic library, where each concrete class denotes an automata template, and an abstract class denotes a concept which is implemented by a number of automata

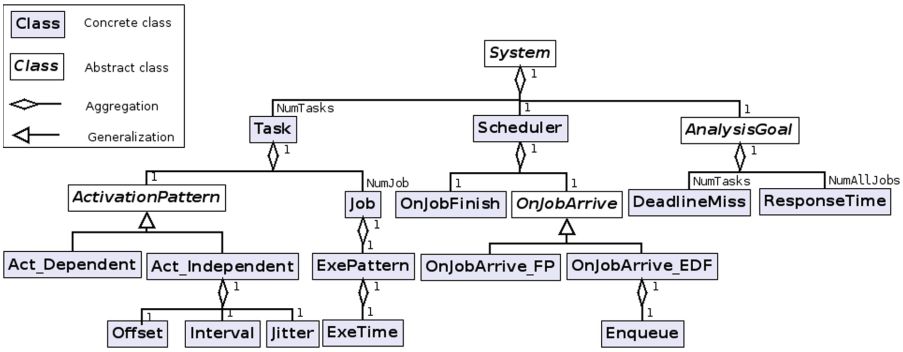

Fig. 2. Structure of the library RTLib Basic

templates. Note that RTLib explicitly specifies analysis goals, but does not have templates related to the platform, as we explain now.

For the moment RTLib only captures multiprocessor/distributed systems. The platform information, i.e. the number of processors in each node, is thus represented by a parameter of schedulers. We leave to future work the extension of RTLib with templates for more complicated platforms. Besides, we have chosen to explicitly specify analysis goals in RTLib, e.g. absence of deadline misses for hard real-time systems, to cover more system-level timing requirements than just schedulability.

In RTLib, an automata template may call its *sub-templates* through synchronization channels. In Fig. 2, an aggregation relation connects a template with its sub-template, and a generalization relation connects a concept with its special case[3]. For instance, the Task template has two sub-templates: ActivationPattern which characterizes the activation pattern of a task, and Job which represents the lifecycle of a task's instance. ActivationPattern has two special cases: Act_Dependent and Act_Independent, for dependent tasks and independent tasks, respectively. The activation pattern of an independent task may have three constraints: Offset, Interval and Jitter, each denoted by a parameter of a task model. The operational semantics of the parameters are represented by separate automata templates.

For example, Interval constrains the separation between job arrivals. Once an independent task τ releases a job τ^i, the automaton Task immediately calls its sub-automaton Act_Independent, which in turns calls its sub-automaton Interval through the channel *Independent_call_Interval*. As shown in Fig. 3, a transition is triggered from the location *Start* to *WaitInterval*, and the clock x is reset

[3] We use the following conventions. Template A_B is a specialization of template A for extension B. Synchronization channels are named as follows:

- A_e_B: automaton A sends a message to B on event e;
- A_call_B: automaton A calls its sub-automaton B;
- A_return_B: automaton A, a sub-automaton of B, returns.

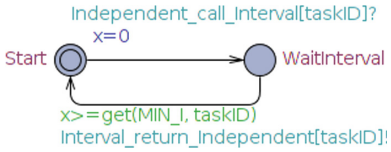

Fig. 3. An example of automata template in RTLib: interval

to zero for recording the time passed since the release of τ^i. At an arbitrary moment after x reaches *MIN_I* (i.e. the minimum interval), Interval returns back to Act_Independent through the channel *Interval_return_Independent*, meaning that the arrival constraint of job τ^{i+1} has been satisfied.

4.2 Data Types

User-defined data types determine the data structure processed by automata. The structure of data type definition in RTLib resembles the structure of the templates in the library.

We classify the data types in RTLib into two groups according to their roles: *parameter types* whose variables represent the parameters of RT systems, hence determined by a timing model; *state types* whose variables represent the dynamic state of RT systems during their execution, hence common to any timing model. Therefore, to formalize a new timing model based on RTLib Basic, *state types* can be reused, while *parameter types* need to be extended.

