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

Juan Ángel Pastor, Pedro Sánchez, Diego Alonso, 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 decissions. 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. This work has been done in the context of the development of robotic applications using a Model-Driven Software Development (MDSD) approach. In this context the model-to-code transformations are designed taking into account both the system requirements and the patterns that satisfy them. MDSD provides the conceptual technology for implementing a pattern-guided transition from component-based models to object-oriented implementations. The results of applying the described story of patterns are shown by an application that initializes, configures and schedules the execution of platform-specific components.

1 Introduction

There is a well established tradition of applying *Component Based Software Development* (CBSD) [16] principles in the robotics community, which has resulted in the appearance of several toolkits and frameworks for developing robotic applications [13]. 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 (or 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 behavior of their components and therefore they lack of formal abstractions to specify it. In this way, robotic framework components have so many platform-specific details that it is almost impossible to reuse the aforementioned components among frameworks [11]. Besides, robotic systems are reactive systems with RT requirements and most of these frameworks do

* 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).

not provide mechanisms for managing such requirements. The Model-Driven Software Development approach can provide the theoretical and practical support to overcome the above drawbacks.

Model-Driven Software Development (MDSD) paradigm [15] is starting to catch the attention of the robotics community [5] 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 [12]. MDSD enables designers to focus on domain concepts, relegating implementation details to a secondary level. In MDSD, models are the primary artifacts leading the whole software development process. In this context, the authors have defined the *3-View Component Meta-Model* (V^3CMM) [9] as a platform-independent modeling 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, and to automatically translate these high-level designs into lower level models. These low-level models should be also carefully designed so that, on the one hand, reflect the high-level design and, on the other hand, comply with the specific requirements of each application and execution platform. One way to design this low-level code is to use design patterns. This paper describes how these transformations have been addressed for designing the task structure of the final application.

In this vein, this paper reports a part of a longer ***pattern story***, which describes the patterns that have been selected to allocate activities to execution tasks, as well as a simple tool (called ATA) for distributing component behaviour to tasks. 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 [7]. 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 remainder of this paper is organized as follows. Section 2 provides a general overview of the overall approach of the paper. Section 3 briefly describes the ATA tool and the architectural decisions involved in its design process. Section 4 gives more detail about the architecture of the generated application and presents the design patterns required to understand the part of the pattern story related in this paper. Section 5 details the role of the COMMAND PROCESSOR pattern to allocate activities to tasks. Section 6 relates this work with other proposals found in the literature. And finally Section 7 discusses future work and concludes the paper.

2 General Overview of the Approach

V^3CMM comprises three complementary views, namely: (1) a *structural* view, (2) a *coordination* view for describing the event-driven behavior of each component (based on UML statecharts), and (3) an *algorithmic* view for describing the algorithm executed by each component depending on its current state (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, it is necessary to provide designers of RT applications with tools that allow them to generate the program code from these high level abstractions (like statecharts), following a MDSD approach that transform them into executable programs compliant with the application requirements.

As stated above, one of the most important and challenging issues is how to distribute the components code into tasks. When deriving the task view of a system from the V^{3}CMM coordination view, a straightforward approach is to directly assign a task per orthogonal region. However, the resulting task set could comprise an excesive number of tasks, perhaps difficult to be scheduled according to their timing requirements. Another solution is to consider that every component is executed in a separate task or in the task assigned to its container component. However, this way of allocating tasks is too rigid and hence useless in a RT based designing context. Therefore, it is needed to adopt a more flexible solution that: (1) allows designing statecharts from a task-independent perspective, and (2) to guide the task derivation from statecharts without imposing a direct relationship between tasks and components (or orthogonal regions). In order to provide this functionality a *Graphical User Interface* (GUI) application entitled ATA (*Activities-to-Tasks Allocator*) has been developed.

