A Framework for Developing Distributed Component-Based Applications with Explicit Concurrency Control*

Francisco Sánchez-Ledesma, Juan Pastor, Diego Alonso

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

Abstract. Reactive system design requires the integration of structural and behavioural requirements with temporal ones (along with V&V activities) to describe the application architecture. This paper describes an implementation framework for component-based application that provides designers with great control over application concurrency (number of threads and their characteristics), the computational load assigned to them, as well as the distribution of components in processes and nodes. The paper presents an improved version of a framework previously developed, putting it in the context of a global Model-Driven Software Development approach for developing, analysing and generating code for reactive applications. This work has been done and validated in the context of the development of robotic applications.

1 INTRODUCTION AND MOTIVATION

There is a well established tradition of applying *Component Based Software Development* (CBSD) [18] principles in the robotics community, which has resulted in the appearance of several tool–kits and frameworks for developing robotic applications [14]. 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. Moreover, most of these frameworks impose the overall internal behaviour of their components. On the other hand, robotic systems are reactive systems with *Real–Time* (RT) requirements by their very nature, and most of the frameworks for robotics do not provide mechanisms for managing such requirements. From our point of view, the design and implementation of CB frameworks for robotic applications development should overcome, among others, the following problems:

* This work has been partially supported by the Spanish CICYT Project EXPLORE (ref. TIN2009-08572).

1. The definition or adoption of a component language for modelling domain applications. This language should allow designers to work with CBSD abstractions rather than with OO ones. It should also take into account the systems requirements, including their timing properties.
2. The translation of the resulting models to executable code and to analysis models, which can later be injected to tools in order to analyse both the CB application and the resulting code properties.

In CBSD, a software component is a unit of composition with well-defined interfaces and explicit context of use. According to the classification of CBSD approaches [10], in this work we consider components as architectural units that communicate only through their ports, in line with the Software Architecture discipline and ADLs (Architecture Description Languages). This decision introduces, however, new problems:

- There are no development tools widely available that can generate efficient code from the architectural model of the application. As far as we know, most of them are academic initiatives, not widely used in industry (for instance, Lagoona [6] or the set of languages derived from Oberon [18]).
- It is difficult to express the RT requirements of architectural components, in part because the concepts used in CBSD (mainly "component" and "port") are not the same as those usually used to model and analyse system schedulability ("threads", "synchronization" and "communication mechanisms" between threads). Some approaches solve this problem by identifying threads or processes with components. However, in our opinion, these solutions overly restrict the flexibility of the CBSD approach, since components should be designed according to their schedulability instead of according to the services offered by their interfaces. Therefore, it is required a more flexible approach.

In order to overcome these problems, we decided to use the *Model Driven Software Development* (MDSD) [17] approach, since it allows addressing them as a whole, and constitutes the theoretical and technological framework where the work presented in this paper is being developed. The MDSD approach has been used to integrate the CBSD approach with RT issues, as well as with the generation of both executable code and analysis models. The paper does not focus on MDSD aspects, but on the use of architectural components for modelling robotic applications, and particularly in a translation of the CBSD concepts to OO concepts to enable both the compliance with the RT requirements of the applications and the components distribution. A brief description of the overall approach is, however, necessary to understand the rest of the paper.

Our global development approach starts by modelling the architecture of the application using the CBSD approach, and then use a series of model transformations to generate both analysis models and executable code. Though any modelling language can be used for performing the first step, we developed our own modelling language, entitled V^3CMM. A more detailed description of the

main characteristics of this language and an explanation of the reasons that led us to its development is not in the scope of this paper, but they can be found in [9]. The V^{3}CMM language provides three complementary but loosely coupled views (see Fig. 1) that allows designers to define and connect software components, namely: (1) an *architectural view* to define components (interfaces, ports, services offered and required, composite components, etc.), (2) a *coordination view* to specify component behaviour, based on timed automata theory [2], and finally (3) an *algorithmic view* to express the sequence of actions executed by a component according to its current state, based on activity diagrams. The algorithmic view considers two kind of activities: periodic and non-periodic ones, with the additional restriction over periodic activities that infinite loops are forbidden, since V^{3}CMM considers that activities model a single iteration of an algorithm.

In order to ease the generation of executable code from the input V^{3}CMM model, an OO framework was designed and implemented. Such framework provides an OO interpretation of the CBSD concepts that allows translating the CB designs into OO applications. Specifically, it provides the base classes for implementing components, and an infrastructure for the user to choose the concurrency features that he ultimately want for the application: number of threads, code allocated to such threads, deadlines, priorities, thread periods, etc. A previous implementation of the framework without hierarchical state-machines nor distribution support was described in [13]. This paper presents an improved version of the cited work fulfilling such limitations.

