A Task-Based Concurrency Scheme for Executing Component-Based Applications

Francisco Sánchez-Ledesma, Juan Pastor, Diego Alonso and Bárbara Álvarez
Division of Systems and Electronic Engineering (DSIE). Universidad Politécnica de Cartagena, Campus Muralla del Mar, E-30202, Spain; email: juanangel.pastor@upct.es

Abstract

This paper describes a flexible development approach for component–based applications with real–time requirements, which enables the performance of schedulability analysis of the resulting application. The work described in this paper is part of a more general approach, and as such it focuses on the design of a concrete part of the approach. Specifically, we describe a task-based concurrency scheme for executing component-based applications, the deployment model that enables us to configure the application execution as well as some examples of the performance of schedulability analysis of the resulting application. The aforementioned deployment model provides the approach with great flexibility, since it enables developers to generate and test different deployments of the same architecture, without modifying it, while at the same time it enables us, as the designers of the approach, to have better control over the resources and facilities required to execute the application, which is mandatory in embedded systems.

1 Introduction

Real-time (RT) systems possess specific characteristics that make them particularly sensitive to the architectural decisions made in the course of their construction. Concurrency design, task scheduling, distributed communication, etc. need to be addressed as soon as possible. However, it is not always possible to test them in the early development stages. Particularly, RT scheduling analysis cannot be performed until the final code is nearly finished and the execution platform has been selected. In case the application does not meet its timing requirements, it can be necessary to re-implement it, thereby increasing the development time and cost. Concurrent programming concepts such as thread, mutex, message, etc. are the common design units in RT systems, since they are also the analysis units. Despite being suitable for performing temporal analysis, they cannot be easily combined or composed in order to build new applications, since usually thread code and thread interaction are application specific.

Architectural software components [**?**] are self-contained units that encapsulate their state and behaviour, that communicate by sending messages through their ports, and that have only explicit context dependencies. They are normally used as the building blocks to model the application architecture, since the abstractions they provide are better suited for this purpose than those provided by concurrency. However, the design concepts that make components very suitable for application construction and code reuse, hinder the performance of schedulability analysis, since there is not a clear mapping between those concepts (i.e. port, interface, service, etc.) and concurrency concepts (i.e. thread, mutex, message, etc.). Typical examples of such mappings are:

- Component models that directly translate components into processes and that use a middleware for message exchange among them. These models provide developers with great flexibility at design time, but penalizes system performance because of the overhead introduced by the middleware. Schedulability analysis is hard to perform, since the developer must know both the threads that are created inside each process (component) and used by the middleware, as well as their timing properties.
- Component models where components are purely passive entities that are invoked sequentially by a single-threaded run-time suffer from "scant concurrency", since the application is normally executed by a cyclic executive. Despite being absolutely predictable they are very fragile in the sense that if the system needs to be modified, a task that before would fit in the slot, may now exceed it [1].

On the other hand, current object-oriented languages and frameworks provide mechanisms and libraries to flexibly manage concurrency in applications, like *java.util.concurrent*, *std::async* in C++ 11, or *android.os.AsyncTask*. In these models, the programmer enqueues the code he wants to be executed, while a pool of worker threads is in charge of dequeuing and executing them concurrently, returning the computed values by means of future objects. This is a very powerful, expressive, flexible, and easy to use model for concurrent execution. But this model has two main drawbacks from our point of view: it has a lower abstraction level for system modelling than architectural components, and its behaviour is not predictable, because worker threads dequeue and execute the activities as soon as there is one available activity and one idle thread. Therefore, it cannot be directly used in RT systems, but instead it must be slightly modified in order to make it more predictable.

The work described in this paper is part of a more general approach, entitled C-Forge [2], where programmers model

applications using architectural components whose behaviour is defined by means of state-machines with orthogonal regions. An object-oriented framework, FraCC [3, 4], provides the execution environment for the resulting application. The execution model is based on a modification of the task model just mentioned, so that schedulability analysis can be performed. In addition, FraCC provides a deployment facility that separates application architecture from its deployment in nodes, processes and threads. This separation allows developers to generate and test different deployments of the same architecture, without modifying it, while at the same time it enables us to have better control over the resources required to execute the application. Given the differences existing between the concepts of each domain (components and concurrency), C-Forge uses the *Model-Driven Software Development* paradigm [5] and its associated technologies to support the whole process. An example of the usage of C-Forge applied to underwater vehicles and the results of the schedulability analysis is described in [6].