Parameter Types. According to their structural relations, the parameter types can be further classified onto three levels: elementary types, composite types and collective types, as shown in Fig. 4.

- Elementary type: a bounded integer, representing the data type of some parameters, e.g. *time_t* as the data type of tasks' timing parameters including period, deadline, WCET, etc.
- Composite type: a struct built upon elementary types, e.g. *task_t* which consists of a set of task parameters.
- Collective type: an array whose elements belong to some composite type, e.g. *taskSet_t* as an array of *task_t* representing the taskset in a system.

Each task model which characterizes a task with a set of parameters is mapped to a definition of *task_t*, and any task characterized by the task model can be expressed as a variable of the type *task_t*. Similarly, elementary types, composite types and collective types are defined to represent scheduler parameters, schedulers and scheduler sets, respectively. Such data types form a hierarchy following the compositional view of a real-time system. The leaves of this hierarchy represent the data types of the parameters defined by a timing model, hence may vary from one timing model to another. The non-leaf part of the hierarchy are reusable for various timing models. This stable hierarchy of data types,

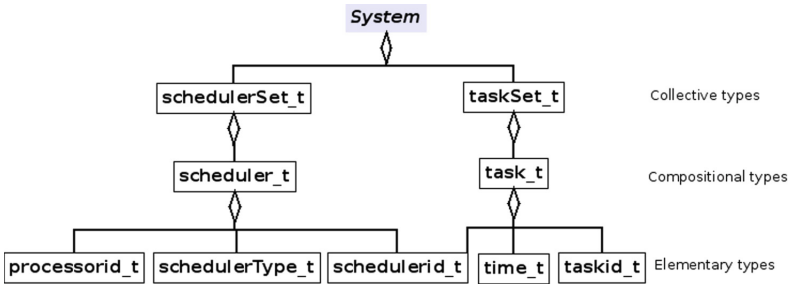

Fig. 4. Hierarchy of parameter data types

together with the stable hierarchy of automata templates, reveals the reusability and extensibility of RTLib. By extending the data types and the automata templates, RTLib can be adapted to formalize a wide range of timing models.

State Types. A RT system formalized as an automata network simulates the timing-related behavior of the system. To track the dynamic state of a system during its execution, the following data types are defined for all timing models:

- For representing the state of tasks:
  - *job_t*: a composite data type with two fields *taskID* and *jobID*, altogether denoting the identifier of a job;
  - *jobQ_t*: an array of *job_t*, denoting the ready job queue.
- For representing the state of a node under the management of a scheduler:
  - *nodeState_t*: a composite data type, whose fields denote the current jobs upon each processor, new arrived job, just finished job, etc.;
  - *nodeStateSet_t*: an array of *nodeState_t*, denoting all nodes in a system.

4.3 Extension of RTLib

Let us now show the extension capabilities of RTLib. By extension, we understand either the introduction of an entirely new concept or a new variant of an existing one. To define an extension, one proceeds in two steps:

1. possibly extend the data type definitions to express the new concepts;
2. add or replace the relevant templates.

To argue that the chosen structure of RTLib achieves sufficient modularity, we illustrate this process with an example.

The *event stream model* [2] is a generalization of the sporadic task model which constrains not only the minimal time distance between any two consecutive activations of task τ, but may for any k impose a stronger constraint for the minimal time interval that may contain k activations. In practice, it is sufficient to consider a strictly increasing constraint sequence only for some first N values for k. Thus, such a *minimum distance* function can be specified by a

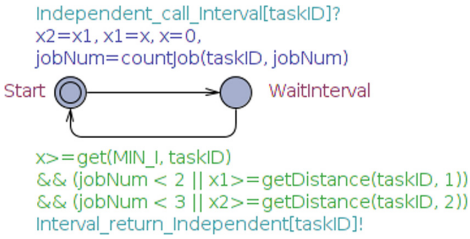

Fig. 5. Interval_MinDistance for 3 distances

vector $[D_0, D_1, \ldots, D_{N-1}]$ of minimal distances D_i between the activation of job τ^k and τ^{k+i+1}, $\forall k$. Note that the minimum interval of the sporadic task model is a minimum distance vector with a single element D_0.