Fig. 1 shows an ideal scenario where it is possible to 'arbitrarily' assign the activities associated to the states of the statechart of a V^{3}CMM model to a set of tasks. In a given system, this allocation would not be arbitrary done but instead driven by the RT requirements of each activity, the selected scheduling algorithms, different heuristics, execution platform constraints, etc. But since these requirements, algorithms, heuristics and constraints could greatly differ from system to system, a great flexibility is then required for allocating activities to tasks. In this context, our purpose is to give support for the development of CB applications with RT requirements in which:

1. V^{3}CMM is the chosen language for modeling the CB software architecture.
2. Code should be automatically generated from V^{3}CMM models through the development of model-to-code transformations. These transformations generate code with the following structure:
   (a) ***Platform-independent code*** comprising two parts: (i) the infrastructure needed for implementing V^{3}CMM concepts (i.e., ports, components, statechart definitions, communication rules and policies, etc.), and (ii) the application-specific but still platform-independent issues (i.e., specific algorithms and statecharts).
   (b) A ***platform-dependent framework*** that provides the run-time support and the 'hotspots' required to integrate the above platform-independent code. The structure and characteristics of this framework depend on both the application-specific requirements and the platform constraints. The framework can be designed using different design patterns applied in the correct sequence.

Fig. 2 shows an scheme of the overall development process (from requirements to V^{3}CMM models and then to code) and the already described platform-independent and platform-dependent generated code. The generated code does not include views for monitoring the state of the components either mechanisms for changing the initial allocation of activities to tasks.

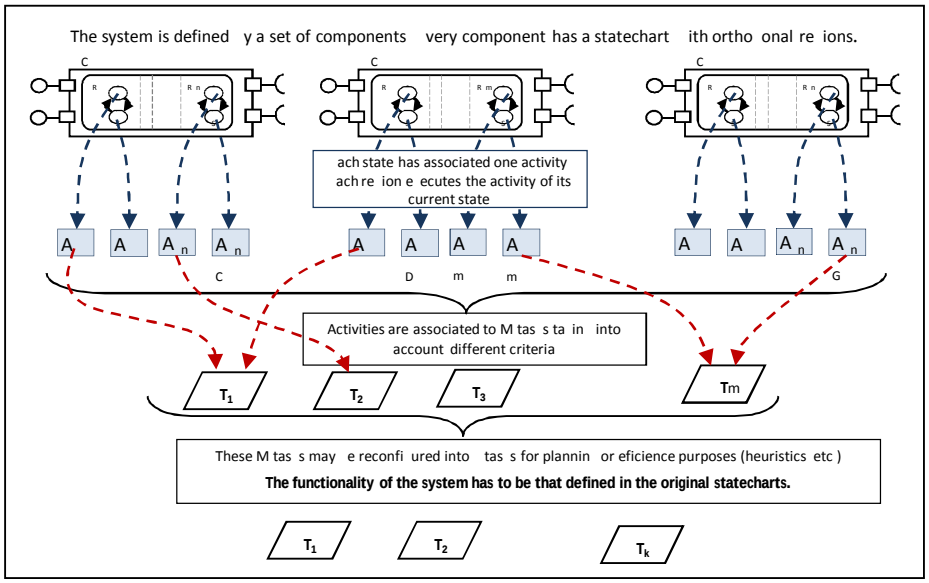

Fig. 1. Ideal scenario for assigning activities to tasks.

3 Overview of the ATA Application

As stated above, the ATA application provides, on the one hand, a view of the inner state of the components and a user interface for controlling them, and on the other hand, some mechanisms for assigning activities to tasks and for specifying some of the timing and scheduling properties of these tasks. In this work, ATA also includes a view of a simulator of a Cartesian robot and some facilities for controlling it. This robot was a part of the results of a research project under the European Union's Fifth Framework Programme (Growth, G3RD-CT-00794) [9]. The purpose of the robot was the cleaning of of a ship's vertical surfaces. It includes a secondary positioning system (XYZ-like table) with three lineal joints, a cleaning tool, and several sensors for detecting movement limits. Fig. 3 shows the different parts of the ATA application. The main criteria that have been followed when designing it are the following:

- Statecharts should show their state in a user interface independent view.
- The user interface should provide access to the joint control commands. The user interface should show the state of the simulator depending on the selected actualization period.
- The simulator should show the behavior of the real device. This includes joint enabling and referencing, and commands for position and speed settings.
- The simulator should show its state in a user interface independent view.
- It should be possible to set from the user interface both activity periods and the correspondences between activities and tasks.