This paper describes the task-based concurrency scheme we have developed for executing component-based applications, the deployment model that enables us to configure the application execution as well as some examples of the performance of schedulability analysis of the resulting application. The rest of the paper is organized as follows. Section 2 compares the described approach with other similar approaches. Section 3 briefly describes the task-based concurrency scheme and its main properties. Section 4 illustrates the flexibility of the approach by defining and testing some examples, while section 5 outlines the conclusions and future research lines.

2 Related work

Given the number of available component models [7], we will focus the rest of the discussion on those aimed to design software for RT systems, their concurrency capabilities and schedulability analysis.

ProCom [8] is the successor of SaveCCM. ProCom is integrated in an MDSD toolchain, which provides C++ source code generator and analysis capabilities, like worst-case execution time. ProCom defines two layers: the upper and the lower layer. The former allows developers to define large-grained components, i.e. subsystems, which are active and which communicate using message passing. The latter consists of basic functional components, which are interconnected inside subsystem, and which are passive and only activated by some external entity. Thus, components are active or passive depending on their size, but smaller components are always passive, independently of their complexity. In C-Forge, all components are active, but this does not mean that they require their own thread.

The *Architecture Analysis & Design Language* (AADL) [9] focuses on the modelling and analysis of the application architecture, both on the software and hardware platform. AADL defines components as the kind of elements that can be used to compose the software and the hardware. AADL does not support the notion of software component, as stated in the introduction, or changing the concurrency of the application, though it does support many analysis types, including schedulability analysis. There are several generators of Ada/C/C++ source code for implementing the application.

RUBUS [10] is a component model for RT systems that supports expressing timing requirements and properties on the architectural level, so that they can be later analysed. It provides schedulability analysis, distributed end-to-end response times, and overall stack analyse of the shared stacks, among others. It does not however model component behaviour, but it is added by the programmer. The execution semantics of software components (implemented as functions) is started based on an input-trigger, then read data on data in-ports, then execute the function, afterwards write data on out-ports, and finally activate the output trigger that will turn on the next connected components. RUBUS does not support concurrent execution.

The *CHESS* project [11] developed a MDSD toolchain for cross-domain modelling of RT embedded systems, that allocates distinct concerns to distinct views. It has been defined as a UML profile, including tailored MARTE profile and others OMG standards. Component behaviour can be defined with state-machines, other standard UML diagrams, as well as the *Action Language for Foundational UML* (ALF, http://www.omg.org/spec/ALF/). The Deployment view models the target execution platform, and software to hardware components allocations. The Analysis view supports Failure Mode Effects & Criticality (FMECA), Failure Mode and Effect Analysis (FMEA), Fault Tree Analysis (FTA), as well as schedulability analysis. CHESS also has generator to Ada/C/C++ /Java source code. Among the reviewed component models, CHESS is perhaps the most similar approach to C-Forge. C-Forge focus on a single way to model component behaviour and manage concurrency, which makes it easier to generate and compose code.

The Real-time Container Component Model (RT-CCM) [12] proposes a methodology for the design of component-based applications with hard real-time requirements. RT-CCM is a aimed at making the timing behaviour of applications predictable, and is inspired in the Lightweight CCM specification with some extensions. The added mechanisms also enable the application designer to configure this scheduling without interfering with the opacity typically required in component management. From the analysis of this model the application designer obtains the configuration values that must be applied to the component instances and the elements of the framework in order to make the application fulfil its timing requirements. However, RT-CCM considers components as black-boxes, while our proposal considers them as white-boxes, with their behaviour modelled by means of state-machines.