Extending the basic activation pattern of the sporadic task model to handle minimum distance functions only requires to modify the Interval template (see Fig. 3). To represent a vector of N minimum distances requires N clocks. Figure 5 shows the template for $N = 3$. It extends the Interval template with two additional clocks: $x1$ records the distance between τ^k and τ^{k+2}, and $x2$ records the distance between τ^k and τ^{k+3}. Function *countJob*() counts the job arrivals up to $N - 1$.

4.4 RTLibEx: A GUI Tool for RTLib Extensions

As already discussed, RTLib is intended both for:

1. defining timing models i.e. system types of RT systems;
2. specifying actual systems i.e. instances of some system types.

It is however not possible to distinguish properly these two activities, which may concern different users, in UPPAAL, as the system type definitions and instance definitions are interleaved and mutually dependent (see Sect. 3).

RTLibEx is a front-end tool for UPPAAL intended to address this issue. It provides two distinct editors (implemented in the same GUI): a *timing model editor* and an *instance editor*. The timing model editor mainly supports the extension and specialization of data types. The instance editor is then automatically generated by our tool from the data type definitions, which allows users to define an actual system by just filling an array of parameters. In the rest of this section, we briefly describe these two editors.

The Instance Editor. As shown in Fig. 6, an instance editor is a GUI for defining an actual system. RTLib Basic defines an RT system as a set of schedulers and tasks whose parameters are defined by the data types *scheduler_t*, resp. *task_t*. Thus, defining a scheduler or a task means providing values for its parameters. The generated instance editor therefore provides two tables for schedulers and tasks, where the system designer inputs the actual parameters

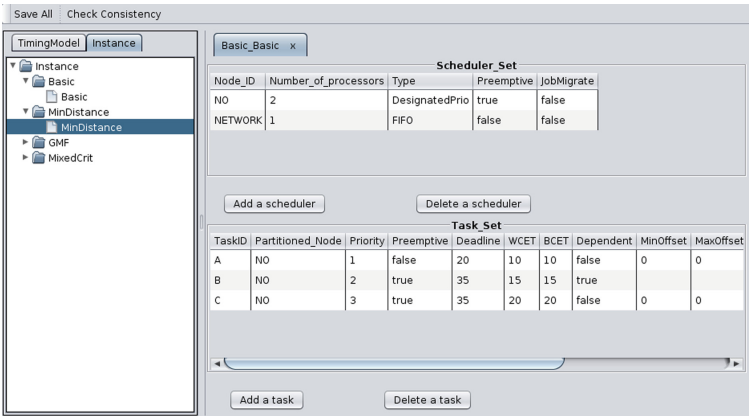

Fig. 6. Snapshot of RTLibEx: instance editor

of a real-time system. The tables look similar to the GUI of existing analyzers, e.g. TIMES [5] and Cheddar [26]. The significant difference is that RTLibEx automatically generates the GUI from the relevant user-defined data types for any user-defined timing model, while the existing timing analysis tools hard-wire specific timing models hence cannot be extended by the user.

Once the system actual parameters are given, RTLibEx checks its consistency and reports diagnostic information. For a consistent definition, it generates the representation of the system as an input file for UPPAAL.

The Timing Model Editor. As stated in Sect. 3.3, a UPPAAL file contains three parts: Declarations, Templates and System declarations. In most cases, to define a new timing model implies extending the data types of RTLib Basic and adding or replacing existing templates. RTLibEx facilitates the extension of the Declarations part and then automatically generates the System declarations part. In parallel, the user should modify Templates with UPPAAL.

Declarations include two subsets: the *generic* subset for a system type, and the *specific* subset for a specific system. The instance editor of RTLibEx allows users to input the parameters of a system, which becomes the *specific* declarations of the system. The timing model editor of RTLibEx provides a GUI for users to specify the *generic* subset of declarations for defining a system type.

