Architecture Models Refinement for Fine Grain Timing Analysis of Embedded Systems

Etienne Borde*, Smail Rahmoun*, Fabien Cadoret*, Laurent Pautet*, Frank Singhoff†, Pierre Dissaux‡
*Institut Telecom;TELECOM ParisTech; LTCI - UMR 5141 - 46 rue Barrault, 75013 Paris, France
firstname.lastname@telecom-paristech.fr
†Lab-STICC UMR 6285, UBO, UEB - 20, Avenue Le Gorgeu CS 93837, 29238 Brest Cedex 3, France
singhoff@univ-brest.fr
‡Ellidiss Technologies - 24, Quai de la Douane, 29200 Brest, France
pierre.dissaux@ellidiss.com

***Abstract*—As real-time systems have become more and more complex, architects rely on abstract models of computation in order to design and analyse these systems. In order to ease the production of source code that respects such models of computation, developper can take advantage of code generators and/or middleware. However, when analyzing an abstract model of computation, timing overheads due to generated code or middleware components are not taken into account. Answering this issue is even more problematic in the domain of embedded systems because of the variability of execution platforms. To tackle this problem, we present in this paper a model refinement and timing analysis framework: abstract models of computation are first transformed in more precise models, which include the timing characteristics of the execution platform. These refined models are then used for a more precise timing analysis. The experiment results we present in this paper show that our method can deal with realistic software architecture of real-time systems.**

I. Introduction

As software architectures of real-time embedded systems have become more and more complex, their designers rely on domain specific languages that capture key properties of a system under design. For systems made of tasks with strict timing requirements, key properties capture scheduling algorithm of tasks, and communications patterns among these tasks. Implementation details of these communication patterns are usually abstracted away in order to focus first on systems validation from a functional point of view. Timing analysis at this stage enable to allocate time budgets for tasks, and to prune task sets that cannot satisfy timing requirements. However, this type of architecture description is not sufficient to assess the schedulability of the system since it does not consider the overhead induced by the implementation of interactions among tasks.

In some cases, this overhead can be reduced to a simple re-evaluation of tasks execution time. But in modern architectures of real-time embedded systems, for instance when time and space partitioning has to be enforced, overhead of communications can difficultly be reduced to a simple addition of execution times. Indeed, service calls of an ARINC653-compliant operating system often require to execute non-interruptible sections of code which are both costly in terms of execution time and in terms of blocking time to consider when proceeding to schedulability analysis.

As a consequence, schedulability analysis performed at design level rely on models with approximated (often pessimistic) blocking times since they must be estimated by designers in order to fit the hypothesis of analysis techniques such as [1]. To improve the precision of such analysis, we propose to rely on a refinement technique that automatically transforms a design model into an implementation model that includes the timing characteristics of the underlying platform services (including accesses to protected shared data). In this paper, we consider additive overheads, which means that adding overheads always lead to longer worst case response time of threads. Pragmatically, this means that sources of scheduling anomalies, such as caches management or multi-core architectures, are avoided. By incorporating such overheads in a schedulability analysis, we can thus assess precisely the final overhead and ensure the schedulability of the architecture.

We illustrate this approach on a case study inspired from the railway domain.

The presentation of our contribution is organized as follows: section II details the problems we adress in this paper. In section III, we give an overview of the refinement based approach we propose. Section IV gives a short presentation of AADL, the modeling language we use for our contribution. The different steps of the approach we propose are then detailed in sections V and VI. In section VII, we provide experimental results to evaluate the usability of our approach. Finally, section VIII compares our approach with the state of the art, and section IX concludes our paper.

II. Problem

In order to abstract away the complexity of real-time embedded systems architectures, several modelling languages have been proposed (*e.g.* [2], [3], [4]). These well-known languages are used to design, validate, and/or implement real-time systems. As a consequence, they identify software components with a model of computation and communication, usually introducing a notion of time [5]. Timing characteristics of components allow to deal with an important requirement of real-time systems: each task must finish its execution within a predefined deadline. Schedulability analysis techniques have been proposed to ensure that every task of a system satisfies this timing requirement (*e.g.* [6], [7]).

In addition, models of computation and communications abstract the implementation of interactions between components thanks to predefined activation and communication patterns. This allows to validate the functions of a system based on abstract models, independently of the concrete implementation of communication and activation patterns.