The remainder of this paper is organized as follows. Section 2 explains the main architectural drivers that guide the design of the implementation framework. Afterwards, Section 3 describe the way in which CBSD modelling elements (components, ports and timed automatas) have been translated into OO concepts to generate executable programs. Section 4 explains the importance of including support for component distribution in the framework, a justification of the approach followed to do so, as well as a brief explanation of the implementation. Finally, Section 5 presents the conclusions and future work.

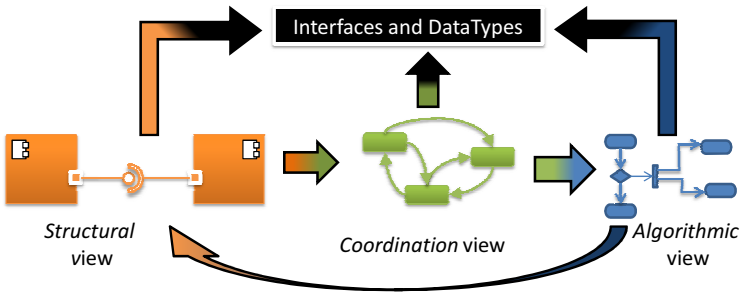

Fig. 1. Schematic representation of the V^{3}CMM views showing the kind of concepts appearing in each view and the loosely coupled relationships existing among them.

Though the paper considers the domain of service robots, we think the approach can be used in other domains with similar characteristics, since both the modelling language (V^3CMM) and the implementation framework consider what we could call "generic" or "domain-independent" concepts.

2 ARCHITECTURAL DRIVERS

In a previous work described in [1], we transformed the input V^3CMM model (which describes the CB application architecture) into a UML implementation model, from which code was generated afterwards. Though this process worked, we soon realised that it had a number of drawbacks, namely (i) the developed transformations were huge, and thus difficult to maintain and evolve, and (ii) too many design decisions were embedded in the transformations, such as the interpretation of the CBSD concepts, the facilities for creating and managing components, communication mechanisms, concurrency characteristics, run-time support, etc. Therefore, we had to modify the model transformations when an application with other characteristics was needed.

Now we follow a different approach. In order to ease the generation of executable code from the input V^3CMM model, we decided to develop an OO framework as the target for a model transformation. This framework will provide the required properties for the final application. And it can be changed, substituted or improved easier than a model transformation. As this paper focuses on the design of such a frameworks, it is worth enumerating (in relevance order) the main architectural drivers that guide its design:

AD1 Control over *concurrency policy*: thread number, thread spawning (static vs. dynamic policies), scheduling policy (fixed priority schedulers vs. dynamic priority scheduler), etc. Unlike most frameworks, these tasking issues are very important for us, and thus we want the users of the framework to be able to select them.

AD2 Control over the *allocation of activities to threads*, that is, control over the computational load assigned to each thread, since V^3CMM considers the activity associated to a state as the minimum computational unit. The framework allows allocating all the activities to a single thread, allocating every activity to its own thread, or a combination of both policies. In any case, the framework design should ensure that only the activities belonging to active states are executed.

AD3 Facilitate the instantiation of the framework from the input CBSD model.

AD4 Control over the communication mechanisms between components (synchronous or asynchronous).

AD5 Control over the component distribution and deployment.

3 FRAMEWORK DESIGN

The design and documentation of the framework was carried out using design patterns, which is a common practice in Software Engineering [5]. In order

to describe the framework we will use figures 2, 3, and 4. Fig. 2 shows the pattern sequence that has been followed in order to meet the architectural drivers described in Section 2, while figures 3 and 4 show the classes that fulfil the roles defined by the selected patterns. At this point, it is worth highlighting that the same patterns applied in a different order would lead to a very different structural design.

Among the aforementioned drivers, the main one is the ability to define any number of threads and control their computational load (architectural drivers AD1 and AD2). This computational load is mainly determined by the activities associated to the states of the timed automata. In order to achieve this goal, the COMMAND PROCESSOR architectural pattern [4] and its highly coupled COMMAND pattern [7] have been selected, and they were the firsts to be applied in the framework design, as shown in Fig. 2. The COMMAND PROCESSOR pattern separates service requests from their execution by defining a thread (the command processor) where the requests are managed as independent objects (the commands). These patterns provide the required level of flexibility, since they impose no constraints over command subscription to threads, number of commands, concurrency scheme, etc. The roles defined by these two patterns are realised by the classes `Activity_Processor` and `State_Activity`, respectively (see Fig. 3).