The *generic* declarations consist of data type definitions, synchronization channels, global variables and functions used within the templates. Among them, data type definitions is the major part to be modified when defining a new timing model based on RTLib Basic. As we stated in Sect. 4.2, we classify data types into *parameter types* and *state types*.

RTLibEx enables users to reuse RTLib Basic as much as possible when defining a new timing model. On the left side of Fig. 7 is a tree of loaded Timing Models, where each one is a tree with five main nodes. The first four nodes, *parameter types*, *state types*, *functions* and *channels* represent the respective parts

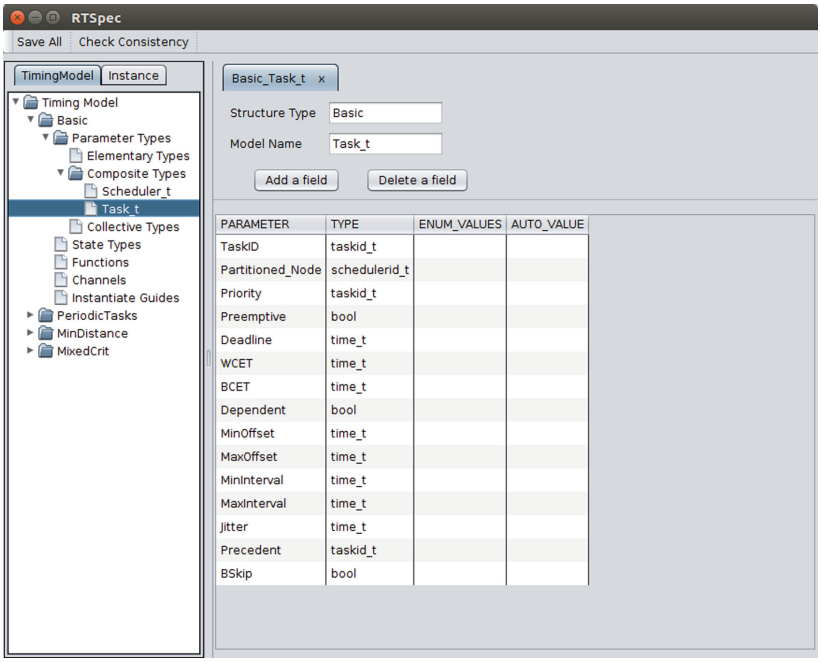

Fig. 7. Snapshot of RTLibEx: timing model editor

of a UPPAAL system type. The last one, *instantiation guides*, specifies a set of rules generating instantiation statements. Given the instantiation statements, UPPAAL can instantiate a set of automata templates into a system instance.

The right side of Fig. 7 is a table of parameters of a composite type, i.e. *task_t* and *scheduler_t* presented in Sect. 4.2. To extend the parameter list of Task or Scheduler, the user adds items into the respective table. Each table comprises 4 columns:

- PARAMETER: name of a parameter.
- TYPE: data type of the parameter.
- ENUM_VALUE: enumerated values of the data type of the parameter. This field is useful only when the parameter has a enumeration data type.
- AUTO_VALUE: default value of the parameter.

To build a new timing model, the user first builds a copy of the Basic library by right-clicking on the node Basic, and then fills in the sub-nodes of the new timing model. When all the nodes of a timing model tree are completed, the timing model editor stores this model and generates the tables *Scheduler_Set* and *Task_Set* in the instance editor. Thus, RTLibEx enables a logical and procedural separation of timing model definition from instance system declaration.

5 The RTSpec Format

Based on the formal library RTLib, we define RTSpec, a human-readable textual format for specifying a wide scope of real-time systems. Following the structure of RTLib, a real-time system specification in RTSpec declares a platform, a task set and a scheduler set. An entity, e.g. a processor or a task or a scheduler, is characterized by a set of attributes, where each attribute has its operational semantics defined in RTLib.

