Towards the Definition of a Pattern Sequence for Real-Time Applications Using a Model-Driven Engineering Approach*

Juan Ángel Pastor, Diego Alonso, Pedro Sánchez, and Bárbara Álvarez

Division of Systems and Electronic Engineering (DSIE)
Technical University of Cartagena, Campus Muralla del Mar, E-30202, Spain
juanangel.pastor@upct.es

Abstract. Real-Time (RT) systems exhibit specific characteristics that make them particularly sensitive to architectural decisions. Design patterns help integrating the desired timing behaviour with the rest of the elements of the application architecture. This paper reports a *pattern story* that shows how a component-based design has been implemented using periodic concurrent tasks with RT requirements. The Model-Driven Software Development (MDSD) approach provides the theoretical and technological support for implementing a pattern-guided translation from component-based models to object-oriented implementations. This work has been done and validated in the context of the development of robotic applications.

1 Introduction

There is a well established tradition of applying *Component Based Software Development* (CBSD) [19] principles in the robotics community, which has resulted in the appearance of several toolkits and frameworks for developing robotic applications [15]. The main drawback of such frameworks is that, despite being *Component-Based* (CB) in their conception, designers must develop, integrate and connect these components using *Object-Oriented* (OO) technology. The problem comes from the fact that CB designs require more (and rather different) abstractions and tool support than OO technology can offer. For instance, the lack of explicit "required" interfaces makes compilers impossible to assure that the components are correctly composed (linked). Also, component interaction protocols are not explicitly defined when using an OO language. Moreover, most of these frameworks impose the overall internal behaviour of their components, and therefore they lack of formal abstractions to specify it. In this way, framework components have so many platform-specific details that it is almost impossible to reuse them among frameworks [12]. In particular, robotic systems are reactive systems with RT requirements by their very nature, and most of the frameworks for robotics do not provide mechanisms for managing such requirements. The *Model-Driven Software Development* (MDSD) paradigm [18] can provide the

* This work has been partially supported by the Spanish CICYT Project EXPLORE (ref. TIN2009-08572), and the Fundación Séneca Regional Project COMPAS-R (ref. 11994/PI/09).

J. Real and T. Vardanega (Eds.): Ada-Europe 2010, LNCS 6106, pp. 167–180, 2010.

theoretical and practical support to overcome the above drawbacks. MDSD is starting to catch the attention of the robotics community [6] mainly due to the very promising results it has already achieved in other application domains (e.g., automotive, avionics, or consumer electronics, among many others) in terms of improved levels of reuse, higher software quality, and shorter product time-to-market [13].

In our opinion, it is needed a new CBSD approach for robotic software development that: (1) considers components as architectural units, (2) enables components to be truly reusable among frameworks (by separating their design from the implementation details), and (3) considers application timing requirements. In the context of the robotics domain and the aforementioned technologies (CBSD and MDSD), the authors have defined the *3-View Component Meta-Model* (V^3CMM) [10] as a platform-independent modelling language for component-based application design. V^3CMM is aimed at allowing developers to model high-level reusable components, including both their structural and behavioural facets. Such behavioural facets are modelled by means of state-charts and activity diagrams. Though these diagrams abstract designers away from run-time issues (such as the number of tasks, the concurrency model, etc.), these details must be realised in further steps. The problem then is how to translate V^3CMM models into executable code that, on the one hand, reflects the behaviour of the original V^3CMM models, and, in the other hand, is organised in a set of tasks compliant with the application-specific timing requirements.

This paper describes the approach we have taken for solving this problem, and the results we have obtained so far. This approach revolves around the definition of a framework that provides the required run-time support, and a set of 'hot-spots' where a model-to-code transformation will integrate the code generated from the V^3CMM models describing the concrete application. The patterns that have been selected to design such framework are described as a ***pattern story*** [8].

In short, the main contributions of the work presented in this paper are:

- An object-oriented interpretation of CBSD architectural concepts.
- A framework supporting such interpretation and taking into account timing requirements.
- A rationale of the framework design following a pattern story.

The remainder of this paper is organised as follows. Section 2 provides a general overview of the overall approach of the paper. Section 3 describes the patterns that comprise the architecture of the developed framework. Section 4 is devoted to detail the main issues of the dynamic of the applications generated using the framework. Section 5 relates this work with other proposals found in the literature. And finally, Section 6 discusses future work and concludes the paper.

2 General Overview of the Approach

The proposed development process starts with the modelling of the application architecture using the V^3CMM language. For the purpose of this paper any language enabling the modelling of components (such as UML, SysML or other UML profiles) could have been used. The reasons why we decided to develop a new modelling

language (V^3CMM) are outside the scope of this paper and are described in [4], but, for the curious reader, they are mainly related to keeping a strong control over the concepts considered in the language and their semantics, and for easing model transformations.