Finally, models are used to automate the validation of real-time embedded systems, but also to automate their synthesis (*i.e.* code generation) [8]. An important requirement of model based code generators is to ensure that the code produced actually enforces the abstract model of computation and communication. This aspect has been studied for several models of computation and communication (*e.g.* [2], [9], [10]). Verifying that the generated code enforces the abstract model of computation and communication of the source modeling language allows to ensure that functional model-based validation will still be valid after code generation. However, it does not deal with the impact of generated code on the timing characteristics of components. As a consequence, generated code might invalidate early schedulability analysis results. This is even more true when a code generator adds additional tasks or makes use of operating systems locks in order to implement communications among components.

The impact of the generated code on the validity of timing analysis results is the problem we tackle in this paper. This problem is even more relevant in the domain of embedded systems since the code generator has to comply with an important variability of programming languages (C, Ada, Java) and execution platforms (ARINC653, OSEK, RT-POSIX, Ravenscar profile, etc.) depending on the application domain. Indeed, the impact of code generator will vary with the underlying execution platform (*i.e.* middleware and/or operating system).

To tackle this problem, we propose in this paper an approach based on model refinements: as part of our code generation framework, abstract models of computation and communication are first transformed into implementation models with precise and analysable timing characteristics. We present this approach in next section.

III. Architecture Refinement and Analysis

In this paper, we propose to automate the refinement of software architecture models in order obtain precise schedulability analysis results. Here, precise means taking into account the overhead due to generated code and/or middleware components used to implement the abstract model of computation and communications. Figure 1 illustrates the main steps of the approach we propose:

1) Model refinement: the input model is transformed into a software architecture description that contains the model of execution platform's components (*e.g.* middleware, operating system).
2) Intermediate model analysis: the model produced by the refinement is then analysed in order to evaluate the schedulability of the system. Note that other types of non-functional requirements could be evaluated on the refined model (memory, safety, security, data flow latency, etc.).
3) Code generation: the refined model is then used to automatically produce the code of communications among components of the input model.

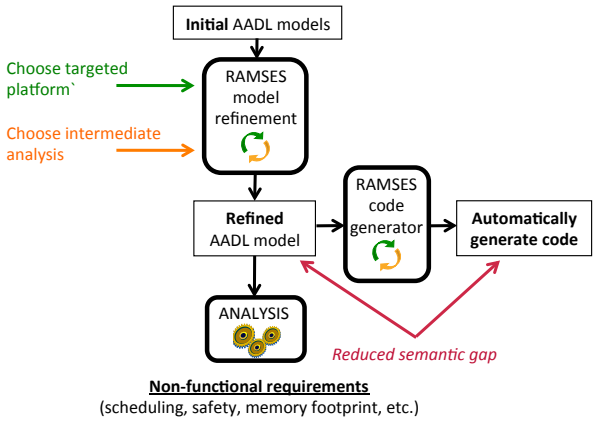

Fig. 1: Approach Overview

We already presented a refinement approach for code generation in [11]. In this paper, we extend this approach to deal with schedulability analysis of intermediate models. In comparison with a direct code generation from the input model, an advantage of this approach is that it reduces significantly the semantic gap between the analysis model and the generated code. We illustrate this result in the remainder of this paper: in section V, we explain the principles of the refinement step, and describe middleware components as we use them to improve the precision of schedulability analysis. In section VI, we show how the refined architecture can be used in order to proceed to schedulability analysis. Finally, we present in section VII some experimentation results, which show that our approach helps to take into account the overhead due to middleware components. This approach was implemented in RAMSES [11], using Cheddar [12] to automate the computation of tasks worst case response time. For the sake of completeness, we first give in next section an overview of AADL, the language we use to prototype and experiment our approach.

IV. Overview of AADL

AADL [13] is a standardized architecture description language that aims at supporting the design, analysis and integration of real-time embedded systems. Using AADL, a system is described as an assembly of software (*e.g.* process, thread, data, subprogram) and hardware (*e.g.* processor, memory, bus, device) components. Interfaces of components are described with to predefined port types (*e.g.* data, event, or event data ports), or component accesses (*e.g.* data or subprogram accesses for software components, bus or memory accesses for hardware components). The internal structure of a component is described by a set of connected subcomponents: connections between interfaces of subcomponents (or between interfaces of a subcomponent and its enclosing component) define communication channels between components.