As an intermediate textual format, RTSpec aims to cover not only the features shared by all timing models underlying existing analyzers, but also attributes which are characteristic of some existing analyzers, such that comparison between different tools is fair. For that reason, and following the two-layer structure of RTLib, RTSpec provides language constructs at two levels: the *basic* elements represent the concepts defined in RTLib Basic, and the *extension* elements characterize concepts which may be handled only by some analyzers. A simple system specification that conforms to the basic timing model can be translated into the input of different analyzers through syntactical mapping, which enables to compare their common subset. A more complex system specification that is beyond the basic model can be translated into the input of different analyzers through semantic mapping, i.e., abstraction and approximation.

5.1 The RTSpec Basic Syntax

We have designed the RTSpec basic syntax with three concerns in mind: conciseness, flexibility and extensibility, as we briefly discuss now.

Conciseness. In RTSpec, default values for parameters allow systems to be specified in a concise manner. For example, if a task has no jitter, instead of declaring `jitter = 0`, the jitter attribute can simply be omitted. As a result, even in presence of language extensions, it is still possible to define a simple entity, e.g. a classic periodic task, using a small set of attributes.

Flexibility. As a unified specification format, RTSpec represents a synthesis not only of the timing models of various analyzers, but also of the style of their input formats. The concrete syntax of RTSpec provides 3 representation styles:

- Positional style: An entity is characterized by a set of parameters, where the position of each parameter indicates its meaning. This style is similar to the input format of the SchedMCore [8] tool. This style can be used to specify simple systems which only need this limited set of attributes.
- Canonical style: An entity is characterized by a set of attributes, where each attribute is declared as a name-value pair and the attributes of an entity can be declared with an arbitrary order.
- Record style: An entity is characterized as a record, where each field represents an attribute such that attributes of an entity can be declared in any order. This style is similar to the input format of pyCPA [12]. It allows interleaving task declarations and referring to attributes declared before.

All task declarations are automatically translated into the canonical style before further processing. The flexibility of the syntax style enables a user to choose a preferred representation according to her/his habits or the complexity of the system to be analyzed. The flexibility in the order of attribute declarations facilitates extensions and modifications of attribute lists when incorporating new timing models. The following listing shows an example of a task set declared using the three representation styles.

```
-- Positional style:
-- Task(name,period, wcet, deadline, offset):
Task("task1", 20, 3, 15, 2)

-- Canonical style:
Task{name="task2", period=23, wcet=4, deadline=8}

-- Record style:
Task("task3")
task3.wcet = 5
task3.period = 23
task3.deadline = 13
task3.offset = 5
Task("task4")
task4.wcet = 9
task4.period = task3.period
task4.deadline = 2 * task3.deadline
task4.offset = 7
```

Extensibility. We are currently extending RTSpec with constructs which are characteristic of some analyzers, e.g. minimum distance functions for pyCPA. It is our intention to have specific keywords identifying extensions in a specification. Such constructs must correspond to extensions of the RTLib UPPAAL library and whenever possible semantic mappings must be provided to transform an extension-dependent specification into a coarser grained basic specification.

5.2 Current Status of the RTSpec-based Tool Chain

So far, we have: (1) defined the basic part of RTSpec based on a subset of the RTLib basic library, (2) implemented a translator which takes any RTSpec input and translates it into an equivalent canonical form, (3) implemented translators which transform an RTSpec (canonical) specification into a number of formats, including the CPAL Language[4], MAST PTE file[5], pyCPA[6], SchedMCore[7] and

[4] http://www.designcps.com/.
[5] http://mast.unican.es/.
[6] http://pycpa.readthedocs.io/en/latest/.
[7] https://forge.onera.fr/projects/schedmcore.

Times[8]. Note that these tools do not provide a formal semantics for their input format. As a consequence it is not possible to prove the correctness of our translations in any formal way. Instead we rely on a tight collaboration with the researchers involved in the development of the targeted analyzers. Another way to look at this issue is then to consider RTSpec and RTLib as the reference semantics for these input formats.

6 Methodology

Our framework is targeted at developers but also users of timing analysis tools. This section describes the methodology that our framework supports for these two categories of people.