V^3CMM comprises three complementary views, namely: (1) a *structural* view, (2) a *coordination* view for describing the event-driven behaviour of each component (this view is based on UML state-charts), and (3) an *algorithmic* view for describing the algorithm executed by each component depending on its current state (this view is based on a simplified version of UML activity diagrams). V^3CMM enables describing the architecture (structure and behaviour) of CB applications, but provides no guidelines for developing implementations. Therefore, as stated in the introduction, it is necessary to provide designers with tools that enable them to generate the program code from these high level abstractions. This code must take into account application-specific timing requirements, and reflect the behaviour of the original V^3CMM models.

The most important and challenging implementation issue is related to the implementation of real-time constraints in the framework structure, and among it, how many tasks must be created and how to distribute the component activities among them. Taking into account that each system might need a different task scheme (e.g. number of tasks, periods, deadlines, etc.), and that even given a system, this scheme can greatly vary (due to different execution resources, varying timing requirements, change of algorithms, etc.), a very flexible solution is required. This solution should allow task derivation from V^3CMM models, and specifically from the coordination view, since this view models both the concurrent behaviour (in the form of orthogonal regions), and the timing requirements of the algorithms (i.e. execution time, period, deadline, etc.) executed by the component.

Fig. 1 shows the pursued ideal solution, where it is possible to 'arbitrarily' allocate the activities associated to the states of the state-chart of a V^3CMM component to a set of tasks. It is worth clarifying that, in the rest of the paper, when we mention 'activity' we really mean activity diagram. The solution must not force a direct one-to-one relationship between components and execution tasks, but instead allow for more flexible schemes. In a given system, activities allocation would be driven by the RT requirements of each activity, the selected scheduling algorithms, different heuristics, execution platform constraints, etc. As these requirements, algorithms, heuristics and constraints could greatly differ from system to system, a great flexibility is then required for allocating activities to tasks. The proposed solution (see Fig. 2), detailed in the following section, considers that application code can be classified into the following three sets:

CS1. Code that provides a run-time support compliant with the domain specific requirements. Normally, this involves making trade-offs among the requirements of the application domain. For instance, in domains where hard real-time is usually needed (e.g. robotics, embedded systems, avionics, etc.), CS1 code should support characteristics such as different concurrency policies, real-time scheduling, event processing and signalling, reliability, memory management, etc., even at the cost of sacrificing other characteristics.

CS2. Code that provides an OO interpretation of the V^3CMM concepts. For instance, how components and ports are mapped to objects, state-charts implementation, port communication issues, etc.

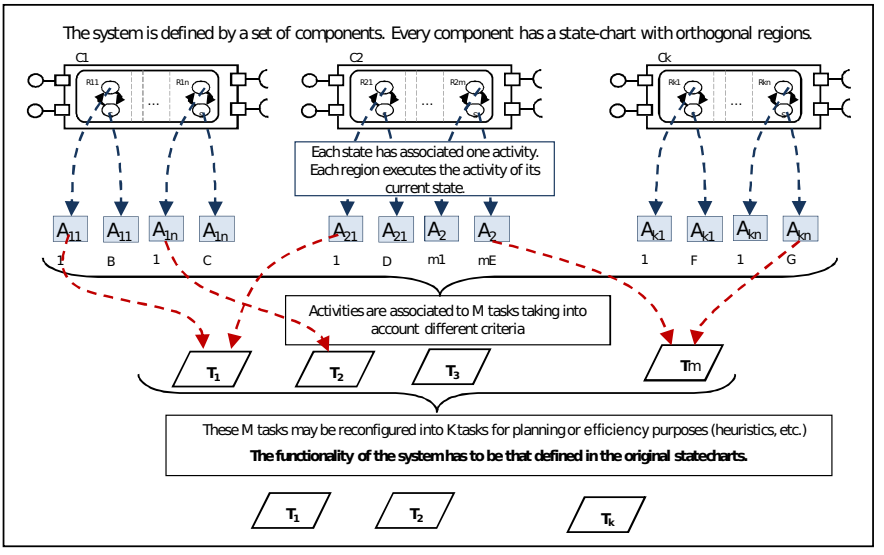


Fig. 1. Ideal scenario for allocating activities to tasks

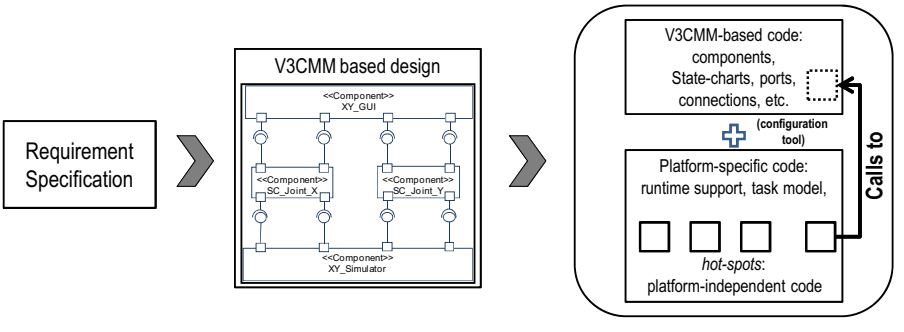


Fig. 2. Global view of the development process

CS3. Code corresponding to the application functionality, described by V^{3}CMM models.