Summarizing, our approach revolves around the following reasons. Firstly, it is mandatory that the number of threads that execute the application, as well as their timing properties (mainly, computational load and period), are known in order to be able to perform a schedulability analysis. Secondly, in order to maintain the coherence between the design model (i.e., the components that define the application architecture) and the concurrency model, this data must be

somehow present in the former, so that the latter can be partially derived from it, and then completed by the developer if needed. These two reasons imply that component models that are purely structural, that is, that only provide primitives for defining the external component shell and its ports, cannot be used for this purpose. There are two viable alternatives to overcome this limitation: (i) to enhance a purely structural component model with the meta–data required to partially derive concurrency characteristics, or (ii) to enable the developer to define component behaviour together with timing requirements. The most important drawback of the first approach is that it is very difficult to assert that the component implementation is coherent with the meta–data that describes its concurrency characteristics and timing properties. We decided to follow the second approach.

3 Task-based concurrency scheme

As said in the introduction, he work described in this paper is part of a more general approach [2], where programmers model applications using architectural components whose behaviour is defined by means of state-machines with orthogonal regions. State-machines do not only model the lifecycle of components, but also enable modelling how components react to messages it receives from other components, to the results of internal computations, as well as to the passage of time. Communication among components only takes place through their ports, and is message-based, asynchronous without response. This mechanism does not only makes it possible to implement any other communication scheme as required, but also decouples component communication, since it does not allow blocking calls. As a good consequence, synchronization and message dependencies must be explicitly modelled in state-machines, which facilitates reviewing and reasoning about the component behaviour.

Regions constitute a very appropriate link between the architecture and concurrency domains. On the component domain, a region defines a part of the whole component behaviour, while on the concurrency domain, a region is assigned to the thread that will execute it. On the component domain, the states contained in a region have been enriched with properties that allow developers to define their timing constraints (mainly execution time, and period or inter-arrival time), while on the concurrency domain the thread's timing properties are derived from those of the states contained in the regions assigned to it. Regions represent computational units of work, since they contain the activities that encapsulate the code that must be executed by the component depending on its internal state. Though a region can contain many activities, only the activity associated to the active state can be executed.

The concurrency model we have developed in order to organize and control region execution in threads is based on a modification of the task-based scheme used in systems like *java.util.concurrent*, *std::async* in C++ 11, *Grand Central Dispatch* in iOS, or *android.os.AsyncTask*, to mention a few. In this model, the main thread enqueues the activities it wants to be asynchronously performed, while a pool of worker threads is in charge of dequeuing and executing them concurrently, returning the computed values by means of future objects. This is a very powerful, expressive, flexible, and easy to use model for concurrent execution, but its behaviour is not predictable, because worker threads dequeue and execute the activities as soon as there is one available activity and one idle thread. As such, it cannot be directly used in RT systems, but instead it must be slightly modified in order to make it more predictable:

- Make the computational load of worker threads static, decided by the user at development time instead of by the system at execution time.
- Convert the main thread into a "normal", worker thread, since there is not such a thing as "a main component" in C-Forge.
- Let the developer decide how many (worker) threads execute the application.
- Create a cyclic executive inside each thread in order to schedule region execution.

The proposed task-based concurrency scheme for executing component-based applications start by characterizing states. States that contain one activity also define its period, deadline, worst case execution time, and activation pattern (periodic or sporadic): $St_i = (T^i_{act}, WCET^i_{act})$; data that is obtained from the application requirements. We assume that period equals deadline, and that period also model the minimum inter-arrival time in the case of sporadic activities. Starting from this data it is possible to calculate the timing properties of the regions of all components by applying equations 1 and 2. This is a pessimistic estimation, since we assume the region will be always executing the activity with the largest execution time, but it is needed in order to perform the schedulability analysis.

$$T^i_{reg} = gcd(T_{act} \in R^i) \tag{1}$$

$$WCET^i_{reg} = max(WCET_{act} \in R^i) \tag{2}$$

On the other hand, the application can be executed in a set of nodes, which represent computational units. They contain a finite set of processes, which represent the unit of resource management. Processes contain a finite set of threads, which represent the unit of concurrent execution. Components are assigned to processes and the regions of a given component can be assigned to any of the threads of the process that contains such component. This is a flexible scheme, which enables threads to execute regions contained in different components, but which does not force to assign all the regions of a component to the same thread.