In addition to these structural definitions, properties decorate AADL models to precise the characteristics of the

software architecture: for instance, the deployment of software components on hardware components, the protocol used to schedule tasks, the periodicity of a periodic task is defined with AADL standardized properties. Last but not least, annexes can be added to AADL components in order to extend the core language with sub-languages. For example, the behaviour annex of AADL is an extension that enables to precise the behavior of AADL components thanks to state machines. In these state machines, the semantic of states and transitions are consistent with the definition of the core AADL language. For instance, states of a thread component are defined as complete state, meaning that the thread becomes idle when this state is reached. In addition, the behaviour annex contains an action language: actions are defined in transitions of the state machine. Actions enable to describe the behavior of components in terms of computation actions (*i.e.* assignment, arithmetic operations, etc.) and interactions with interfaces of a component (*i.e.* get a message from a port, put a message on a port, etc.).

In this paper, we focus on a subset of AADL made of thread and subprogram components, connected through ports or data access, and deployed on processor components. In addition, the behavior annex can be used to detail internal behavior of thread and subprogram components. Besides, we use AADL at different levels of abstraction as illustrated in Figure 1. These levels of abstraction are detailed in next sections.

V. Automatic Integration of Middleware Components

The approach we propose in this paper relies on model refinements, *i.e.* endogeneous model transformations that increase the level of details in the output model. To explain this part of our contribution, we first present the type of input models we consider. We then explain how middleware components should be modelled to be used in this approach. Finally, we describe the principles of AADL model refinements for timing analysis, and more specifically how to integrate middleware components in a refined version of the input architecture.

A. Input Architecture Models

Architecture models we consider in this paper are based on the AADL standard and its Behavior Annex. Figure 2 illustrates the type of architecture we consider with a simple example made of two periodic tasks $T1$ and $T2$. The behavior of each task is described thanks to a state machine formalized with the AADL Behavior Annex. States and transitions of thread $T1$ are visible on figure (states with circles and transitions with arrows). Different types of actions can be associated to a given transition, including interactions with components interfaces, and abstract computations represented by timing consumption. For instance, in the the state machine of task $T1$, transition from $S1$ to $S2$ includes a computation of 1 to 2 milliseconds (noted ***computation****(1ms..2ms)* in the behavior annex), then an emission of value x (data subcomponent of $T1$, typed as $Float$) through port Po (noted $Po!(x)$ in the behavior annex).

In order to implement interactions among tasks, architecture designers usually rely on a middleware and/or a code generator. In this paper, we propose to use model transformations

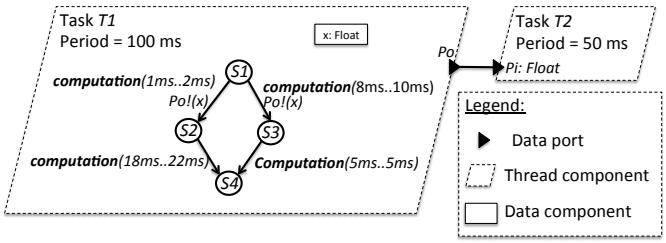

Fig. 2: Example of Input Architecture

in order to integrate middleware (or operating systems) components in a refined AADL model. This, of course, requires to model middleware components in AADL. In next subsection, we explain how to model middleware components with AADL in order to take into account their timing characteristics during schedulability analysis.

B. Modelling Middleware Components

In this paper, we consider middleware components as a set of threads, subprograms and data types. For these components, it is important to provide a model of the following characteristics:

- timing consumption of each subprogram or thread component: either as a timing interval (bounded by best and worst case execution time) for subprograms or threads, timed behavior actions (in the behavior annex), or a set of properties that enable to compute such timing consumptions;
- accesses to shared data, in order to describe which component has access to a shared data, when does it access it, and what is the access policy to be considered for schedulability analysis;
- scheduling properties of a task set: scheduling protocol, periods, deadlines, and priorities (if needed).

```
1  subprogram put_value
2  prototypes
3    value_type: data;
4    lock_type: data;
5  features
6    lock_access: requires data access lock_type;
7    value_in: out parameter value_type;
8    data_storage: requires data access value_type;
9  annex behavior_specification {**
10   states
11     s1: initial final state;
12   transitions
13     t1: s1 -[]-> s1
14     {
15       computation(1 ms .. 2 ms) in binding (x86);
16       lock_access!<;
17       data_storage[current_slot] := value_in;
18       lock_access!>;
19     }
20 **};
21 end put_value;
22
23 processor x86
24 properties
25   Assign_Time => [Fixed => 0us; Per_Byte => 50 us];
26 end x86;
```