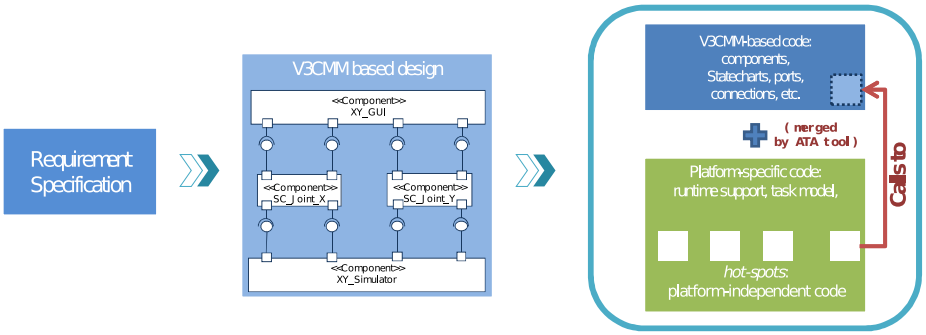

Fig. 2. Global view of the development process.

As shown in Fig. 3, ATA includes a separate simulator view for the above mentioned XYZ table (bottom right part). The bottom-left part enables users to set the execution activity periods, an estimated execution time, and the allocation of activities to tasks. The central section of the tool shows, for each task, its execution period, its execution cycle, and its current activity list. The upper-right part shows the current state of each region of the statechart. It is also offered a user interface for controlling the XYZ table by absolute bi-dimensional positions (upper left part), or discrete commands (movement, stopping and speed setting).

Fig. 4 shows the statechart describing the behaviour of a single joint of the Cartesian robot. The statechart comprises four orthogonal regions (from left to right): limits of range, axis referencing, movement enabling, and movement control. As mentioned above, the screenshot of the ATA application shows the views of the statecharts corresponding to the two components that control each joint of the robot, since the behaviour of both joints is described using the same statechart.

ATA enables users to allocate activities to tasks in an arbitrary way and afterwards to modify this allocation at execution time. Configuration, initialization, starting and stopping of components can be done as many times as required while ATA is in execution. This feature provides us with great flexibility to test different allocation scenarios withouth having to regenerate the code.

With respect to the V^{3}CMM model, ATA instantiates the components and their corresponding ports. It also extracts the activities associated to the components and creates a set with them. The user is then able (selecting the option '*Choose Concurrency Policy*') to assign these activities to the existing tasks by considering a desired concurrency policy (a unique task for all the activities, one activity per task, or a particular correspondence). A global perspective of the different parts involved in the developed work that summarizes the above discussion was already shown in Fig. 2. The first step of this development process is the V^{3}CMM component model definition. This model is manually derived from a requirement specification document and is platform independent. The following transformations are involved in the generation of the application code:

1. A transformation that translates the V^{3}CMM concepts (port, component, state, activity) into OO concepts (class, interface, method). This is a particular interpreta-