Threads can be characterized by their period and their worst case execution time: $Th^i = (T^i_{th}, WCET^i_{th})$, which can be derived from the assigned regions by applying equations 3 and 4.

$$T^i_{th} = gcd(T_{reg} \in Th^i) \tag{3}$$

$$WCET^i_{th} = \sum(WCET^i_{reg} \in Th^i) \tag{4}$$

A cyclic executive scheduler is created inside each thread in order to control the execution of the regions assigned to it. Given that the assignment of regions to threads is static and is made at design time, it is possible to automatically calculate the parameters needed by the cyclic executive, primary cycle (H) and secondary cycle (T_s), and build the execution table from such assignment. The primary cycle (H) can be calculated by means of equation 5, while the secondary cycle coincides with the thread period, calculated by means of equation 3.

$$H^i = lcm(T_{reg} \in Th^i) \tag{5}$$

It should be highlighted that FraCC does not give any guidance as to the number of threads that have to be created or how regions should be assigned to them, but rather it provides the necessary mechanisms to enable developers to choose the appropriate heuristic methods, like the ones defined in [13], for instance. Both the number of threads as well as the allocation of regions to them can be done arbitrary, but the main objective should be to "*ensure application schedulability*". Two heuristics we normally use are to assign to the same thread regions that have similar periods, or that have states which activities communicate with each other.

3.1 Schedulability Analysis

A deployment model in C-Forge enables developers to set the application distribution in computational nodes, as well as the number of processes and threads in which the application will be run, as described previously. This organization makes it possible to perform schedulability analysis of a given application deployment. The deployment model provides great flexibility, since it does not impose a fixed relationship between component and processes/threads, but rather allows developers to define it, within certain limits. It also enables us to better control the resources and facilities needed by the platform in order to execute the application, and use only the necessary ones, as well as the performance of RT schedulability analysis. For instance, if all the application components run in the same node, no middleware is really needed, and thus lighter mechanisms, like shared memory, can be used instead for message exchange.

Cheddar [14] is a RT scheduling tool, designed for checking task temporal constraints of a RT system. In order to perform the schedulability analysis, Cheddar requires the number of tasks, their timing properties (mainly wcet and period) and the number of shared resources of the application. *Thread*s of the deployment model are directly transformed into Cheddar tasks, but shared resources must be derived from the deployment model, according to the buffer structures implemented in FraCC, as described in [3]. It must be highlighted that shared resources do not use synchronization primitives, only mutual exclusion, due to the fact that communication among components is only asynchronous. This makes it possible to bound blocking times.

According to the memory structure, only the buffers are candidates to be structures shared among threads. Among the generated buffers, only those that hold messages sent or processed by activities contained in regions assigned to different threads need to be protected from concurrent access. Buffers that hold messages produced or consumed by activities contained in regions assigned to the same thread need not be protected, since they will be accessed sequentially by activities. These shared buffers use the immediate ceiling priority protocol. It must also be noted that there is only one active state per region, and thus only one activity per region will access these buffers. All the needed information can be derived from the architectural and deployment models.

Table 1: Regions' calculated timing properties

Region	Period (ms)	WCET (ms)
R1	10	0.5
R2	20	1
R3	2	0.5
R4	40	0.8
R5	20	1
R6	2	1

In case the schedulability analysis concludes that the application is not schedulable, the developer can first generate new deployment models, mainly by changing the number of threads and the assignment of regions to threads. If the applications continues to be not schedulable, he/she has to start modifying the architecture, mainly by changing the algorithms used in the activities to faster ones, or by relaxing the timing constraints of the states. The last option if none of the previous generates a schedulable application is to change the components themselves, and thus the application architecture.

4 Sample Application

In order to illustrate the system and execution models described in the previous section, as well as the schedulability analysis, the sample application shown in figure 1 will be used. As can be seen, it comprises three components and six regions, which timing properties are also depicted in the figure. We assume that the application will run in one node and one process.

4.0.1 Region characterization.

The timing properties of the regions are derived from their activities by applying equations 1 and 2 as shown in Table 1.

4.0.2 Region to threads assignment.

Regions can be assigned to threads in an arbitrary way. A possible thread scheme considers four threads to execute the application, with the following assignment: $Th1 = \{R2\}$, $Th2 = \{R1, R4, R5\}$, $Th3 = \{R3\}$, and $Th4 = \{R6\}$. The following subsection will present some deployment examples in which we change this assignment and the results of the Cheddar analysis.

