HAL
archives-ouvertes.fr

Validation of real-time properties of a robotic software architecture

Charles Lesire, David Doose, Hugues Cassé

▶ **To cite this version:**

Charles Lesire, David Doose, Hugues Cassé. Validation of real-time properties of a robotic software architecture. 6th National Conference on Control Architectures of Robots, May 2011, Grenoble, France. 7 p., 2011. <inria-00599689>

HAL Id: inria-00599689
https://hal.inria.fr/inria-00599689

Submitted on 10 Jun 2011

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Validation of real-time properties of a robotic software architecture

Charles Lesire and David Doose
ONERA - The French Aerospace Lab
F-31055, Toulouse, France
{charles.lesire, david.doose}@onera.fr

Hugues Cassé
IRIT / University of Toulouse
F-31062, Toulouse, France
casse@irit.fr

***Abstract*— In this paper, we propose a mechanism allowing to evaluate the schedulability of a robotic software architecture, and then validate its real-time properties. The robotic software architecture is described through a Domain Speci c Language (DSL), MAUVE, that allows to model communicating components. The evaluation of schedulability of the architecture consists in rst computing the Worst-Case Execution Time (WCET) of the elementary functions of the components. Then the Worst Case Response Time (WCRT) of the component is computed from the elementary WCET and the component models, allowing to validate the schedulatiblity of the architecture. We illustrate our methodology on the evaluation of a control architecture for a ground mobile robot.**

I. Introduction

Robots are more and more used in very diverse situations: service robotics, where they are directly in contact with persons; military missions, where they navigate in complex and dangerous environments; crisis management, where they may be responsible for the survival of human-beings.

In all these situations, robots must be *safe* and *reliable*. Robotic manufacturers must be able to provide the evidences that the robot behavior will fulfill some safety requirements. Such requirements aim at guaranteeing that the robotic system will hurt nobody, will not damage infrastructure, and if possible, will accomplish the goal for which the robot is used.

Among these requirements, we are more specifically interested in the requirements that concern the embedded software. Software architectures indeed support the behavior of the robot, in term of movement and navigation, communication management, and mission achievement.

Software requirements usually deal with the correctness of the code itself (i.e., verifying that the code corresponds to a formal algorithm, e.g. for laser mapping, or obstacle avoidance), and with the evaluation of the *schedulability* of the software architecture, i.e. to guarantee that the embedded software will be executed on time, ensuring for instance that the motion command will be sent to the actuators at a specific rate.

Next section presents the state of the art related to validation of robotic architectures. Then, in section III, we propose a methodology to analyze the schedulability of a robotic software architecture, based on a Domain Specific Language called Mauve. Finally, we illustrate this methodology on the control architecture of a mobile robot in section IV.

II. Validation of robotic architectures

The safety of robotic application can be considered either online or off-line. Online, the safety of the robot behavior is ensured by monitoring and controlling the robot's actions. [1] models the safe behaviors as LTL formulas, and then checks the validity (i.e. whether the formulas are true or not) at execution time, using the robot state and the actions to perform. Similarly, [2] has developed a *Request Checker* component that checks the feasibility of actions using state and resource information, based on constraint satisfaction techniques. Although these approaches help monitoring and controlling the behavior of the robot, no guarantee is given on this behavior before execution.

Guaranteeing a safe behavior of the robot before execution is addressed in [3] by first specifying the robot controllers as Petri nets. From these models, classical Petri net algorithms allow to verify some safety properties, such as boundedness or liveness. From these models, transformation techniques allow to generate some embeddable code, that ensures the safety properties satisfied by the models. The BIP (Behavior Interaction Priorities) model [4] is a component-based specification of the architecture, where components are described by ports (used as inputs or outputs), and a behavior, modeled as an automaton. Interaction between components correspond to either synchronization of the automata, or data exchange between components. From these models, some model checker tools allow to verify some properties on the system architecture, such as deadlock freedom, or reachability of states. Moreover, code generation consists in synthesizing a controller, that will schedule the execution of the components and synchronize their interactions. Nevertheless, these approaches do not consider real-time schedulability and properties of the architecture.

Real-time analysis in robotic applications is usually done empirically. For instance, [5], [6] have compared the real-time properties of robotic applications built with several software environments. Formally analyzing the real-time behavior of the robot architecture has not been studied in the literature.