Another key aspect, related to AD4, is to provide an OO implementation of the timed automata compatible with the selected patterns for concurrency control, in order to integrate it in the scheme defined by the aforementioned COMMAND PROCESSOR pattern. It is also an aspect that has a great influence on the whole design, since timed automatas model the behaviour of the components. We decided that both regions and states should be treated homogeneously, and thus we selected a simplified versions of the COMPOSITE pattern. The timed automata is managed following the METHODS FOR STATES pattern [4], and the instances of the classes representing it are stored in a hash table. The roles defined by this pattern are realised by the classes `State, Hierarchical_State, Region` and `Leaf_State`.

Each activity associated to a state of the timed automata is implemented as an object, following again the COMMAND pattern, represented by the class `State_Activity`. In this way, activities can be allocated to any command processor. This constitutes the link between concurrency control and timed automata implementation, since activities play roles in both patterns. The distinction between states and regions led us to define two hierarchies of `State_Activity`, which were implemented following the STRATEGY pattern: those associated to leaf states (represented by the root class `Leaf_Activity`), and those activities associated to regions (represented by the class `Hierarchical_Activity`). The latter is aimed at managing the region states and transitions, and thus is provided as part of the framework. The formers, shown in Fig. 4, are related to (i) the activities defined in the V^3CMM models, represented by `Native_Activity` subclasses, and (ii) activities to manage the flow of data and control among component through their ports.

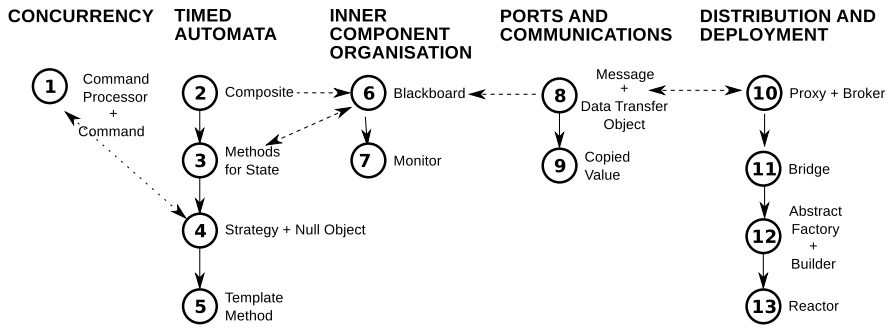

Fig. 2. Dependency relationships between the patterns considered in the framework development and the V^{3}CMM views. Though the patterns are numbered, the design was iterative, and most of the patterns had to be revisited, leading to many design modifications.

Conditions, transitions and events are modelled as separate classes, shown in Fig. 4, to facilitate the framework instantiation from the input V^{3}CMM model (AD4). `Condition` is an abstract class used to model transitions' conditions. It provides an abstract method to evaluate the condition. Concrete subclasses are (i) `ConditionStateActive`, which tests whether a specific state is active; (ii) `ConditionActivityDone`, which tests whether an activity is finished; and (iii) `ConditionPort`, which tests if a message has been received by a specific port. On the other hand, the class `Transition` include the source and target states, and groups a set of conditions vectors that must be evaluated to determine if the transition should be executed. To end with timed automata implementation, it is worth mentioning two additional patterns. The Null Object pattern is used for smoothly integrating states that have no associated activity, while the Template Method pattern is related to defining the activity in charge of region management.

The next challenge is how to store and manage the component internal data, including all the states and activities mentioned above, the data received or that must be sent to other components, the transitions among states, event queues, etc. All these data is organised following the Blackboard pattern. The idea behind the blackboard pattern is that a collection of different threads can work cooperatively on a common data structure. In this case, the threads are the command processors mentioned above. The main liabilities of the Blackboard pattern (i.e. difficulties for controlling and testing, as well as synchronization issues in concurrent threads) are mitigated by the fact that each component has its own blackboard, which maintains a relatively small amount of data. Besides, the data is organized in small hash tables with different access policies (monitors, *1-writer/n-readers*, etc.). The roles defined by this pattern are realised by the classes `V3Data` and `V3ComponentData`.

As shown in Fig. 2, the Blackboard pattern serves as a joint point between timed automata and the input/output messages sent by components