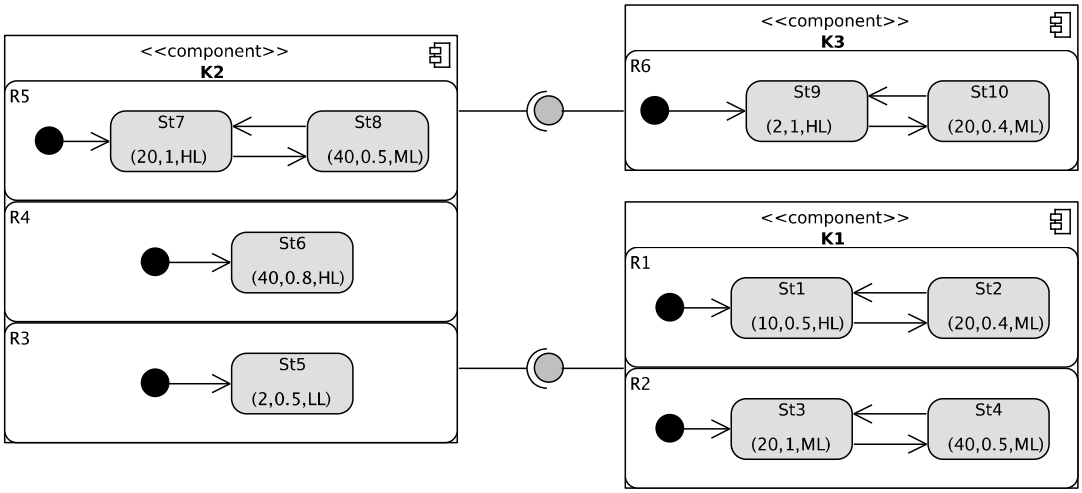

Figure 1: Sample application architecture with timing properties.

Table 2: Threads' calculated timing properties

Thread	Region/s	Period (ms)	WCET (ms)	Priority
Th1	R2	20	1	4
Th2	R1, R4, R5	10	2.3	3
Th3	R3	2	0.5	2
Th4	R6	2	1	1

4.0.3 Threads characterization.

Given this assignment, the timing properties of the thread can be calculated by applying equations 3 and 4, as shown in Table 2. The rate monotonic algorithm is used to calculate the concrete priority level. The lower the priority number (Pr), the higher the thread priority.

4.0.4 Scheduling regions inside threads.

Threads $Th1, Th3$, and $Th4$ do not need to schedule the regions inside them because they only have one region each, but thread $Th2$ does need to schedule regions $R1$, $R4$, and $R5$. To schedule these regions we need to calculate the primary and secondary cycles, as well as to build the scheduling table. Primary cycle is calculated by applying equation 3: $H_2 = lcm(T_{R1}, T_{R4}, T_{R5}) = lcm(10ms, 40ms, 20ms) = 40ms$, while the secondary cycle coincides with the thread period, $T_{s2} = 10ms$. Thus, the scheduling table will have four secondary cycles of $10ms$ each:

$t = 0ms$ Executes $R1$, $R4$ and $R5$
$t = 10ms$ Executes $R1$
$t = 20ms$ Executes $R1$ and R_5
$t = 30ms$ Executes $R1$

4.1 Deployment Examples

Table 3 shows the results of the Cheddar analysis for the sample deployments we describe below. The default deployment model created by the tool, deployment 1, defines one node with a single process hosting just one thread. All components are assigned to this process, while all regions of the components are assigned to such thread. Given the periods and worst execution times of the components regions, it is clear that the application resulting from the default deployment is not schedulable.

In deployment 2 the developer has defined three threads, one for executing each component. At a glance, it is possible to determine that the application is again not schedulable. Note that the period of thread 3 is lesser than its WCET. The developer can change the regions assignment, as it is shown in deployment 3, in order to reduce the WCET of thread 3. This new deployment is now schedulable. Deployment 4 shows the case where all the components' regions have been assigned to different threads, resulting, for this example, in the best of the four deployments from the point of view of processor usage.