Hence, we propose to start this formal validation of real-time properties of robotic software architecture, by first modeling the architecture components and their interactions, and then analyze the worst case execution time of the

functions that are executed by these components.

III. Validation of real-time properties

Formal analysis of real-time systems is often supported by specific languages and tools, such as temporal logic, automata, Petri nets... However, using this kind of language has some drawbacks: (1) the resulting models are often too detailed, making the modeling task quite complex; (2) each language is relevant for a specific set of properties, forcing to use several tools and models to analyze complete system; and (3) the modeling task must be redone each time a new system is considered.

To overcome these issues, the general approach is to define a modeling language for the target domain. This Domain Specific Language (DSL) is defined in order to model all the system information that are useful for the analyses. It is then used as a pivot language for several model transformations, that will allow to use different tools and methods for the system analyses.

A. MAUVE: a component-based DSL

In order to analyse robotic systems, we have defined a DSL called MAUVE (Modeling AUtomatic VEhicles). Mauve allows to model mono-processor real-time architectures in which the robotic control architecture is defined by communicating components.

1) The component model: Mauve allows to model libraries of reusable components. A component is described by several elements (Fig. 1):

- **Services:** each component can provide a set of services (called *operations*) to the other components; it can also need a specific service (called *method*); execution modes are associated to services in order to define which task will execute the corresponding function;
- **Ports:** components can exchange data through *ports*; a component define its *input* and *output ports*, with the type of the received or sent data; connections between ports are defined by *connectors* the specify the connection policy (buffered or not) and some parameters (size of the buffer, initial data, ...); a specific input port, the *event port*, wakes up the component when a new data is received;
- **Execution:** each instantiated component is described by a set of dynamic properties: its *period*, its *priority* and its *deadlines*; these parameters are used for the analysis and the execution;
- **Properties:** each component has a set of *properties* that define the component parameters;
- **Behavior:** each component has a specific behavior defined by a finite state machine (FSM); each state of the FSM is associated to several pieces of elementary codes (called *codels*):
  - the *entry* codel is executed at the step when the component enters this state;
  - the *message* codel executes the queuing messages;
  - the *method* codel executes the queuing methods;
  - the *run* codel is executed at each step when this state is active;
  - the *exit* codel is executed at the step when the component leaves this state.

This component description is consistent with a lot of component-based middleware classicaly used in robotics [7], [8], [4], [9].

2) Architecture description: The whole embedded system on which the analysis is performed is composed of several components, that interacts either by port connections, or by service calls. Each component is hosted by a task with the same execution properties as the component. The tasks are executed on a preemptive real-time system with a fix priority scheduler.

The first step towards the temporal validation of the control architecture is to evaluate the Worst Case Execution Time (WCET) of each component codel.

B. WCET Computation

We show in this section that safe WCET can only be estimated by static analysis for critical embedded system and how this analysis is performed.

1) Motivation: There are several approaches to obtain a WCET. The simple one consists in testing the application with different input sets, to measure the execution time and to take the worst one. This approach may be improved by stopping tests when some level of application coverage rate is reached.

While this approach works well with simple architectures, it becomes quickly unsafe with more powerful and modern microprocessors. To achieve power required by more and more complex embedded applications, statistic mechanisms like caches, branch predictors and prefetchers are added to the hardware. These facilities are designed to enhance performances in the average case but, in some situations, they may exhibits very long timings, even worse than if they were not used.

Usually, such a configuration is hard to reach using the measurement approach because (1) it is hard or almost impossible to accurately set the state of these facilities and (2) it is not straight-forward to determine what is the worst configuration for a given program.

In the opposite, the static analysis approach attempts to model the whole embedded system, software and hardware, in order to compute an overestimation of the WCET. Current software and hardware are so complex that it is specious to hope to cope with the full system behavior. Yet, the method used in static analysis ensures that the WCET is safe, by providing an overestimation of the actual WCET. The issue is now to reduce as much as possible the difference between the estimated and the real WCET. Such a difference is not usually computable; otherwise we can exhibit easily the real WCET. Yet, we can identify parts of the computation where overestimation may arise. The more the parts are overestimated, the more the WCET is inaccurate.