These three code sets are arranged in a way that code sets CS1 and CS2 constitute a framework, where CS3 must be integrated in order to obtain the final application. The hot-spots for specialising the framework are defined in CS2. CS2 also serves for minimising the coupling between CS3 and CS1, enabling their separate evolution and reuse. As long as the interpretation of the V^{3}CMM concepts (CS2) remains the same, it would be possible to reuse the run-time support (CS1) in different applications (CS3). And, even more interesting, it would be possible to select a given run-time support (CS1) for the same application functionality (CS3), depending on the application domain requirements. CS1 and CS2 have been designed and implemented manually,

following a set of design patterns, while CS3 is automatically generated from the V^{3}CMM models and integrated in the framework by means of a model transformation.

3 Global Architecture of the Generated Applications

This section explains how the framework design (shown in Fig. 3) has been obtained starting from its main requirements. Some of the most important patterns that comprise the pattern story are highlighted in the figure by the classes that fulfil the roles defined by such patterns. The correspondence between the classes and the code sets will be described at the end of the section where it will be better understood. Due to space limitations, this section is focused on the three main challenges that were faced when designing the framework, namely: how to allocate state activities to tasks, how to implement state-charts, and finally how to manage the component internal data.

Among the aforementioned challenges, the main one is how to allocate the activities associated to the states of the state-charts to different tasks. In order to achieve it, the COMMAND PROCESSOR architectural pattern [7] and the highly coupled COMMAND pattern have been selected. The COMMAND PROCESSOR pattern separates service requests from their execution. For this purpose, the pattern defines a task (the command processor) where the requests are managed as independent objects (the commands). Each activity associated to a state is implemented as a separate command, which can be allocated to any command processor. The roles defined by these two patterns are realised by the classes **`Activity_Processor`** and **`State_Activity`**, respectively (see Fig. 3). The COMMAND PROCESSOR pattern provides the required flexibility, since it imposes no constraints over activity subscription, number of activities, activity duration, concurrency scheme, etc. Nevertheless, this pattern has the following liabilities:

- It leads to a large number of subclasses since it is necessary to define a subclass of **`State_Activity`** for each and every activity defined in the V^{3}CMM models. As these subclasses will be generated by the model transformation, it is not a relevant drawback for this work.
- Loss of performance due to the additional indirection levels. This loss is paid off given the obtained flexibility.
- The component internal data can be simultaneously accessed by activities belonging to the same component but allocated to different tasks by the implementation. It will be necessary to synchronize such concurrent accesses, as detailed below.

The second challenge is how to interpret and implement state-charts in a way that enables its integration in the scheme defined by the aforementioned COMMAND PROCESSOR pattern. Providing an implementation that considers all the possibilities offered by hierarchical states and orthogonal regions is an extremely complex issue, which can be afforded by following different techniques [16]. In our case, we decided that both regions and states should be treated homogeneously, and their activities allocated to different command processors without knowing (or caring about) their type. This need is fulfilled by using a simplified versions of the COMPOSITE pattern. The roles defined by this pattern are realised by the classes **`State, Orthogonal_Region`**

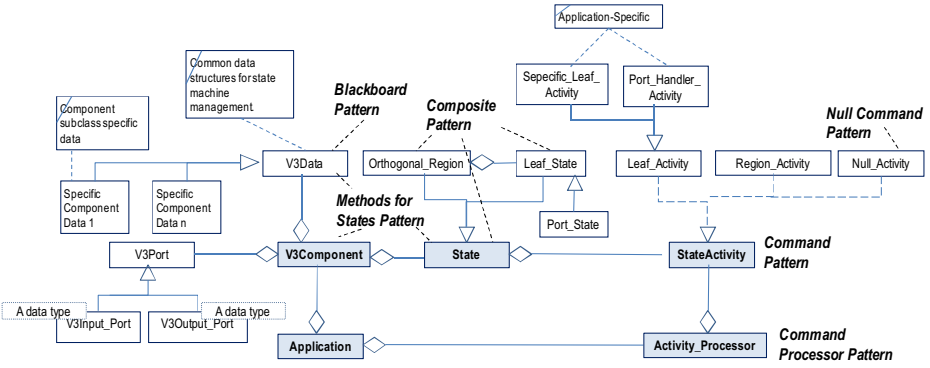


Fig. 3. Simplified class diagram of the generated code

and **`Leaf_State`**. The state-chart is managed following the METHODS FOR STATES pattern [7], where the instances of the classes representing the state-chart are stored in a hash table, while orthogonal regions store the keys of their leaf states in order to manage them. To shorten the implementation of the first working version of the framework, we only considered orthogonal regions comprising non-hierarchical states. In spite of this limitation, a broad range of systems can be still modelled.