Listing 1: AADL Middleware Component

Listing 1 shows one component (a subprogram definition) of the middleware we use to implement communications between AADL threads (*e.g.* based on connections represented on figure 2). In addition, this listing provides the definition of a processor component which according to the AADL standard, is an abstract execution platform that represents both the hardware execution unit, and the operating system running on it. The reason for modelling the processor at this stage is that execution times of threads and subprograms obviously depend on the processor they are executed on. The execution time of subprogram *put_value* is represented in its behavior annex (starting line 9 of the listing), with a single state (initial and final) and a single transition. In the remaining of this subsection, we present the timing characteristics that can be deduced from these simple components:

- An abstract computation first describes that an execution time interval of one to two milliseconds is necessary at the beginning of the execution of this subprogram (see line 15 of listing 1). Note that these timing characteristics are only valid when the subprogram is executed on the *x86* processor (modelled in the same listing) as specified by the *"in binding"* statement line 15.
- Execution time of subprogram put_value can also be deduced from the assignment action line 17, combined with (i) the data size of operands of the assignment (unknown when the middleware is developed, but deduced from the input model during the refinement), and (ii) the assignment time property given in line 25 of the listing (which also depends on the processor on which the assignment action is executed).
- Data accesses are represented in lines 16 and 18, with a locking (line 16) and unlocking (line 18) access to shared data that will be connected to interface *lock_access* (line 6).

Note that the modelling of middleware components makes use of AADL prototypes defined in the version two of this language. AADL prototypes enable to define incomplete architectures along with consistency rules to complete the architecture later in the design process. For instance, in lines 3 and 4 of listing 1, we defined two prototypes of data components that are then used in the definitions of interfaces of subprogram *put_value* (lines 6 to 8). The reason for using these prototypes is that when the middleware components are modelled, we don't know yet what will be the data type used in communications. By combining the middleware models with the input architecture model (for instance the one given in figure 2), we know from the connections in the input architecture model which data types are actually used in each connection; we can then refine this model into one that includes data structures and refined subprogram to model the implementation of connection. This process, automated in RAMSES, is called architecture refinement and is described in next subsection.

C. Architecture Refinement

Figure 3 illustrates the principles of the architecture refinement implemented in RAMSES in order to provide fine-grain schedulability analysis.