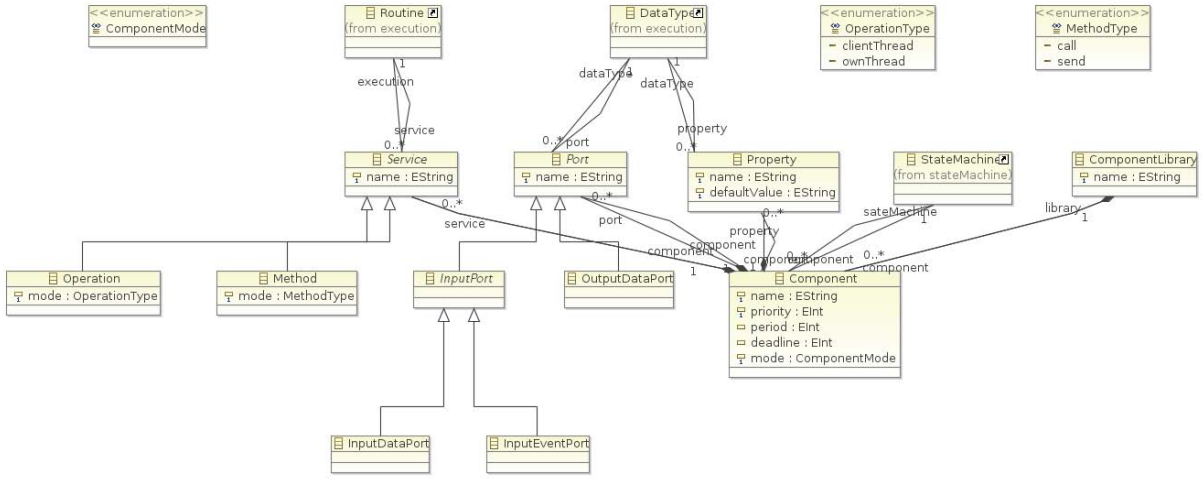


Fig. 1: Mauve component modeling

Let's take the example of a memory cache, a small fast memory put between the fast microprocessor and the slow main memory. It contains a subset of main memory blocks: when the processor performs a memory access, if the block is in the cache, we have a hit and a fast access, else a miss and the block must obtained from main memory (slow access). Usually, the static analysis assigns a category to each cache block access. This may be *Always Hit*, *Always Miss* or *Persistent* (once loaded in the cache, it is no more evicted). The last category copes with the fact that the cache behavior is too complex to be modeled, *Not Classified*. In this case, the WCET computation is evaluated taking into account both cases, either a *Miss*, or a *Hit*. This is not required [10] but the computation may consider each *Not Classified* block as a *Miss*. Although such a case is likely not to arise, the obtained WCET remains safe because we can assert it is an overestimation, but at the price of loosing precision.

Unfortunately, the main limitation of the static analysis approach remains that some hardware features does not exhibit predictable-enough behavior to be efficiently modeled. Therefore, a tradeoff must be found between hardware and program complexity to ensure safe WCET computation.

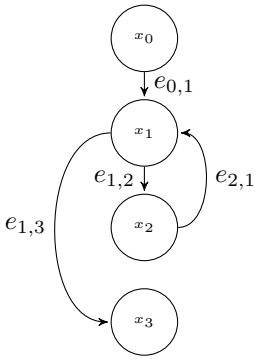
2) Computation: In this article, the OTAWA framework [11] has been used. It implements the Implicit Path Enumeration Technique (IPET) that has currently shown the best results. The execution is modeled by an Integer Linear Programming (ILP) system whose maximization produces the WCET. The WCET computation is divided in 3 stages:

- path analysis,
- timing analysis,
- WCET computation.

The first stage extracts the Control Flow Graph (CFG) from the binary of the application. A CFG provides a way to represent compactly the execution paths of the program. Its nodes are blocks of machine instructions and its edges represent the control flow between the blocks. We have to work at the machine code level because we need to model precisely the instruction execution inside the pipeline of the microprocessor. When the control flow of an application is too complex to be automatically extracted (for example with function pointers), the user can help the construction by providing annotations on the control flow. Figure 2 shows an illustrative source code (Fig. 2a) and the corresponding CFG (Fig. 2b).

```
int sum(int t[100]) {
  int i, s;
  s = 0;
  for(i = 0; i < 100; i++)
    s += t[i]
  return s;
}
```

(a) Source program



(b) CFG

Fig. 2: Example of a CFG corresponding to a source code.