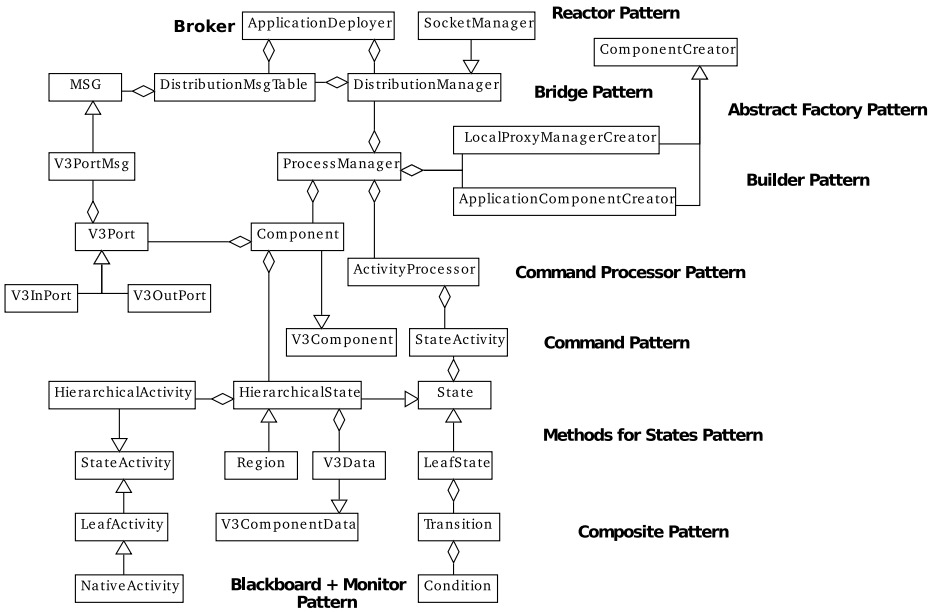

Fig. 3. Simplified class diagram of the developed framework showing some of the patterns involved in its design.

through their ports. Component ports and messages exchanged between them are modelled as separate classes. The classes representing these entities are the classes `V3Port` and `V3Port_Msg`, shown in Fig. 3. The communication mechanism implemented by default in the framework is the asynchronous without reply scheme, based on the exchange of messages following the MESSAGE pattern. In order to prevent the exchange of many small messages, we use the DATA TRANSFER OBJECT pattern to encapsulated in a single message all state information associated to a port interface, which is later serialized and sent through the port. Finally, because components encapsulate their inner state, we use the COPIED VALUE pattern to send copies of the relevant state information in each message.

The framework adds extra regions to manage the flow of messages through ports, the internal memory of the component, and each region of the component's timed automata. The activities for performing such a duty are predefined in the framework, and they shall be assigned to threads in a similar way as the user does with the activities of the input V^{3}CMM model. It is worth highlighting that this design facilitates schedulability analysis, since no code is "hidden" in the framework implementation, but it must be explicitly allocated to a particular thread.

While the temporal characteristics of the input model activities are specified in the model itself, the characteristics of those added by the framework cannot be set arbitrarily, but must be derived from the model, although currently

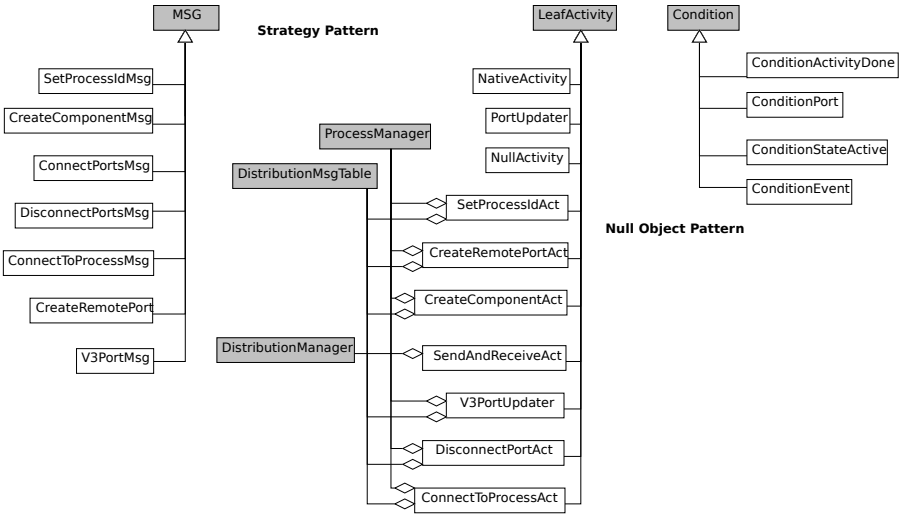

Fig. 4. Hierarchy of classes that complement Fig. 3. Classes filled in grey represent the joint points with the classes shown in that figure.

this is done manually. For example, the period of the activity that manages a region must be less than the period of the most frequent activity defined inside the region. Fig. 5 shows a scenario that illustrates these features. The original V^{3}CMM timed automata are extended with regions for managing the component state and the messages flow through ports. Notice that, in the figure, regions have marks indicating the period of the activity managing it. For instance, region RgMotion is managed every 10 ms. Please also notice that new regions have been added to the original model in order to manage communications among ports. The framework allows designers to select the number of threads (three in this case) and the way activities are allocated to them (in this case, all the region activities are allocated to the same thread, according to their periods). In any case, the framework user can select any other combination of threads–activities once the framework has been instantiated from the input V^{3}CMM model. The framework does not provide any guidance on the number of threads to be created, or how to make the assignment of activities to threads, but provides the mechanisms needed for users can make such assignment according to some heuristics as those defined in [8].