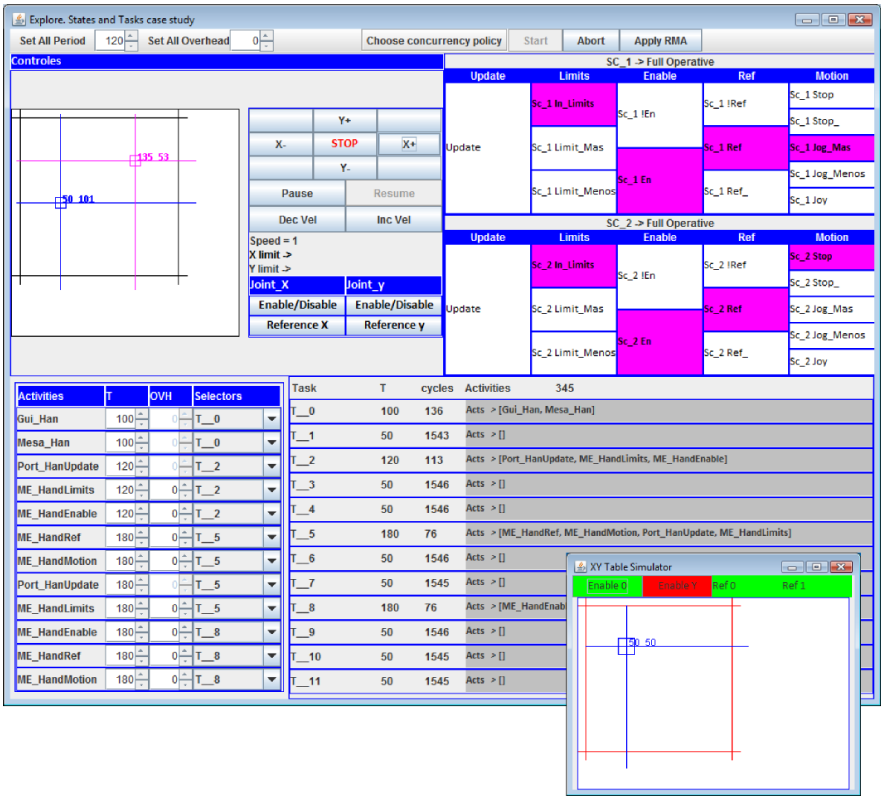

Fig. 3. Screenshot of ATA. The simulator view of the Cartesian robot is shown in the bottom-right side.

tion of the V^{3}CMM concepts suitable to be reused in several applications (since it answers questions such as how to manage state transitions in statecharts, how to implement the communication through the ports, or how to tackle with asynchronous or synchronous communications). This transformation generates all the application elements that are independent of the platform (see the upper right part of Fig. 2).

2. A transformation that generates a platform-dependent execution framework with hotspots, in which it is possible to integrate the code obtained from the previous transformation. This framework is also conceived to be reused in different applications, since it implements the task model, associates the activity code to the tasks, fixes the task execution policy, etc. (see bottom-right side of Fig. 2).

ATA is in charge of merging the code generated in both transformations. This integration is made in a semi-automated way and taking into account user parameters and the configuration of the final application (such as number of tasks, scheduling policies, mechanisms for distributing the components, etc.).

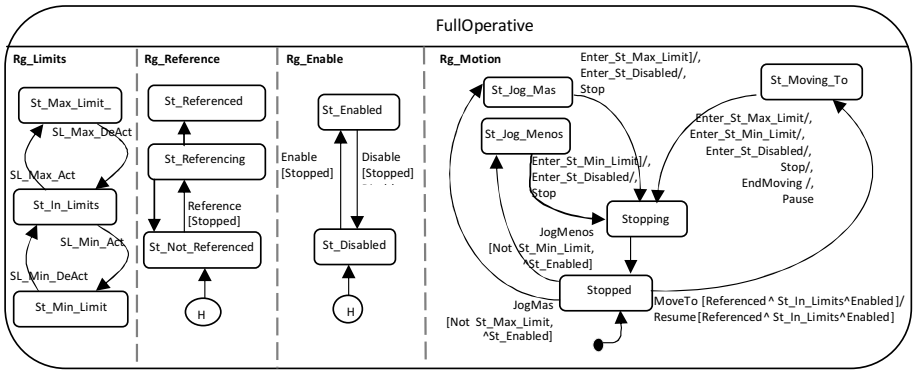

Fig. 4. Statechart modelling the behaviour of a joint of the Cartesian robot.