The timing analysis stage is usually split in the global timing analysis and and the local timing analysis. In the first phase, different analyses are applied to model behavior of each long time-effect facilities as memory caches or branch prediction. In the second phase, the results of the previous analysis are used to compute the execution time t_i of each block of the CFG. Usually, several times for a block may be produced to cope with different executions. For example, block 2 has two timings t_2 and t_2^m to support, respectively, hit and miss in the memory cache.

In the last stage, the ILP system is built from the results of the previous stages. For instance, the ILP system corre-

sponding to Fig. 2 is:

$$W_{CET} = MAX(t_0x_0+t_1x_1+t_2^hx_2^h+t_2^mx_2^m+t_3x_3) \quad (1)$$
$$x_0 = 1 \quad (2)$$
$$x_1 = e_{0,1} = e_{1,3} + e_{2,1} \quad (3)$$
$$x_2 = e_{1,2} = e_{2,1} \quad (4)$$
$$x_3 = e_{1,3} \quad (5)$$
$$e_{2,1} \leq 100 \quad (6)$$
$$x_2 = x_2^h + x_2^m \quad (7)$$
$$x_2^m \leq 1 \quad (8)$$

The WCET is the sum of the execution time of each block multiplied by the number of time the block is executed (eq. (1)). Then constraints are added to the ILP to model the hardware and software effects on the WCET:

- to model CFG (for example, a block is executed as many times as the predecessors are executed – eq. (2)-(5)),
- to bound loop iterations (that causes circularity in the CFG – eq. (6)).
- to take into account global effects (like cache misses and hits – eq. (7) and (8) for Persistent category),

Passed to an ILP solver, the maximization of the system returns the WCET and the number of times each block is executed in the worst case.

C. Schedulability analysis

We propose a methodology to analyze the schedulability of the architecture, i.e. that each component will respect its deadlines. This analysis is based on the computation of the Worst Case Response Time (WCRT) of each component, i.e. the delay between its waking and the end of its execution.

1) Methodology: Several methods are classically used to evaluate the WCRT of a task. However, they cannot be simply used as the Mauve components are communicating together, creating some interaction between tasks. To evaluate the WCRT of such complex components, we will first transform the Mauve models into Periodic State Machines (PSM), and then approximate these PSM models into classical tasks, on which classical evaluation methods can be used (Fig. 3).

2) Mauve to PSM transformation: Each component is mapped into a periodic state machine. A PSM (Fig. 4) is a finite state machines in which one transition fires at each activation (i.e. at each period for periodic components). Each transition is associated to a computation time, obtained from the WCET evaluation of the corresponding codel. The PSM model is not deterministic: at each step, several transitions may be enabled. The PSM also owns the execution properties of the component (period, priority, deadline).

The PSM transformation associated one PSM for each *active* component. Passive (or non active) components correspond to *libraries*: they provide services to other components but are not mapped into a task. The active components that wake up on data reception (*event ports*) have specific PSM

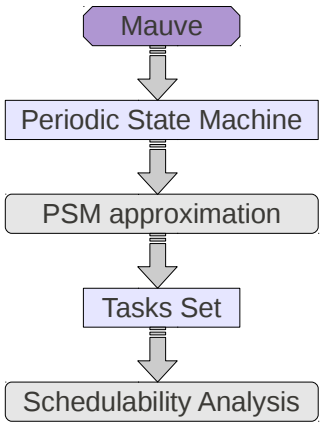


Fig. 3: Validation process. Model transformations are based on the rewriting tool Tom [12].

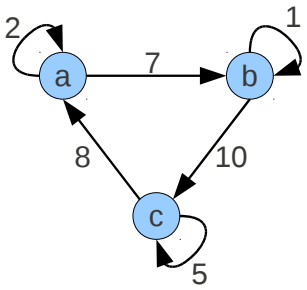


Fig. 4: A Periodic State Machine.

models where the activation frequency is upper bounded. The PSM model takes into account service calls between component. Data exchange are considered as non blocking and instantaneous (null duration).