6.1 For System Architects

With a variety of available tools, it can be difficult for the architect of a real-time system to select the best fit to analyze his/her particular system. Our framework enables her/him to compare tools by taking the following steps:

1. System specification: specify the system in RTSpec and apply automatic translations to generate an input for some existing tools.
2. Comparative experiment: analyze the system with several tools and compare their analysis results.

6.2 For Developers of Analyzers

With the advance of research on schedulability analysis, researchers propose new methods and implement new tools. Such new methods often incorporate new concepts for characterizing real-time systems more faithfully than existing models, so as to obtain more precise analysis results. The incompatibility and incomparability between the timing models underlying all these tools lead to difficulties for the developer of a new analyzer to argue about the advantages of their new tool over existing analyzers.

Our framework enables tool developers to formally relate their timing models to existing ones and to conduct comparative experiments. More specifically, let T denote a new tool, and M_T the timing model underlying tool T. The workflow for the developer of T is as follows:

1. Semantics definition: Formalize M_T by extending RTLib, i.e. providing an operational semantics for the new concepts in M_T by refining or modifying the related concepts in RTLib. The extension of automata templates can be conducted with the UPPAAL model checker. Our tool RTLibEx helps to extend the data types in RTLib.

[8] http://www.timestool.com/.

2. Syntax definition: Extend the syntax of RTSpec so as to represent the new concepts in M_T. In principle this could be automated in RTLibEx but the process is manual at the moment.
3. Translator development: Develop a translator between RTSpec and the input format of T. Additionally, whenever possible a semantic mapping of RTSpec extensions for T to basic elements must be provided to enable comparison with tools which cannot handle such features.
4. Benchmark specification: Design some benchmarks in RTSpec, or in the input format of T if a translator from T to RTSpec is implemented.
5. Comparative experiment: Conduct an analysis of the benchmarks with various tools, and compare their analysis results.

7 Conclusion

This paper presents our ongoing work on a framework for evaluating schedulability analysis tools. We propose RTSpec as a common format for specifying real-time systems. The formal semantics of RTSpec is represented by a modular and extensible UPPAAL model library, which covers a wide range of terminology in schedulability analysis. We are developing a tool chain which translates RTSpec files into an input for diverse tools, such that a system only has to be specified once to be analyzed by several tools.

Compared to more general formalisms such as Amalthea [1] or MARTE [20], the expressiveness of RTSpec is constrained to only support features that are useful for rigorous timing analysis, i.e. supported by static analyzers. This principle allows us to formalize all concepts used in RTSpec, and hence enables rigorous mappings between various timing models.

For evaluating various timing analysis tools, researchers have developed tool-neutral languages and implemented their mappings to different tools, e.g. in [22] and Waruna. However, this approach has two defects: (1) it cannot ensure the correctness of the mappings; (2) a special feature which can be expressed by one tool may have no direct mapping in other tools. With our approach, the semantics of the terminology is defined independent from the translations between tools. The semantics of the formalism of each tool helps users to understand the concepts and the assumptions underlying the tool. With the explicit semantics, different but related concepts can be mapped to each other through approximation/abstraction.

RTSpec serves as an extensible framework for connecting different formalisms. Currently, the expressivity of RTSpec Basic is the common subset of the existing formalisms. Next, we will extend RTSpec, in collaboration with the tool developers, to incorporate the special features of their formalisms. The aim is to incorporate the important features of existing tools, so that the tools can be evaluated with a comprehensive set of benchmarks. We are still developing the benchmarks, and will present them later.

Note that in this paper the primary usage of UPPAAL models is as the operational semantics of the terminology concerning real-time systems. Given

specific RT systems, the UPPAAL model checker can be used for exact timing analysis, if the complexity of the model does not exceed the capability of the model checker.

Ongoing and Future Work. Let us summarize here the topics on which we are currently working or plan to work in the near future.