4 Global Architecture of the Generated Applications

The purpose of this section is to detail the global architecture of any application generated from a V^{3}CMM model. This global architecture reflects the platform-dependent and independent division of the generated code mentioned above. These transformations involve around twenty design patterns. The application of these patterns have been documented in the form of a pattern story with the aim of extracting the experience. For space reasons, this article only describes the application of a reduced subset of all the patterns involved in the transformations, specifically those strongly related to the allocation of activities to tasks. Table 1 lists an excerpt of the requirements for the transformation design, where requirements R1 to R7 are related to activity and task implementation. The rest of the requierements are simply mentioned in the last two rows.

Fig. 5 shows a simplified version of the class diagram of the generated code. Some of the most important patterns that comprise the pattern story are highlighted in the figure by the classes that fulfill the roles defined by such patterns. There are other patterns, such as OBSERVER, COPIED VALUE, DATA TRANSFER OBJECT and PROXY, which are not shown in the figure since the roles defined by such patterns cannot be clearly identified in the figure. Taking into account the naming convention shown in Fig. 2, each element appearing in the class diagram belongs to one of the following sets:

V^{3}CMM concepts. This set integrates the classes and interfaces named **`V3Component`**, **`State_Activity`**, **`Hieralchical_State`**, **`V3Input_Port`**, **`V3Output_Port`**, **`V3Data`**, **`State`**, **`V3Port`** and **`Leaf_State`**. The last three classes, related by the COMPOSITE pattern, enables the representation of the statechart structure and its hierarchy. **`V3Component`**, **`V3Port`**, and **`V3Data`** provides the implementation according to the V^{3}CMM concepts 'component', 'port', and 'data' (internal to the component), respectively. There are several well known alternatives for modeling objects with a state-dependent behaviour [6]. In this work, we have adopted the METHOD FOR STATE pattern, which implements the state-dependent behaviour in

Requirements related to activities
R1. The activities of the states may be assigned to different tasks.
R2. Activities are self-contained. In doing so, it is minimized the coupling between activitiesand tasks.
R3. Users should model the minimum time interval between to successive executions of each activity.
R4. Activities have to be as shorter (time) as possible.
Requirements related to tasks
R5. It should be possible to assign activities to the task in which is going to be executed (R1).
R6. Tasks should plan the execution of their activities.
R7. The execution of the tasks may be planned. It is needed to be able to apply RMA.
Requirements related to ports and communications: It should be possible to support the interface and communication types defined in the V3CM component ports. Deben soportarse las interfaces y los tipos de comunicación definidos en los puertos de los componentes V3CM. The exchange of data through the port needs to be safe (data consistency and interlock avoiding). The implementation should facilitate the synchronism between tasks and/or components. It should be possible to extend the solution in order to support component distribution.
Requirements related to statemachines: It should be explicitly considered the action of entering or exiting from a state. It should be implemented orthogonal regions in non-hierarchical statemachines. Changes in a region may trigger state transitions in other regions of the same component. It is needed a way to animate the statemachine. Los cambios que se produzcan por acciones realizadas en una región pueden disparar transiciones en otras regiones.

Table 1. Summary of the requirements for the transformation development.

internal subprograms of the object and by means of inner data structures. Finally, `State_Activity` represents the interface of the activities associated to the states.

Application-specific code. This set integrates all the subclasses of `V3Input_Port`, `V3Output_Port`, `Leaf_Activity`, `Hierarchical_Activity`, `V3Data`, and `V3Component`. Two specializations of `Leaf_Activity` are included in order to achieve two objectives (see Fig. 5): (1) `A_Particular_Leaf_Activity` is an activity associated with the leaf states (i.e., those states that do not contain any other state) of the statechart, and (2) `A_Particular_Port_Handler` is an activity associated to an orthogonal region that is added to the original statechart during the transformation in order to manage the component ports. The specialization of `Hieralchical_Activity` entitled `St_Machine_Handler` manages the transitions between states within an orthogonal region and is also automatically generated and added to the code. `V3Input_Port` and `V3Output_Port` are generic types designed to be specialized by the user with the data types sent or received between component ports. When the class `V3Data` is specialized, the user adds the specific data types of the defined component, as `V3Data` models internal values of the component. The specializations of the superclass `V3Component` are the components defined by the user (for instance, joint controllers, man-machine interfaces, coordinators, sensors, actuators, etc.).