The PSMs generated from the Mauve models have three interesting properties:

- the PSMs are strongly connected, i.e. each state is reachable from any other state;
- each state has loop transitions, leading to a possible infinite stay time;
- for each couple of states (a, b), it exists at most one transition from a to b.

3) PSM to tasks transformation: The next step consists in mapping the PSM models to task models. The tasks are built from the computation of upper bounds of possible execution run whose length is bounded.

Let consider the PSM of the i-th component. We compute all the possible runs of length $k_i = \lceil \frac{max_j D_j}{T_i} \rceil$ of the PSM, where D_j is the deadline of the j-th component and T_i is the period of component i. k_i represents the number of activation of component i. An execution run (or *trace*) is a sequence of execution durations $(e_j)_{1 \leq j \leq k_i}$ that correspond to the duration of execution of the transition fired at the j-th activation of the component. For instance, the PSM of Fig. 4

has $abbc$ as a run of length 3, whose sequence is $(7, 1, 10)$.

Then we define an approximation A_i of the traces $\mathcal{T}_i$ of length k_i for component i, according to equations (9) and (10).

$$e_1(A_i) = max_{t \in \mathcal{T}_i}\ e_1(t) \tag{9}$$

$$\forall l \in [\![2, k]\!],\quad e_l(A_i) = \left(max_{t \in \mathcal{T}_i} \sum_{j=1}^{l} e_j(t) \right) - e_{l-1}(A_i) \tag{10}$$

Figure 5 shows traces $abbc = (7, 1, 10)$, $bbca = (1, 10, 8)$, $ccaa = (5, 8, 2)$ and their approximation $A = (7, 6, 6)$.

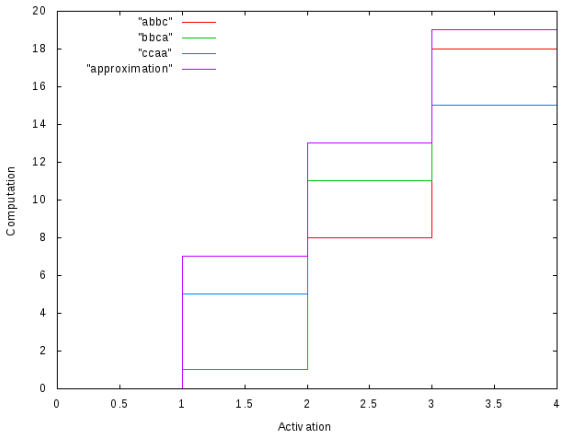


Fig. 5: PSM Approximation.

4) Worst Case Response Time: From the approximation of the component PSMs, each element of the approximation traces are considered as individual instantiated tasks. We can then perform a classical schedulability analysis on this set of tasks.

The computation of the WCRT consists in considering that all tasks wake up at the same time, that correspond to the worst case . Then we incrementally consider the interactions (preemption and delays) coming from tasks with a higher priority. The computation ends when either no more interaction is possible with higher-priority tasks or if the deadline is exceeded. The process then follows a fix point computation.

Let R_i be the WCRT of task τ_i. hp_i represents the set of tasks whose priority is higher than task τ_i. r_{ij} is the waking time of the j-th instance of task τ_i ($r_{ij} = j.T_i$). The fix point computation is defined by the following process:

1) $R_i^0 = C_{i,1}$
2) $R_i^{n+1} = C_{i,1} + \sum_{(j,l), j \in hp(i), l=1..k_j / r_{j,l} \leq R_i^n} C_{j,l}$
3) If $R_i^{n+1} \geq D_i$, the deadline is exceeded;
4) If $R_i^{n+1} = R_i^n$ then $R_i = R_i^n$
5) Otherwise, $R_i^n := R_i^{n+1}$ and go back to step 2.

IV. Evaluation of the control architecture of a mobile robot

A. The robot architecture

We consider a custom tracked robot, built on a TTRK-KT base (Fig. 6). The architecture of the robot is made of two processors: an ARM7 processor is used as an interface towards the TTRK base (direct control of motors, odomerty); an ARM9 processor is used to embed all the processing (sensor management, control laws, ...). In this study, we are only considering the latter processor, whose frequency is 200MHz, and that runs a Linux Ubuntu distribution with the real-time Xenomai patch.

