AN IMPLEMENTATION FRAMEWORK FOR COMPONENT-BASED APPLICATIONS WITH REAL-TIME CONSTRAINTS
Extensions for Achieving Component Distribution

Francisco Sánchez-Ledesma, Juan A. Pastor, Diego Alonso and Francisca Rosique
School of Telecommunication Engineering, Technical University of Cartagena
Plaza del Hospital n^{o}1, 30202 Cartagena, Spain

Keywords: Component distribution, Middleware, Component-based software development.

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. We have adopted the Model-Driven Software Development approach to address these problems globally: from the definition of the application architecture to the generation of both code and analysis models. An Object Oriented framework was developed in order to ease the generation of code, as well as to provide the required properties for the final application (specifically, temporal behaviour). This paper describes how distribution support was added to the framework in a regular way without disrupting its design, and allowing users to integrate communication overload in timing analysis.

1 INTRODUCTION

There is a well established tradition of applying *Component Based Software Development* (CBSD) (Szyperski, 2002) principles in the robotics community, which has resulted in the appearance of several tool–kits and frameworks for developing robotic applications (Rosta, 2010). 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. In particular, 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. Additionally, these systems normally comprise several computational nodes, and thus distribution is also an important issue. From our point of view, the design and implementation of CB frameworks for robotic applications development should overcome, among others, the following problems:

1. The definition or adoption of a component language for modelling applications. This language should allow designers to work with CB 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 that can be injected to tools in order to analyse both the CB application properties and the properties of the resulting executable code.
3. It is necessary to provide services for component distribution that are compatible with the CB architecture and its RT requirements.

As explained below, the *Model Driven Software Development* (MDSD) (Schmidt, 2006) approach allows addressing these problems globally, and constitutes the technological framework where the work presented in this paper is being developed. MDSD enables us to define a component language for modelling the application architecture, to define a set of model transformations for both generating implementation code and analysis models from the previously defined models, and to integrate component distribution and timing properties in an orthogonal way.

Sánchez-Ledesma F., A. Pastor J., Alonso D. and Rosique F..
AN IMPLEMENTATION FRAMEWORK FOR COMPONENT-BASED APPLICATIONS WITH REAL-TIME CONSTRAINTS - Extensions for Achieving Component Distribution.
DOI: 10.5220/0003500402900293
In *Proceedings of the 6th International Conference on Software and Database Technologies* (ICSOFT-2011), pages 290-293
ISBN: 978-989-8425-77-5

The remainder of this paper is organized as follows. Section 2 explains the overall approach and the way in which CB concepts have been translated into OO concepts to generate executable programs, resulting in the development of an OO implementation framework. Section 3 explains the importance of including in the framework support for component distribution, as well as to justify the approach taken to do so. Section 4 explains in detail component deployment. Finally, Section 5 presents the conclusions and future work.

The paper considers the domain of service robots, but it could be extended to other domains with similar characteristics, since we consider "generic" or "domain-independent" concepts.

2 GENERAL OVERVIEW AND PREVIOUS WORKS

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^{3}CMMV3CMM V^{3}CMMV3CMM (Iborra et al., 2009). This language provides three complementary but loosely coupled views 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, 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.

In order to ease the generation of executable code from the application expressed in terms of architectural components, an OO framework was designed and implemented (Pastor et al., 2010). Such framework provides the base classes for implementing the components of the architectural design defined in V^{3}CMMV3CMM, and an infrastructure for the user to choose the concurrency features that he ultimately wants for the application: number of threads, code allocated to such threads, deadlines, priorities, etc. The framework provides an OO interpretation of the CBSD concepts that allows translating the CB designs to OO applications. The design and documentation of the framework was carried out using design patterns, which is a common practice in Software Engineering (Buschmann et al., 2007). Figure fig:secuenciaPatrones shows the sequence of patterns applied to the design of the framework in order to meet the following requirements:

1. Control over *concurrency policy*: thread number, thread spawning (static vs. dynamic policies) and thread characteristics (deadline, period, priority, etc.), scheduling policy (fixed priority schedulers vs. dynamic priority scheduler). Unlike most frameworks, these tasking issues are very important for us, and can be selected by the user of the framework.
2. Control over the *allocation of activities to threads*, that is, control over the computational load assigned to each thread. 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 ensures that only the activities belonging to active states are executed.
3. Control over the communication mechanisms between components. The communication mechanism implemented by default in the framework is the asynchronous without reply scheme, based on the exchange of messages.
4. Facilitate the instantiation of the framework to obtain OO code from the CBSD model.
5. Control over the distribution policy of the components in computational nodes.

The framework defines extra regions and activities to manage the flow of messages through ports, the internal memory of the component, and each region of the component's state machine. 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.

With all this, the framework code is organized into three groups with clearly defined interfaces: (C1) provides the runtime support, (C2) 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 specific application. This classification enables providing an alternative interpretation of CBSD concepts (C2) that use the same run-time (C1), and reusing the same application (C3) in a different runtime (C1), provided that C2 is not changed.