The distinction between states and regions led us to define specific subclasses of **`State_Activity`**. Basically, we needed two hierarchies of subclasses: activities associated to leaf states (represented by the root class **`Leaf_Activity`**), and activities associated to regions (represented by the class **`Region_Activity`**). The latter is aimed at managing the region states and transitions, and thus is provided as part of the framework. The formers are related to (1) the activities defined in the V^{3}CMM models, which are generated by the model transformation and are represented by **`Specific_Leaf_Activity`** subclasses, and (2) activities to manage ports, which are also provided by the framework and are represented by **`Port_Handler_Activity`** subclasses. Following the NULL OBJECT pattern, the **`Null_Activity`** class has been defined in order to smoothly integrate those states that have no associated activity.

The third and last challenge covered in this paper is how to provide concurrent access to the component internal data. This data is organised following the BLACKBOARD pattern. The idea behind the blackboard pattern is that a collection of different tasks can work cooperatively on a common data structure. In this case, the tasks are the command processors mentioned above, and the data comprise input/output port information and the hash table that stores state information. The main liabilities of the BLACKBOARD pattern (i.e. difficulties for controlling and testing, as well as synchronization issues in concurrent applications) are mitigated by the fact that each component has its own blackboard, which maintains a relatively small amount of data. In addition, it is possible to optimize the access to the blackboard in some important cases. For instance, the hash table that stores the component state is accessed following a *1-writer/n-readers* scheme.

The full pattern story comprises eighteen patterns, from which only the most important ones from the point of view of the global architecture have been described. There are other patterns, such as OBSERVER, COPIED VALUE, DATA TRANSFER OBJECT, TEMPLATE METHOD, STRATEGY, PROXY, etc., which are not shown in the

figure since the roles defined by them cannot be clearly identified in Fig. 3 and there is no space left to explain how they have been used in the framework design.

The classes shown in Fig. 3 fall into the code sets described in the previous section as follows:

CS1: Run-time support. This set comprises the classes **`Activity_Processor`** and **`State_Activity`**, which have been manually coded.

CS2: Interpretation of V^{3}CMM concepts. This set comprises almost the rest of the classes shown in Fig. 3: **`State_Activity`**, **`Leaf_State`**, **`Orthogonal_Region`**, **`State`**, **`Port_State`**, **`V3Input_Port`**, **`V3Output_Port`**, **`Region_Activity`**, **`Leaf_Activity`**, **`V3Data`**, and **`V3Component`**. Notice that **`State_Activity`** is the link between CS1 and CS2. Classes **`V3Input_Port`** and **`V3Output_Port`** are defined as generics (or templates), which are instantiated with the concrete messages types the ports exchange. The classes comprising CS2 have been manually coded, and define the main framework hot-spots. Although the framework is mainly specialised by sub-classing them (and therefore it can be considered a white-box framework), it provides some concrete subclasses. These subclasses, which are enumerated below, are defined for implementing the port behaviour, and are meant to be directly instantiated.

- **`Port_State`**: concrete leaf states modelling the state of a given port.
- **`Port_Handler_Activity`**: concrete strategies for managing input ports. By default, port to port communication is implemented following the asynchronous without response policy, since it is the basic block for distributed systems and for designing more complex interaction schemes.
- **`Region_Activity`**: concrete strategy for managing orthogonal regions.

CS3: Application functionality. This set integrates (1) new subclasses of the hot-spots defined in CS2, and (2) instances of these new subclasses and of the classes comprising CS1 and CS2. All these classes and instances are automatically generated by a model-to-code transformation from the V^{3}CMM models. The most relevant elements of this set, generated for each component, are:

- Data types representing the messages exchanged by components through their ports.
- Instances of the **`V3Input_Port`** and **`V3Output_Port`** generic classes with the concrete messages that the ports exchange. Notice that these instances represent only the static structure of ports. Their dynamic behaviour is defined in the item below.
- New orthogonal regions (instances of **`Orthogonal_Region`**), added to the original state-chart in order to manage the behaviour of the component ports. These orthogonal regions comprise leaf states (instances of **`Port_State`**), as well as the activities corresponding to these states (instances of **`Region_Activity`** and **`Port_Handler_Activity`**, respectively). This design decision provides regularity and flexibility to the framework, since (1) all regions, both those derived from the V^{3}CMM models and those added to manage ports, are treated homogeneously by the command processors, and (2) ports handling is explicit and can be allocated to different tasks, depending on the timing requirements.