Fig. 6: The TTRK tracked base.

B. The Orocos middleware

The software architecture of our robot is developed on top of the Orocos middleware [8]. Orocos has some specific elements that match quite well with the Mauve DSL: the software architecture is described by *components*, that are defined by *ports*, *services*, *state machines*, *hooks* (that correspond to *codels*), port *connections*, and *execution engines* (that own the execution parameters – period and priority).

Each component's behavior is described by a Finite State Machine (FSM). The default FSM is shown in Fig. 7; where each transition corresponds to a codel call. It is also possible

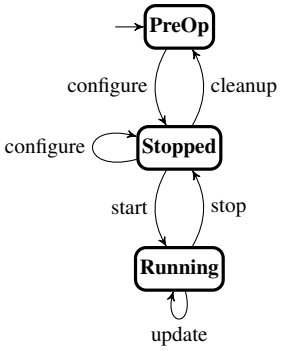


Fig. 7: Orocos standard FSM.

to define a custom FSM, while attaching codel calls to either transition, or specific state functions (entry, run, exit).

C. Architecture components

The robot software architecture is represented on Fig. 8. It is composed of six components: `IG500` acquires position data from a GPS sensor; `CHR-6dm` acquires attitude data

from an IMU sensor; `CICAS` sends control command to the robot tracks; `StateFusion` agregates position and attitude information and provides the robot state vector; `Command` computes the track command from the system state and the target pose.

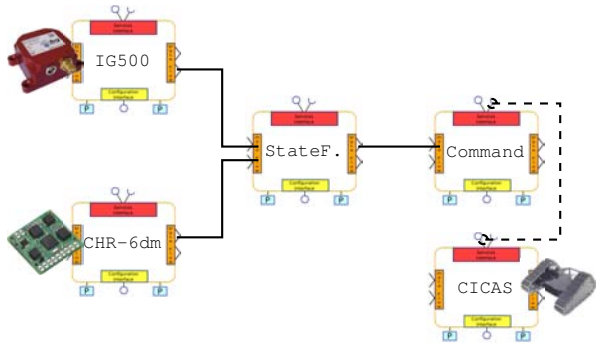


Fig. 8: Robot control architecture. Plain lines are data flows between components. Dashed lines represent operation calls.

`IG500`, `CHR-6dm` and `StateFusion` are defined by the default Orocos FSM. These components are periodic, with a period of 10ms for `IG500` and `StateFusion` and 1ms for `CHR-6dm`.

`CICAS` has no FSM and only provides a `send` service that sends the track command frame to the TTRK control processor.

`Command` has a specific FSM, defined by two control modes: `Rotating` when the control command consists in a rotation, and `Reaching` when the control command consists in reaching a given point. In each mode, a specific codel is called to compute the control command to apply, then the `send` service of component `CICAS` is called. The `Command` component is periodic with a period of 10ms.

D. Real-time evaluation

The scedulability analysis of the robot software architecture is then performed on the Orocos components (modeled with the Mauve DSL). The WCET computation is made on the codels, while the WCRT computation is made on the component models. As defined in [9], the codels are independent of the middleware, allowing more flexibility of the design.

This approaches induces that no analysis is performed on the Orocos code itself: we make the assumption that Orocos-specific execution time is null. This assumptions allows to make some schedulability analysis on the architecture, and will be discussed in the conclusion.

Moreover, as neither priority nor deadlines are defined for the Orocos component, we allocate them in a classical way for real-time systems: shortest period has the higher priority, and the deadline are defined equal to the periods.

Table I summarizes the Orocos components, their execution parameters, the WCET of the codels, and the resulting WCRT. The results of this analysis are that the system is *schedulable*, and that the processor load is $0,67$. It means that all the component will always satisfy their deadline.

This result points out the safety of our control architecture, as far as the real-time properties are concerned.

V. Conclusion and future works

In this paper, we have proposed an approach for the validation of the real-time properties of a software architecture. This approach is based on the Mauve DSL, that allows to model the architecture as communicating components. A first step consists in computing the WCET of the component codels, and then deduce the component's WCRT from the WCET and the component model. These WCRT lead to a conclusion on the schedulability of the architecture.

We have also shown an application of this approach on a robot control architecture developed and deployed using the Orocos middleware. The architecture, that only contains basic components for navigation control, has been proved to be schedulable on a 200MHz ARM9 platform.