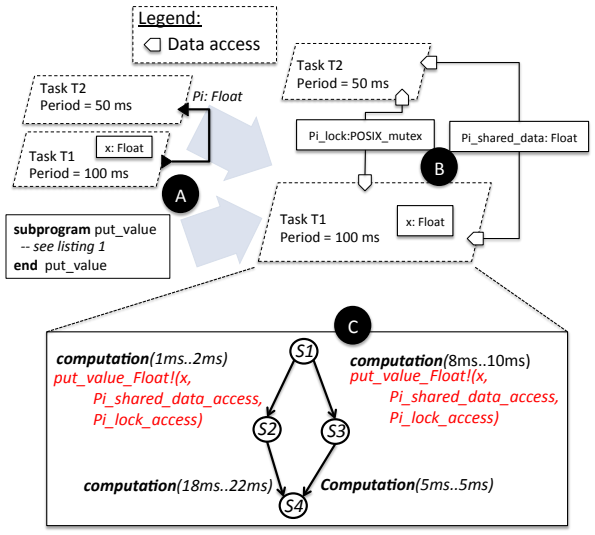

Fig. 3: Model Refinement for Timing Analysis

On the part **A** of the figure, we represented a summary of the modelling elements presented in previous subsections: the input model architecture on top, and the middleware components (see subprogram put_value) on the lower part. Part **B** of the figure represents the result of the refinement implemented as an automatic model transformation in RAMSES. This transformation expands abstract communication interfaces (*i.e.* AADL ports) into data accesses and subprogram calls. Data accesses enable threads to write or read the value shared variable Pi_shared_data, which contains data exchanged between threads through these ports. Global variable Pi_lock enables to protect accesses to Pi_shared_data. Subprogram calls are represented on part **C** of the figure (see $put\_value\_Float!(...)$). Part **C** of the figure illustrates the state machine resulting from the refinement of $T1$'s behavior: instructions $Po!(x)$ (see figure 2) have been replaced by calls to put_value_Float, which is a refined version of subprogram put_value (presented in listing 1). Listing 2 gives the definition of subprogram put_value_Float, which simply extends put_value and defines prototypes (line 2 of the listing). For instance, data type $value\_type$ is deduced from the input model: the data exchanged between threads are of type $Float$ (see figure 2). This subprogram copies the value of x into a global variable $shared\_data$. This global variable is made accessible to $T1$ in part **B** thanks to the structural model transformation that expanded the connection between Po and Pi into two global variables, four data access interfaces, and four connections.

```
1 subprogram put_value_Float extends put_value
2 (value_type => Float; lock_type => POSIX_Mutex;)
3 end put_Value_Float;
```

Listing 2: Refined Middleware Component

As a consequence of the refinement presented in this section, the behavior of threads is defined more precisely: timing characteristics given in both the input model, and the middleware components are merged into a unique model produced automatically. In next section, we explain how we

use the information contained in refined models in order to proceed to fine-grained schedulability analysis.

VI. Precise Schedulability Analysis

This section describes the method we propose to proceed to schedulability analysis of refined models. This method is made of two steps, presented in subsections VI-A and VI-B respectively. These steps aim at transforming the models obtained after refinement in models that can be processed by schedulability analysis tools. The model of computation we consider for schedulability analysis is a very classical model in which a system is made of a set of processors, tasks, and shared data. A processor represents a computation unit with a scheduler type. Each task runs on a processor, and has a set of scheduling attributes (e.g. a period, a deadline, and a capacity). Tasks also reference shared resources they use, and each usage is parametrized by an earliest and latest date respectively for the acquisition and release of this shared resource. This model of computation is typically the one implemented in Cheddar for schedulability analysis [12]. This model is of course much simpler than the one obtained as a result of the refinement since it reduces the behavior of tasks to a timing intervals with critical sections, which means we need to proceed to a new model transformation from the refined model to the analysis model. We present the principles of this transformation in the remainder of this section.

A. Extraction of Timing Execution Trees

Refined models, obtained from transformations presented in section V, contain a set of tasks with a description of their behavior. The behavior of each task is defined with a state machine made of execution time intervals, data accesses, and subprogram calls. The behavior of subprograms is also defined with similar state machines. For each task, we concatenate the information contained in its state machine into a set of execution trees in which a node is either the initial state of the task, a lock or an unlock action, or the final state of the task. Arcs of the obtained tree are decorated with execution time intervals resulting from the concatenation of execution time intervals of state machines: the lower bound (respectively upper bound) of a time interval associated with an arc is the sum of lower bounds (resp. upper bounds) of each time interval of transitions going from the element in the state machine that corresponds to the source node of the arc, and going to the element in the state machine that corresponds to the target node of this arc. A node may have several outgoing arcs to represent conditional branches, which exist when a state has several outgoing transitions or when actions of a given transition are conditional (*i.e.* controlled by an "if" statement). Note that all the timing intervals have to be represented with the same timing unit, defining the timing precision of the analysis. End-users of our framework can select the timing precision by defining the default unit of the schedulability analysis. When conversions are necessary, the floor (respectively the ceiling) of the resulting number is taken for the lower bound (resp. upper bound) of the timing interval associated to an arc.

Figure 4 represents the timing execution tree of task $T1$ on an $x86$ processor with an analysis precision of a millisecond. We explain the left branch of the tree, given that the explanations for the other part is very similar:

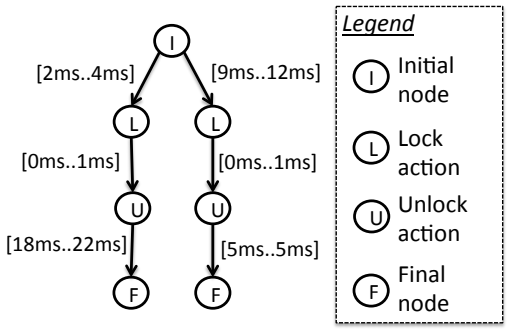

Fig. 4: Execution tree of $T1$

the first time interval is the result of concatenating abstract computations $computation(1ms..2ms)$ from T1 and $computation(1ms..2ms)$ from put_value. The second time interval results from the computation of the assignments time, given the property $Assign\_Time$ line 25 of listing 1. The resulting value is 0.5 millisecond, which means an execution time between 0 to 1 millisecond. Finally, the last time interval is simply a copy of the abstract computation from state $S2$ to state $S4$ in $T1$'s behavior.