After a couple of refactoring steps, we soon realised that the code implementing these architectural drivers could be organized into three groups with clearly defined interfaces: (**C1**) the code that provides the runtime support (modelling threads), (**C2**) the code that provides an OO interpretation of the CBSD concepts and the framework '*hot-spots*', and (**C3**) the application-specific code that supplement the 'hot-spots' of the framework to create a

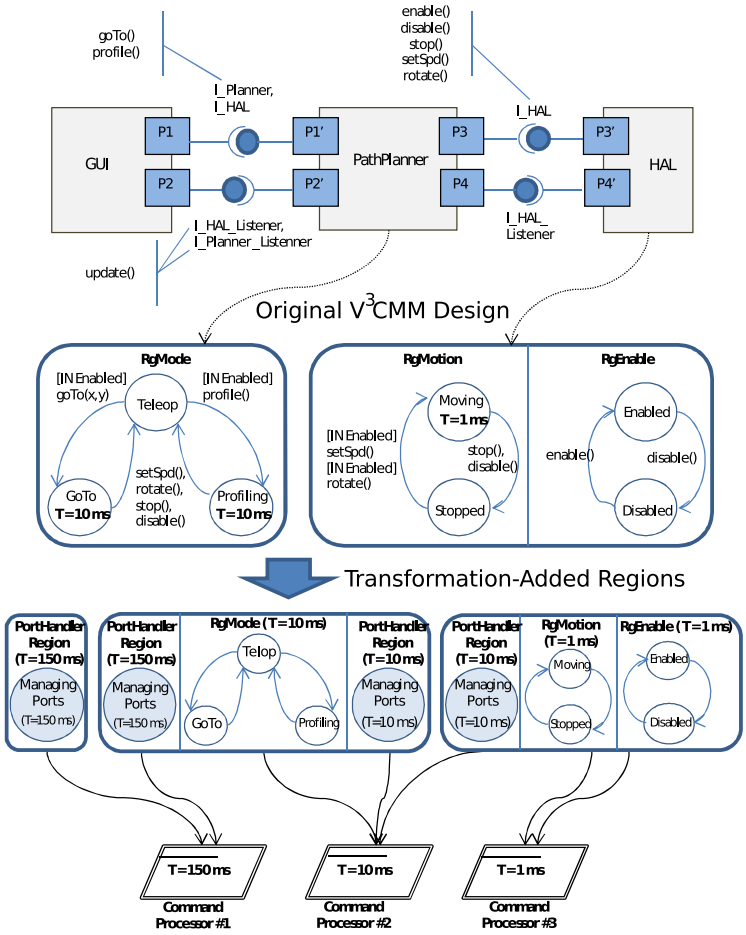

Fig. 5. An example of the use of the framework. Definition of thread number and allocation of activities to them.

specific application. In this way, it would be possible to provide an alternative interpretation of CBSD concepts (C2) using the same run–time support (C1), as well as to reuse the same application (C3) in a different run–time (C1), provided that C2 is not changed.

Distribution issues were the last to be considered. They are described in the following section due its importance and extension. Nevertheless, it is worth highlighting that its implementation did not require modifying the framework structure, but defining several new classes and instantiating some of the previously describe base classes.

4 OVERVIEW OF COMPONENT DISTRIBUTION

Being able to distribute components among nodes should be one of the main characteristics of the framework, and thus one of our priorities, due to the following reasons:

- By the very nature of robotic applications, which normally include several computing nodes.
- Isolation of components with hard RT requirements from those with soft RT requirements. In our case, this characteristic can be achieved by separating both types of components and assigning them to different nodes.
- The case studies considered in our research lines explicitly consider component distribution.
- The ability to modify the components deployment depending on the available computing resources and the state of the environment.

In fact, each and every available robotic framework include distribution in one way or another, being the use of a middleware technology, like CORBA, the most common solution used all of them. A survey of the most well-known robotics frameworks can be found in [14]. In our work, however, we decided to develop an *ad-hoc* middleware for carrying out component distribution for the following reasons:

- The users of commercial middleware technologies normally lose the control over the execution of the application (the "inversion of control" problem), as well as some RT characteristics (like number of threads, computing time, etc.) that must be taken into account if RT analysis is required, as in our case.
- Commercial middleware normally target OO applications, while our approach uses components for modelling the application architecture. Thus, it would be awkward to combine components with objects when designing the application architecture.
- Though it would be possible to achieve a certain degree of control over the middleware characteristics mentioned in item (i), the changes needed for introducing these modifications would require a deep knowledge of the middleware implementation (and its source code).