– A subclass of **`V3Data`** comprising the specific component data as described previously.
– An instance of the class **`V3Component`**. This object acts as a container for all the previous elements.

Finally, when all the components have been generated, the transformation connects the components ports, creates a set of command processors, and allocates activities to them.

4 Allocation of Activities to Tasks

This section deals with relevant aspects of the application dynamics and with the criteria followed by the transformation to allocate activities to command processors. From a dynamic point of view, it is important to remark that command processors can execute activities defined in the state-charts of different components. Among

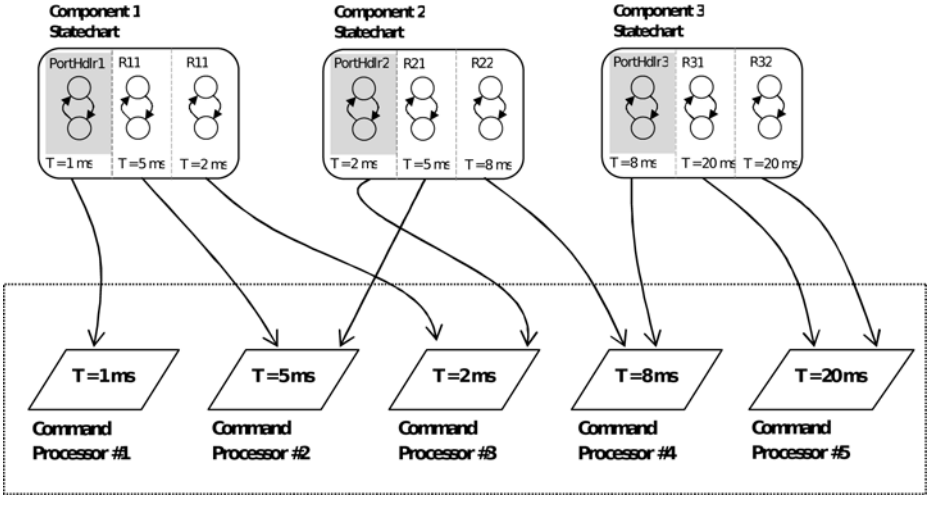


Fig. 4. Sample allocation scenario. From state-charts to command processors

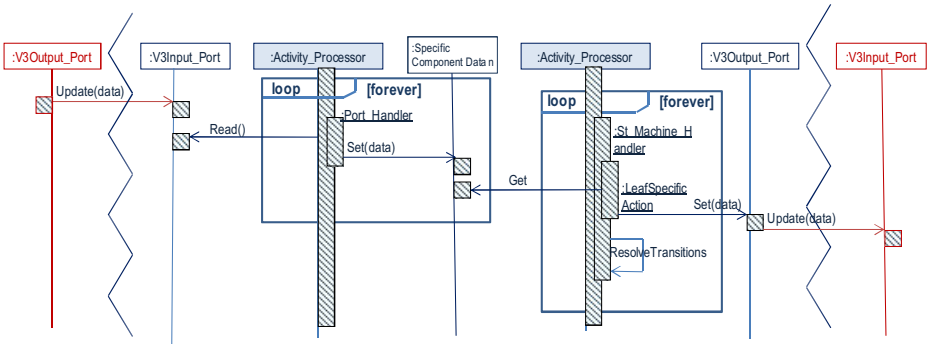


Fig. 5. A sequence diagram with a typical execution scenario

the many feasible possibilities for allocating activities to command processors, the main criteria is based on activity periods, since it facilitates the further schedulability analysis of the command processors. Fig. 4 shows a sample allocation scenario from the activities defined in state-charts to the tasks defined in command processors. This scenario comprises three components, each of them associated to a state-chart with three orthogonal regions, including the region added for port management (the shaded one). For the shake of simplicity, the example assumes that all the activities associated to the states contained in a given region have the period of its associated region activity. Notice that command processors 2 and 3 access the internal data corresponding to component 2 concurrently.

A typical execution scenario is shown in the sequence diagram of Fig. 5, which comprises the communication among three components. A **`V3Input_Port`** object stores the data received from an output port. Then, a task (i.e. an **`Activity_Procesor`**) will asynchronously put this data into a **`V3Data`** object (global to the component). Afterwards, another task will asynchronously process the incoming data depending on the current component state. As a consequence of this processing, state transitions in one or more regions of the component may occur. Moreover, this processing includes the execution of the activities of the set of current active states, and the updating of new data in output ports (sub-program **`set(data)`** in Fig. 5).