For each task, we thus obtain a timing execution tree similar to the one presented on figure 4. However, such sets of execution trees do not directly match hypothesis of schedulability analysis frameworks like Cheddar. As a consequence, we have to extract, from these sets of execution trees, analysable configurations from which we can ensure the schedulability of the system. We present this extraction in next subsection.

B. Computation of Analyzable Task Set Configurations

The execution trees obtained from the extraction presented in section VI-A are closer to the model of computation dedicated to schedulability analysis since actions from the behavior annex have been translated into timing intervals. However, execution trees contain conditional executions that are not part of models analyzable with Cheddar. We thus transform execution trees in a set of analyzable configurations. This transformation is rather simple:

1) we consider each complete branch of the execution tree is an execution scenario for a task of the system. By complete branch, we mean: from the initial node until a final node of the tree. We thus obtain a set of execution scenarii for each task of the system.
2) we compute the Cartesian product of these sets of execution scenarii. Each resulting set of tasks scenarii is thus an analyzable configuration.

To illustrate the complexity of the analysis we propose, we consider $T2$ has an execution tree made of three branches. Since $T1$ has an execution tree of two branches, this means we have to analyze six task configurations. In other words, the number of configurations to consider is $\prod_{0<i<N} P_i$, where N is the number of tasks and P_i is the number of path of an execution tree, going from the initial node to a final node. We analyse the schedulability of each of theses configurations (combination of tasks execution scenario). If one configuration is not schedulable, a counter example is given to the software

architect. In next section, we present our experimental results obtained with our approach.

VII. Experimentations results

In order to evaluate our contribution, we realized two types of experiments that aim at measuring, respectively, the utility and the usability of our approach. To evaluate the utility of our approach, we consider an input model from which we measure the difference between the worst case response time of tasks with an without considering the overhead of execution platform operations. To evaluate the usability of our apporach, we focus on its scalability, and provide a performance measurement for executing our refinement process on a realistic model. We present our experimental results in next subsections.

A. Case Study

The evaluation of our approach relies on a realistic case study inspired from the railway domain. In terms of scheduling and communications, this use-case relies on a partitionned system with same principles as ARINC653 software architectures. Our case-study is made of two partitions, eight threads (four thread in each partition), and twelve connections among ports of these threads. Figure 5 illustrates the architecture of this case study in order to show rapidly its complexity.

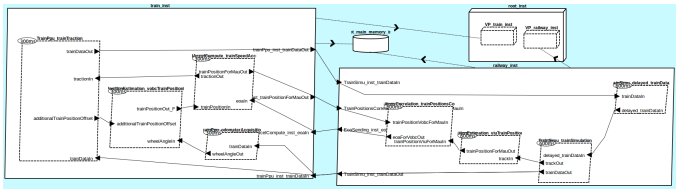

Fig. 5: AADL Model of our Case Study

B. Importance of Overhead Insertions

Figure 6 provides the worst case response time of each of the threads of our model, before and after refinement. The results provided by this figure are not surprising: the worst case response time is higher when considering overheads due to communications, and the impact of this overhead is more important for tasks with a lower priority than for tasks with a higher priority (tasks are scheduled with a fixed priority scheduling inside partition windows).

The results of this experiment show that the overhead due to communication grows rapidly with the number of tasks and the number of communications among tasks. Indeed, even if we did not consider real execution time estimations, the evolution of the execution time clearly shows a rapid growth because of the overhead. This tends to confirm that considering this overhead is necessary to validate the schedulability of complex software-intensive systems.