– The overall design approach to CBSD applications we follow, as described in previous sections, do not need all the distribution services normally required by distributed applications and provided by middleware technologies. In our case, the components that make the application architecture up and the connections among them are defined in the input models, and therefore services like naming, registering, searching, etc. are not needed.

Under all these conditions, we considered that the best choice was to develop an *ad-hoc* middleware with the minimum services needed for managing component distribution. As such, we considered only services for component allocation and de-allocation inside processes, component and application start and stop services, and connection and disconnection of compatible ports of components allocated in the same or in different processes. These processes can be executed in different nodes, and thus the framework considers two levels of component distribution: (i) components are allocated to a process, which means that all the activities belonging to the component must be assigned to threads belonging to such process, and (ii) processes are assigned to different computational nodes.

According to what has been stated, in order to make the distribution possible and feasible, two artefacts have been defined: one belonging to the CBSD domain (the `LocalProxyManager` component), and the other one is a class added to the framework (the `ApplicationDeployer` class). These elements are in charge of realizing the deployment of the application components, and of managing the messages sent among the nodes. At this point, it is worth highlighting that the implementation of both elements did not require modifying the original framework structure, but they only instantiate the base classes provided by the framework. The objectives of the `LocalProxyManager` component are (i) to create and connect component instances in node it manages, and (ii) to act as a proxy of the ports of remote components. This component is not meant to be directly added by the application developer, but, for each deployment process, one of such components is automatically added to the application architecture, and then created by the framework distribution services. These services are mainly provided by the `ApplicationDeployer` class, which acts as the master node for application deployment, and thus it must be executed in its own process before the application can be deployed. This class performs the application deployment according to the specification of a configuration file.

Internally, the `LocalProxyManager` contains the same constructive elements (regions, states, transitions and activities) as any other component that has to be instantiated in the framework. And like the rest of the components, it allows their internal activities to be assigned to different threads in any arbitrary fashion. It was decided to maintain the same structure because this ensures that the overhead added by this component is completely measurable and afterwards analysable. Although from the point of view of the framework user is irrelevant how the `LocalProxyManager` component works internally, it is worth mentioning that the current implementation is based on the use of the Reactor pattern [16].

The `ApplicationDeployer` can be considered as a mini BROKER [16], without explicit register nor look-up services. These services are not explicitly invoked because all the information required to deploy and connect the application components is read from the application configuration file, which is used by the `ApplicationDeployer` to communicates `LocalProxyManager`s which components should be created, and how to connect their ports. If any of these ports belong to a component that runs in a separate process, the `ApplicationDeployer` sends to the `LocalProxyManager` a PROXY [7] of such port, so that the `LocalProxyManager` can make a local connection. The creation of the concrete application components, which also involves the creation of its ports, states, transitions, etc., is carried out by the classes `LocalProxyManagerCreator` and `ApplicationComponentCreator`, which play the roles defined in the ABSTRACT FACTORY and BUILDER patterns.

The communication protocol currently used is TCP, though we plan to use deterministic protocols like CAN bus. The decoupling of `ApplicationDeployer` and `LocalProxyManager` from the communication infrastructure is achieved by means of the BRIDGE pattern [7]. Both artefacts use the communication services defined in the abstract class `DistributionManager` (see Fig. 3). Currently, there is only one concrete subclass of this class, `SocketManager`, which relies on the sockets libraries. But this design allows us to easily extend the framework with new classes for changing the underlying communication protocol.

In the current implementation, based on TCP sockets, the `ApplicationDeployer` is also in charge of sending to each `LocalProxyManager` the port and IP address of the rest of the nodes the application is going to be deployed into, in order for them to establish and manage communication among them. Ports are taken starting from number 50.000 to avoid possible collisions with other protocols. Messages exchanged among components (both intra and inter nodes) as well as those related to components deployment are marshalled as ASCII string characters, where each character has the size of a byte. These messages use labels to separate and identify their fields, and to separate consecutive messages.

Many of the subclasses shown in Fig. 4 are related to component distribution. Specifically, those derived from the class `MSG` define the kind of messages exchanged by the `ApplicationDeployer` and the `LocalProxyManager`s, while the ones deriving from `LeafActivity` define how the services requested by such messages are carried out.

Newly, the activities added by the framework to achieve distribution are treated as "normal" activities, and thus have to be allocated to threads just like the rest of the component activities. The artefacts in charge of managing component distribution and deployment are considered "normal" components, in the sense that they use the same elements and behave exactly like any other component in the application. This allows us to regularly include the communication overhead in the RT analysis, provided that transmission times are known and can be incorporated to the execution time associated to the activities that manage communications.