Code listing 1 shows an excerpt of the Ada specification of the **`Activity_Processor`**, which has been implemented as a generic package, while code listing 2 shows the body of the task corresponding to a command processor. The main characteristics of this generic package are the following:

- The priority of the task contained in the package body is assigned according to both the timing requirements of the subscribed activities and the chosen scheduling algorithm. As this data is known before the transformations generate the code, it is possible to derive the priority of each **`Activity_Processor`**. Thus, a fixed priority static scheduling algorithm can always be used if required.
- The transformation takes into account that a task may include activities with different periods. The period assigned by the transformation to each task (**`Activity_Processor`**) is equal to the lowest period of its subscribed activities.

Listing 1. Code excerpt of the specification of the **`Activity_Processor`** generic package

```
1  generic
2    Listener        : access I_Activity_Processor_Listener'Class;
3    Name            : Unbounded_String;
4    Worker_Priority : System.Any_Priority;
5  package Common.Activity_Processor is
6    function Get_Name return Unbounded_String;
7    procedure Set_Priority (Priority : System.Any_Priority);
8    function Get_Priority return System.Any_Priority;
9    procedure Start;
10   procedure Stop;
11   procedure Set_Period (Period: Time_Span);
12   function Get_Period return Time_Span;
13   procedure Add_Activity (Act : access I_State_Activity'Class);
14   procedure Del_Activity (Act : access I_State_Activity'Class);
15 end Common.Activity_Processor;
```

Listing 2. Code excerpt of the body of the **Activity_Processor** generic package showing the task corresponding to a command processor

```
task body Worker is
      Next_Exec   : Time := Clock;
      Iterator    : P_Dll.Cursor;
      Element     : State_Activity_All;
   begin
      Suspend_Until_True (Start_Lock);
      while Continue loop
            delay until Next_Exec;
            Next_Exec := Next_Exec + Period;
            Iterator := Activity_List.First;
            while (P_Dll.Has_Element (Iterator)) loop
                  Element := P_Dll.Element (Iterator);
                  Element.Execute_Tick;
                  P_Dll.Next (Iterator);
            end loop;
      end loop;
   end Worker;
```

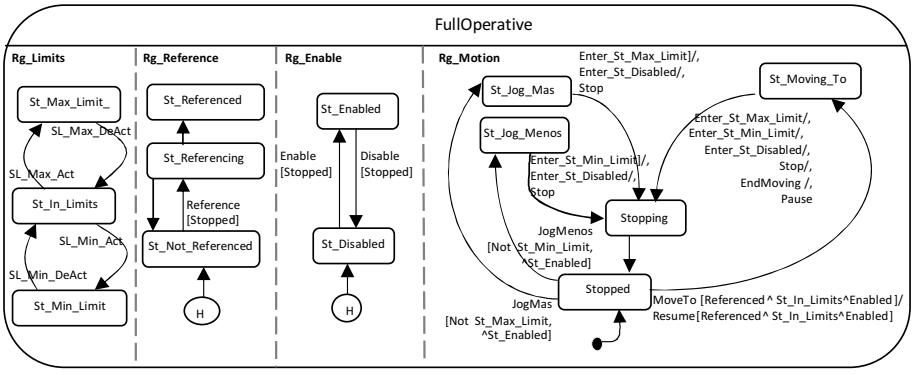
It is important to highlight that activities may execute periodically or not. When activities are sporadic, the period attribute represents the minimum separation between two consecutive executions. The activities are executed in the same order as they have been subscribed to the **Activity_Processor**, although any alternative policy could have been chosen. Tasks are executed by the operating system according to the chosen scheduling algorithm.

– The sub-program **Add_Activity** enables subscribing activities to tasks.

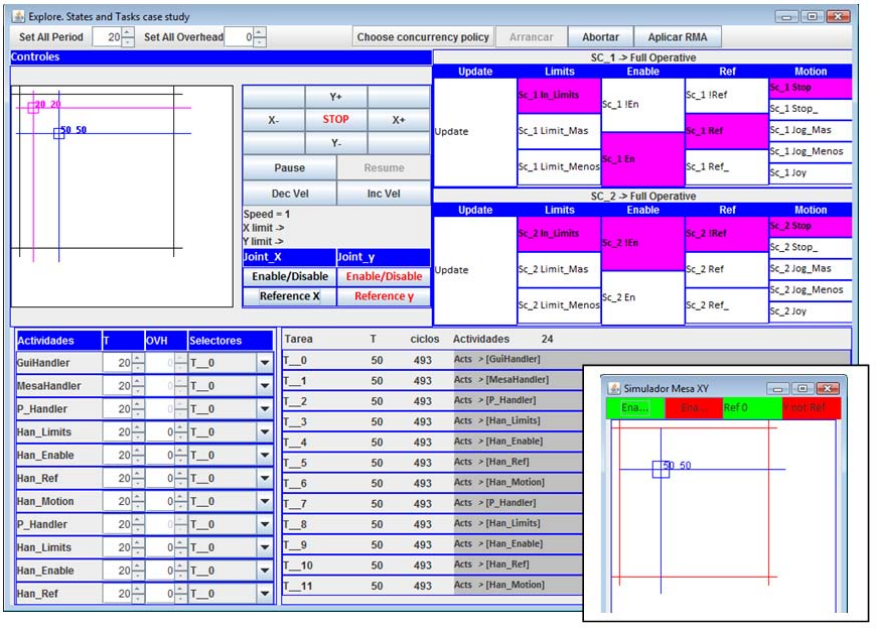
This design assumes that activities are defined to have an execution time as short as possible to simplify scheduling. When an algorithm includes a big number of iterations or considers a continuous control action, then the activity should be divided into a set of sub-activities with a bounded execution time (for example, an algorithm step or a discrete control action).