C. Performance Evaluation

The complexity of this analysis grows rapidly with the number of branches in tasks execution trees. The number of configurations to consider mainly depends on the complexity of input models: when shared data are acquired and released

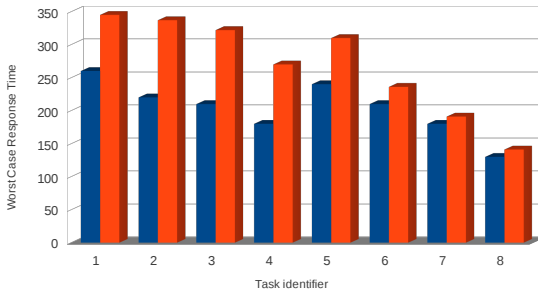

Fig. 6: Worst Case Response Times With and Without Overheads

in different conditinal executions of a task, the number of configuration to analyze grows rapidly. On the other hand, if shared data are all accessed in a unique branch of an execution tree, the number of configurations to analyze remains low.

The timing analysis of our case study required the analysis of 64 tasks configuration resulting from the cartesian product of tasks execution scenari. Experimentations were conducted on a 2.7 GHz Intel processor (Intel Core i7-3740QM; 4 cores) with 3.9 GiB memory and a SSD hard drive disk. The complete process, from the begining of the model refinement, until the compilation of generated code, passing by the analysis of 64 tasks configuration took 2 minutes and 17 seconds. Considering the complexity of the input architecture, this result seems to be very satisfactory: of course, the number of configuration to analyse can grow very fast by increasing the complexity of the input model, but the analysis of every single configuration is the price to pay for an exhaustive result.

VIII. Related Works

Many model-based frameworks have been proposed in order to ease the analysis and implementation of embedded systems. In [14], authors present an extension of Ocarina for ARINC653 operating systems. In both cases (ravenscar or ARINC653), AADL models used for code generation could also be used for schedulability analysis. However, models refinement was not considered in this work, which means that timing overheads due to execution platform functions (middleware and/or operating system functions) were not considered during timing analysis. An AADL model transformation process was introduced in [15], and extended in [11] in order to ease the production and maintenance of code generators for multiple target platforms. However, these works did not explain how to proceed to schedulability analysis of refined models. Going beyond the state of art of these approaches, the method we presented in this paper combines both refinements for code generation and analyse purposes.

More generally, refinements can be seen as part of design space exploration: a refinement being in this context a set of design decisions. For instance, [16] and [17] propose model based design exploration thanks to optimization techniques. They both consider input models with degrees of freedom, and produce automatically output models in which decisions have been taken with respect to these degrees of freedom. These approaches are compelementary with the method we

presented in this paper, since design decisions could be part of our refinement process.

Refinement for timing analysis can also be considered with the perpective of execution time evaluation. In [18], refinement is considered as a way to improve the precision of worst case execution time analysis when taking into account caches effect. This work could be integrated in our approach by modeling more precisely the effect of execution plateform components on execution times considered for analysis.

Finally, authors of [19] present a method to analyse real-time systems schedulability from the specification of their timing behavior as state machines. As opposed to our hypothesis, tasks considered in [19] are independent (*i.e.* no locks) and mult-rate (*i.e.* not strictly periodic tasks). However, an interesting idea in this work is to be able to reduce the exploration of potential solution by identifying execution scenari that dominates others.

IX. Conclusion

In this paper, we have presented a method to reduce the gap between models used for timing analysis and for code generation. This method relies on model transformations in order to lower automatically the level abstraction of models used for timing analysis. During this transformation, execution platform components are refined and integrated in the architecture under design. This approach was implemented in the RAMSES framework, and uses Cheddar for the schedulability analysis. In this paper, we also present experimental results to evaluate the scalability and utility of our approach. These experiments were made on a realistic software architecture inspired from the raiway domain. However, the number of tasks set to analyse with this method grows rapidly with the number of tasks and braches in their behavior. To tackle this issue, we plan to extend this work in order to reduce the number of configurations to consider, either by relying on design patterns that limit the number of branches in tasks execution, or by prunig tasks configuration that dominate others.

References

[1] L. Sha, R. Rajkumar, and J. Lehoczky, "Priority inheritance protocols: an approach to real-time synchronization," *Computers, IEEE Transactions on*, vol. 39, no. 9, pp. 1175–1185, Sep 1990.

[2] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice, "Lustre: A declarative language for real-time programming," in *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL '87. New York, NY, USA: ACM, 1987, pp. 178–188. [Online]. Available: http://doi.acm.org/10.1145/41625.41641

[3] P. H. Feiler and D. P. Gluch, *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language*, 1st ed. Addison-Wesley Professional, 2012.