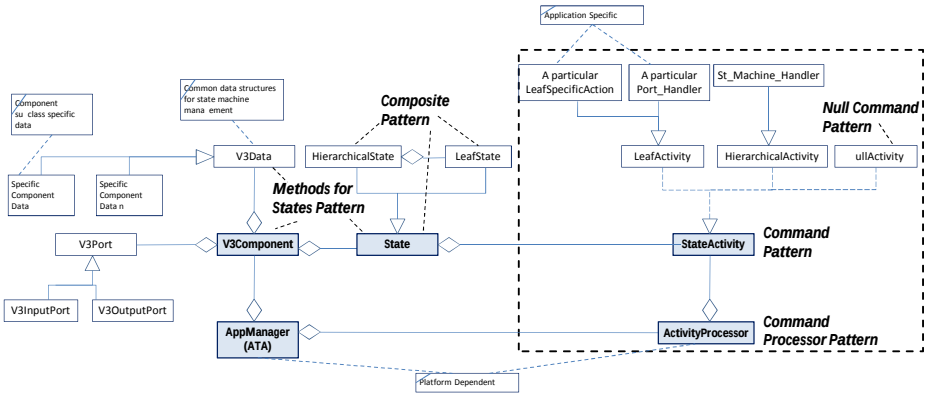

Fig. 5. Simplified class diagram of the generated code.

Platform-specific code. This last set integrates the classes **`Activity_Processor`** and **`App_Manager`**. The class **`App_Manager`** represents the ATA application, which merges the platform-dependent aspects with the V^{3}CMM concepts implementation, as last section described. The class **`Activity_Processor`** is the result of applying the COMMAND PROCESSOR pattern [6]. This pattern separates service requests from their execution by managing these requests as independent objects. In doing so, each task is modeled as a command processor and the activities (instances of subclasses of **`State_Activity`**) as commands themselves. Tasks as command processors provides a great flexibility to the overall design since this decission imposes no constraints over activity subscription, number of activities, activity duration, etc. From now on, the terms "COMMANDPROCESSOR" and "**`Activity_Processor`**" are used throughout the text. In order to avoid confusion, the first term refers to the pattern name, while the second represents the concrete implementation of the pattern used in this article.

From a dynamic point of view, a typical component execution scenario is shown in the sequence diagram of Fig. 6, where broken lines represent the boundary of a component. 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 the same task or another will asynchronously process the incoming data. 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 states, and the updating of new data in output ports (subprogram **`set(data)`** in Fig. 6).

5 Allocation of Activities to Tasks. The Command Processor Pattern

This section deals with the application of the COMMAND PROCESSOR pattern, since it captures the essential decisions of the application design. As stated before, the COM-

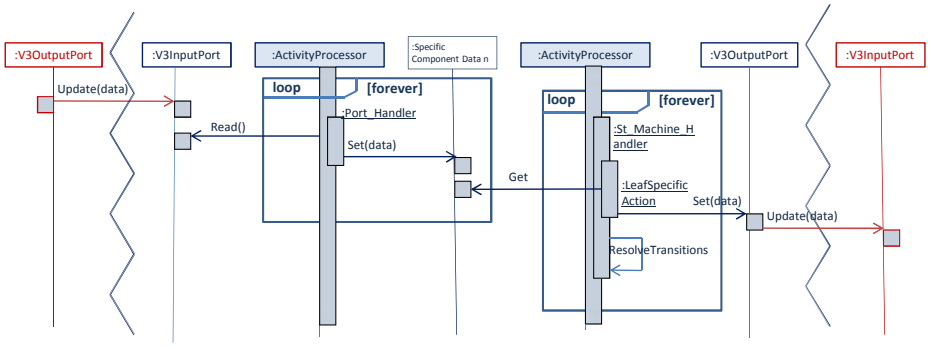

Fig. 6. A sequence diagram with a typical execution scenario.