The framework design requirements impose many constraints to the flexibility provided by the COMMAND PROCESSOR pattern. These constraints are mainly enforced by the real-time nature of the application domain. Some examples of the impact of these requirements are that command processors do not spawn new tasks to execute subscribed activities (which is permitted by the pattern), and that it is not allowed subscribing activities to command processor or modifying periods at execution time, to mention a few.

In order to validate the framework we have developed (1) several case studies, and (2) a tool to monitor the execution of each application and to change the number of command processors, and the allocation of activities to tasks. The tool enables us to experiment with the number of command processors and different activities allocation criteria, by reconfiguring the application generated by the transformation. The case study shown in this paper corresponds to a Cartesian robot, developed in the context of a research project (European Union's Fifth Framework Programme, Growth, G3RD-CT-00794) [10]. Fig. 6 shows an excerpt of the state-chart corresponding to the controller of one of the robot joints, and the part of the aforementioned configuration tool in charge of configuring command processors.



a) State-chart modelling the behaviour of a joint of the Cartesian robot.



b) Configuration tool. The left part enables users to set activity periods, an estimated execution time, and to allocate activities to tasks. The right part shows, for each task, its execution period, number of execution cycles, and the activities allocated to it.

Fig. 6. Case study of a Cartesian robot and the reconfiguration tool

5 Related Work

As said in the introduction, there is a well established tradition of applying CBSD principles for developing robotic applications. However, there are not many initiatives for applying MDSD principles to robotic software development. In general, existing robotic frameworks cannot be considered to be model-driven, since they have no meta-model foundation supporting them. Among the main examples of applying the

MDSD approach to robotics is the work related to the Sony Aibo robot presented in [5]. Another initiative, described in [11], revolves around the use of the Java Application Building Center (jABC) for developing robot control applications. Although jABC provides a number of early error detection mechanisms, it only generates Java code and, thus, its applicability to systems with real-time requirements is rather limited. Finally, Smartsoft [17] is one of the most interesting initiatives for applying a MDSD approach to robotic software development. Nevertheless, as far as we know, none of these initiatives considers real-time issues.

The current state of the application of MDSD to robotic software development contrasts with what happens in other similar domains, where big efforts are being carried out in this line. For instance, the ArtistDesign Network of Excellence on Embedded Systems Design [1] and the OpenEmbeDD [3] project address highly relevant topics regarding real-time and embedded systems, while the automotive industry has standardised AUTOSAR [2] for easing the development of software for vehicles.

As Buschmann et al. [7] states, not all domains of software are yet addressed by patterns. However, the following domains are considered targets to be addressed following a pattern-language based development: service-oriented architectures, distributed RT and embedded systems, Web 2.0 applications, software architecture and, mobile and pervasive systems. The research interest in the RT system domain is incipient and the literature is still in the form of research articles. A taxonomy of distributed RT and embedded system design patterns is described in [9], allowing the reader to understand how patterns can fit together to form a complete application. The work presented in this paper is therefore a contribution to the definition of pattern languages for the development of this kind of systems with the added value of forming part of a global MDSD initiative.

6 Conclusions and Future Research Lines

This paper has described an approach to provide a run-time support (framework) to a component-based approach for modelling RT applications. To do that, it has been necessary to provide an OO interpretation of the high-level architectural concepts defined in V^3CMM (components, ports, state-charts, etc.), taking into account real-time requirements. The proposed solution is not general nor closed to future improvements, but it is a stable and validated starting point for further development.

The adoption of a pattern-driven approach has greatly facilitated the design of such framework. In addition, the selected patterns have been described like a pattern story. A further step would be the definition of a ***pattern sequence***, which comprises and abstracts the aforementioned pattern story, so that developers can use it in other applications as long as they share similar requirements. With several pattern stories and pattern sequences it would be possible to define a true ***pattern language*** for a given domain, which gives a concrete and thoughtful guidance for developing or refactoring a specific type of system. The greatest difficulties in reporting this story have been how to synthesize in a few pages the motivations for choosing the patterns that have been used, and the lack of consensus about the best way of documenting pattern stories.