- *Extension of RTLib.* To encompass the key elements of the pyCPA and MAST timing models, RTLib is currently being extended in two directions: the platform model, to take shared resources into account, and the task model, to handle more complex dependencies between tasks.
- *Semantic mappings.* We want to investigate mappings from timing model extensions to the basic timing model. In particular we need to rigorously define mappings between timing models based on their formal semantics, and prove their correctness in the sense that schedulability analysis on the approximated system specification is possibly pessimistic but still correct.
- *Extension of RTSpec and its associated tool chain.* The current RTSpec is based on the basic timing model defined by RTLib Basic, which is the common subset of the timing models of various analyzers. Further, we will extend RTSpec with typical special features supported by some of the existing analyzers and extend the corresponding translators.
- *Comparative experiments and benchmarks.* Finally, we will develop a set of benchmarks derived from synthetic and industrial cases, and conduct a comparative evaluation of existing analyzers. The analysis results thus obtained with these benchmarks will show the relative strengths of the analyzers on different types of systems, and help users to choose tools according to the features of their systems.

Acknowledgment. The authors would like to thank Claire Pagetti and Eric Noulard from Onera, Olivier Cros from ECE Paris, and Jean-François Monin from Verimag for the fruitful and inspiring discussions.

References

1. The Amalthea project. http://www.amalthea-project.org/
2. Albers, K., Slomka, F.: An event stream driven approximation for the analysis of real-time systems. In: Proceedings of the 16th Euromicro Conference on Real-Time Systems, ECRTS 2004, pp. 187–195. IEEE (2004)
3. Alur, R.: Timed automata. In: Halbwachs, N., Peled, D. (eds.) CAV 1999. LNCS, vol. 1633, pp. 8–22. Springer, Heidelberg (1999). doi:10.1007/3-540-48683-6_3
4. Amnell, T., et al.: UPPAAL - now, next, and future. In: Cassez, F., Jard, C., Rozoy, B., Ryan, M.D. (eds.) MOVEP 2000. LNCS, vol. 2067, pp. 99–124. Springer, Heidelberg (2001). doi:10.1007/3-540-45510-8_4
5. Amnell, T., Fersman, E., Mokrushin, L., Pettersson, P., Yi, W.: TIMES: a tool for schedulability analysis and code generation of real-time systems. In: Larsen, K.G., Niebert, P. (eds.) FORMATS 2003. LNCS, vol. 2791, pp. 60–72. Springer, Heidelberg (2004). doi:10.1007/978-3-540-40903-8_6