In order to illustrate the whole process, the feasibility of the approach, and its main advantages, we will use a simple example consisting in three components deployed in two nodes. The selected example revolves around an application for tele-operating a mobile robot. The architecture of the application is shown in the left side of Fig. 6: a Human-Machine Interface (HMI) component sends the movement and control commands issued by the user to the robot, a Display component shows the information obtained from the robot sensors and the commands sent by the user, and finally a controller component, entitled LazaroRC, acts as a hardware abstraction layer that facilitates the sending and receiving of messages to the robot. The figure only shows the structural view of the application, since the timed automatas that describe component behaviour are not relevant in this example. The right side of the figure shows the same application distributed in two process, which requires the addition of two `LocalProxyManager`.

The next code listing shows an excerpt of the configuration file that describes the deployment of the case study application. It is possible to identify networking characteristics (IP address and port for every node) and the connection between component ports, among others:

```
<Deployment>
  <Master Ip="localhost" SocketPort="50001"/>
  <Process Id="1" Ip="localhost" SocketPort="50002"/>
  <Process Id="2" Ip="localhost" SocketPort="50003"/>
  <ComponentInstance Id="1" Type="1" ProcessId="1"/>
  <ComponentInstance Id="2" Type="1" ProcessId="2"/>
  <PortConnection>
    <PortInstanceProcessId="1" ComponentInstanceId="1"
              PortId="2" PortType="InPort"/>
    <PortInstanceProcessId="2" ComponentInstanceId="2"
              PortId="3" PortType="OuPort"/>
  </PortConnection>

  <RegionThreadRegionId="1" ProcessId="2"
              ComponentInstanceId="2" ThreadId="1"/>
...
</Deployment>
```

At the time of making application deployment, every `LocalProxyManager` create ports that replicates local ports of the components hosted on others processes. The Display component, that according to the architecture of the application shown in Fig. 6 should be connected with component LazaroRC, is really connected to a mirror port that was created in a `LocalProxyManager`. In this way, communication is carried out as if both components were contained within the same process. Therefore, the communication mechanisms become transparent for the application components.

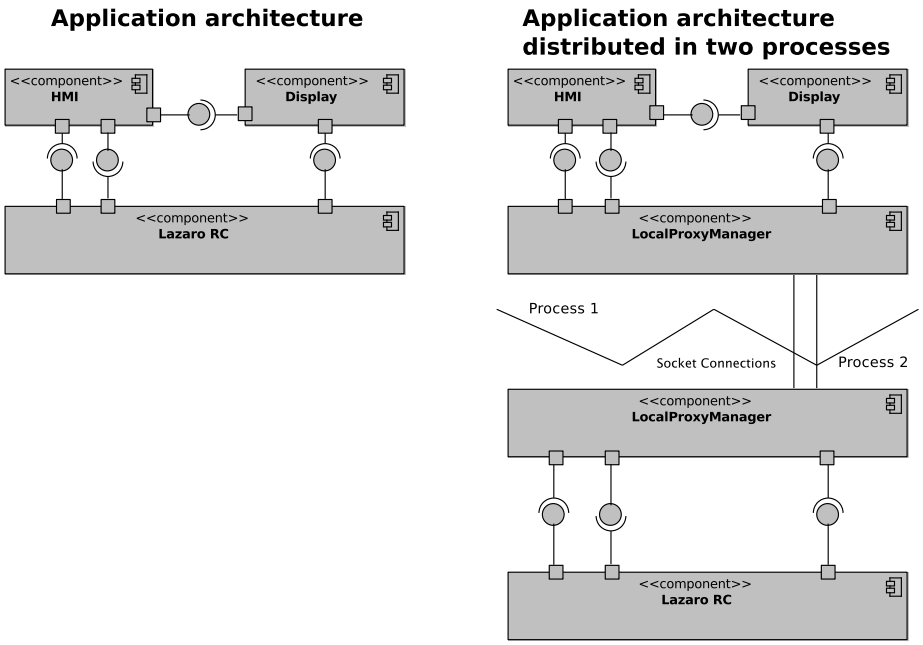

Fig. 6. Architecture of the robot case study, where components are assigned to one process (left side) or distributed in two processes (right side).

5 CONCLUSIONS AND FUTURE WORK

This paper has described an approach to provide a run–time support (framework) to a CB approach for modelling applications with RT requirements. To do so, it has been necessary to provide an OO interpretation of the high–level architectural concepts (components, ports, timed automatas, etc.) providing enough flexibility to also control application concurrency characteristics in order to take into account RT requirements. The proposed solution is not closed to future improvements, but it is a stable starting point for further development. The approach has been validated with small–scale applications, targeted to "academic" platforms (the in–house developed vehicle Lazaro, the Pioner P3AT commercial robot, and a simple electrical vehicle). Therefore it still needs to be tested in larger applications.