MAND PROCESSOR design pattern separates service requests from their execution, and builds on the COMMAND design pattern. Both patterns follow the idea of encapsulating requests into objects, and thus clients do not invoke the component services directly but send messages that command processors enqueue and execute. In this work, the class **`App_Manager`** plays the client role. During the application initialization phase this class allocates all the activities to a set of **`Activity_Processor`** objects, that is, the tasks. Code listing 1.1 shows an excerpt of the Ada specification of the **`Activity_Processor`**, which has been implemented as a generic package. 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. As this is the chosen scheduling policy there is no **`Set_Priority`** subprogram in the current implementation.
- The transformation takes into account that a task may include several activities with different periods. The period assigned by the transformation to each task (**`Activity_Processor`**) is equal than the lowest periods of its subscribed activities. It is important to highlight that activities may execute periodically or not. When tasks are sporadic the period attribute represents the minimum gap between two consecutive activations of each activity. The activities are executed in the same order as they have been subscribed to the **`Activity_Processor`** (requirement R6), and the way in which tasks are executed is given by the chosen scheduling algorithm.
- The subprogram **`Add_Activity`** enables subscribing activities to tasks (requirement R5).

This design assumes that activities are defined to have an execution time as short as possible to simplify scheduling. For instance, when an algorithm includes a big number of iterations or considers a continuos control action, then the activity is divided into a

Listing 1.1. Code excerpt of the specification of the `Activity_Processor` generic package.

```
generic
   Listener        : access I_Activity_Processor_Listener'Class;
   Name            : Unbounded_String;
   Worker_Priority : System.Any_Priority;
package Common.Activity_Processor is
   function Get_Name return Unbounded_String;
   procedure Set_Priority (Priority : System.Any_Priority);
   function Get_Priority return System.Any_Priority;
   procedure Start;
   procedure Stop;
   procedure Set_Period (Period: Time_Span);
   function Get_Period return Time_Span;
   procedure Add_Activity (Act : access I_State_Activity'Class);
   procedure Del_Activity (Act : access I_State_Activity'Class);
end Common.Activity_Processor;
```

set of sub-activities with a bounded execution time (for example, an algorithm step or a discrete control action).

With regards to the allocation of activities to tasks it is needed to emphasize some aspects. Due to the encapsulation property of components, an activity belonging to a particular component has no visibility of the data of other components. Consequently, there is no problem with merging in a unique task all the component activities. When the activities of a component are allocated to different tasks then concurrent access to component data occurs. In order to assure data consistency, component data is structured in a set of protected objects.

6 Related Work

There is a well established tradition of applying CBSD principles in the robotics community, which has resulted in the appearance of several toolkits and frameworks for developing robotic applications. An actualized state-of-the-art with references to the most important robotic frameworks and toolkits can be found in the RoSta project (Robot Standards and Reference architecture) [13]. Robotic frameworks are excellent examples of the application of design patterns and are oriented to code reuse by their very nature. However, they strongly depend on a specific platform or middleware and this dependency makes component and design reuse across different frameworks almost impossible [11]. Furthermore, most of these frameworks do not consider hard RT requirements. Anyway, the main drawback of these frameworks is that, despite being CB in their conception, designers develop, integrate and connect components using OO technology. The main problem lies in the fact that CB designs require more abstractions and tool support than OO technology can offer. Thus, we think that OO languages

Listing 1.2. Code excerpt of the specification of the `Activity_Processor` generic package.

```
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;
```

must not be used for expressing CB concepts, although OO technology can be perfectly used for implementing them. In this respect, this work enriches the existing initiatives by providing a CB design level that uses platform-independent CB abstractions (port, statechart, activity, etc.) that are implemented following a set of design patterns and using OO technology within a MDSD approach.