[4] S. Sentilles, A. Vulgarakis, T. Bures, J. Carlson, and I. Crnkovic, "A component model for control-intensive distributed embedded systems," in *Proceedings of the 11th International Symposium on Component Based Software Engineering (CBSE2008)*, M. R. Chaudron and C. Szyperski, Eds. Springer Berlin, October 2008, pp. 310–317, lecture Notes in Computer Science, volume 5282/2008. [Online]. Available: http://www.es.mdh.se/publications/1299-

[5] A. Jantsch and I. Sander, "Models of computation and languages for embedded system design," *Computers and Digital Techniques, IEE Proceedings*, vol. 152, 2005.

[6] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *J. ACM*, vol. 20, no. 1, pp. 46–61, Jan. 1973. [Online]. Available: http://doi.acm.org/10.1145/321738.321743

[7] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. J. Wellings, "Applying new scheduling theory to static priority pre-emptive scheduling," *Software Engineering Journal*, vol. 8, pp. 284–292, 1993.

[8] S. Edwards, L. Lavagno, E. A. Lee, and A. Sangiovanni-Vincentelli, "Readings in hardware/software co-design," G. De Micheli, R. Ernst, and W. Wolf, Eds. Norwell, MA, USA: Kluwer Academic Publishers, 2002, ch. Design of Embedded Systems: Formal Models, Validation, and Synthesis, pp. 86–107. [Online]. Available: http://dl.acm.org/citation.cfm?id=567003.567011

[9] E. Borde and J. Carlson, "Towards verified synthesis of procom, a component model for real-time embedded systems," in *Proceedings of the 14th International ACM Sigsoft Symposium on Component Based Software Engineering*, ser. CBSE '11, 2011, pp. 129–138.

[10] F. Cadoret, T. Robert, E. Borde, L. Pautet, and F. Singhoff, "Deterministic implementation of periodic-delayed communications and experimentation in aadl," Paderborn, Allemagne, Jun. 2013.

[11] F. Cadoret, E. Borde, S. Gardoll, and L. Pautet, "Design patterns for rule-based refinement of safety critical embedded systems models," in *ICECCS*, 2012, pp. 67–76.

[12] F. Singhoff, J. Legrand, L. Nana, and L. Marcé, "Scheduling and memory requirements analysis with aadl," *Ada Lett.*, vol. XXV, no. 4, pp. 1–10, Nov. 2005. [Online]. Available: http://doi.acm.org/10.1145/1104011.1103847

[13] S. A. E. S. of Automotive Engineers), "Architecture analysis and design language. sae as5506b, sae, 2009."

[14] J. Delange, L. Pautet, A. Plantec, M. Kerboeuf, F. Singhoff, and F. Kordon, "Validate, simulate, and implement arinc653 systems using the aadl," in *Proceedings of the ACM SIGAda Annual International Conference on Ada and Related Technologies*, ser. SIGAda '09. New York, NY, USA: ACM, 2009, pp. 31–44. [Online]. Available: http://doi.acm.org/10.1145/1647420.1647435

[15] G. Lasnier, L. Pautet, and J. Hugues, "A model-based transformation process to validate and implement high-integrity systems," in *Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2011 14th IEEE International Symposium on*, March 2011, pp. 67–74.

[16] A. Aleti, S. Bjornander, L. Grunske, and I. Meedeniya, "Archeopterix: An extendable tool for architecture optimization of aadl models," in *Model-Based Methodologies for Pervasive and Embedded Software, 2009. MOMPES '09. ICSE Workshop on*, May 2009, pp. 61–71.

[17] C. Mraidha, S. Tucci-Piergiovanni, and S. Gerard, "Optimum: A marte-based methodology for schedulability analysis at early design stages," *SIGSOFT Softw. Eng. Notes*, vol. 36, no. 1, pp. 1–8, Jan. 2011. [Online]. Available: http://doi.acm.org/10.1145/1921532.1921555

[18] S. Chattopadhyay and A. Roychoudhury, "Scalable and precise refinement of cache timing analysis via model checking," in *Real-Time Systems Symposium (RTSS), 2011 IEEE 32nd*, Nov 2011, pp. 193–203.

[19] M. Stigge and W. Yi, "Combinatorial abstraction refinement for feasibility analysis," in *Real-Time Systems Symposium (RTSS), 2013 IEEE 34th*, Dec 2013, pp. 340–349.