5 Conclusions

This paper describes a flexible development approach for component–based applications with real–time requirements, which provides developers with enough control over the concurrency characteristics of the application execution so that schedulability analysis can be performed. These objectives have been achieved by means of (i) defining a component model that includes structure and behaviour; (ii) establishing a clear separation between these concerns, decoupling the structural elements from the behavioural and the algorithmic ones; (iii) defining a clear and consistent association between the elements of the system and execution models through a deployment model. The approach is supported by a model-driven toolchain developed in Eclipse (C-Forge).

The explicit modelling of component behaviour by means of state-machines with orthogonal regions offers several advantages, namely it enables developers to describe the temporal requirements at the architectural level; orthogonal regions explicitly reflect the concurrent nature of the component behaviour; regions have proven to be an excellent way to link the architecture and concurrency domains, since on the component domain regions define a part of the whole component behaviour, while on the concurrency domain they define the unit of computational work assigned to a thread. The concurrency scheme we developed in order to organize and control

Table 3: Summary of the four considered deployments and the results of the Cheddar schedulability analysis

Deployment 1 (T, WCET)	Deployment 2	Deployment 3	Deployment 4
Thread1 (T=2, WCET=4.8) Reg1 (10, 0.5) Reg2 (20, 1.0) Reg3 (2, 0,5) Reg4 (40, 0.8) Reg5 (20, 1.0) Reg6 (2, 1.0)	Thread1 (T=10, WCET=1.5) Reg1 (10, 0.5) Reg2 (20, 1.0) Thread2 (T=2, WCET=1) Reg6 (2, 1.0) Thread3 (T=2, WCET=2.3) Reg3 (2, 0,5) Reg4 (40, 0.8) Reg5 (20, 1.0)	Thread1 (T=10, WCET=1.5) Reg1 (10, 0.5) Reg2 (20, 1.0) Thread2 (T=20, WCET=1.8) Reg4 (40, 0.8) Reg5 (20, 1.0) Thread3 (T=2, WCET=1.5) Reg3 (2, 0,5) Reg6 (2, 1.0)	Thread1 (T=10, WCET=0.5) Reg1 (10, 0.5) Thread2 (T=20, WCET=1.0) Reg2 (20, 1.0) Thread3 (T=2, WCET=0.5) Reg3 (2, 0,5) Thread4 (T=40, WCET=0.8) Reg4 (40, 0.8) Thread5 (T=20, WCET=1.0) Reg5 (20, 1.0) Thread6 (T=2, WCET=1.0) Reg6 (2, 1.0)
Cheddar analysis results: ***Feasibility test based on the processor utilization factor:*** – Processor utilization factor with deadline is 2.4 – In the pre-emptive case, with RM, cannot prove that the task set is schedulable: processor utilization factor is more than 1.0 ***Feasibility test based on worst case task response time:*** Processor utilization exceeded: cannot compute bound on the response time with this task set.	**Cheddar analysis results:** ***Feasibility test based on the processor utilization factor:*** – Processor utilization factor with deadline is 1.52 – In the pre-emptive case, with RM, cannot prove that the task set is schedulable: processor utilization factor is more than 1.0 ***Feasibility test based on worst case task response time:*** Processor utilization exceeded: cannot compute bound on the response time with this task set.	**Cheddar analysis results:** ***Feasibility test based on the processor utilization factor:*** – Processor utilization factor with period is 0.99 – 200 µs are unused in the base period. – In the pre-emptive case, with RM, the task set is schedulable. ***Feasibility test based on worst case task response time:*** Bound task response time: Thread2 ⇒ 19800 µs Thread1 ⇒ 6000 µs Thread3 ⇒ 1500 µs	**Cheddar analysis results:** ***Feasibility test based on the processor utilization factor:*** – Processor utilization factor with period is 0.92 – 3200 µs are unused in the base period. – In the pre-emptive case, with RM, the task set is schedulable. ***Feasibility test based on worst case task response time:*** Bound task response time: Thread4 ⇒ 15800 µs Thread2 ⇒ 10000 µs Thread5 ⇒ 6000 µs Thread1 ⇒ 2000 µs Thread3 ⇒ 1500 µs Thread6 ⇒ 1000 µs

region execution in threads revolves around a modification of the thread-pool design, where regions are the units of work; developers define at design time both the number of threads that execute the application, as well as their computational load, by assigning the regions they will execute; and a cyclic executive inside each threads manages region execution. The regularity of this scheme enables the performance of schedulability analysis, and thus its use in applications with timing requirements.