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. As said in [4], 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.

Although this article describes only a story of patterns, in the sense described in [4], we hope that it is sufficiently valuable to contribute to the definition of a true pattern language for the development of CB applications 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 stories. It is also very difficult to synthesize the problems encountered along the way, and therefore it is impossible to describe the problems that have arisen during the development of the framework and how they have been resolved. As mentioned in the paper the design has been was iterative, and most of the patterns had to be revisited, leading to many design modifications.

This paper also has described the evolution of the ongoing work, incorporating the ability to distribute components to an OO framework that support CB development. Distribution capacity was added in a regular way to the framework, which allows to analyse the impact on the temporal characteristics of the application that has a certain distribution of its components. For the type of applications that have been done, the distribution implementation has proven to be more than enough, although we must see how it scales as the number of components increases. Having added distribution capacities to the framework, the most urgent future work is to validate the framework, using it in larger applications that comprise both safety critical requirements in their reactive behaviour and intensive processing. In this way, it is crucial to integrate the algorithms libraries offered by other robotic frameworks. This integration is problematic, as is explained in [11], but it is crucial to make the approach (not only the framework) attractive enough to the robotics community.

The work described in this article is a work in progress. Currently, work is continuing to extend the framework with additional features following a pattern driven approach. Among these extensions, it should be noted (1) the addition of heuristics to determine the number of threads and to carry out the assignment of the component activities to these threads, (2) the development of more sophisticated model transformations to instantiate the framework from a V^{3}CMM input model, (3) the refinement and improvement of standards used for implementing hierarchical state machines and timed automata [15], (4) the adaptation of the implementation to be compliant with the Ravenscar profile [3] for designing hard RT applications, (5) the addition of deterministic network protocols, (6) the integration of algorithms defined in other frameworks, which are available as open source code. And last but not least, we are working on generating input models for analysis tools compliant with the UML MARTE profile [12] from instances of the framework.

References

1. Alonso, D., Vicente-Chicote, C., Ortiz, F., Pastor, J., Álvarez, B.: 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 (Jan 2010), http://joser.unibg.it/index.php?journal=joser&page=article&op=viewFile&path[]=18&path[]=3
2. Baier, C., Katoen, J.: Principles of Model Checking. The MIT Press (May 2008)
3. Burns, A., Dobbing, B., Vardanega, T.: Guide for the use of the ada ravenscar profile in high integrity systems. Tech. Rep. YCS-2003-348, University of York (2003)
4. Buschmann, F., Henney, K., C. Schmidt, D.: Pattern-Oriented Software Architecture, Volume 4: A Pattern Language for Distributed Computing. John Wiley and Sons Ltd (2007)
5. Buschmann, F., Henney, K., Schmidt, D.: Pattern-Oriented Software Architecture, Volume 5: On Patterns and Pattern Languages. John Wiley and Sons Ltd. (2007)
6. Fröhlich, P., Gal, A., Franz, M.: Supporting software composition at the programming language level. Science of Computer Programming 56(1-2), 41–57 (Apr 2004)
7. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns: elements of reusable object-oriented software. A-W Prof. (Jan 1995)
8. Gomaa, H.: Designing Concurrent, Distributed, and Real-Time Applications with UML. Object Technology, Addison-Wesley (2000), iSBN: 0-201-65793-7
9. Iborra, A., Alonso, D., Ortiz, F., Franco, J., Sánchez, P., Álvarez, B.: Design of service robots. IEEE Robot. Automat. Mag., Special Issue on Software Engineering for Robotics 16(1), IEEE (Mar 2009)
10. Lau, K., Wang, Z.: Software component models. IEEE Trans. Software Eng. 33(10), IEEE (Oct 2007)
11. 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'07). IEEE (2007)
12. 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
13. Pastor, J., Alonso, D., Sánchez, P., Álvarez, B.: Towards the definition of a pattern sequence for real-time applications using a model-driven engineering approach. In: Proc. of the 15^{th} Ada-Europe International Conference on Reliable Software Technologies, Ada Europe 2010. pp. 167–180. LNCS, Springer-Verlag (Jun 2010)
14. 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
15. Samek, M.: Practical UML Statecharts in C/C++, Second Edition: Event-Driven Programming for Embedded Systems. Newnes (2008)
16. Schmidt, D., Stal, M., Rohnert, H., Buschmann, F.: Pattern-oriented software architecture, volume 2: patterns for concurrent and networked objects. Wiley (2000)
17. Schmidt, D.: Model-driven engineering. IEEE Computer 39(2), IEEE (Feb 2006)
18. Szyperski, C.: Component software: beyond object-oriented programming. A-W, 2 edn. (2002)