6. Bengtsson, J., Yi, W.: Timed automata: semantics, algorithms and tools. In: Desel, J., Reisig, W., Rozenberg, G. (eds.) ACPN 2003. LNCS, vol. 3098, pp. 87–124. Springer, Heidelberg (2004). doi:10.1007/978-3-540-27755-2_3
7. Boudjadar, A., David, A., Kim, J.H., Larsen, K.G., Mikučionis, M., Nyman, U., Skou, A.: Statistical and exact schedulability analysis of hierarchical scheduling systems. Sci. Comput. Program. **127**, 103–130 (2016)
8. Cordovilla, M., Boniol, F., Forget, J., Noulard, E., Pagetti, C.: Developing critical embedded systems on multicore architectures: the Prelude-SchedMCore toolset. In: 19th International Conference on Real-Time and Network Systems (2011)
9. David, A., Illum, J., Larsen, K.G., Skou, A.: Model-based framework for schedulability analysis using UPPAAL 4.1. Model-Based Des. Embed. Syst. **1**(1), 93–119 (2009)
10. Davis, R.I., Burns, A.: A survey of hard real-time scheduling for multiprocessor systems. ACM Comput. Surv. (CSUR) **43**(4), 35 (2011)
11. Decotigny, D., Puaut, I.: Artisst: an extensible and modular simulation tool for real-time systems. In: Proceedings of the Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, (ISORC 2002), pp. 365–372. IEEE (2002)
12. Diemer, J., Axer, P., Ernst, R.: Compositional performance analysis in python with pyCPA. In: Proceedings of WATERS (2012)
13. Feiler, H., Lewis, B., Vestal, S.: The SAE architecture analysis and design language (AADL) standard. In: IEEE RTAS Workshop (2003)
14. Fersman, E., Krcal, P., Pettersson, P., Yi, W.: Task automata: schedulability, decidability and undecidability. Inf. Comput. **205**(8), 1149–1172 (2007)
15. Harbour, M.G., García, J.G., Gutiérrez, J.P., Moyano, J.D.: MAST: Modeling and analysis suite for real time applications. In: 13th Euromicro Conference on Real-Time Systems, pp. 125–134. IEEE (2001)
16. Henia, R., Hamann, A., Jersak, M., Racu, R., Richter, K., Ernst, R.: System level performance analysis - the SymTA/S approach. IEE Proc.-Comput. Digit. Tech. **152**(2), 148–166 (2005)
17. Larsen, K.G., Pettersson, P., Yi, W.: UPPAAL in a nutshell. Int. J. Softw. Tools Technol. Transf. (STTT) **1**(1), 134–152 (1997)
18. Liu, C.L., Layland, J.W.: Scheduling algorithms for multiprogramming in a hard-real-time environment. J. ACM (JACM) **20**(1), 46–61 (1973)
19. Mikučionis, M., Larsen, K.G., Rasmussen, J.I., Nielsen, B., Skou, A., Palm, S.U., Pedersen, J.S., Hougaard, P.: Schedulability analysis using UPPAAL: Herschel-Planck case study. In: Margaria, T., Steffen, B. (eds.) ISoLA 2010. LNCS, vol. 6416, pp. 175–190. Springer, Heidelberg (2010). doi:10.1007/978-3-642-16561-0_21
20. OMG: UML profile for MARTE: Modeling and analysis of real-time embedded systems (2011). http://www.omg.org/spec/MARTE/1.1/PDF/
21. OMG: Modeling and analysis of real-time and embedded systems. Object Management Group (2008)
22. Perathoner, S., Wandeler, E., Thiele, L.: Evaluation and comparison of performance analysis methods for distributed embedded systems. Master's thesis, Swiss Federal Institute of Technology, Zürich (2006)
23. Perathoner, S., Wandeler, E., Thiele, L., Hamann, A., Schliecker, S., Henia, R., Racu, R., Ernst, R., González Harbour, M.: Influence of different abstractions on the performance analysis of distributed hard real-time systems. Des. Autom. Embed. Syst. **13**(1), 27–49 (2009)

24. Sha, L., Abdelzaher, T., Årzén, K.E., Cervin, A., Baker, T., Burns, A., Buttazzo, G., Caccamo, M., Lehoczky, J., Mok, A.K.: Real time scheduling theory: a historical perspective. Real-Time Syst. **28**(2–3), 101–155 (2004)
25. Shan, L., Wang, Y., Fu, N., Zhou, X., Zhao, L., Wan, L., Qiao, L., Chen, J.: Formal verification of lunar rover control software using UPPAAL. In: Jones, C., Pihlajasaari, P., Sun, J. (eds.) FM 2014. LNCS, vol. 8442, pp. 718–732. Springer, Cham (2014). doi:10.1007/978-3-319-06410-9_48
26. Singhoff, F., Legrand, J., Nana, L., Marcé, L.: Cheddar: a flexible real time scheduling framework. In: ACM SIGAda Ada Letters, vol. 24, pp. 1–8. ACM (2004)
27. Stigge, M., Yi, W.: Graph-based models for real-time workload: a survey. Real-Time Syst. **51**(5), 602–636 (2015)
28. Team, SysML Merge: Systems modeling language (SysML) specification. OMG document: ad/2006-03-01 (2006)
29. Thiele, L., Chakraborty, S., Naedele, M.: Real-time calculus for scheduling hard real-time systems. In: Proceedings of the 2000 IEEE International Symposium on Circuits and Systems, ISCAS 2000 Geneva, vol. 4, pp. 101–104. IEEE (2000)