Besides, there are not many initiatives for applying MDSD principles to robotic software development. In general, the 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 [4]. Another initiative, described in [10], 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 is rather limited. Finally, Smartsoft [14] is one of the most interesting initiatives for applying a MDSD approach to robotic software development. 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 standardized AUTOSAR [2] for easing the development of software for vehicles.

As Buschmann et al. [6] 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 [8], 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.

7 Conclusions and Future Research Lines

As already discussed throughout the paper, the adoption of a pattern-driven approach greatly facilitates the design of complex RT software, such as robotics applications. The purpose is to apply different patterns aimed to solve different problems synergistically in a way that can be extrapolated to other systems with similar requirements. The definition of such kind of pattern languages is certainly a difficult task that can be initiated by the availability of pattern stories. The pattern story reported in this article shows the feasibility of combining a statechart-based design with an implementation based on periodic concurrent tasks with RT requirements.

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 sequences (although there are some proposals, as the already mentioned in [6]). The pattern story reported in this paper will become a pattern sequence when validated by its application in other systems and improved by the critics of the scientific community.

In the context of pattern language research [7], the set of patterns used for designing the transformation constitutes a *pattern story*. Pattern sequences are more general than pattern stories. A system that shares the requirements addressed by the previous application could be designed using a pattern sequence that abstracts away the details of the concrete example described in this paper. If a software requirement is not met by the sequence then the design advice offered by the sequence is at best limited. It is important to highlight that the pattern sequence denotes only one possible system design under a given set of requirements. Different requirements would require alternative pattern sequences.

We are currently working on extending the pattern story to (1) consider distribution aspects, (2) on automating the allocation of state activities onto tasks according to their time requirements and to the chosen scheduling algorithm (in the current ATA version this allocation is done manually), and (3) on refining and improving the pattern used for implementing hierarchical and timed statecharts.

Bibliography

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

[2] "AUTOSAR: Automotive Open System Architecture", 2008-2011. Available online: `http://www.autosar.org/`

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

[4] Blanc, X.; Delatour, J. and Ziadi, T.: "Benefits of the MDE approach for the development of embedded and robotic systems. Application to Aibo". In: *Proc of the 3^{rd} National Conference onControl Architectures of Robots*, , 2007.

[5] Bruyninckx, H.: "Robotics Software: The Future Should Be Open". *IEEE Robot. Automat. Mag.*, 2008, **15(1)**, IEEE. ISSN 1070-9932. doi: 10.1109/M-RA.2008.915411.

[6] Buschmann, F.; Henney, K. and C. Schmidt, D.: *Pattern-Oriented Software Architecture, Volume 4: A Pattern Language for Distributed Computing*. John Wiley and Sons Ltd, 2007. ISBN 0471486485.

[7] Buschmann, F.; Henney, K. and Schmidt, D.: *Pattern-Oriented Software Architecture, Volume 5: On Patterns and Pattern Languages*. John Wiley and Sons Ltd., 2007. ISBN 0471486485.

[8] Dipippo, L. and Gill, C.: *Design Patterns for Distributed Real-Time Embedded Systems*. Real-Time. Springer, 2009. ISBN 0387243577.

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

[10] Jorges, Sven; Kubczak, Christian; Pageau, Felix and Margaria, Tiziana: "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 '07*, pp. 137–148. IEEE, 2007. doi: 10.1109/EASE.2007.17.

[11] Makarenko, A.; Brooks, A. and 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'07)*, IEEE, 2007.

[12] OMG: "MDA success stories", 2008. Available online: `http://www.omg.org/mda/products_success.htm`

[13] Robot Standards and Reference Architectures (RoSTa), Coordination Action funded under EU's FP6:. Available online: `http://wiki.robot-standards.org/index.php/Current_Middleware_Approaches_and_Paradigms`

[14] Schlegel, C.; Hassler, T.; Lotz, A. and 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, 2009.

[15] Stahl, T. and Völter, M.: *Model-Driven Software Development: Technology, Engineering, Management*. Wiley, 2006.

[16] Szyperski, C.: *Component software: beyond object-oriented programming*. A-W, 2th edition, 2002. ISBN 0201745720.