3 COMPONENT DISTRIBUTION

Despite the number of currently available middleware technologies, we decided to develop an *ad-hoc* mid-

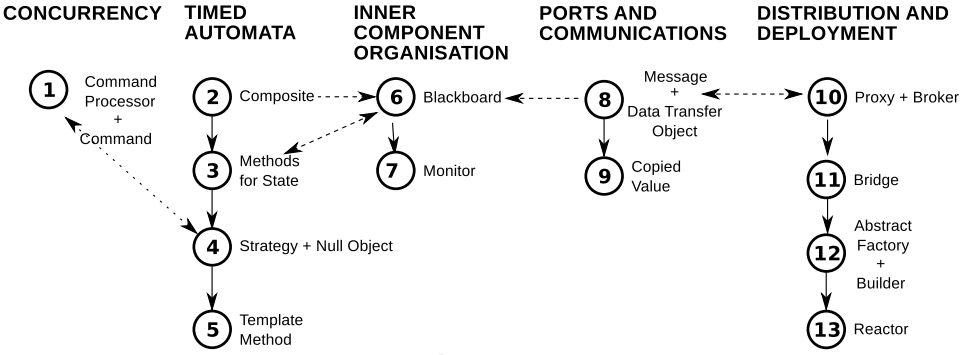

Figure 1: Dependency relationships existing between the patterns considered in the framework development and the V^{3}CMMV3CMM V^{3}CMMV3CMM views.

dleware for carrying out component distribution for the following reasons:

- The users of commercial middleware 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, thread periods, computing time, etc.) that must be taken into account if RT analysis is required, as in our case.
- We do not need all the distribution services normally provided by middlewares like CORBA. In our case, the components that make the application up and the connections among them are defined in the input models, and therefore services like naming, registering, searching, etc. are not needed.

Newly, 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 make the distribution possible and feasible, two artefacts have been defined: one belonging to the CBSD domain (the LOCALPROXYMANAGER component) and the other a class of the framework (the APPLICATIONDEPLOYER class). 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 it.

The APPLICATIONDEPLOYER class acts as the master node for application deployment, and thus it must be executed in its own node before the application can be deployed. The APPLICATIONDEPLOYER is in charge of notifying each LOCALPROXYMANAGER component (previously deployed in each and every node that make the application up) which component it must create (or destroy), which ports it must connect (or disconnect), and to which other nodes it must establish a TCP connection. Thus, the APPLICATIONDEPLOYER can be considered as a simplified Broker (Schmidt et al., 2000), without explicit register nor look-up services.

The LOCALPROXYMANAGER component is defined using V^{3}CMMV3CMM, in terms of ports, operations, states, etc., and its objectives are (i) to create and connect component instances in the 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 node, one of such components is automatically added to the architecture of the application, and then implemented by the framework distribution service. At the time of making application deployment, every LOCALPROXYMANAGER create ports that replicates local ports of the components hosted on others nodes as commanded by the APPLICATIONDEPLOYER. In this way communication is carried out as if both components were contained within the same node.

4 CONCLUSIONS AND FUTURE WORK

This paper has described the evolution of a previous work, where a OO framework for implementing CB designs was described. The new features consist on the support for component distribution. Distribution capacity was added in a regular way to the framework, respecting its original design. This regularity allows us to analyse the impact on the temporal char-

acteristics of the application that has a certain distribution of its components. Only the strictly necessary distribution services have been incorporated to the framework in order to perform component distribution, taking into account that the architecture is defined previously, and thus it is an input parameter to the master node. 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 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 is worth highlighting the following (1) the addition of heuristics to determine the number of threads and carry out the assignment of the activities of the components to these threads, (2) development of model transformations to instantiate the framework from an input component model, (3) adapt the implementation to be compliant with the Ravenscar profile for designing strict RT applications, and (4) development of a model transformation for generating input models for analysis tools.

ACKNOWLEDGEMENTS

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

REFERENCES

Buschmann, F., Henney, K., and Schmidt, D. (2007). *Pattern-Oriented Software Architecture, Volume 5: On Patterns and Pattern Languages*. John Wiley and Sons Ltd.

Iborra, A., Alonso, D., Ortiz, F., Franco, J., Sánchez, P., and Álvarez, B. (2009). Design of service robots. *IEEE Robot. Automat. Mag., Special Issue on Software Engineering for Robotics*, 16(1):24–33.

Pastor, J., Alonso, D., Sánchez, P., and Álvarez, B. (2010). 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*, LNCS, pages 167–180. Springer-Verlag.

Rosta (2010). Robot Standards and Reference Architectures (RoSTa), Coordination Action funded under EU's FP6.

Schmidt, D. (2006). Model-driven engineering. *IEEE Computer*, 39(2):25–31.

Schmidt, D., Stal, M., Rohnert, H., and Buschmann, F. (2000). *Pattern-oriented software architecture, volume 2: patterns for concurrent and networked objects*. Wiley.

Szyperski, C. (2002). *Component software: beyond object-oriented programming*. A-W, 2 edition.