The characteristics of the Ada language that have revealed most useful for the development of the framework have been its mature concurrency facilities, strong typing, the generics mechanism, and the flexibility provided by packages in order to organise and encapsulate component structure. In addition, the new container library has proven very useful for implementing the internal blackboard of each component. The main difficulty comes from the fact that Ada is an extensive language and requires a deep understanding of the its mechanisms in order to successfully combine them.

Regarding future research lines, we are currently working on extending the framework with additional capabilities following a pattern-driven approach. Among these extensions, it is worth mentioning the following: (1) component distribution, (2) testing and adding heuristics for activities allocation and task grouping, (3) refining and improving the patterns used for implementing hierarchical and timed state-charts, and (4) comply with the Ravenscar profile for designing safety-critical hard real-time systems. Assessing timing requirements fulfilment in an automated way is also very important, and thus we plan to study strategies to generate analysis models for different scheduling analysis tools. The usage of the UML profile for MARTE [14] as a mechanism to formalize the models involved is also an approach to be explored.

References

[1] ArtistDesign - European Network of Excellence on Embedded Systems Design (2008-2011), http://www.artist-embedded.org/

[2] AUTOSAR: Automotive Open System Architecture (2008-2011), http://www.autosar.org/

[3] OpenEmbeDD project, Model Driven Engineering open-source platform for Real-Time & Embedded systems, (2008-2011), http://openembedd.org/home_html

[4] Alonso, D., Vicente-Chicote, C., Ortiz, F., Pastor, J.: V^3CMM: a 3-View Component Meta-Model for Model-Driven Robotic Software Development. Journal of Software Engineering for Robotics (JOSER) 1(1), 3–17 (2010)

[5] Blanc, X., Delatour, J., Ziadi, T.: Benefits of the MDE approach for the development of embedded and robotic systems. Application to Aibo. In: Proc. of the 3rd National Conference on Control Architectures of Robots (2007)

[6] Bruyninckx, H.: Robotics Software: The Future Should Be Open. IEEE Robot. Automat. Mag. 15(1) (2008) ISSN 1070-9932

[7] Buschmann, F., Henney, K., Schmidt, D.: Pattern-Oriented Software Architecture. A Pattern Language for Distributed Computing, vol. 4. John Wiley and Sons Ltd., Chichester (2007) ISBN 0471486485

[8] Buschmann, F., Henney, K., Schmidt, D.: Pattern-Oriented Software Architecture. On Patterns and Pattern Languages, vol. 5. John Wiley and Sons Ltd., Chichester (2007) ISBN 0471486485

[9] Dipippo, L., Gill, C.: Design Patterns for Distributed Real-Time Embedded Systems. In: Real-Time, Springer, Heidelberg (2009) ISBN 0387243577

[10] Iborra, A., Alonso, D., Ortiz, F.J., Franco, J.A., Sánchez, P., Álvarez, B.: Design of service robots. IEEE Robot. Automat. Mag., Special Issue on Software Engineering for Robotics 16(1) (2009), doi:10.1109/MRA.2008.931635, ISSN 1070-9932

[11] Jorges, S., Kubczak, C., Pageau, F., Margaria, T.: Model Driven Design of Reliable Robot Control Programs Using the jABC. In: Proc. Fourth IEEE International Workshop on Engineering of Autonomic and Autonomous Systems EASe 2007, pp. 137–148. IEEE, Los Alamitos (2007)

[12] Makarenko, A., Brooks, A., Kaupp, T.: On the Benefits of Making Robotic Software Frameworks Thin. In: Proc. of the IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS 2007). IEEE, Los Alamitos (2007)
[13] OMG: MDA success stories (2008), Available online: http://www.omg.org/mda/products_success.html
[14] OMG: UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems, formal/2009-11-02 (2009), http://www.omg.org/spec/MARTE/1.0
[15] Robot Standards and Reference Architectures (RoSTa), Coordination Action funded under EU's FP6: http://wiki.robot-standards.org/index.php/Current_Middleware_Approaches_and_Paradigms
[16] Samek, M.: Practical UML Statecharts in C/C++, Second Edition: Event-Driven Programming for Embedded Systems. Newnes (2008), ISBN 0750687061
[17] Schlegel, C., Hassler, T., Lotz, A., Steck, A.: Robotic software systems: From code-driven to model-driven designs. In: Proc. International Conference on Advanced Robotics ICAR 2009, pp. 1–8. IEEE, Los Alamitos (2009)
[18] Stahl, T., Völter, M.: Model-Driven Software Development: Technology, Engineering, Management. Wiley, Chichester (2006)
[19] Szyperski, C.: Component software: beyond object-oriented programming. A-W, 2nd edn. (2002), ISBN 0201745720