This case study is being extended to consider more components, including more consuming processing, such as mapping and obstacle avoidance from laser data, and onboard decision making. The schedulability analysis, even if it fails at the beginning, will force to better define the architecture, by optimizing the codel code, or adjusting priorities or periods.

Another step of the validation process will be to investigate, first, an analysis of the Orocos code, for instance by evaluating the WCET of the data exchange (writing/reading on ports), or the service call, and second to evaluate our approach on another target middleware.

Finally, the objective will be to provide an environment for the development of a robotic architecture, that will allow to specify the components and the architecture, to write the associated codels and evaluate their WCET, and finally to generate the target code (either Orocos, another middleware, or bare C code) while verifying the schedulability of the resulting embedded architecture.

References

[1] P. Doherty, J. Kvarnström, and F. Heintz, "A temporal logic-based planning and execution monitoring framework for unmanned aircraft systems," *Journal of Autonomous Agents and Multi-Agent Systems (JAAMAS)*, vol. 19, no. 3, pp. 332–377, 2009.

[2] F. Py and F. Ingrand, "Dependable execution control for autonomous robots," in *International Conference on Intelligent Robots and Systems (IROS)*, Sendai, Japan, 2004.

[3] L. Montano, F. Garcia, and J. Villarroel, "Using the time Petri net formalism for specification, validation, and code generation in robot-control applications," *International Journal of Robotics Research (IJRR)*, vol. 19, no. 1, pp. 59–76, 2000.

[4] A. Basu, M. Gallien, C. Lesire, T.-H. Nguyen, S. Bensalem, F. Ingrand, and J. Sifakis, "Incremental component-based construction and verification of a robotic system," in *European Conference on Artificial Intelligence (ECAI)*, Patras, Greece, 2008.

[5] A. McKenzie, D. Gay, R. Nori, J. Davis, and M. Anderson, "Comparing temporally aware mobile robot controllers built with Sun's Java Real-Time System, Orocos' Real-Time Toolkit and Player," in *International Conference on Intelligent Robots and Systems (IROS)*, Taipei, Taiwan, 2010.

Component	Period (ms)	Priority	Codel	WCET		WCRT (μs)
				Cycles	Time (μs)	
CICAS	-	-	send	$1'030'335$	$5'512$	-
CHR-6dm	1	1	update	$28'846$	145	145
IG500	10	2	update	168	1	146
StateFusion	10	3	update	267	2	413
Command	10	4	update	$1'064'752$	$5'324$	$6'607$
			Rotating	$13'782$	69	
			Reaching	$34'417$	173	

TABLE I: Component description and WCRT results.

[6] A. Shakhimardanov and E. Prassler, "Comparative evaluation of robotic software integration systems: a case study," in *International Conference on Intelligent Robots and Systems (IROS)*, San Diego, CA, USA, 2007.

[7] R. Alami, R. Chatila, S. Fleury, M. Ghallab, and F. Ingrand, "An Architecture for Autonomy ," *International Journal of Robotics Research*, vol. 17, no. 4, 1998.

[8] P. Soetens and H. Bruyninckx, "Realtime Hybrid Task-Based Control for Robots and Machine Tools," in *International Conference on Robotics and Automation (ICRA)*, Barcelona, Spain, 2005.

[9] A. Mallet, C. Pasteur, M. Herrb, S. Lemaignan, and F. Ingrand, "GenoM3: Building middleware-independent robotic components," in *International Conference on Robotics and Automation (ICRA)*, Anchorage, AK, USA, 2010.

[10] T. Lundqvist and P. Stenström, "Timing anomalies in dynamically scheduled microprocessors," in *IEEE Real-Time Systems Symposium (RTSS)*, Washington, DC, USA, 1999.

[11] C. Ballabriga, H. Cassé, C. Rochange, and P. Sainrat, "OTAWA: an Open Toolbox for Adaptive WCET Analysis," in *IFIP Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS)*, 2010.

[12] E. Balland, P. Brauner, R. Kopetz, P.-E. Moreau, and A. Reilles, "Tom: Piggybacking rewriting on java," in *Conference on Rewriting Techniques and Applications (RTA)*, Paris, France , 2007.