The deployment model has also proven to be essential in the approach, since it separates application architecture from its deployment in terms of nodes, processes and threads, enabling the separation of roles in the development team, as well as the rapid testing of different deployment scenarios. This model also enables us to determine the computational resources required by the application, as well as to estimate memory consumption, which is very important in embedded systems. Unlike other reviewed component models, C-Forge does not enforce a rigid association between components and processes/threads, but it can be easily configured thanks to the deployment model. It also means that C-Forge components are not forced to use a communication software for message exchange in all scenarios, but only on those where the application is distributed in more than one node.

Regarding future works, we are currently enhancing the deployment model for supporting multi-core systems, and end-to-end transactions specification, as well as automatically generating and testing different deployments, in order to find an optimum one. We are also interested in generating a less pessimistic analysis file, since we now assume that components are always executing the states with the longest computation, which cannot be possible in some cases. A more exhaustive analysis of the state-machines will enable us to make less pessimistic analysis.

References

[1] M. Ben-Ari (2006), *Principles of Concurrent and Distributed Programming*, Addison-Wesley.

[2] D. Alonso, F. Sánchez-Ledesma, P. Sanchez, J. A. Pastor, and B. Álvarez (2014), *Models and frameworks: a synergistic association for developing component-based applications*, The Scientific World Journal, pp. 1–17.

[3] D. Alonso, F. Sánchez-Ledesma, P. Sánchez and B. Álvarez (2014), *Embedded and Real Time System Development: A Software Engineering Perspective*, A flexible framework for Component based Application with Real-Time Requirements and its Supporting Execution Framework, pp. 3–22, Springer-Verlag.

[4] J. A. Pastor, D. Alonso, P. Sanchez and B. Álvarez., *Towards the definition of a pattern sequence for real-time applications using a model-driven engineering approach*, The Scientific World Journal, pp. 1–17.

[5] J. Bezivin (2005), *On the unification power of models*, Journal of Systems and Software, pp. 171–188.

[6] F. J. Ortiz, C. Insaurralde, D. Alonso, F. Sanchez and Y. Petillot (2014), *Model-driven analysis and design for software development of autonomous underwater vehicles*, Robotica, pp. 1–20.

[7] I. Crnkovic, S. Sentilles, A. Vulgarakis and M. R. V. Chaudron (2011), *A classification framework for software component models*, IEEE Trans. Software Eng., pp. 37(5):593–615.

[8] A. Vulgarakis, J. Suryadevara, J. Carlson, C. Seceleanu and P. Pettersson (2009), *Formal semantics of the procom real-time component model*, Proc. of the 35^{th} Euromicro Conference on Software Engineering and Advanced Applications, pp. 478–485, IEEE.

[9] P. Feiler and D. Gluch (2012), *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Languages*, Addison Wesley Professional.

[10] K. Hanninen et al (2008), *The rubus component model for resource constrained real-time*, International Symposium on Industrial Embedded Systems, pp. 177–183, IEEE.

[11] A. Cicchetti et al (2012), *Chess: a model-driven engineering tool environment for aiding the development of complex industrial systems*, in Proc. of the 27^{th} IEEE/ACM International Conference on Automated Software Engineering, pp. 362–365, ACM Press.

[12] Patricia López Martínez, L. Barros and J. M. Drake (2013), *Design of component-based real-time applications*, Journal of Systems and Software, pp. 86(2):449–467.

[13] P. Feiler and D. Gluch (2000), *Designing Concurrent, Distributed, and Real-Time Applications with UML*, Object Technology, Addison-Wesley.

[14] F. Singhoff, A. Plantec, P. Dissaux and J. Legrand (2009), *Investigating the usability of real-time scheduling theory with the cheddar project*, Journal of Real Time Systems, pp. 43(3):259–295.